VIA C3 Processor Alternate Instruction Set

Programming Reference

Version 0.25

(Review version, Incomplete)

VIA Confidential

November 2002

This is **Version 0.25** of the VIA C3 Processor
Alternate Instruction Set Programming Reference.

Table of Contents

CHAPTER

1

INTRODUCTION

This document describes an alternate set of instructions that may be used on the VIA C3 processor. The alternate instructions are the internal instructions of the VIA C3 processor and provide substantial additional function over the x86 instruction set. The *VIA C3 Alternate Instruction Set Application Note* describes how system software can enable these alternate instructions. This document is a programming reference describing the encoding and operation of alternate instructions.

1.1 BASIC CONCEPTS

The VIA C3 processor family is intended as a plug-replaceable, software-compatible alternative to the Intel Pentium III processor. Accordingly, the VIA C3 processor normally executes compatible instructions. The internal design of the VIA C3 processor, however, is quite different from the Pentium III internal design. In particular, the VIA C3 processor comprises two major components: a front-end that fetches x86 instruction bytes and translates them from x86 into internal instructions, and an internal microprocessor that executes these internal instructions.

1.2 OVERVIEW OF THIS PROGRAMMING REFERENCE

This Programming Reference is divided into sections describing internal instructions according to the registers used:

- **Chapter 1 – Introduction.** Describes the different execution units and programmer's model of the registers.

- **Chapter 2 –Instruction format.** Describes the instruction format and bit field definitions.
- **Chapter 3 – General instructions.** These instructions operate on the general x86 registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI as well as additional temporary general registers.
- **Chapter 4 – Floating-point instructions.** These instructions operate on the floating-point registers as well as additional temporary floating-point registers.
- **Chapter 5 – MMX™ instructions.** These instructions operate on the x86 MMX™ registers as well as additional temporary MMX™ registers.

1.3 GENERAL PURPOSE REGISTERS

These are 32 general purpose registers (GPRs) with similar usage to the x86 GPRs. GPR 0 always returns zero and can never be written. GPR 31 has a different special meaning in alternate instruction mode. It is the forward path data from the EA unit when referenced on load/store instructions (not LEA) as the base.

The GPRs have the required x86 functionality in that there are instruction controls that can select byte-oriented subsets (such as the low byte) of the 32-bit result data to be written into the result register.

The x86 instruction translator and associated microcode use the GPRs to store some x86 architecture registers such as the x86 GPRs and the x86 selector registers. These registers are directly referenced by code generated by the translator; thus, the mapping of EAX etc. into the native GPRs is fixed and considered part of the ISA. Other use of the GPRs is known only to the x86 microcode and thus is not defined as part of the ISA. A table below shows the usage of all GPRs and whether their use is known to the hardware (T means that the translator references the register, and H means that there are special hardware semantics for this register).

GPR Registers

Reg	Asm Label	Description	Hdw Vis
R0	0	Constant 0	H,T
R1	tmp1,X_EA	Xlator-ucode communication	T
R2	tmp2,X_ED	Xlator-ucode communication	T
R3	tmp3,X_ED2	Xlator-ucode communication	T
R4	tmp4	Normal microcode temp	
R5	tmp5	Normal microcode temp	
R6	tmp6	Normal microcode temp	
R7	tmp7	Normal microcode temp	
R8	ES	(1)	T,H
R9	CS	(1)	T,H
R10	SS	(1)	T,H
R11	DS	(1)	T,H
R12	FS	(1)	T,H
R13	GS	(1)	T,H
R14	LDTR	(1)TR (upper), LDTR (lower)	H
R15	tmp15	(1)Special emulator temp	H
R16	EAX		T
R17	ECX		T
R18	EDX		T
R19	EBX		T
R20	ESP		T
R21	EBP		T
R22	ESI		T
R23	EDI		T
R24	tmp24	Exception handler temp	
R25	tmp25	Exception handler temp	
R26	tmp26	Exception handler temp	
R27	tmp27	Exception handler temp	
R28	tmp28	Exception handler temp	
R29	tmp29	Exception handler temp	
R30	tmp30	Exception handler temp	

R31	tmp31	Exception handler temp	
	LEA_FWD	EA forward path for load/store instructions/ Normal data for non load/store instructions	T,H

(1) XPUSH of 32-bits pushes 0x0000 | [15:0]

1.4 FLOATING POINT REGISTERS

The floating point data registers are similar to those in x86 architecture except that:

- there are extra scratch registers available, and
- all of the registers may be directly accessed in addition to the x86 stack semantics.

Reg	Asm Label	Description	Type
FP0	FP0	x86 FP Stack Register 0	RW
FP1	FP1	x86 FP Stack Register 1	RW
FP2	FP2	x86 FP Stack Register 2	RW
FP3	FP3	x86 FP Stack Register 3	RW
FP4	FP4	x86 FP Stack Register 4	RW
FP5	FP5	x86 FP Stack Register 5	RW
FP6	FP6	x86 FP Stack Register 6	RW
FP7	FP7	x86 FP Stack Register 7	RW
FP8	FP8		RW
FP9	FP9		RW
FP10	FP10		RW
FP11	FP11		RW
FP12	FP12		RW
FP13	FP13		RW
FP14	FP14		RW
FP15	FP15		RW
FP16: FP31	FP16:FP31	FP Scratch registers 16 to 31	RW

1.5 MMX™ REGISTERS

Xxxx

1.6 INTERNAL PROCESSOR REGISTERS

xxxx.

CR0 …

CHAPTER

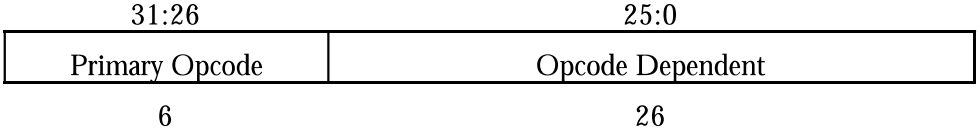
2

INSTRUCTION FORMAT

This chapter describes the format and bit fields of the alternate instructions.

2.1 GENERAL FORMAT & PRIMARY OPCODES

Alternate instruction formats are all instructions 32 bits long with a 6-bit primary opcode field:



Some of the primary opcodes have extended opcodes in other bits of the instruction.

2.1.1 PRIMARY OPCODES

The primary opcodes are:

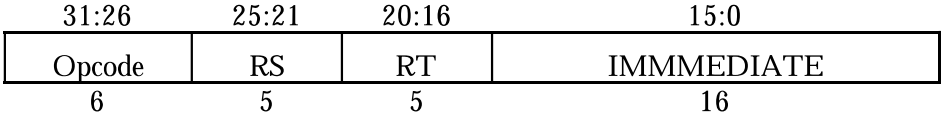
28:26→ 31:29↓	0	1	2	3	4	5	6	7
0							XJ	
1	ORIU	ADDI	ANDIU	ANDIL	ANDI	ORI	XORI	XORIU
2		FPU	XLFP	XSFP	MMX		XLMMX	XSMMX
3								
4	XALU	XALUI	XALUR	XALUIR				
5	XMISC			XLEAI	XLEAD			
6	XL	XL2	XL3	XLBI	XLDESC	XIOR	XPOPBR	XPOP
7	XS	XS2	XPUSHI	XSI	XPUSHIP	XIOW	XSU	XPUSH

The shaded opcodes have extended opcodes as defined in subsequent sections. Cross-hatched unlabeled opcodes represent primary opcodes that are not included in the alternate instruction set.

2.2 INSTRUCTION FORMATS USED

2.2.1 IMMEDIATE (OR I-TYPE) INSTRUCTION FORMATS

Some alternate instructions use the immediate instruction format:



The opcode is one of the primary opcodes. The source and target operands for these I-type instructions are:

GPR[RT] ← GPR[RS] *opcode* IMMEDIATE

Note that the destination is RT in this case rather than RD as for R-type instructions (which are described in a later section).

The I-type instructions are:

28:26→ 31:29↓	0	1	2	3	4	5	6	7
1	ORIU	ADDI	ANDIU	ANDIL	ANDI	ORI	XORI	XORIU

2.2.1.1 XALU-Type Instruction Formats

The XALU-type instruction format is used for x86-style ALU instructions defined using the XALU[I][R] primary opcodes. It has special control fields to allow most x86 ALU semantics to be specified in a single 32-bit instruction.

31:26	25:21	20:16	15:11	10:0
XALU[R]	RS	RT	RD	Function
XALUI[R]	RS	Const	RD	Function
6	5	5	5	11

The source and target operands for XALU[R] instructions is basically the same as for R-type instructions. XALUI[R] instructions are similar except that they allow an encoded immediate value, to be used instead of RT. is a multi-part field that defines the function to be performed.

GPR[RD] ← GPR[RS] *function* GPR[RT]
GPR[RD] ← GPR[RS] *function* Const value

The R versions of the XALU[I][R] instructions cause the x86 result flags to be set as defined for the particular instruction type.

Implementation Note: As described in the instruction definition section, not every version of every extended opcode is needed for all of the XALU[I][R] instructions.

2.2.1.2 Const Field (for XALUI forms)

The 5-bit field allows a small constant to be used as an immediate value. The following table lists the values for .

Const	Mnemonic	Description
00000_2	0	Constant zero
00001_2	1	Constant one
00010_2	2	Constant two
00011_2	4	Constant four
00100_2	8	Constant eight
00101_2	16	Constant 16
00110_2	24	Constant 24
00111_2	32	Constant 32
01000_2	LoWdMask	0x0000FFFF (used for AltInst sel register moves)
01001_2	-1	Constant minus one
01010_2	-2	Constant minus two
01011_2	-4	Constant minus four
01100_2	-8	Constant minus eight
01101_2		
01110_2		
01111_2	5	Constant 5

10000_2	OS	Operand size: +2 or +4 depending on TSR.OS
10001_2	3	Constant 3
10010_2	6	Constant 6
10011_2	7	Constant 7
10100_2	9	Constant 9
10101_2	LoByteMask	Constant 0x00FF
10110_2	OSSH	OS-related shift magic count: 16 for OS==0, 0 for OS==1
10111_2		
11000_2	MOS	Minus operand size: -2 or -4 depending on TSR.OS
11001_2	MGS	
11010_2		
11011_2		
11100_2	DFind1	Minus or Plus 1 respectively depending on EFLAGS.DF
11101_2	DFindOS	Minus or Plus Operand Size (2 or 4) (defined in TSR.OS) respectively depending on EFLAGS.DF
11110_2	IMMED	Use value in the IMMED Register
11111_2	DISP	Use value in the DISP Register

2.2.1.3 Function Codes for XALU-Type

10:8	7:5	4:0
Ind	DPcntl	SubOp
3	3	5

SubOp Field

This field describes the ALU function. It is similar in concept to the 6-bit extended opcode for special instructions. The extended opcodes for the primary opcodes are:

2:0→ / 4:3↓	0	1	2	3	4	5	6	7
0	**SHL**		**SHR**	**SAR**	**ROL**	**ROR**	**RCL**	**RCR**
1	**INC**	**CMPS**	**DEC**		**IMUL**	**MUL**	**IDIV**	
2	**ADD**	**ADC**	**SUB**	**SBB**	**AND**	**OR**	**XOR**	**NOR**
3		**CTC2**				**SETCC**	**MFLOU**	**MFLOI**

The fourth-row opcodes have special semantics over the normal logical operations: the C2 suffix indicates that the destination is a CP2 control register (vs. a GPR. Only some CP2 registers may be used as the destination.

DPcntl

The DPcntl field controls (1) the source byte selection, (2) the size of the writes into the register file, and (3) the size of the result over which condition codes are calculated.

In the table below, BE indicates which bytes of the register file are written back ( ); and CC is the portion of the result that the condition codes other than AF and PF are calculated over

(these are always based on carry from bit 3 and the low order 8 bits of the result). Condition codes are always calculated over the low-order bits of the result.

DPcntl	Mnemonic	BE	CC
000_2	32	00001111_2	32 bits
001_2	16	00000011_2	16 bits
010_2	LL	00000001_2	8 bits
011_2	HL	00000010_2	8 bits
100_2	LH	00000001_2	8 bits
101_2	HH	00000010_2	8 bits
110_2	(reserved)		
111_2	(reserved)		

32

Provides 32 bit operations. All 32 bits are written back to the register file. Condition codes are calculated with an operand size of 32 bits.

16

Provides for 16-bit operations. The low 16 bits are written back to the register file. Condition codes are calculated on the low-order 16 bits of the result.

LL

Provides for 8-bit low-low byte operations. The low 8 bits of each operand are used as the sources and the low 8 bits of the result are written back to the register file. Condition codes are calculated on the low-order 8 bits of the result.

HL

Provides for 8-bit high-low byte operations. Bits 15:8 of the left operand are shifted right and operated on with the low byte of the right operand. Condition codes are calculated on the low-order 8 bits of the result. The low 8 bits are shifted left and written into bits 15:8 of the target register.

HH

Provides for 8-bit high-high byte operations. Bits 15:8 of both operands are shifted right and operated on. Condition codes are calculated on the low-order 8 bits of the result. . The low 8 bits are shifted left and written into bits 15:8 of the target register.

LH

Provides for 8-bit low-high byte operations. Bits 15:8 of the right operand are shifted right and operated on with the low byte of the left operand. Condition codes are calculated on the low-order 8 bits of the result. The low 8 bits are into the low byte of the target register.

(reserved)

Using reserved DPcntl codes will result in unpredictable results.

Ind

The field is not useful for alternate instruction mode and must always be all zeros. Non-zero values will result in unpredictable results.

2.2.2 XMISC-TYPE INSTRUCTION FORMATS

The XMISC-type instruction format is used for miscellaneous x86-related instructions defined using the XMISC primary opcode. It is like the XALU format except that the field is special for each particular instruction.

31:26	25:21	20:16	15:11	10:6	5:0
XMISC	RS	RT	RD	SubFunc	Defined By Instruction
6	5	5	5	5	6

Subfunction Field

This field describes the specific instruction. It is similar in concept to the 6-bit extended op-code for R-form instructions. The extended opcodes for the new primary opcode are

8:6→ / 10:9↓	0	1	2	3	4	5	6	7
0	**XRET**	**XCNULL**	**XRFP**	**XHALT**	**SBF/SBN**	**MFHI**		**CFPFL**
1	**XTI**	**XTII**			**MTC0**	**MFC0**	**MTCNT**	**MFCNT**
2	**XMDB**	**XMDBI**	**DMTC1**	**DMFC1**	**MTC1**	**MFC1**	**CTC1**	**CFC1**
3		**DMTMD**	**DMTC2**		**MTC2**	**MFC2**		**CFC2**

2.2.3 XLS-TYPE INSTRUCTION FORMATS

The new XLS-Type instruction format is used for x86-style load and store (Xlx, XSx) instructions This form is highly encoded to allow most x86 load/store semantics to be specified in a single 32-bit instruction.

31:26	25:21	20:16	15:11	10:0
Opcode	RS	Base	Offset	Function
6	5	5	5	11

The source and target operands for XL-type instructions are similar to those for load and store instructions (except that the Base register is in a different register field). The x86-style EA is calculated as a base register plus an offset. The x86 selector register and other x86 addressing semantics are specified in the field.

Load: RS ← memory(selector, GPR[Base] + Offset)

Store: RS → memory(selector, GPR[Base] + Offset)

This general instruction format is also used for XLEA instructions which do not actually perform load or stores but rather calculate an offset address. However, the XLEA instructions have some unique fields and should really be considered as special format instructions.

Offset

The 5-bit field allows a small constant to be used as an immediate value. The following table lists the values for .

Offset	Mnemonic	Description
00000_2	0	Constant zero
00001_2	1	Constant one
00010_2	2	Constant two
00011_2	4	Constant four
00100_2	8	Constant eight
00101_2	16	Constant 16
00110_2	24	Constant 24
00111_2	32	Constant 32
01000_2	10	Constant ten
01001_2	-1	Constant minus one
01010_2	-2	Constant minus two
01011_2	-4	Constant minus four
01100_2	-8	Constant minus eight
01101_2		
01110_2		
01111_2	5	Constant 5

10000_2	OS	Operand size: +2 or +4 depending on TSR.OS
10001_2	PDOS	
10010_2		
10011_2		
10100_2		
10101_2		
10110_2		
10111_2		
11000_2	MOS	Minus operand size: -2 or -4 depending on TSR.OS
11001_2	MGS	
11010_2	MDOS	
11011_2		
11100_2	DFind1	Minus or Plus 1 respectively depending on EFLAGS.DF
11101_2	DFindOS	Minus or Plus Operand Size (2 or 4) (defined in TSR.OS) respectively depending on EFLAGS.DF
11110_2		
11111_2	DISP	Use value in the DISP Register

Function Codes

The XLx and XSx instructions are used to implement x86 load and store instructions. The function field encodes most of the x86 peculiar load/store semantics:

10:9	8	7:6	5:2	1	0
SubOp	AddrSize-1	Size-2:1	Sel	Size-0	AddrSize-0
2	1	2	4	1	1

Note that GPR indirection (via the IIR) is not available in the normal load/store format. If register indirection is needed, a XLEAx instruction (which allows indirection) must be used to calculate the address followed by the load/store instruction using the output register of the XLEA as the Base register.

AddrSize

Indicates the address size for the effective address calculation of this XL , or XS instruction.

AddrSize	Mnemonic	Address Size Used	
00_2	AS	TSR.AS	
01_2	SAS	TSR.SAS	
01_2	OS	TSR.OS	If XLEAD or XLEAI
10_2	16	16	
11_2	32	32	

Sel

Specifies the selector descriptor used for virtual address calculation and limit checking. The bit encoding is:

SEL Value	Mnemonic	Descriptor Used
0000_2	ES	ES
0001_2	CS	CS
0010_2	SS	SS
0011_2	DS	DS
0100_2	FS	FS
0101_2	GS	GS
0110_2	GDT	GDT
0111_2	LDT	LDT
1000_2	IDT	IDT
1001_2	TSS	TSS
1010_2	FLAT	Flat 32-bit address space
1011_2	T0	Temp0
1100_2		
1101_2		
1110_2		
1111_2	indSEL	Use value in field of the TSR

Architecture Note: The register encoding values for GDT and LDT are important: they differ only in the low-order bit which corresponds to the TI bit in a selector that selects GDT or LDT. This bit has a mux on it that selects the saved TI bit or the bit in the field based. This mux is set-up be the XTI instruction, but affects the following instruction —assumed to be a XLDES. See these instructions for more description of this magic.

Usage Note: The FLAT descriptor is set-up (by microcode) as a special "flat" segment used for implementing I/O accesses. This is a 32-bit, flat RW segment.

Size

The size field indicates which portion of the destination register to update for loads, and the size of the source operand for stores. It has different encodings for the various types of x86 load/store opcodes. The high order two bits come from the Size-2:1 field, and the low order bit is bit 1 of the instruction. The XLEAD and XLEAI instructions have an additional (highest order) bit which comes from bit 2 of the instruction.

All Load/Store Instructions (Include LEAx) Except XLDESC,XLFP/XSFP

Size	Mnemonic	Data Size	
000_2	16	16 bits-low	
010_2	32	32 bits	
100_2	AS	address size-16 or 32	If 16 bits, it is low 16 bits.
110_2	OS	operand size - 16 or 32	If 16 bits, it is low 16 bits.
001_2	8L	8 bits-low	
011_2	8H	8 bits-high	
011_2	16H	XPUSH: operand size 16 (31:16) or 32 (31:16) zero extended	
101_2	64	64 bits In this case, the destination register (RS) must have an even address; the data will be loaded into RS and RS | 1.	Note that there a store of 64 bits is an invalid (missing) instruction.
111_2	IND	defined by TSR.DPcntl	
111_2	GS	16 or 32 depending on XBR bit 27	Gate Size (only on XPUSH)

XLEAD and XLEAI same as previous plus:

Size	Mnemonic	Data Size
1000_2	SAS	stack address size-16 or 32
1001_2		undefined
1010_2		undefined
1011_2		undefined
1100_2		undefined
1101_2		undefined
1110_2		undefined
1111_2		undefined

SubOp

This field controls the use of the effective-address, linear-address, physical-address, and protection-calculation hardware for performing operations other than simple loads and stores. It is used mostly to precipitate exceptions before modifying the architectural state (thus facilitating instruction restartability). It is also used to control bus operations.

Various types of load/store instructions have different encodings for this field:

SubOp Type 1 – For All Load/Store Instructions Except Below Types

SubOp	Mnemonic	Load Desc	Store Desc
00_2	[norm]	Normal Load	Normal Store
01_2	rwv	Verify Writeable	Verify Writeable
10_2	sup	Assume CPL = 0	Assume CPL = 0
11_2	lock	Locked Load	UnLock Store

SubOp Type 2 – XSU & XPOP Instructions

SubOp	Mnemonic	Load Desc	Store Desc
00_2	[norm]	Normal Load	Normal Store
01_2	str_int	Decr/test COUNT Allow Interrupt	String Semantics Allow Interrupt
10_2	str_testcnt	Test COUNT Don't allow inter-rupt	Test COUNT Don't allow inter-rupt
11_2			

SubOp Type 3 – For XLDESC (details on xldesc in page **Error! Bookmark not defined.**)

SubOp	Mnemonic	Load Desc	
00_2	dschk	Data Segment checks	
01_2	sschk	Stack Segment checks	N/A
10_2			
11_2	NoChk		N/A

SubOp Type 3 – For XLDESC_CS (details on xldesc_cs in page **Error! Bookmark not defined.**)

SubOp	Mnemonic	Load Desc	
00_2	jmp_call	JMP and CALL checks	
01_2	retf	Return far checks	N/A
10_2	spec	Checks for special microcode	
11_2	NoChk		N/A

SubOp Type 4 – XL2 & XS2 Instructions

SubOp	Mnemonic	Load Desc	Store Desc
00_2	[norm]	Normal Load	
01_2	tickle	Tickle Load	Tickle Store
10_2	tickle_ lock	Tickle Locked Load	
11_2	suplk	Assume CPL = 0, Locked Load	Assume CPL = 0, Un-Lock Store

SubOp Type 5 – For XIO

SubOp	Mnemonic	Load Desc	Store Desc
00_2	[norm]	I/O Read	I/O Write
01_2	special	Interrupt Ack	Special Bus Cycle (type defined by low three address bits)
10_2			
11_2			

SubOp Type FP – XLFP & XSFP Instructions

SubOp	Mnemonic	Load Desc	Store Desc
00_2	[norm]	Normal Load	
01_2	tickle	Tickle Load	Tickle Store
10_2	NORM_REC	Load w/ RFP	Store w/ RFP
11_2	TICK_REC	Tick Load w/ RFP	Tick Store w/ RFP

norm

Performs a normal load or store operation as specified in the rest of the instruction. This is the default in assembler instructions and does not have to be specified.

rwv (used only with loads)

Performs the load operation as specified in the instruction, except that it performs all protection and access right checks as though this were a store operation. This is meant to be used to force read-modify-write operations that are going to fault on the write to fault on the read instead. This is used to force exceptions to occur before the flags are modified, and to prevent partial modification of the target memory location on unaligned references.

tickle

Performs the effective-address, virtual-address, and physical-address calculations as specified in the instruction including all protection and access-right checks. However, the actual transfer of data is inhibited. This is meant to be used to verify that all parts of a data structure will not generate faults before a portion of it is modified.

lock (used only with loads)

Performs the load operation as specified in the instruction with x86 Lock semantics. The fully compatible definition invalidates the cache line containing the data and asserts the LOCK#

signal on the external bus. Locked loads are also tested for write privileges as described above in LSrmv.

The bus control unit manages the synchronization of asserting and deasserting LOCK#. It basically counts consecutive locked loads operations (reaching the bus unit) and deasserts LOCK# after the last of an equivalent number of stores. However, the locked store portions of the locked RMW sequence must be specified with a Lock SubOp.

Note that the TSR.LK bit (the x86 instruction had a valid LOCK prefix) forces a LOCK sequence only if the SubOp is RVW. If the SubOp is nom, for example, the LOCK prefix is ignored.

str_int

Adds the following semantics to the load/store instruction:

- If the COUNT register is 0 (considering any effect of the previous instruction), do not perform the load or store operation and signal "stop string generation" to the translator.
- Else, perform the operation and decrement COUNT by one (forwarding results to next instruction test of COUNT).

In addition, an external interrupt, data breakpoint trap exception, or TF exception is allowed to occur following successful completion of the associated load/store instruction.

str_testcnt

Adds the following semantics to the load/store instruction:

If the COUNT register is 0 (considering any effect of the previous instruction), do not perform the load or store operation and signal "stop string generation" to the translator.

special

Allows generation of interrupt acknowledge on loads, and special cycles on stores the low-order bits of the address defines the special bus operation (the byte enable lines on the bus define the cycle type).

sup

Performs the otherwise specified operation but performs all access right and paging tests as though CPL == 0. This is used for descriptor table accesses.

suplk

Performs the load with the semantics of and combined.

Unaligned Operations

There are two different meanings of

1. The data operand spans across two 8-byte-aligned memory units. This case is handled automatically by the hardware by decomposing the load/store operation into the correct two load/store operations for each portion. This will called .

Note that this is technically different than the P54 which uses a four-byte boundary for generating multiple bus cycles (if the data size is four bytes or less) rather than eight.

2. The data is aligned such that it meets the x86 architecture definition of unaligned. In this case, if CPL == 3 && (EFLAGS.AC & CR0.AM) then an alignment exception occurs. This will called

The architecture alignment boundary for data is the same as for the P54 (some of this must be enforced by microcode).

MMX Load and Store Instructions

The MMX load and store instructions are formatted identically to the LS form load store instructions, with the exception that the load target and the store source registers are MMX operand registers. The size 64 store is a true size_64 store in that page protection checking, etc. is performed for the entire 64 bit operand. Note that this is not the case in the standard form store-64 instructions.

ADD LOAD_ALU INFO HERE!!

FPU Load and Store Instructions

The FPU load and store instructions are formatted identically to the LS form load store instructions, with the exception that the load target and the store source registers are FPU operand registers and the size field (bits 7,6,1 in the instruction) are encoded to represent the format of the FPU load/store (see table below). The size 64 store is a true size_64 store in that page protection checking, etc. is performed for the entire 64 bit operand. Note that this is not the case in the standard form store-64 instructions.

2.2.3.1 XLFP and XSFP size encodings

Size	Mnemonic	Data Size
000_2	Int 16 (H)	16
001_2	Int 64 (L)	64
010_2	Int 32 (W)	32
011_2	--	
100_2	Exp 16 (Exp)	16
101_2	FP Ext (E)	64
110_2	FP Sng (S)	32
111_2	FP Dbl (D)	64

2.2.4 CP1 (FLOATING POINT) INSTRUCTION FORMATS

See the floating-point instruction description section.

2.2.5 CP4 (MMX) INSTRUCTIONS

Coprocessor 4 primary opcodes are used for implementation of the MMX instruction set. Please refer to the MMX section for more detail.

CHAPTER

3

GENERAL INSTRUCTIONS

This chapter describes the instructions that operate on the general purpose registers (GPRs) as well as the additional temporary registers.

3.1 ALU INSTRUCTIONS

3.1.1 IMMEDIATE INSTRUCTIONS

Usage Note: The following table summarizes what gets modified how by the 16-bit immediate logical instructions:

Instruction	Upper RT	Lower RT
ANDI	0	RS(15:0) & Immed
ANDIL	RS(31:16)	RS(15:0) & Immed
ANDIU	RS(31:16) & Immed	RS(15:0)
ORI	RS(31:16)	RS(15:0) | Immed
ORIU	RS(31:16) | Immed	RS(15:0)
XORI	RS(31:16)	RS(15:0) ^ Immed
XORIU	RS(31:16) ^ Immed	RS(15:0)

3.1.1.1 ADDI - Add Immediate

Encoding

	31:26	25:21	20:16	15:0
ADDI:	001001_2	RS	RT	Immediate
	6	5	5	16

Format

```
ADDI    RT,RS,0x1234
```

Description

The contents of GPR is added to the field extended with 0x0000. The result is written back to GPR .

Operation

```
GPR[RT] = GPR[RS] + IMMEDIATE;
```

Flags Setting

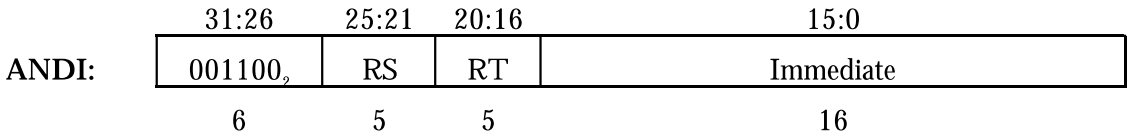
None

Exceptions

None

3.1.1.2 ANDI - AND Immediate

Encoding



Format

```
ANDI    RT,RS,0x1234
```

Description

the field and ANDs it to 32-bit GPR with the result replacing 32-bit GPR .

Operation

```
GPR[RT] = GPR[RS] & IMMEDIATE;
```

Flags Setting

None

Exceptions

None

3.1.1.3 ANDIL - AND Immediate Lower

Encoding

	31:26	25:21	20:16	15:0
ANDIL:	001011_2	RS	RT	Immediate
	6	5	5	16

Format

```
ANDIL   RT,RS,0x1234
```

Description

The contents of GPR are ANDed with the field extended with 0xFFFF. The result is written back to GPR .

Operation

```
GPR[RT] = GPR[RS] & (0xFFFF0000 | IMMEDIATE);
```

Flags Setting

None

Exceptions

None

3.1.1.4 ANDIU - AND Immediate Upper

Encoding

	31:26	25:21	20:16	15:0
ANDIU:	001010_2	RS	RT	Immediate
	6	5	5	16

Format

```
ANDIU   RT, RS, 0x6789
```

Description

The contents of GPR are ANDed with the immediate field shifted left 16 and extended on the right with 0xFFFF. The result is written back to GPR .

Operation

```
GPR[RT] = GPR[RS] & ((IMMEDIATE << 16) | 0x0000FFFF);
```

Flags Setting

None

Exceptions

None

3.1.1.5 ORI - OR Immediate

Encoding

	31:26	25:21	20:16	15:0
ORI:	001101_2	RS	RT	Immediate
	6	5	5	16

Format

```
ORI   RT, RS, 0x1234
```

Description

the field and ORs it to 32-bit GPR with the result replacing 32-bit GPR .

Operation

```
GPR[RT] = GPR[RS] | IMMEDIATE;
```

Flags Setting

None

Exceptions

None

3.1.1.6 ORIU – OR Immediate Upper

Encoding

	31:26	25:21	20:16	15:0
ORIU:	001000_2	RS	RT	Immediate
	6	5	5	16

Format

```
ORIU    RT, RS, 0x1234
```

Description

The contents of GPR are ORed with the immediate field shifted left 16. The result is written back to GPR .

Operation

```
GPR[RT] = GPR[RS] | (IMMEDIATE << 16);
```

Flags Setting

None

Exceptions

None

3.1.1.7 XORI - XOR Immediate

Encoding

	31:26	25:21	20:16	15:0
XORI:	001110_2	RS	RT	Immediate
	6	5	5	16

Format

```
XORI    RT,RS,0x1234
```

Description

the field and XORs it to 32-bit GPR with the result replacing 32-bit GPR .

Operation

```
GPR[RT] = GPR[RS] ^ IMMEDIATE;
```

Flags Setting

None

Exceptions

None

3.1.1.8 XORIU - XOR Immediate Upper

Encoding

	31:26	25:21	20:16	15:0
XORIU:	001111_2	RS	RT	Immediate
	6	5	5	16

Format

```
XORIU   RT,RS,0x1234
```

Description

The contents of GPR are XORed with the immediate field shifted left 16. The result is written back to GPR .

Operation

```
GPR[RT] = GPR[RS] ^ (IMMEDIATE << 16);
```

Flags Setting

None

Exceptions

None

3.1.2 X86-SEMANTIC INSTRUCTIONS

3.1.2.1 XADC[I][R] - X86 Add with Carry

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XADC: **XADCR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	ADC 10001_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:06
XADCI: **XADCIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	ADC 10001_2
	6	5	5	5	3	3	5

Format

```
XADC    RD,RS,RT
XADCI   RD,RS,2
```

Description

The contents of GPR are added to either the contents of GPR or the immediate value specified in depending on the primary opcode. The EFLAGS.CF is also added to the result.

Operation

```
GPR[RD] = GPR[RS] + GPR[RT] + CF
GPR[RD] = GPR[RS] + Const   + CF
```

Flags Setting

CCarith

Exceptions

None

3.1.2.2 XADD[I][R] - X86 Add

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XADD: **XADDR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	ADD 10000_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:06
XADDI: **XADDIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	ADD 10000_2
	6	5	5	5	3	3	5

Format

```
XADD    RD,RS,RT
XADDI   RD,RS,2
```

Description

The contents of GPR RS are added to either the contents of GPR or the immediate value specified in depending on the primary opcode.

Operation

```
GPR[RD] = GPR[RS] + GPR[RT]
GPR[RD] = GPR[RS] + Const
```

Flags Setting

CCarith

Exceptions

None.

3.1.2.3 XAND[I][R] - X86 And

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XAND: **XANDR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	AND 10100_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:06
XANDI: **XANDIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	AND 10100_2
	6	5	5	5	3	3	5

Format

```
XAND    RD,RS,RT
XANDI   RD,RS,2
```

Description

The contents of GPR RS are logically ANDed to either the contents of GPR RT or the value immediate specified in depending on the primary opcode.

Operation

```
GPR[RD] = GPR[RS] & GPR[RT]
GPR[RD] = GPR[RS] & Const
```

Flags Setting

CClog

Exceptions

None.

3.1.2.4 XCMPS - X86 A-Stage Compare String

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XCMPS:	100000_2	RS	RT	$xxxxx_2$	xxx_2	DPcntl	XCMPS 01001_2
	6	5	5	5	3	3	5

Format

```
XCMPS   RS,RT
```

Description

If the COUNT register modulo TSR.AS is not zero then it is decremented by one. The contents of GPR are compared in the to the contents of GPR according to the size specified by the field as described in the following table. If any of the conditions to terminate or interrupt an x86 string compare are true then the instruction is nullified and the translator is signalled to stop generating the sequence for a repeat string and control is transferred to microcode. The conditions under which a string compare terminate are either the COUNT register modulo TSR.AS is zero or the comparison results in equality and TSR.REPN is set or the comparison results in inequality and TSR.REPN is clear. The conditions under which a string compare is interrupted is an external interrupt or debug trap. If the string compare is interrupted or a debug trap is triggered before the terminating condition is met then hardware will set XCR[STRINT_BIT] at the .T that completes the current x86 instruction; IP is not advanced in this case. See the operation description below.

DPcntl	Mnemonic	Compare
000_2	32	31:0
001_2	16	15:0
010_2	LL	7:0
110_2	OS	31 or 15:0

Usage Note: XCMPS is needed to allow the translator to generate the instruction sequence for x86 string compares and scan strings.

A sample usage is

```
MTCNT       ECX                 // mod AS automatically
XPOP.8L.AS  tmp2,DSdesc,ESI,1,str_testcnt
XPOP.8L.AS  tmp3,ESdesc,DSI,1,str_testcnt
XCMPS.8L    tmp2,tmp3
```

which loads the COUNT register with CX or ECX based on the address size, then checks for COUNT equal to zero and if not zero then loads a byte from DS:[E]SI and from ES:[E]DI and

compares the two bytes. The loads and the compare are repeatedly generated by the translator until COUNT goes to zero or the terminating condition of equality is reached or there is an interrupt or debug trap. Note that XCMPS does not modify EFLAGS, microcode must determine if the original count in [E]CX was non-zero and set EFLAGS in that case.

Operation

```
if (COUNT.AS != 0) {
    COUNT = COUNT - 1;
}
REP_DONE = (COUNT.AS == 0 or not((GPR[    ] == GPR[    ]) ^ TSR.REPN));
if (REP_DONE or interrupt or IT or TF or DBRKPT) {
    send signal to xlator to stop;
    nullify inst;
    nullify pipeline;
    transfer control to corresponding microcode entry point;
}
```

//--The following action taken at .T that completes current x86 instruction

```
if (REP_DONE == 0) {
    IP          advance;
    XCR[STRINT_BIT] = 1;
}
```

Flags Setting

None

Exceptions

None.

3.1.2.5 XDECIR - X86 Decrement

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XDECIR:	100011_2	RS	Const	RD	000_2	DPcntl	DEC 01010_2
	6	5	5	5	3	3	5

Format

```
XDECIR  RD,RS,1
```

Description

The contents of the immediate specified in are subtracted from the contents of GPR RS.

Usage Note: XDECIR is needed to perform a single-cycle X86 DEC function. It is the same as a XSUBIR instruction (subtract immediate) except that it sets EFLAGS differently than XSUBIR.

Operation

```
GPR[RD] = GPR[RS] - Const
```

Flags Setting

CCinc

Exceptions

None.

3.1.2.6 XIDIV - X86 Signed Divide Step

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:5	4:0
XIDIV:	100000_2	$xxxxx_2$	$xxxxx_2$	00000_2	xx_2	0_2	Cntl	IDIV 01110_2
	6	5	5	5	2	1	3	5

Format

```
XIDIV.STEP
```

Description

Performs an signed divide of the dividend in (loaded by DMTMD.DVD) by the divisor in (loaded by DMTMD.DVS).. The size of the dividend, divisor, and quotient are specified by the field and summarized in the following table. The dividend and divisor must be loaded using the DMTMD.DIV.size instruction before issuing XDIV. For 64b-by32b divide the dividend is loaded separately using DMTMD.DIVidend.32 instruction. XIDIV does not allow register indirection or (immediate) values. XIDIV computes as many quotient bits as are specified in the table below. The quotient is written to the LO register, the remainder is written to the HI register. A MFLOU.size instruction is used to get the quotient. A MFHI instruction is used to get the remainder. This DIVIDE instruction is a divide styep instruction that produces one bit of result per cycle. Unlike previous implementations, C3 microcode will control all steps of the multiply process.

Cntl	Mnemonic	Details
000_2	Step	General Divide Step
001_2	OVF	Detect Divide Ovf
010_2	REM	Remainder Adjust
011_2	QUO	Quotient Adjust
100_2		
101_2		
110_2		
111_2		

Flags Setting

None

Exceptions

None

3.1.2.7 XIMUL[I] - X86 Signed Multiply Step

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:5	4:0
XIMUL:	100000_2	RS	RT	00000_2	xx_2	0_2	Cntl	IMUL 01100_2
	6	5	5	5	2	1	3	5

	31:26	25:21	20:16	15:11	10:9	8	7:5	4:0
XIMULI:	100001_2	RS	Const	00000_2	xx_2	0_2	Cntl	IMUL 01100_2
	6	5	5	5	2	1	3	5

Format

```
XIMUL     RS,RT
XIMULI    RS,2
```

Description

Performs a signed multiply of the operands in the MUL operand registers. The product is written to the LO register. The size of the factors and product are specified by the field and summarized in the following table. XIMUL does not allow register indirection. If is used it must be coded as IMMED.

Cntl	Mnemonic	Info
[2]	First	First Multiply Clock (choose ops)
[1]	LastClk	Load result into LO
[0]		

Operation

Flags Setting

None

Exceptions

None

3.1.2.8 XMUL - X86 Unsigned Multiply Step

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:5	4:0
XMUL:	100000_2	RS	RT	00000_2	xx_2	0_2	Cntl	MUL 01101_2
	6	5	5	5	2	1	3	5

Format

```
XMUL     RS,RT
```

Description

Performs an unsigned multiply of and . The product is written to the LO register. The size of the factors and product are specified by the field and summarized in the following table. XIMUL does not allow register indirection. If is used it must be coded as IMMED.

Cntl	Mnemonic	Info
[2]	First	First Multiply Clock (choose ops)
[1]	LastClk	Load result into LO
[0]		

Operation

Flags Setting

None.

Exceptions

None

3.1.2.9 XINCIR - X86 Increment

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XINCIR:	100011_2	RS	Const	RD	000_2	DPcntl	INC 01000_2
	6	5	5	5	3	3	5

Format

```
XINCIR    RD,RS,1
```

Description

The contents of the immediate specified in are added to the contents of GPR RS.

Usage Note: XINCIR is needed to perform a single-cycle X86 INC function. It is the same as a XADDIR instruction (add immediate) except that it sets EFLAGS differently than XADDIR.

Operation

```
GPR[RD] = GPR[RS] + Const
```

Flags Setting

CCinc

Exceptions

None.

3.1.2.10X8NOR[I][R] - X86 NOR

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
X8NOR: **X8NORR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	NOR 10111_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:06
X8NORI: **X8NORIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	NOR 10111_2
	6	5	5	5	3	3	5

Format

```
X8NOR   RD,RS,RT
X8NORI  RD,RS,0      // Same as RD = NOT(RS)
```

Description

The contents of GPR are logically NORed with the immediate value specified in depending.

Usage Note: By specifying a constant of zero, this instruction directly performs an X86 NOT instruction function. This is the intended function of this instruction.

Operation

```
GPR[RD] = ~ (GPR[RS] | GPR[RT])
GPR[RD] = ~ (GPR[RS] | Const)
```

Flags Setting

None

Exceptions

None

3.1.2.11X8OR[I][R] - X86 OR

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
X8OR: **X8ORR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	OR 10101_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:06
X8ORI: **X8ORIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	OR 10101_2
	6	5	5	5	3	3	5

Format

```
X8OR    RD,RS,RT
X8ORI   RD,RS,0     // Moves RS to RD
```

Description

The contents of GPR RS are logically ORed to either the contents of GPR RT or the value immediate specified in depending on the primary opcode.

Operation

```
GPR[RD] = GPR[RS] | GPR[RT]
GPR[RD] = GPR[RS] | Const
```

Flags Setting

CClog

Exceptions

None.

3.1.2.12 XRCL - X86 Rotate Left Thru Carry

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:06
XRCLI:	100001_2	RS	00001_2	RD	000_2	DPcntl	RCL
XRCLIR:	100011_2						00110_2
	6	5	5	5	3	3	5

Format

```
XRCLI   RD,RS
```

Description

This instruction implements a rotate left through carry of a GPR. The low-order portion of GPR (as defined by the operand size in with the carry flag concatenated on the left end is rotated left

Usage Note: This instruction directly performs an X86 RCL instruction of . The multi-bit X86 instructions are trapped to the microcode and performed one bit at a time.

Architecture Note: This instruction only rotates through carry one bit because (1) it is much harder to do multi-bit rotates through carry, and (2) the multi-bit forms are rarely used in X86.

Operation

```
Temp_bit = EFLAGS.CF
EFLAGS.CF = MSB(GPR[RS])   // MSB = most significant bit
GPR[RD] = (GPR[RS] << 1) | Temp_bit
```

Flags Setting

CCrl

Exceptions

None.

3.1.2.13XRCR - X86 Rotate Right Thru Carry

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:06
XRCRI: **XRCRIR:**	100001_2 100011_2	RS	00001_2	RD	000_2	DPcntl	RCR 00111_2
	6	5	5	5	3	3	5

Format

```
XRCRI   RD,RS
```

Description

This instruction implements a rotate right through carry of a GPR.. The low-order portion of GPR (as defined by the operand size in with the carry flag concatenated on the right end is rotated right

Usage Note: This instruction directly performs an X86 RCR instruction of . The multi-bit X86 instructions are trapped to the emulator and performed one bit at a time.

Architecture Note: This instruction only rotates through carry one bit because (1) it is much harder to do multi-bit rotates through carry, and (2) the mult-bit forms are rarely used in X86.

Performance

If there is no data dependency stall, this instruction executes in one clock.

Operation

```
Temp = EFLAGS.CF << N    // N = 32, 16, or 8 depending on
DPcntl
EFLAGS.CF = GPR[RS] & 1
GPR[RD] = Temp | (GPR[RS] >> 1)
```

Flags Setting

CCrr

Exceptions

None.

3.1.2.14XROL[I][R] - X86 Rotate Left

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XROL: **XROLR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	ROL 00100_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:06
XROLI: **XROLIR::**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	ROL 00100_2
	6	5	5	5	3	3	5

Format

```
XROL    RD,RS,RT
XROLI   RD,RS,2
```

Description

This instruction implements a rotate left of a GPR. The low-order portion of GPR as defined by the operand size in is rotated left the value in GPR or modulo 32.

Operation

```
Temp = xxxxxx
```

Flags Setting

CCrl

Exceptions

None.

3.1.2.15XROR[I][R] - X86 Rotate Right

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XROR: **XRORR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	ROR 00101_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:06
XRORI: **XRORIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	ROR 00101_2
	6	5	5	5	3	3	5

Format

```
XROR    RD,RS,RT
XRORI   RD,RS,2
```

Description

This instruction implements a rotate right of a GPR. The low-order portion of GPR as defined by the operand size in is rotated right the value in GPR or modulo 32.

Operation

```
Temp = xxxxxx
```

Flags Setting

CCrr

Exceptions

None.

3.1.2.16XSBB[I][R] - X86 Subtract with Borrow

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSBB: **XSBBR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	SBB 10011_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSBBI: **XSBBIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	SBB 10011_2
	6	5	5	5	3	3	5

Format

```
XSBB    RD,RS,RT
XSBBI   RD,RS,2
```

Description

The contents of GPR or the immediate specified in , depending on the primary opcode, are subtracted from the contents of GPR .

Operation

```
GPR[RD] = GPR[RS] - GPR[RT] - EFLAGS.CF
GPR[RD] = GPR[RS] - Const   - EFLAGS.CF
```

Flags Setting

CCarith

Exceptions

None.

3.1.2.17XSETCC - X86 SETcc

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSETCC:	100000_2	00000_2	00000_2	RD	000_2	DPcntl	XSETCC 11101_2
	6	5	5	5	3	3	5

Format

```
XSETCC    RD
```

Description

The contents of GPR (expected to be coded as 0) are logically ORed to the contents of GPR (also expected to be coded as 0), the low order bit of the result is then logically ORed with the condition specified by the tttn field of the IIR.

Architecture Note: This instruction is intended for the translator to implement the x86 SETcc instruction, its use in microcode is limited by the fact that the condition to test is specified in the tttn field of IIR.

Architecture Differences: this instruction is new to C2.

Operation

Flags Setting

None.

Exceptions

None.

3.1.2.18XSHL[I][R] - X86 Shift Left Logical

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSHL: **XSHLR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	SLL 00000_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSHLI: **XSHLIR::**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	SLL 00000_2
	6	5	5	5	3	3	5

Format

```
XSHL    RD,RS,RT
XSHLI   RD,RS,2
```

Description

This instruction implement a shift left of a GPR. The low-order portion of GPR as defined by the operand size in is shifted left the value in GPR or modulo 32.

Operation

```
GPR[RD] = xxxxxx
```

Flags Setting

CCshl

Exceptions

None.

3.1.2.19XSAR[I][R] - X86 Shift Right Arithmetic

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSAR: **XSARR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	SRA 00011_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSARI: **XSARIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	SRA 00011_2
	6	5	5	5	3	3	5

Format

```
XSAR    RD,RS,RT
XSARI   RD,RS,2
```

Description

This instruction implement a shift right arithmetic of a GPR. The low-order portion of GPR as defined by the operand size in is shifted right arithmetically the value in GPR or modulo32.

Operation

```
GPR[RD] = xxxxxx
```

Flags Setting

CCsar

Exceptions

None.

3.1.2.20XSHR[I][R] - X86 Shift Right Logical

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSHR: **XSHRR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	SHR 00010_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSHRI: **XSHRIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	SHR 00010_2
	6	5	5	5	3	3	5

Format

```
XSHR    RD,RS,RT
XSHRI   RD,RS,2
```

Description

This instruction implement a shift right logical of a GPR. The low-order portion of GPR as defined by the operand size in is shifted left logically the value in GPR or modulo 32.

Operation

```
GPR[RD] = xxxxxx
```

Flags Setting

CCshr

Exceptions

None.

3.1.2.21XSUB[I][R] - X86 Subtract

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSUB: **XSUBR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	SUB 10010_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
XSUBI: **XSUBIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	SUB 10010_2
	6	5	5	5	3	3	5

Format

```
XSUB    RD,RS,RT
XSUBI   RD,RS,2
```

Description

The contents of GPR or the immediate specified in , depending on the primary opcode, are subtracted from the contents of GPR .

Operation

```
GPR[RD] = GPR[RS] - GPR[RT]
GPR[RD] = GPR[RS] - Const
```

Flags Setting

CCarith

Exceptions

None.

3.1.2.22X8XOR[I][R] - X86 XOR

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
X8XOR: **X8XORR:**	100000_2 100010_2	RS	RT	RD	000_2	DPcntl	XOR 10110_2
	6	5	5	5	3	3	5

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
X8XORI: **X8XORIR:**	100001_2 100011_2	RS	Const	RD	000_2	DPcntl	XOR 10110_2
	6	5	5	5	3	3	5

Format

```
X8XOR     RD,RS,RT
X8XORI    RD,RS,2
```

Description

The contents of GPR RS are logically XORed to either the contents of GPR RT or the value immediate specified in depending on the primary opcode.

Operation

```
GPR[RD] = GPR[RS] ^ GPR[RT]
GPR[RD] = GPR[RS] ^ Const
```

Flags Setting

CClog

Exceptions

None.

3.2 EFLAGS UPDATE FORMS

The EFLAGS update forms specify how the EFLAGS register is modified by certain XALU instructions. These update forms specify how the CF, PF, AF, ZF, SF, and OF are modified; there is no hardware support for modifying the other bits in EFLAGS. In the following sections the exact semantics of each EFLAG update form will be specified.

In the following table the EFLAG update forms are shown along with the corresponding operations performed on the flag bits. An indicates the flag is modified, a – indicates the flag is specified in an X86 as undefined (the actual value used on a C1 is the same as on a Pentium and is documented in the detail descriptions), and a blank indicates that the flag is unmodified.

Label	OF	SF	ZF	AF	PF	CF	X86 Instruction	C1 Instructions
CCnop							NOT MENTIONED BELOW	NOT MENTIONED BELOW
CCarith	M	M	M	M	M	M	ADC, ADD, CMP, CMPS, CMPXCHG, NEG, RSM, SBB, SCAS, SUB, XADD	XADC, XADD, XSBB, XSUB
CClog	0	M	M	–	M	0	AND, OR, TEST, XOR	XAND, X8OR, X8XOR
CCinc	M	M	M	M	M		DEC, INC	XDEC, XINC
CCshl	M	M	M	–	M	M	SAL1, SAL N, SHL 1, SHL N	XSHL
CCshr	M	M	M	–	M	M	SAR1, SAR N, SHR 1, SHR N	XSHR, XSAR
CCrl	M					M	RCL1, RCL N, ROL 1, ROL N	XRCL, XROL
CCrr	M					M	RCR1, RCR N, ROR 1, ROR N	XRCR, XROR
CCdiv	–	–	–	–	–	–	DIV	XDIV
CCidiv	–	–	–	–	–	–	IDIV	XIDIV
CCimul	M	–	–	–	–	M	MUL	XMUL
CCmul	M	–	–	–	–	M	IMUL	XIMUL

Default Flag Setting

In most of the following descriptions is the 32 bit result of an operation, and are the carry outs from bits 3, 6, 7, 14, 15, 30, and 31 of the adder respectively. and are encodings of the DPcntl field specifying that the operation is 8, 16, or 32 bits wide. The variable SubOp indicates that the instruction is a subtract.

For the shift and rotate flag setting operations, is the 34 bit output of the shifter. This allows the code to specify [-1] and [33] which are required to generate the carry out for the shift and rotate instructions. The high and low bits (33 and -1) only go to the condition code

unit, they never appear in registers. The semantics of the flag operations are specified in a pseudo-C language.

CF - Carry Flag

Set on a carry out or not a borrow from the high-order bit of an add or subtract calculation respectively; cleared otherwise. The high-order bit is determined by the DPcntl field in the extended opcode which specifies bit 7 for eight bit operations, bit 15 for sixteen bit operations, and bit 31 for thirty-two bit operations.

```
if DP8
  CF = SubOp ^ CF7;
if DP16
  CF = SubOp ^ CF15;
if DP32
  CF = SubOp ^ CF31;
```

PF - Parity Flag

Set if the low order eight bits of the result have an even number of ones; cleared otherwise.

```
PF =!(Result[7] ^ Result[6]^ Result[5]^ Result[4]^
   Result[3] ^ Result[2]^ Result[1]^ Result[0]);
```

AF - Auxiliary Carry Flag

Set on a carry out of bit 3 or not a borrow from bit 3 of an add or subtract calculation respectively; cleared otherwise

```
AF = SubOp ^ CF3;
```

ZF - Zero Flag

Set if the result of an operation modulo the operation size, specified in DPcntl, is all zeros; cleared otherwise.

```
if DP8
  ZF = Result[7:0] == 0;
if DP16
  ZF = Result[15:0] == 0;
if DP32
  ZF = Result[31:0] == 0;
```

SF - Sign Flag

Set to equal the high-order bit of the result. The high order bit of the result is specified by the size in DPcntl.

```
if DP8
  SF = Result[7];
if DP16
  SF = Result[15];
if DP32
  SF = Result[31];
```

OF - Overflow Flag

Set if the carry out of the high-order two bits differ. The high-order bit of the result is specified by the size in DPcntl. Note that for adds and subtracts, OF may also be determined from the signs of the result versus the sign of the sources.

```
if DP8
  OF = CF7 ^ CF6;
if DP16
  OF = CF15 ^ CF14;
if DP32
  OF = CF31 ^ CF30;
```

CCarith

The CCarith flags setting causes all the flags to be set as described in the Default Flag Setting section above

```
SF = defaultSF;
ZF = defaultZF;
PF = defaultPF;
OF = defaultOF;
CF = defaultCF;
AF = defaultAF;
```

CCinc

The CCinc flags setting causes all the flags except the CF to be set as described in the Default Flag Setting section above. The CF is unmodified.

```
SF = defaultSF;
ZF = defaultZF;
PF = defaultPF;
OF = defaultOF;
AF = defaultAF;
```

CClog

The CClogic flags setting causes the SF, ZF, and PF to be set as described in the Default Flag Setting section above.. The CF and OF are cleared. AF is architected as undefined; its real behavior is: that it is cleared.

```
SF = defaultSF;
ZF = defaultZF;
PF = defaultPF;
OF = 0;
CF = 0;
AF = 0; // Verified on Pentium
```

CCnop

The CCnop flags setting leaves all flags unchanged.

CCshl

Note the special setting of CF and OF for byte shifts with shift counts greater than operand size.

```
if (shiftCnt != 0) {
  ZF = defaultZF;
  PF = defaultPF;
  SF = defaultSF;
  AF = 1;
  if (shiftCnt >= operand_size)
      CF = ( (shiftCnt % operand_size) == 0)
              & bit 0 of original data;
  else
    CF = result[operand_size];
  OF = SF ^ CF;
}
```

CCsar

```
if (shiftCnt != 0) {
  ZF = defaultZF;
  PF = defaultPF;
  SF = defaultSF;
  AF = 1;
  OF = 0;
  CF = result[-1];
}
```

CCshr

Note the special setting of CF and OF for byte shifts with shift counts greater than operand size.

```
if (shiftCnt != 0) {
  SF = defaultSF;
  ZF = defaultZF;
  PF = defaultPF;
  AF = 1;
  if (shiftCnt >= operand_size)
      CF = ( (shiftCnt % operand_size) == 0)
               & MSB of original data;
  else
    CF = result[-1];
  if (shiftCnt == 1)
    OF = high bit of result;
  else
    OF = 0;
}
```

CCrl

The SF, ZF, AF, and PF flags are not affected.

```
if (shiftCnt != 0) {
  OF = result[len] ^ result[len+1];
  CF = result[len+1]
}
```

CCrr

The SF, ZF, AF, and PF flags are not affected.

```
if (shiftCnt != 0) {
  OF = result[len] ^ result[len-1];
  CF = result[-1]
}
```

3.3 LOAD/STORE INSTRUCTIONS

There are no MIPS format load/store instructions and no way to perform load/stores to native-mode address space. All load/store operations use X86 address and load/store semantics.

3.3.1.1 XIOR - X86 I/O Read

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XIOR:	110101_2	Offset	Base	RS	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	2	1	2	4	1	1

Format

Description

Performs an I/O read operation into GPR

The effective address is calculated by adding the value specified in to the contents of GPR modulo the address size specified in . The linear address is calculated with respect to the segment descriptor specified in , The field specifies the number of bytes to load and the location within the target register. The field specifies further special I/O read semantic information.

No virtual-to-physical address translation is performed on the linear address. The operation bypasses the cache and performs an I/O read bus cycle.

Usage Note: To perform an X86 IN operation, the Seg should be a flat 32-bit linear address segment, the Base register contains the I/O address (from DX or the immediate field), the Offset is 0. The 16-bit base address value should be zero extended to 32-bits since the address size from the translator could be 32 bits.

Exceptions

None.

3.3.1.2 XIOW - X86 I/O Write

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XIOR:	111101_2	Offset	Base	RS	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	2	1	2	4	1	1

Format

Description

Performs an I/O write operation of the data in GPR

The effective address is calculated by adding the value specified in to the contents of GPR modulo the address size specified in . The linear address is calculated with respect to the segment descriptor specified in , The field specifies the number of bytes to load and the location within the target register. The field specifies further special I/O read semantic information.

No virtual-to-physical address translation is performed on the linear address. The operation bypasses the cache and performs an I/O read write cycle.

Usage Note: To perform an X86 OUT operation, the Seg should be a flat 32-bit linear address segment, the Base register contains the I/O address (from DX or the immediate field), the Offset is 0. The 16-bit base address value should be zero extended to 32-bits since the address size from the translator could be 32 bits.

Exceptions

None.

3.3.1.3 XL - X86 Load

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XL:	110000_2	Offset	Base	RS	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	2	1	2	4	1	1

Description

Performs a load with X86 addressing semantics into GPR Data Register (or RS | 1 for load of 64 bit size)

The effective address is calculated by adding the value specified in to the contents of GPR modulo the address size specified in . The linear address is calculated with respect to the segment descriptor specified in , The field specifies the number of bytes to load and the location within the target register. The field specifies further special load semantic information.

Performance

If there is no data dependency stall, and the requested data is in the cache and contained within an eight-byte aligned data unit, the load executes in one clock. Note that the loaded data is available to the next ALU or store instruction without a pipeline stall. However, if the load is used as a source to an addressing calculation in the next instruction, there is a one-cycle data dependency stall (AGI) on the subsequent instruction.

If the requested data is not in the cache, the instruction (and all subsequent instruction) execution stalls until the data is found and returned.

If the data spans an eight-byte aligned unit, this instruction is automatically decomposed into two sequential loads to get the two data portions. The timing of each follows the rules above except that the second load can't cause an data dependency stall.

If LOCK is specified, timing gets more complicated depending on the compatibility mode in effect.

Operation

Exceptions

SEGERR, DPAGE, ALIGN, DBRKPT

3.3.1.4 XLDESC - X86 Load Descriptor

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XLDESC:	110100_2	Offset	Base	DRS	SubOp	Addr Size1	10_2	Seg	1_2	Addr Size0
	6	5	5	5	2	1	2	4	1	1

Description

Even though the load descriptor instruction is a load instruction, its documentation has moved to the Segment Register function section to be with its tightly-bound companion the XTI instruction.

3.3.1.5 XLEAD - X86 Load Effective Address - Displacement

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:3	2	1	0
XLEAD:	101100_2	Offset	Base	RS	xx_2	Addr Size1	Size-2:1	xxx_2	Size3	Size-0	Addr Size0
	6	5	5	5	2	1	2	3	1	1	1

Description

Performs an X86 (EA) calculation by adding GPR to the value specified by the field (typically the DISP register) modulo the size defined by the field and storing the result back into GPR with size defined by the field. On C2 an of operand size (TSR.OS) is encoded as 01_2 which C1-A interprets as stack address size (TSR.SAS).

Architecture Note: The XLEAx instructions are not really load and store instructions and don't need all of the load/store semantics (such as a segment register and the load/store SubOp field), but they are included in this section since they have many features in common with load/store instructions and are used with them.

Usage Note: This instruction is needed for three reasons: (1) to (partially) perform an X86 LEA function, (2) to partially perform address calculations using a base register and an index register and a displacement, and (3) to perform address calculations in trapped microcode since the XL and XS instructions can't use register indirection.

This XLEAD instruction performs an X86 EA calculation using a base register and a displacement. When combined with the XLEAI instruction, a full three component X86 LEA can be performed.

Architecture Differences: The extension of Size for XLEAD to 4 bits is new to C2 as is allowing operand size encoding in .

Performance

Same rules as for the XLEAI instruction.

Exceptions

none

3.3.1.6 XLEAI - X86 Load Effective Address - Indexed

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:3	2	1	0
XLEAI:	101011_2	Index	Base	RD	shift count	Addr Size1	Size-2:1	xxx_2	Size3	Size-0	Addr Size0
	6	5	5	5	2	1	2	3	1	1	1

Description

Performs an X86 (EA) calculation by adding GPR to GPR storing the result modulo the size defined by the field back into with size defined by the field. . On C2 an of operand size (TSR.OS) is encoded as 01_2 which C1-A interprets as stack address size (TSR.SAS).

Usage Note: This instruction is needed for three reasons: (1) to (partially) perform an X86 address calculation involving both a base and an index register, and (2) to perform address calculations in trapped microcode since the XL and XS instructions can't use register indirection.

This XLEAI instruction performs an X86 EA calculation using a non-shifted index register and a base register. X86 LEA calculations using a scaled index value require that a separate shift instruction be performed. X86 LEA calculations involving a register and a displacement can use the XLEAD instruction.

Architecture Differences: The extension of Size for XLEAI to 4 bits is new to C2 as is allowing operand size encoding in .

Performance

If there is no data dependency stall, this instruction executes in one clock.

If either RS or RT are the result of a load or an ALU operation on the immediately preceding instruction, there is an additional one-cycle data dependency stall.

Architecture Note: This is effectively a register–register add operation that executes in the A-stage versus a normal ALU instruction that executes in the D-stage. Thus, address calculations instructions that follow this one don't have a data-dependency stall on the XLEA instruction results.

Exceptions

none

3.3.1.7 XPOP[BR] - X86 POP (Load with Post Update)

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XPOP:	110111_2	Offset	Base	RS	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	2	1	2	4	1	1

	31:26	25:21	20:16	15:11	10:98	8	7:6	5:2	1	0
XPOPBR:	110110_2	Offset	Base	RS	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	23	1	2	4	1	1

Description

Performs a load with X86 addressing semantics into GPR and updates the base register with a new address.

The effective address is the value in GPR modulo the address size specified in . The linear address is calculated with respect to the segment descriptor specified in , The field specifies the number of bytes to load. The field specifies further special load semantic information.

, the address size portion of GPR is replaced with the result of adding the to the contents of . If and are the same register, then the behavior of the instruction depends on the implementation:

- On C1-A the load data is loaded into and the address offset is not used.
- On C2 the update of with number of bytes from the address offset can be thought of as occuring first, then the load of with S number of bytes from memory. This is relevant when the number of bytes specified in is larger than S .

Usage Note: This instruction is intended to directly perform an X86 POP function in one clock.. In order to perform the POP SP function correctly, the load data must have priority over the updated Base value, if both registers are the same.

To perform the X86 POP function, set:

= SS, = OS, = indOS, = LSnop.

The XPOPBR version of the instruction will read the linear address contained in the top of the call/return stack (LINEAR_RET). If LINEAR_RET is in the instruction cache then a branch is initiated to it and a hidden latch (RETSTK_HIT) is set. If LINEAR_RET is not in the instruction cache then RETSTK_HIT is cleared. See the description of the POP version of XJ for how RETSTK_HIT is used.

Architecture Differences: XPOPBR instruction is new to C2.

Usage Note: The XPOPBR instruction is intended to speed up subroutine return. It is used to initiate a return to the address pushed on the call/return stack by a previous XBcc.PUSH. Whether this branch was in fact correct is verified by a subsequent XJ.POP.

Performance

Same rules as for the XL instruction.

Operation

```
if (XPOPBR) {                                   // Subroutine return
   LINEAR_RET = RETSTK[TOS];                    // Linear instruction pointer of expected return
   if (in_I_Cache(LINEAR_RET)) {                // Expected return is in instruction cache
         start_branch(LINEAR_RET);              // Get fetcher and translator going
      RETSTK_HIT = 1;                           // Latch to remember branch started
                              }                      else                                    {
      RETSTK_HIT = 0;                           // Indicate branch was not started
                                                                                            }
}
```

Exceptions

SEGERR, DPAGE, ALIGN, DBRKPT

3.3.1.8 XPUSH - X86 PUSH (Store with Pre-Update)

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XPUSH:	111111_2	Offset	Base	RS	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	2	1	2	4	1	1

Description

Performs a store with X86 addressing semantics of the data in GPR and updates the base register with a new address.

The effective address is calculated by adding the displacement specified in to the contents of GPR modulo the address size specified in . The linear address is calculated with respect to the segment descriptor specified in , The field specifies the number of bytes to store. The field specifies further special store semantic information.

If the field indicates that four bytes are to be pushed and is in the range 8 to 15 (a segment register) then the upper two bytes of the data pushed are zero.

If the field is 16H (encoded as 8L) then it is the upper 2 bytes of GPR that are stored. In this case the store size is controlled by TSR.OS. If TSR.OS is 32 then the upper two bytes of the data pushed are zero.

, the address size portion of GPR Base is replaced by the effective address of the store.

Usage Note: This instruction is intended to perform an X86 PUSH function in one clock. Note that the pre-update function correctly handles the PUSH SP function: the old SP value is stored before SP is updated.

To perform the PUSH function, set:

= SS, = OS, = indMOS, = LSnop.

Performance

Follows the same rules as for the XS instruction.

Exceptions

SEGERR, DPAGE, ALIGN, DBRKPT

3.3.1.9 XPUSHIP - X86 PUSH NSIP (Store with Pre-Update)

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XPUSHIP:	111100_2	Offset	Base	00000_2	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	2	1	2	4	1	1

Description

Performs a store with X86 addressing semantics of the NSIP register

The effective address is calculated by adding the displacement specified in to the contents of GPR modulo the address size specified in . The linear address is calculated with respect to the segment descriptor specified in , The field specifies the number of bytes to store. The field specifies further special store semantic information.

More information on the semantics of the function is found on page 2-7.

The address size portion of GPR is updated to contain the effective address of the store; thus, this is a store with pre-update type instruction.

Architecture Note: There is no general store of a CP2 data register instruction (which is where NSIP lives). Thus, a special form is provided for NSIP.

Usage Note: This instruction is intended to perform a "push IP" function needed for fast CALL execution. To perform this operation set

= SS, = OS, = indMOS, = LSnop.

Performance

Follows the same rules as for the XS instruction.

Exceptions

SEGERR, DPAGE, ALIGN, DBRKPT

3.3.1.10XS - X86 Store

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XS:	111000_2	Offset	Base	RS	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	2	1	2	4	1	1

	31:26	25:21	20:16	15:11	10:98	8	7:6	5:2	1	0
XS2:	111001_2	Offset	Base	RS	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	23	1	2	4	1	1

Description

Performs a store with X86 addressing semantics of the data in GPR .

The effective address is calculated by adding the displacement specified in to the contents of GPR modulo the address size specified in . The linear address is calculated with respect to the segment descriptor specified in , The field specifies the number of bytes to load and the location within the target register. The field specifies further special store semantic information. The difference between XS and XS2 is merely the choice of data sizes and locations as defined in section 5.1.5.

Performance

If there is no data dependency stall, and the requested data is in the cache and contained within an eight-byte aligned unit, and there is no pending store operation. the store executes in one clock.

If either base or RS are the result of a load or an ALU operation on the immediately preceding instruction, there is an additional one-cycle data dependency stall.

There is a store buffer between the D-stage and the cache/bus unit. Thus, stores that miss in the cache or that are blocked by previous loads or store operations take an indeterminate time. The rules for store queuing are complex and are discussed elsewhere.

If the data spans an eight-byte aligned unit, this instruction is automatically decomposed into two sequential stores to handle the two data portions. The timing of each follows the rules above except that the second load can't cause an data dependency stall.

If LOCK is in effect (specified on a previous load operation), timing gets more complicated depending on the compatibility mode in effect.

Operation

Exceptions

SEGERR, DPAGE, ALIGN, DBRKPT

3.3.1.11XSI, XPUSHI - X86 Store/PUSH Immediate

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XSI:	111011_2	Offset	Base	00000_2	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	2	1	2	4	1	1

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XPUSHI:	111010_2	Offset	Base	00000_2	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	2	1	2	4	1	1

Description

Performs a store or a PUSH with X86 addressing semantics of the value specified in the IMMED register. The semantics are the same as for the XS and XPUSH instructions except that the IMMED register is stored instead of the RS register.

Architecture Note: There is no way to address the IMMED register as a source in a normal store instruction. Thus, a special form is provided for store IMMED.

Usage Note: This instruction is intended to directly perform the corresponding X86

MOV [ea],immed or PUSH 10H

instructions in one clock:

To perform this MOV operation set

= AS, = OS, = LSnop.

Operation

Follows the same rules as for the XS and XPUSH instruction.

Exceptions

SEGERR, DPAGE, ALIGN, DBRKPT

3.3.1.12XSU - X86 Store with Post-Update

Encoding

	31:26	25:21	20:16	15:11	10:9	8	7:6	5:2	1	0
XSU:	111110_2	Offset	Base	RS	SubOp	Addr Size1	Size-2:1	Seg	Size-0	Addr Size0
	6	5	5	5	2	1	2	4	1	1

Description

Performs a store with X86 addressing semantics of the data in GPR and updates the base register with a new address.

The effective address is the value in GPR modulo the address size specified in . The linear address is calculated with respect to the segment descriptor specified in , The field specifies the number of bytes to load. The field specifies further special load semantic information.

, the address size portion of GPR is replaced with the result of adding the to the contents of .

Performance

Follows the same rules as for the XS instruction.

Exceptions

SEGERR, DPAGE, ALIGN, DBRKPT

3.4 CONTROL REGISTERS AND MICRO-OPERATIONS

This section includes micro-operations that move to/from control registers that may be useful as Alternate Instructions.

CP2 Control Registers

Reg	Asm Lbl	Description
XC5	TSC_U	Time Stamp Counter (Upper 32 bits)
XC19	TSC_L	Time Stamp Counter (Lower 32 bits)
XC30	CR0	X86 CR0
XC31	EFLAGS	X86 EFLAGS

3.4.1.1 CTC2 - Store To CP2

Encoding

	31:26	25:21	20:16	15:11	10:8	7:5	4:0
CTC2	100000_2	RS	00000_2	C2RD	000_2	DPcntl	11001_2
	6	5	5	5	3	3	5

Description

Moves the contents of general purpose register RS to the CP2 control register C2RD.

DPcntl does not control the size of the result stored to the CP2 control register except for EFLAGS: in this case, a size of 16 or 32 bits can be specified directly or indirectly through OS.

3.4.1.2 CFC2 - Move Control From CP2

Encoding

	31:26	25:21	20:16	15:11	10:6	5:3	2:0
CFC2	XMISC 101000_2	00000_2	RT	C2RD	CFC2 11111_2	xxx_2	xxx_2
	6	5	5	5	5	3	3

Description

The contents of CP2 control register are loaded into GPR .

3.4.1.3 XJ[XAI] - X86 Jump and optionally exit Alternate Instruction Execution Mode

Encoding

	31:26	25:21	20:16	15:11	10	9	8	7:6	5:2	1:0
XJ:	000110_2	00000_2	Base	00000_2	0_2	0_2	0_2	XJSize	0001_2	00_2
XJXAI:										11_2
	6	5	5	5	1	1	1	2	4	2

XJSize	Mnemonic	Data Size
00_2	16	16 bits-low
01_2	32	32 bits
10_2	AS	address size-16 or 32
11_2	OS	operand size - 16 or 32

Description

XJ Performs an absolute branch with X86 addressing semantics. XJXAI will branch like XJ and also exit Alternate Instruction Execution Mode (encoded with bits 1:0 = '11'). The target address is the contents of the GPR modulo the current operand size (as opposed to address size as for load/store instructions). The linear address is calculated with respect to the CS segment descriptor. XJSize controls the size of the EA calculation.

The NSIP is updated with the calculated target address. This has the effect of clearing the upper 16 bits of NSIP when an XJ instruction is executed with a 16 bit operand size.

Exceptions

SEGERR

CHAPTER

4

X87 FLOATING-POINT MICRO-OPERATIONS (C5XL)

This chapter describes the x87 floating point registers and micro-operations available for use in Alternate Instruction enabled mode in the VIA C5XL (Nehemiah) processor. The encoding of x87 floating point micro-operations is different in earlier versions of the VIA C3 Processor family (C5A, C5B, C5C) and are not included in this document.

4.1 X87 FLOATING POINT REGISTERS (C5XL)

All VIA C3 processors implement eight 80-bit registers corresponding to the standard x87 floating point registers. In addition to these eight registers the processor implements additional extended x87 floating-point registers, as described in the following table:

Processor	C5XL (Nehemiah)
Standard x87 floating-point registers	Eight standard x87 floating-point registers FP0-FP7
Extended x87 floating-point registers	Ten extended x87 floating-point registers FP8-FP17

4.2 X87 FLOATING-POINT MICRO-OPERATIONS (C5XL)

X87 Floating-point micro-operations have certain fields in common to control the effects on the top-of-stack (TOS) field in FPSW and to control the precision, rounding, and response to exception cases.

Fmt2 Precision Controls

Fmt2[15:13]	Round	Mask	PC	C1	Description
000_2	FPCW	FPCW	FPCW	Per result	Normal
001_2	FPCW	All	64	A	Set FPLE.PE, Clear DE
010_2	FPCW	FPCW	FPCW	StkFlt	Update C1 on stack fault only
011_2	FPCW	PE	64	Don't clear	?
100_2	FPCW	FPCW	53	?	?
101_2	Nearest	All	64	A	Set FPLE.PE, Clear DE
110_2	-	-	-	-	Undefined
111_2	Nearest	All	64	-	Mask Overflow/Underflow

TOSCtrl – Top Of Stack Control

TOSCtrl[7:6]	Description
00_2	Hold
01_2	FPSW.TOS += 1
10_2	FPSW.TOS -= 1
11_2	FPSW.TOS += 2

4.2.1 FADD, FSUB, FSUBR, FMUL, FDIV, FDIVR

FADD, FSUB, FSUBR, FMUL, FDIV, FDIVR – x87 Floating-point Add, Subtract, Subtract Reverse, Multiply, Divide, Divide Reverse

Encoding

	31:26	25:21	20:16	15:13	12:9	8	7:6	5:0
FADD FSUB FSUBR FMUL FDIV FDIVR	010001_2	FRD	FRS	Fmt2	0000_2	R	TOSCtrl	SubOp
	6	5	5	3	4	1	2	6

Description

The FADD, FSUB, SUBR, FMUL, FDIV and FDIVR instructions operate on the floating-point values in x87 floating-point register FRS and FRD and store the result in FRD. Note that FSUBR and FDIVR are like FSUB and FDIV except the input operands (FRS and FRD) are reversed (FRD is always the destination).

Fields

- FRS – Source x87 FP register
- FRD – Source and Destination x87 FP register
- Fmt2 - Controls rounding, exception masking, precision, and flags
- R

  0 – No record

  1 – Record FP environment
- TOSCtrl - encode the function to be performed on the FPSW.TOP field

Instruction encoding and operation

Instruction	SubOp	Operation
FADD	000000_2	FRD <- FRD + FRS
FSUB	000001_2	FRD <- FRD - FRS
FSUBR	001001_2	FRD <- FRS – FRD
FMUL	000010_2	FRD <- FRD * FRS
FDIV	000011_2	FRD <- FRD ÷ FRS
FDIVR	001011_2	FRD <- FRS ÷ FRD

4.2.2 FSQRT, FABS, FCHS

FSQRT, FABS, FCHS – x87 Floating-point Square Root, Absolute Value, Change Sign

Encoding

	31:26	25:21	20:16	15:13	12:9	8	7:6	5:0
FSQRT FABS FCHS	010001_2	FRD	FRS	Fmt2	0000_2	R	TOSCtrl	SubOp
	6	5	5	3	4	1	2	6

Description

The unary operations FSQRT, FABS and FCHS operate on the value in x87 floating-point register FRS and store the result in FRD. FABS and FCHS will copy FRS to FRD and set the sign bit of FRD to '0' (FABS) or invert it (FCHS).

Fields

- FRS – Source x87 FP register
- FRD – Destination x87 FP register
- Fmt2 - Controls rounding, exception masking, precision, and flags
- R

  0 – No record

  1 – Record FP environment
- TOSCtrl - encode the function to be performed on the FPSW.TOP field

Instruction encoding and operation

Instruction	SubOp	Operation
FSQRT	000100_2	FRD <- SQRT(FRS)
FABS	000101_2	FRD <- ABS(FRS)
FCHS	000111_2	FRD <- - (FRS)

CHAPTER

5

MMX MICRO-OPERATIONS

This chapter describes the MMX registers and MMX micro-operations available for use in Alternate Instruction execution mode. There are micro-operations corresponding to all register-to-register x86 MMX instructions. There are no alternate instructions for loading data directly from memory to an MMX register. Also lacking are alternate instructions that combine a memory load with another operation. This is because these micro-operations require more than 32-bits and alternate instructions are restricted to 32-bits. Use x86 MMX instructions to move data to/from memory.

5.1 MMX REGISTERS

All VIA C3 processors implement eight 64-bit registers corresponding to the standard x86 MMX registers. In addition to these eight registers the processor implements additional extended MMX registers, as described in the following table:

Processor	Samuel, Samuel 2, Ezra	C5XL (Nehemiah)
Standard X86 MMX registers	Eight standard x86 MMX registers MM0-MM7	Eight standard x86 MMX registers MM0-MM7
Extended MMX registers	Two extended MMX registers MM8-MM9	Five extended MMX registers: MM8-MM9 MM13-MM15

5.2 MMX MICRO-OPERATIONS

5.2.1 MMXADD / MMXSUB

Encoding

	31:26	25:21	20:16	15:11	10:8	7:6	5:4	3	2	1	0
MMXADD MMXSUB	010100_2	RT	RS	RD	000_2	Sz	01_2	K	S	T	0
	6	5	5	5	3	2	2	1	1	1	1

Description

Implements the x86 instructions with operands in two MMX registers and destination in MMX register. Note that the x86 instruction encoding requires that one of the source registers also be the destination register, this micro-operation allows the destination MMX register (RD) be different from the two source registers (RT,RS).

Fields

- Sz –Source /Dest Size
  - 00 - 8 bit
  - 01 - 16 bit
  - 10 - 32 bit
- RT – Source MMX register
- RS – Source MMX register
- RD – Destination MMX register
- K
  - 0 - Addition
  - 1 - Subtraction
- S – Signed
  - 0 – Unsigned
  - 1 - Signed
- T – Saturation
  - 0 – Wrap
  - 1 - Saturate

Equivalent x86 instruction encoding

x86 instruction	Sz	K	S	T
PADDB	00_2	0	0	0
PADDW	01_2	0	0	0
PADDD	10_2	0	0	0
PADDSB	00_2	0	1	1
PADDSW	01_2	0	1	1
PADDUSB	00_2	0	0	1
PADDUSW	01_2	0	0	1
PSUBB	00_2	1	0	0
PSUBW	01_2	1	0	0
PSUBD	10_2	1	0	0
PSUBSB	00_2	1	1	1
PSUBSW	01_2	1	1	1
PSUBUSB	00_2	1	0	1
PSUBUSW	01_2	1	0	1

5.2.2 MMXPACK

Encoding

	31:26	25:21	20:16	15:11	10:8	7:6	5:3	2	1:0
MMXPACK	010100_2	RT	RS	RD	000_2	Sz	000_2	S	00_2
	6	5	5	5	3	2	3	1	2

Description

Implements the x86 instructions with operands in two MMX registers and destination in MMX register. Note that the x86 instruction encoding requires that one of the source registers also be the destination register, this micro-operation allows the destination MMX register (RD) be different from the two source registers (RT,RS).

Fields

- Sz –Source /Dest Size
  - 00 - 8 bit
  - 01 - 16 bit
- RT – Source MMX register
- RS – Source MMX register
- RD – Destination MMX register
- S – Signed
  - 0 – Unsigned
  - 1 - Signed

Equivalent x86 instruction encoding

x86 instruction	Sz	S
PACKSSWB	00_2	1
PACKSSDW	01_2	1
PACKUSWB	00_2	0

5.2.3 MMXUNPACK

Encoding

	31:26	25:21	20:16	15:11	10:8	7:6	5:3	2:1	0
MMXUNPK	010100_2	RT	RS	RD	000_2	Sz	001_2	00_2	H
	6	5	5	5	3	2	3	2	1

Description

Implements the x86 instructions with operands in two MMX registers and destination in MMX register. Note that the x86 instruction encoding requires that one of the source registers also be the destination register, this micro-operation allows the destination MMX register (RD) be different from the two source registers (RT,RS).

Fields

- Sz –Source /Dest Size
  - 01 - 16 bit
  - 10 - 32 bit
  - 11 - 64 bit
- RT – Source MMX register
- RS – Source MMX register
- RD – Destination MMX register
- H – Source Half
  - 0 – Low
  - 1 - High

Equivalent x86 instruction encoding

x86 instruction	Sz	H
PUNPCKHBW	01_2	1
PUNPCKHWD	10_2	1
PUNPCKHDQ	11_2	1
PUNPCKLBW	01_2	0
PUNPCKLWD	10_2	0
PUNPCKLDQ	11_2	0

5.2.4 LOGICALS

Encoding

	31:26	25:21	20:16	15:11	10:8	7:6	5:3	2:1	0
MMXAND MMXANDN MMXOR MMXXOR	010100_2	RT	RS	RD	000_2	11_2	100_2	L	0
	6	5	5	5	3	2	3	2	1

Description

Implements the x86 instructions with operands in two MMX registers and destination in MMX register. Note that the x86 instruction encoding requires that one of the source registers also be the destination register, this micro-operation allows the destination MMX register (RD) be different from the two source registers (RT,RS).

Fields

- L – Logical Operation
  - 00 - AND
  - 01 – AND NOT
  - 10 - OR
  - 11 - XOR
- RT – Source MMX register
- RS – Source MMX register
- RD – Destination MMX register

Equivalent x86 instruction encoding

x86 instruction	L
PAND	00_2
PANDN	01_2
POR	10_2
PXOR	11_2

5.2.5 MOVES

Encoding

	31:26	25:21	20:16	15:11	10:8	7:6	5:3	2	1	0
MMXMOV	010100_2	00000_2	RS	RD	000_2	11_2	100_2	0	S	1
	6	5	5	5	3	2	3	1	1	1

Description

Implements the x86 instructions with source operand in an MMX register and destination in MMX register.

Fields

- S – Replicate Low 32-bits in High
  - 0 – High 32-bits from high 32-bits
  - 1 – High 32-bits copy of low 32-bits
- RS – Source MMX register
- RD – Destination MMX register

Equivalent x86 instruction encoding

x86 instruction	S
MOVQ	0

5.2.6 COMPARES

Encoding

	31:26	25:21	20:16	15:11	10:8	7:6	5:3	2	1:0
MMXCMP	010100_2	RT	RS	RD	000_2	Sz	101_2	E	00_2
	6	5	5	5	3	2	3	1	2

Description

Implements the x86 instructions with operands in two MMX registers and destination in MMX register. Note that the x86 instruction encoding requires that one of the source registers also be the destination register, this micro-operation allows the destination MMX register (RD) be different from the two source registers (RT,RS).

Fields

- Sz –Source /Dest Size
  - 00 - 8 bit
  - 01 - 16 bit
  - 10 - 32 bit
- RT – Source MMX register
- RS – Source MMX register
- RD – Destination MMX register
- E – Compare Type
  - 0 – Greater Than
  - 1 - Equal

Equivalent x86 instruction encoding

x86 instruction	Sz	E
PCMPEQB	00_2	1
PCMPEQW	01_2	1
PCMPEQD	10_2	1
PCMPGTPB	00_2	0
PCMPGTPW	01_2	0
PCMPGTPD	10_2	0

5.2.7 MULTIPLIES

Encoding

	31:26	25:21	20:16	15:11	10:9	8:6	5:3	2:0
MMXMULL MMXMULH MMXMULADD	010100_2	RT	RS	RD	M	000_2	110_2	000_2
	6	5	5	5	2	3	3	3

Description

Implements the x86 instructions with operands in two MMX registers and destination in MMX register. Note that the x86 instruction encoding requires that one of the source registers also be the destination register, this micro-operation allows the destination MMX register (RD) be different from the two source registers (RT,RS).

Fields

- M – Multiply Type
  - 00_2 - Low
  - 01_2 - High
  - 10_2 – Multiply Add
- RT – Source MMX register
- RS – Source MMX register
- RD – Destination MMX register

Equivalent x86 instruction encoding

x86 instruction	M
PMULLW	00_2
PMULHW	01_2
PMADDWD	10_2

5.2.8 SHIFT

Encoding

	31:26	25:21	20:16	15:11	10:8	7:6	5:3	2:1	0
MMXSHL MMXSHR MMXSAR	010100_2	RT	RS	RD	000_2	Sz	111_2	S	0
	6	5	5	5	3	2	3	2	1

Description

Implements the x86 instructions with operands in two MMX registers and destination in MMX register. Note that the x86 instruction encoding requires that one of the source registers also be the destination register, this micro-operation allows the destination MMX register (RD) be different from the two source registers (RT,RS).

Fields

- Sz –Source /Dest Size
  - 01 - 16 bit
  - 10 - 32 bit
  - 11 - 64 bit
- RT – MMX register with shift count
- RS – Source MMX register
- RD – Destination MMX register
- S – Shift type
  - 00 – Arithmetic Right
  - 01 – Right
  - 10 - Left

Equivalent x86 instruction encoding

x86 instruction	Sz	S
PSLLW	01_2	10_2
PSLLD	10_2	10_2
PSLLQ	11_2	10_2
PSRLW	01_2	01_2
PSRLD	10_2	01_2
PSRLQ	11_2	01_2
PSRAW	01_2	00_2
PSRAD	10_2	00_2