ALLEGREX™ User's Manual

Table of Contents

1. Overview

1.1. ALLEGREX™ RISC Core

The ALLEGREX™ RISC Core is a proprietary 32-bit MIPS core, independently developed by Sony Corporation, which has an architecture based on a combination of the MIPS II ISA instruction set and an R4000-type system control coprocessor (CP0). Instructions and functions were both added and removed to optimize the design for a SoC (System on Chip) implementation.

The PSP™ system chip uses the ALLEGREX™ 3.1 f v version of the processor. This version includes a single-precision floating-point unit (FPU) and a single-precision vector floating-point unit (VFPU) as coprocessors.

Figure 1-1 shows the configuration of the ALLEGREX™ CPU, FPU and VFPU. This manual describes the ALLEGREX™ RISC CPU Core. For information about the FPU and VFPU coprocessors, please see the FPU and VFPU user's manuals.

Figure 1-2 is a functional block diagram of the ALLEGREX™ CPU.

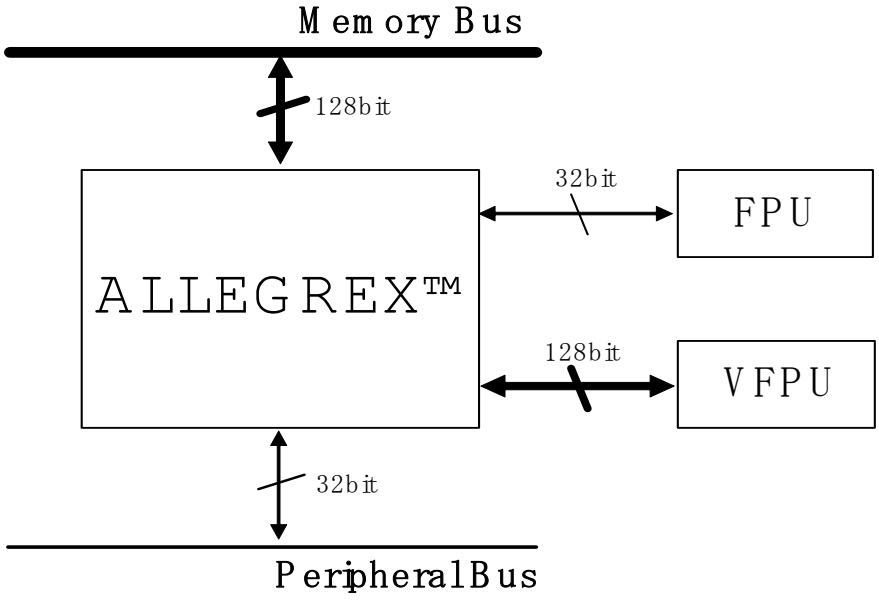


Figure 1-1: ALLEGREX™ Configuration

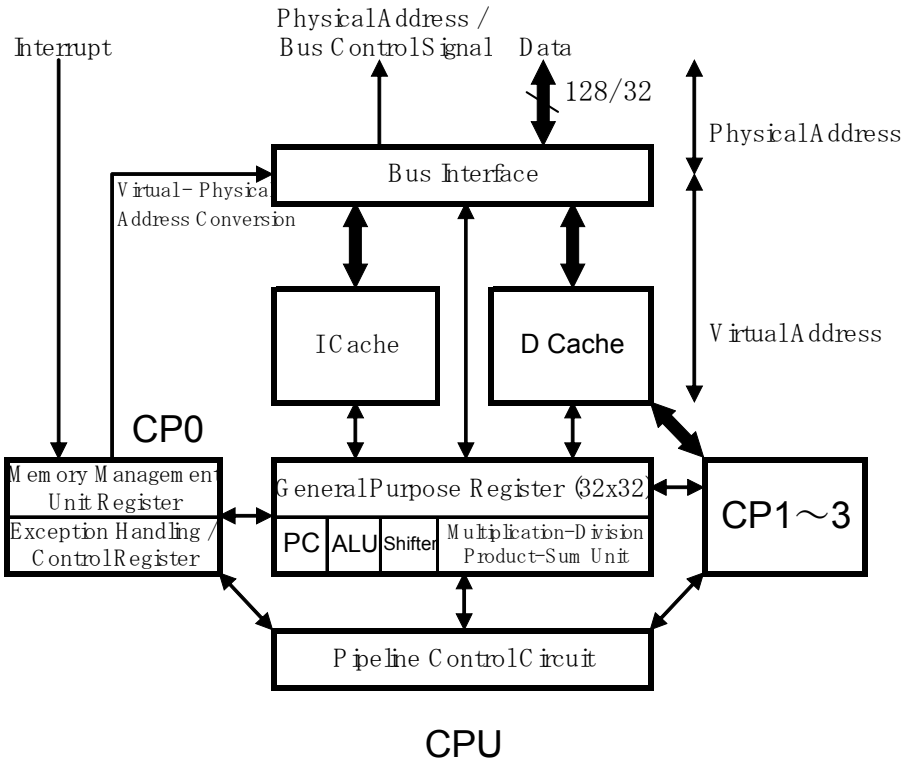


Figure 1-2: ALLEGREX™ CPU Functional Block Diagram

1.1.1. Main Features of the ALLEGREX™ RISC Core

1. MIPS R4000-like 32-bit RISC core

ALLEGREX™ is a 32-bit RISC CPU that has the same instruction set and exception handling mechanism as a MIPS R4000-type processor operating in 32-bit mode. ALLEGREX™ has thirty-two 32-bit general-purpose registers, and all instructions and addresses are 32 bits long.

2. 7-stage pipeline

The ALLEGREX™ CPU uses a 7-stage single pipeline. An independent arithmetic unit is also provided for performing multiplication, division, multiply and add, and multiply and subtract operations. These operations require multiple cycles to execute.
An FPU and VFPU are implemented as independent coprocessors. The FPU has an 8-stage pipeline and the VFPU has a variable-length pipeline whose length depends on the instruction being executed. These pipelines operate independently of the CPU pipeline.

3. MMU

ALLEGREX™ does not use the TLB to map the 4 GB virtual address space to physical addresses, but instead maps segments directly. The MMU supports three operating modes (kernel mode, supervisor mode, and user mode) similar to those of an R4000-type processor. The MMU also provides memory protection functions.

4. Caches

ALLEGREX™ has an independent 16 KB instruction cache and 16 KB data cache built into the chip. Each cache is two-way set associative to improve the cache hit rate. The caches also have a locking mechanism which allows critical instructions and data to always remain in the cache.
Writes to the data cache are done using the writeback method.

5. Cache Writeback Buffer

ALLEGREX™ is equipped with a cache writeback buffer which can hold one cache line. The cache writeback buffer enables new data to be read before old data is written back during cache replacement.

1.1.2. CPU Registers

The ALLEGREX™ RISC Core has thirty-two general-purpose registers, a program counter, and two registers for storing the results of integer multiplication and division.
All of these registers are 32-bit. Two of the general-purpose registers have special meanings as shown below.

- Register r0: Always 0
- Register r31: Link register for the Jump And Link instruction

Three additional registers are also used by certain instructions.

- PC: Program counter
- HI: Multiply/divide register high word
- LO: Multiply/divide register low word

The multiplication-division product-sum registers (HI and LO) store the 64-bit result of an integer multiplication, or the quotient and remainder of an integer division.

In addition to these registers, the single LL bit is used by the LL and SC instructions.

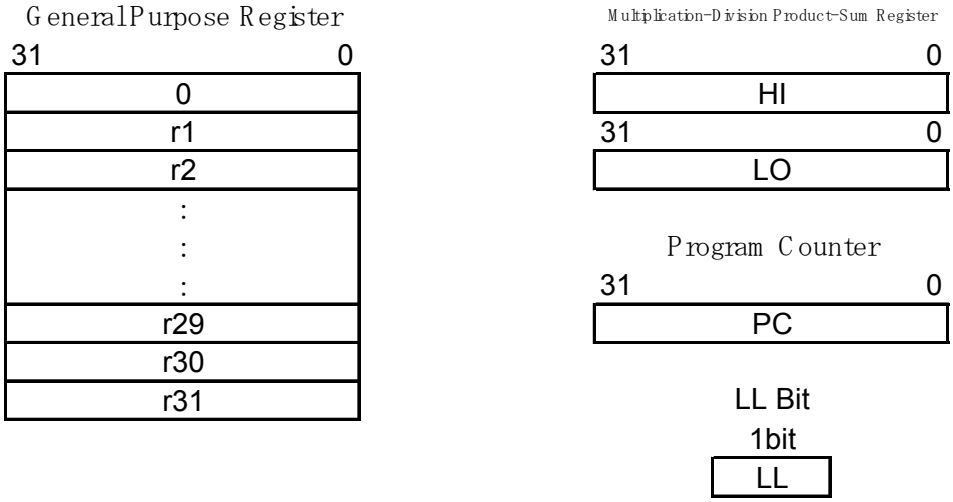


Figure 1-3: CPU Registers

The system control coprocessor (CP0) registers are described later in this section.

1.1.3. CPU Instruction Set Overview

CPU instructions are 32 bits long. The instruction set is divided into the following groups.

1. Load and Store Instructions

These instructions transfer data between memory and general-purpose registers. Only one addressing mode is supported. The effective address is always calculated by adding a 16-bit signed immediate offset to a base register.

2. Arithmetic Instructions

These instructions perform arithmetic, logical, and shift operations on register values. They are divided into two types. In one type, both source operands and the result are in registers. With the other type, one operand is a 16-bit immediate value.

3. Jump and Branch Instructions

These instructions change the control flow of a program. Jump instructions are divided into two types. One type jumps to an absolute address calculated by concatenating a 26-bit target address to 4 bits of the program counter. The other type is the register indirect jump, in which a 32-bit register contains the target address. Branch instructions use a 16-bit offset address relative to the program counter to calculate the branch target address.

4. Coprocessor Instructions

These instructions perform operations on a coprocessor. Coprocessor instructions include load and store instructions, arithmetic instructions, and coprocessor branch instructions.

The format of coprocessor arithmetic instructions is different for each coprocessor.

5. System Control Coprocessor (CP0) Instructions

These instructions perform operations on CP0 registers. They are used for memory management and exception handling.

6. Extended Instructions

These instructions perform operations unique to ALLEGREX™ such as making system calls and setting breakpoints.

Table 1-1 lists the MIPS I ISA instructions common to all MIPS machines.
Table 1-2 lists the MIPS II ISA instructions supported by ALLEGREX™.
Table 1-3 lists instructions unique to ALLEGREX™.
Table 1-4 lists the R4000-type system control coprocessor (CP0) instructions implemented by ALLEGREX™.
Table 1-5 lists the R4000-type FPU instructions that are used for communication between the Core and the FPU.
CPU instructions are described in Chapter 2. For complete descriptions of FPU and VFPU instructions, refer to the FPU and VFPU user's manuals, respectively.

Table 1-1: MIPS I ISA Instructions

Category	Instruction	Description
Load and store instructions	LB	Load Byte
	LBU	Load Byte Unsigned
	LH	Load Halfword
	LHU	Load Halfword Unsigned
	LW	Load Word
	LWL	Load Word Left
	LWR	Load Word Right
	SB	Store Byte
	SH	Store Halfword
	SW	Store Word
	SWL	Store Word Left
	SWR	Store Word Right
Arithmetic instructions (ALU immediate)	ADDI	Add Immediate
	ADDIU	Add Immediate Unsigned
	SLTI	Set on Less Than Immediate
	SLTIU	Set on Less Than Immediate Unsigned
	ANDI	AND Immediate
	ORI	OR Immediate
	XORI	Exclusive OR Immediate
	LUI	Load Upper Immediate

Category	Instruction	Description
Arithmetic instructions (3 operands, R-type)	ADD	Add
	ADDU	Add Unsigned
	SUB	Subtract
	SUBU	Subtract Unsigned
	SLT	Set on Less Than
	SLTU	Set on Less Than Unsigned
	AND	AND
	OR	OR
	XOR	Exclusive OR
	NOR	NOR
Shift instructions	SLL	Shift Left Logical
	SRL	Shift Right Logical
	SRA	Shift Right Arithmetic
	SLLV	Shift Left Logical Variable
	SRLV	Shift Right Logical Variable
	SRAV	Shift Right Arithmetic Variable
Multiply and divide instructions	MULT	Multiply
	MULTU	Multiply Unsigned
	DIV	Divide
	DIVU	Divide Unsigned
	MFHI	Move From HI
	MTHI	Move To HI
	MFLO	Move From LO
	MTLO	Move To LO
Jump and branch instructions	J	Jump
	JAL	Jump And Link
	JR	Jump Register
	JALR	Jump And Link Register
	BEQ	Branch on Equal
	BNE	Branch on Not Equal
	BLEZ	Branch on Less than or Equal to Zero
	BGTZ	Branch on Greater Than Zero
	BLTZ	Branch on Less Than Zero
	BGEZ	Branch on Greater than or Equal to Zero
	BLTZAL	Branch on Less Than Zero And Link
	BGEZAL	Branch on Greater than or Equal to Zero And Link
Coprocessor instructions	LWCz	Load Word to Coprocessor z
	SWCz	Store Word from Coprocessor z
	MTCz	Move To Coprocessor z
	MFCz	Move From Coprocessor z
	CTCz	move Control to Coprocessor z
	CFCz	move Control from Coprocessor z
	COPz	Coprocessor z Operation
	BCzT	Branch on Coprocessor z True
	BCzF	Branch on Coprocessor z False
Special instructions	SYSCALL	System Call
	BREAK	Break Point

Table 1-2: MIPS II ISA Instructions

Category	Instruction	Description
Load and store instructions	LL	Load Linked
	SC	Store Conditional
Jump and branch instructions	BEQL	Branch on Equal Likely
	BNEL	Branch on Not Equal Likely
	BLEZL	Branch on Less than or Equal to Zero Likely
	BGTZL	Branch on Greater Than Zero Likely
	BLTZL	Branch on Less Than Zero Likely
	BGEZL	Branch on Greater than or Equal to Zero Likely
	BLTZALL	Branch Less Than Zero And Link Likely
	BGEZALL	Branch Greater than or Equal to Zero And Link Likely

Table 1-3: ALLEGREX™Extended Instructions

Category	Instruction	Description
Arithmetic instructions	MAX	Select Max
	MIN	Select Min
Move instructions	MOVZ	Move Conditional on Zero
	MOVN	Move Conditional on Not Zero
	EXT	Extract Bit Field
	INS	Insert Bit Field
	SEB	Sign-Extend Byte
	SEH	Sign-Extend Halfword
Shift instructions	ROTR	Rotate Word Right
	ROTRV	Rotate Word Right Variable
	WSBH	Word Swap Bytes within Halfword
	WSBW	Word Swap Bytes within Word
	BITREV	Bit Reverse
Multiply and add, multiply and subtract instructions	MADD	Multiply and Add
	MADDU	Multiply and Add Unsigned
	MSUB	Multiply and Subtract
	MSUBU	Multiply and Subtract Unsigned
Count leading 0 and 1 instructions	CLZ	Count Leading Zero
	CLO	Count Leading One
Special instructions	SYNC	Synchronize Shared Memory

Table 1-4: MIPS CP0 Instructions

Category	Instruction	Description
CP0 move instructions	MTC0	Move To CP0
	MFC0	Move From CP0
Special instructions	ERET	Exception Return
	CACHE	Cache Operation

Table 1-5: FPU Transfer Instructions

Category	Instruction	Description
FPU transfer instructions	LWC1	Load Word to FPU
	SWC1	Store Word from FPU
	MTC1	Move To FPU
	MFC1	Move From FPU
	CTC1	Move Control To FPU
	CFC1	Move Control From FPU
FPU branch instructions	BC1T	Branch on FPU True
	BC1TL	Branch on FPU True Likely
	BC1F	Branch on FPU False
	BC1FL	Branch on FPU False Likely

1.1.4. Data Formats and Addressing

ALLEGREX™ supports 3 data sizes, namely, word (32 bits), halfword (16 bits), and byte (8 bits). Byte order can be either little-endian or big-endian.

In little-endian byte ordering, byte 0 is the lowest order (rightmost) byte within a multiple byte structure. In big-endian, the order of bytes is reversed and byte 0 is the highest order (leftmost) byte. Figure 1-4 shows byte positions within words for little-endian byte ordering. Figure 1-5 and Figure 1-6 show byte positions on a 128-bit bus in little-endian.

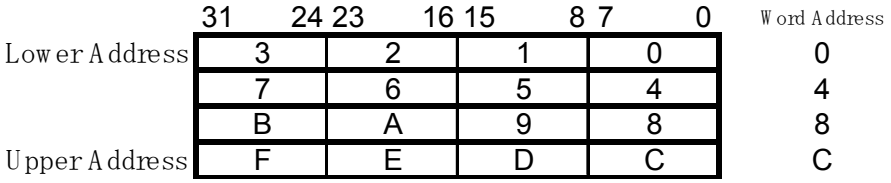


Figure 1-4: Byte Addresses Within Words (Little-Endian)

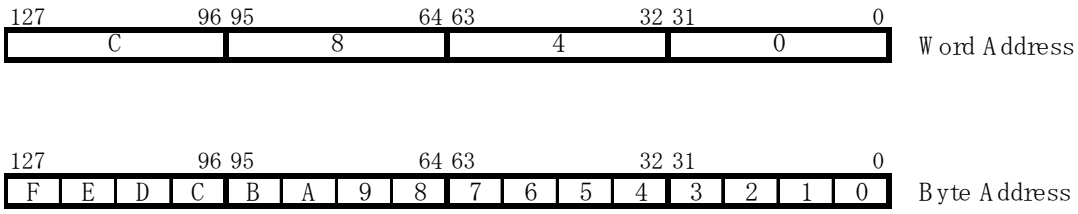


Figure 1-5: Word and Byte Addresses on a 128-Bit Bus (Little-Endian)

ALLEGREX™ also uses byte addresses to access halfwords and words. Halfword data must be located on a 2-byte boundary (0, 2, 4, …), whereas word data must be located on a 4-byte boundary (0, 4, 8, …).

However, the LWL / LWR and SWL / SWR instructions can be used to access word data that is not aligned on a 4-byte boundary. These instructions must be used in pairs to access unaligned word data, so one extra instruction cycle is needed compared to an aligned word access. Figure 1-6 shows how bytes are accessed when the byte order is little-endian and an unaligned word is addressed beginning with byte address 3.

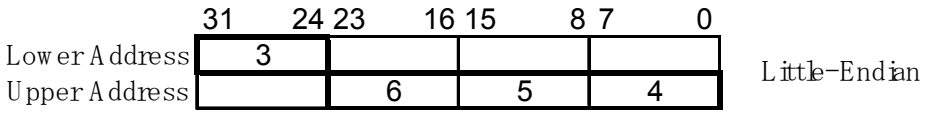


Figure 1-6: Unaligned Word (Byte Address = 3)

1.1.5. System Control Coprocessor (CP0)

The ALLEGREX™ Core can operate the external coprocessors CP1-CP3.

CP1: FPU

CP2: VFPU

CP3: Reserved

For details, please refer to each user's manual.

System control coprocessor (CP0) is built into the CPU, and provides memory management and exception handling support for the operating system. Table 1-6 presents simplified descriptions of the system control coprocessor (CP0) registers. For information on exception handling, please see Chapter 6.

Table 1-6: System Control Coprocessor (CP0) Registers

No.	Register	R/W	Function
0	---		Reserved
1	---		Reserved
2	---		Reserved
3	---		Reserved
4	---		Reserved
5	---		Reserved
6	---		Reserved
7	---		Reserved
8	BadVAddr	R	Virtual address where the last error occurred
9	Count	R/W	Free-running timer count
10	---		Reserved
11	Compare	R/W	Timer comparison value
12	Status	R/W	Status register
13	Cause	R/W	Cause of last exception
14	EPC	R/W	Exception program counter
15	PRId	R	Processor Revision ID register
16	Config	R	Configuration register
17	---		Reserved
18	---		Reserved
19	---		Reserved
20	---		Reserved
21	---		Reserved
22	---		Reserved
23	---		Reserved
24	---		Reserved

No.	Register	R/W	Function
25	EBace	R/W	Virtual address of exception vector address
26	---		Reserved
27	---		Reserved
28	TagLo	R/W	Cache instruction register
29	TagHi	R/W	Cache instruction register
30	ErrorEPC	R/W	Error exception program counter
31	---		Reserved

1.1.6. Internal Caches

ALLEGREX™ has internal instruction and data caches that help to achieve efficient use of the high-performance pipeline. The caches are independent and can be accessed in parallel. Each cache has a 128-bit wide data and can be accessed in 1 cycle. The caches are described in Chapter 4.

1.2. Memory Management System

ALLEGREX™ has a 4GB (32-bit) address space. However, very few systems will actually implement such a large physical memory. Instead, ALLEGREX™ provides an address mapping function that enables the virtual address space to be mapped into actual physical memory addresses.

1.2.1. Direct Segment Mapping

Virtual addresses are mapped into physical addresses using a process known as direct segment mapping. This method does not use the TLB, which is implemented in R4000-type processors.

In direct segment mapping the 4 GB address space is divided into five segments. Within each segment the low-order bits of the virtual address are used directly for the physical address. This enables two different virtual addresses to be mapped to the same physical address simply by switching segments.

For example, the user-mode segments, kuseg0 and kuseg1, provide cached and non-cached accesses, respectively. By switching between these segments, the same virtual address can perform cached and non-cached accesses to the same physical memory address.

1.2.2. Operating Modes

ALLEGREX™ has three operating modes (user mode, supervisor mode, and kernel mode) just like an R4000-type processor. Normally, the CPU executes programs in user mode and

switches to kernel mode when an exception occurs. It returns to user mode by executing the ERET instruction.
Details about the MMU and operating modes are explained in Chapter 5.

1.3. ALLEGREX™ Pipeline Architecture

ALLEGREX™ has a 7-stage pipeline for performing instruction-level parallelism. Normally, when the pipeline isn't stalled, one instruction can be issued every cycle. The CPU pipeline is used to execute general instructions and operates independently of the FPU and VFPU pipelines. In addition to the CPU pipeline, an independent arithmetic unit is provided for performing multiplication, division, multiply and add, and multiply and subtract operations.

Figure 1-7 shows how the 7-stage pipeline of the ALLEGREX™ CPU Core can overlap the execution of seven instructions.

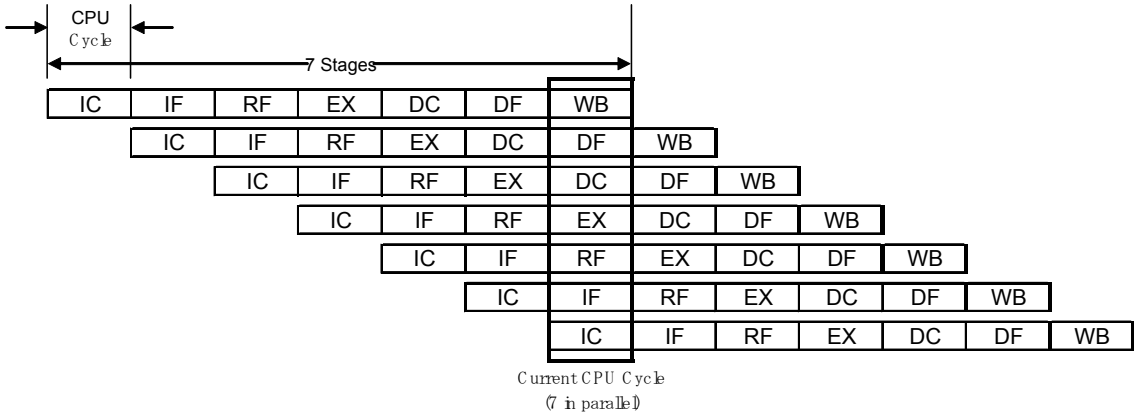


Figure 1-7: ALLEGREX™ Pipeline Overlap

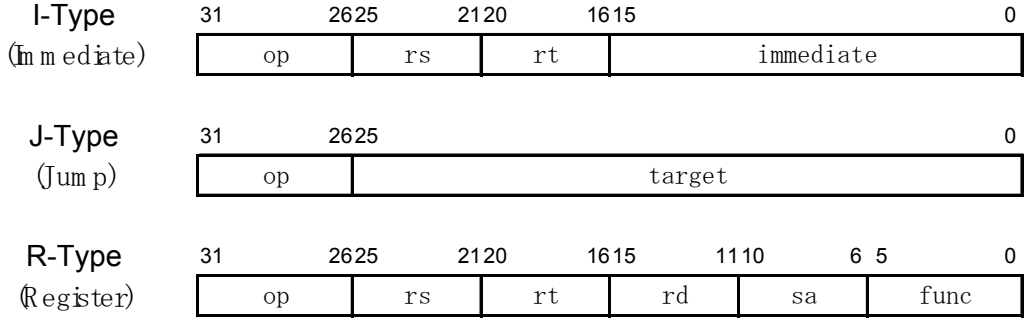
For details about the ALLEGREX™ pipeline, see Chapter 3.

2. CPU Instruction Set Overview

This chapter presents an overview of the ALLEGREX™ instruction set which includes MIPS instructions as well as extended instructions unique to ALLEGREX™.

2.1. Instruction Formats

CPU instructions are all one word in length (32 bits), and are always aligned on a word boundary. There are three instruction formats, which are shown in Figure 2-1. By cutting the number of instruction formats, instructions can be decoded easily. More complex (less frequently used) operations and addressing modes are implemented by the compiler.



op: Operation Code
rs: Source Register Specifier
rt: Target (Source or Destination) Register Specifier or Branch Condition
rd: Destination Register Specifier
sa: Shift Amount
immediate: Immediate or Address Offset
target: 2 Bit Right Shift of the Jump Destination Target Address
func: Function (Sub-Operation Code)

Figure 2-1: CPU Instruction Formats

2.2. Load and Store Instructions

Load and store instructions are I-type (immediate format) instructions that transfer data between memory and general-purpose registers. Only one addressing mode is supported by load and store instructions. The effective address is always generated by adding a signed 16-bit immediate offset to the base register.

If a register is loaded by a load instruction, and that load instruction is immediately followed by another instruction that uses the register that was just loaded for its source or

destination operand, the pipeline will interlock when the register is accessed by the instruction.

The access type of a load/store instruction (word, byte, etc.) is determined by the opcode, and that in turn determines the data size. The address used by a load/store instruction is the lowest order byte address (highest order byte for big-endian and lowest order byte for little-endian), regardless of the access type or byte order (endian).

Which bytes are accessed in the word at the specified address is determined by the access type and low-order 2 bits of the address as shown in Figure 2-2. Other combinations of access type and address bits 0-1 not shown in Figure 2-2 are not permitted and their use will cause an address error exception to occur.

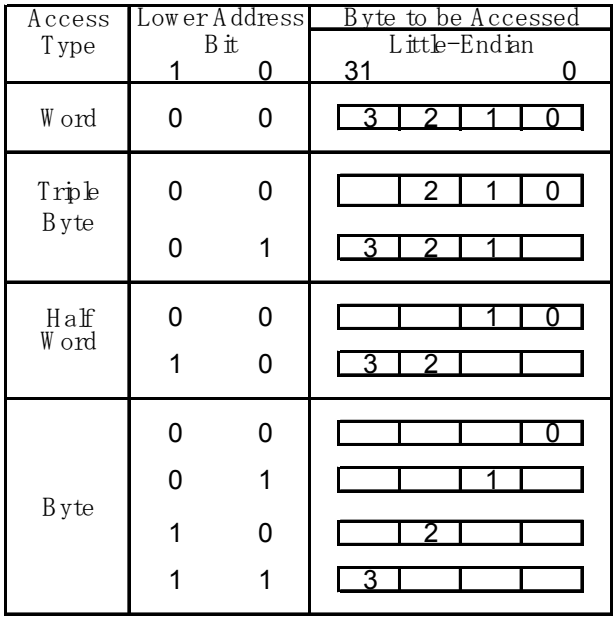


Figure 2-2: Byte Addressing in Load/Store Instructions

Table 2-1: Load/Store Instruction lists the load and store instructions that are supported by ALLEGREX™.

Table 2-1: Load/Store Instructions

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
op	base	rt	offset
6	5	5	16

Instruction	Format	Description
Load Byte	LB rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The byte in memory at that address is sign-extended and loaded into register rt.

Instruction	Format	Description
Load Byte Unsigned	LBU rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The byte in memory at that address is zero-extended and loaded into register rt.
Load Halfword	LH rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The halfword in memory at that address is sign-extended and loaded into register rt.
Load Halfword Unsigned	LHU rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The halfword in memory at that address is zero-extended and loaded into register rt.
Load Word	LW rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The word in memory at that address is loaded into register rt.
Load Word Left	LWL rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The word in memory at that address is shifted left so that the byte at that address is at the leftmost end of the word. The result of the shift operation is merged into register rt.
Load Word Right	LWR rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The word in memory at that address is shifted right so that the byte at that address is at the rightmost end of the word. The result of the shift operation is merged into register rt.
Load Linked	LL rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The word in memory at that address is loaded into register rt and the LLbit is set to 1.
Store Byte	SB rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The low-order byte of register rt is stored in memory at that address.
Store Halfword	SH rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The low-order halfword of register rt is stored in memory at that address.
Store Word	SW rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The contents of register rt are stored in memory at that address.
Store Word Left	SWL rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The contents of register rt are shifted right so that the leftmost byte is in the same position in the word as the byte at the generated address. The byte (bytes) that contains the original data of register rt is stored in the byte (bytes) from the byte position of the specified address to the word boundary on the right side.

Instruction	Format	Description
Store Word Right	SWR rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The contents of register rt are shifted left so that the rightmost byte is in the same position in the word as the byte at the generated address. The byte (bytes) that contains the original data of register rt is stored in the byte (bytes) from the byte position of the specified address to the word boundary on the left side.
Store Conditional	SC rt,offset(base)	If the LLbit is 1, the 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The contents of register rt are stored in memory at that address and 1 is returned in register rt. If the LLbit is 0, no store is performed and 0 is returned in register rt.

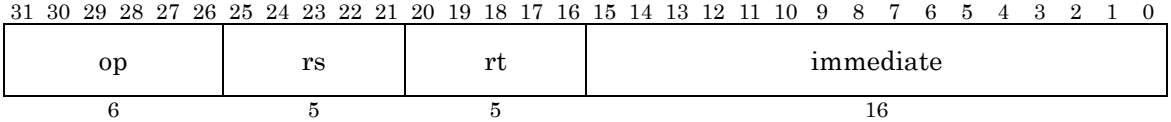
2.3. Arithmetic Instructions

Arithmetic instructions perform arithmetic, logical and shift operations. These instructions are either in register format (R-type) in which both operands are in registers, or immediate format (I-type) in which one operand is an immediate value. Arithmetic instructions are classified as follows.

1. MIPS ALU immediate instructions (Table 2-2)
2. MIPS 3 operand instructions (Table 2-3)
3. MIPS shift instructions (Table 2-4)
4. MIPS multiplication/division instructions (Table 2-5)
5. ALLEGREX™ extended arithmetic instructions (Table 2-6)
6. ALLEGREX™ extended move instructions (Table 2-7)
7. ALLEGREX™ extended shift instructions (Table 2-8)
8. ALLEGREX™ extended multiply and add/multiply and subtract instructions (Table 2-9)
9. ALLEGREX™ extended count leading 0/1 instructions (Table 2-10)

Table 2-2: MIPS ALU Immediate Instructions

Format

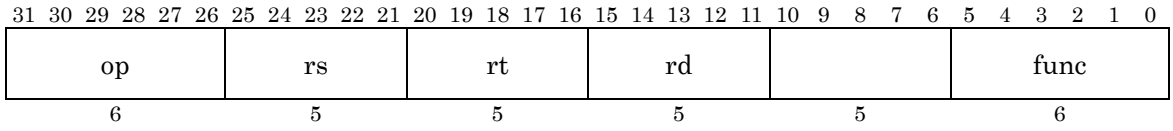


Instructions

Instruction	Format	Description
ADD Immediate	ADDI rt,rs,immediate	The 16-bit immediate field is sign-extended to 32 bits and added to register rs. The 32-bit result is stored in register rt. If a two's complement overflow occurs, an exception is generated.
ADD Immediate Unsigned	ADDIU rt,rs,immediate	The 16-bit immediate field is sign-extended to 32 bits and added to register rs. The 32-bit result is stored in register rt. No exception is generated even if an overflow occurs.
Set on Less Than Immediate	SLTI rt,rs,immediate	The contents of register rs and the 16-bit immediate value sign-extended to 32 bits are compared as 32-bit signed integers. If the comparison result is rs < immediate, 1 is stored in register rt, otherwise 0 is stored.
Set on Less Than Immediate Unsigned	SLTIU rt,rs,immediate	The contents of register rs and the 16-bit immediate value sign-extended to 32 bits are compared as 32-bit unsigned integers. If the comparison result is rs < immediate, 1 is stored in register rt, otherwise 0 is stored.
AND Immediate	ANDI rt,rs,immediate	The 16-bit immediate field is zero-extended and ANDed together with the contents of register rs. The result is stored in register rt.
OR Immediate	ORI rt,rs,immediate	The 16-bit immediate field is zero-extended and ORed together with the contents of register rs. The result is stored in register rt.
Exclusive OR Immediate	XORI rt,rs,immediate	The 16-bit immediate field is zero-extended and exclusive ORed together with the contents of register rs. The result is stored in register rt.
Load Upper Immediate	LUI rt,immediate	The 16-bit immediate value is shifted left by 16 bits to produce a 32-bit word. The low-order 16 bits are set to 0 and the result is stored in register rt.

Table 2-3: MIPS 3 Operand Register Type Instructions

Format

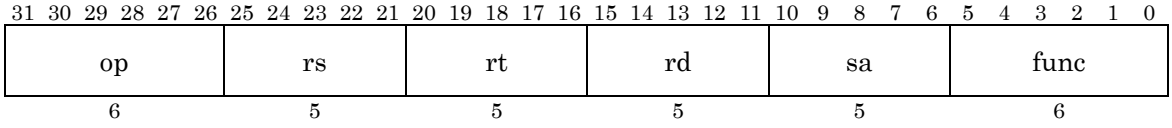


Instructions

Instruction	Format	Description
ADD	ADD rd,rs,rt	The contents of registers rs and rt are added together and the 32-bit result is stored in register rd. If a two's complement overflow occurs, an exception is generated.
Add Unsigned	ADDU rd,rs,rt	The contents of registers rs and rt are added together and the 32-bit result is stored in register rd. No exception is generated even if a two's complement overflow occurs.
Subtract	SUB rd,rs,rt	The contents of register rt are subtracted from the contents of register rs and the 32-bit result is stored in register rd. If a two's complement overflow occurs, an exception is generated.
Subtract Unsigned	SUBU rd,rs,rt	The contents of register rt are subtracted from the contents of register rs and the 32-bit result is stored in register rd. No exception is generated even if a two's complement overflow occurs.
Set on Less Than	SLT rd,rs,rt	The contents of registers rs and rt are compared as 32-bit signed integers. If the comparison result is rs < rt, 1 is stored in register rd, otherwise 0 is stored.
Set on Less Than Unsigned	SLTU rd,rs,rt	The contents of registers rs and rt are compared as 32-bit unsigned integers. If the comparison result is rs < rt, 1 is stored in register rd, otherwise 0 is stored.
AND	AND rd,rs,rt	The contents of registers rs and rt are ANDed together and the result is stored in register rd.
OR	OR rd,rs,rt	The contents of registers rs and rt are ORed together and the result is stored in register rd.
Exclusive OR	XOR rd,rs,rt	The contents of registers rs and rt are exclusive ORed together and the result is stored in register rd.
NOR	NOR rd,rs,rt	The contents of registers rs and rt are NORed together and the result is stored in register rd.

Table 2-4: MIPS Shift Instructions

Format

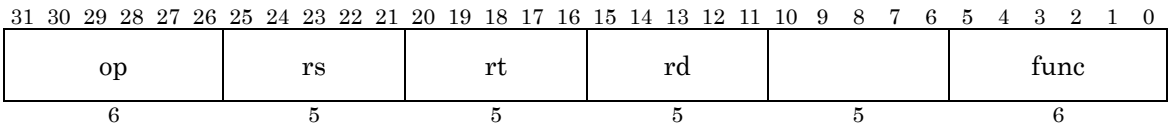


Instructions

Instruction	Format	Description
Shift Left Logical	SLL rd,rt,sa	The contents of register rt are shifted left sa bits. Zeroes are inserted into the low-order bit positions from the right. The 32-bit result is stored in register rd.
Shift Right Logical	SRL rd,rt,sa	The contents of register rt are shifted right sa bits. Zeroes are inserted into the high-order bit positions from the left. The 32-bit result is stored in register rd.
Shift Right Arithmetic	SRA rd,rt,sa	The contents of register rt are shifted right sa bits. The sign is extended into the high-order bit positions. The 32-bit result is stored in register rd.
Shift Left Logical Variable	SLLV rd,rt,rs	The contents of register rt are shifted left. Zeroes are inserted into the low-order bit positions from the right. The shift amount is specified by the low-order 5 bits of register rs. The 32-bit result is stored in register rd.
Shift Right Logical Variable	SRLV rd,rt,rs	The contents of register rt are shifted right. Zeroes are inserted into the high-order bit positions from the left. The shift amount is specified by the low-order 5 bits of register rs. The 32-bit result is stored in register rd.
Shift Right Arithmetic Variable	SRAV rd,rt,rs	The contents of register rt are shifted right. The sign is extended into the high-order bit positions. The shift amount is specified by the low-order 5 bits of register rs. The 32-bit result is stored in register rd.

Table 2-5: MIPS Multiplication/Division Instructions

Format



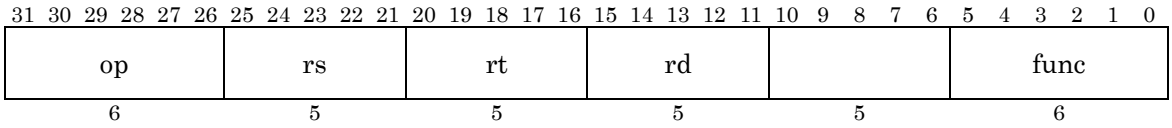
Instructions

Instruction	Format	Description
Multiply	MULT rs,rt	The contents of register rs are multiplied by the contents of register rt and the 64-bit result is stored in special registers HI and LO. The operands are treated as two's complement value.
Multiply Unsigned	MULTU rs,rt	The contents of register rs are multiplied by the contents of register rt and the 64-bit result is stored in special registers HI and LO. The operands are treated as unsigned integers.

Instruction	Format	Description
Divide	DIV rs,rt	The contents of register rs are divided by the contents of register rt. The 32-bit quotient is stored in special register LO, and the 32-bit remainder is stored in special register HI. The operands are treated as two's complement value.
Divide Unsigned	DIVU rs,rt	The contents of register rs are divided by the contents of register rt. The 32-bit quotient is stored in special register LO, and the 32-bit remainder is stored in special register HI. The operands are treated as unsigned integers.
Move From HI	MFHI rd	The contents of special register HI are transferred to register rd.
Move From LO	MFLO rd	The contents of special register LO are transferred to register rd.
Move To HI	MTHI rs	The contents of register rs are transferred to special register HI.
Move To LO	MTLO rs	The contents of register rs are transferred to special register LO.

Table 2-6: ALLEGREX™ Extended Arithmetic Instructions

Format

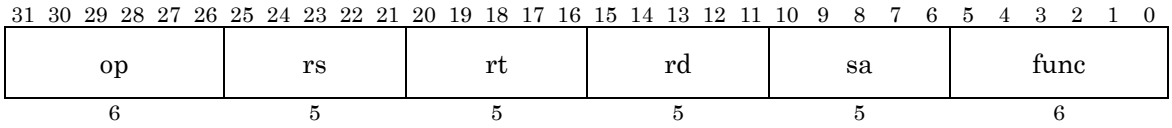


Instructions

Instruction	Format	Description
Select Max	MAX rd,rs,rt	The contents of registers rs and rt are compared as 32-bit signed integers. The contents of the register containing the larger value are stored in register rd.
Select Min	MIN rd,rs,rt	The contents of registers rs and rt are compared as 32-bit signed integers. The contents of the register containing the smaller value are stored in register rd.

Table 2-7: ALLEGREX™ Extended Move Instructions

Format



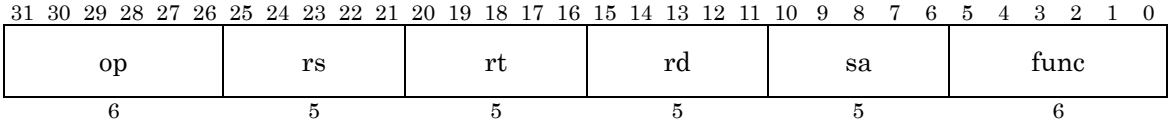
Instructions

Instruction	Format	Description
Move Conditional on Not Zero	MOVN rd,rs,rt	If the contents of register rt are not equal to zero, the contents of register rs are copied to register rd.

Instruction	Format	Description
Move Conditional on Zero	MOVZ rd,rs,rt	If the contents of register rt are equal to zero, the contents of register rs are copied to register rd.
Extract Bit Field	EXT rt,rs,pos,size	size(=rd+1) bits are extracted from register rs starting at offset bit position pos (=sa) within the word. The result is stored right-justified in register rt. The high-order bits of register rt are filled with zeros.
Insert Bit Field	INS rt,rs,pos,size	size(=rd-sa+1) bits are extracted from register rs starting from the low-order bit position and inserted into register rt at bit position pos(=sa) within the word.
Sign-Extend Byte	SEB rd,rt	The lowest byte of register rt is sign-extended to 32 bits. The result is stored in register rd.
Sign-Extend Halfword	SEH rd,rt	The low-order halfword of register rt is sign-extended to 32 bits. The result is stored in register rd.

Table 2-8: ALLEGREX™ Extended Shift Instructions

Format

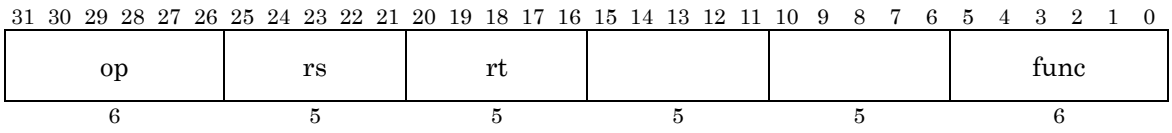


Instructions

Instruction	Format	Description
Rotate Word Right	ROTR rd,rt,sa	The contents of register rt are rotated right sa bits. The 32-bit result is stored in register rd.
Rotate Word Right Variable	ROTRV rd,rt,rs	The contents of register rt are rotated right by an amount specified by the low-order 5 bits of register rs. The 32-bit result is stored in register rd.
Word Swap Bytes within Halfword	WSBH rd,rt	The contents of register rt are swapped byte-for-byte within each halfword of the register. The 32-bit result is stored in register rd.
Word Swap Bytes within Word	WSBW rd,rt	The contents of register rt are swapped byte-for-byte within the word. The 32-bit result is stored in register rd.
Bit Reverse	BITREV rd,rt	The contents of register rt are swapped bit-for-bit within the word. The 32-bit result is stored in register rd.

Table 2-9: ALLEGREX™ Extended Multiply and Add/Multiply and Subtract Instructions

Format

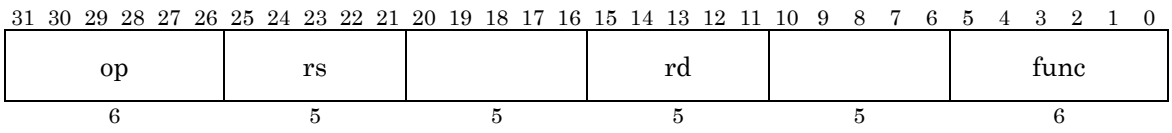


Instructions

Instruction	Format	Description
Multiply Add	MADD rs,rt	The contents of register rs are multiplied by the contents of register rt. The 64-bit product is added to the 64-bit value obtained by concatenating special registers HI and LO. The result is stored back into the HI/LO register pair. The operands are treated as two's complement value.
Multiply Add Unsigned	MADDU rs,rt	The contents of register rs are multiplied by the contents of register rt. The 64-bit product is added to the 64-bit value obtained by concatenating special registers HI and LO. The result is stored back into the HI/LO register pair. The operands are treated as unsigned integers.
Multiply Subtract	MSUB rs,rt	The contents of register rs are multiplied by the contents of register rt. The 64-bit product is subtracted from the 64-bit value obtained by concatenating special registers HI and LO. The result is stored back into the HI/LO register pair. The operands are treated as two's complement value.
Multiply Subtract Unsigned	MSUBU rs,rt	The contents of register rs are multiplied by the contents of register rt. The 64-bit product is subtracted from the 64-bit value obtained by concatenating special registers HI and LO. The result is stored back into the HI/LO register pair. The operands are treated as unsigned integers.

Table 2-10: ALLEGREX™ Extended Count Leading 0/1 Instructions

Format



Instructions

Instruction	Format	Description
Count Leading Zero	CLZ rd,rs	The number of consecutive zeroes in register rs are counted starting from the leftmost bit (MSB). The result (0-32) is stored in register rd.
Count Leading One	CLO rd,rs	The number of consecutive ones in register rs are counted starting from the leftmost bit (MSB). The result (0-32) is stored in register rd.

Number of Cycles Needed for Multiply, Divide, Multiply and Add, and Multiply and Subtract

Unlike other integer arithmetic instructions, the multiply, divide, multiply and add, and multiply and subtract instructions need multiple cycles to execute. If you try to load the results from these instructions into a general-purpose register by executing either the MFHI or MFLO instruction before these multi-cycle arithmetic instructions have completed, the pipeline will interlock. Since the arithmetic unit for executing these instructions operates independently of the normal pipeline, an instruction other than MFHI or MFLO that follows one of these multi-cycle arithmetic instructions can be executed in parallel. Table 2-11 shows the number of cycles required to execute these multi-cycle arithmetic instructions.

Table 2-11: Number of Cycles Required for the Multiplication, Division, Multiply and Add, and Multiply and Subtract Instructions

Instruction	Number of Cycles Required
MULT	5
MULTU	5
DIV	36
DIVU	36
MADD	5
MADDU	5
MSUB	5
MSUBU	5

2.4. Jump and Branch Instructions

Jump and branch instructions change the control flow of a program.

Every branch instruction and every jump instruction in the MIPS instruction set is followed by a one instruction delay slot. This means that the instruction immediately following the branch or jump instruction is always executed while the branch target instruction is being fetched. This is true except for the branch likely instructions which are MIPS extended instructions. With a branch likely instruction, if the branch condition is not satisfied, the instruction in the delay slot is nullified and the delay slot becomes a bubble slot.

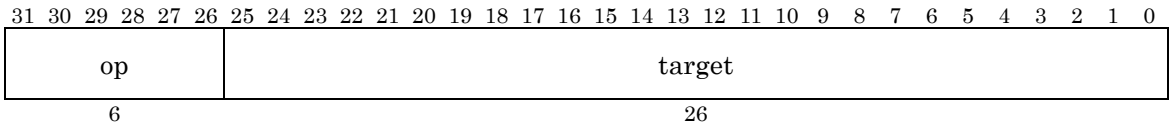
Normally, the Jump instruction (J) or Jump-And-Link instruction (JAL) is used to perform subroutine calls in high-level languages. The format of both the Jump and Jump-And-Link instruction is jump (J-type). In this format, the 26-bit target address is shifted 2 bits to the left to generate a 32-bit absolute address that is combined with the high-order 4 bits of the current program counter.

Normally, the Jump-Register instruction (JR) or Jump-And-Link-Register instruction (JALR) is used to return from a subroutine, to perform dispatching with a function pointer,

or to perform a long jump that exceeds a page. Both of these are R-type instructions in which the contents of a general-purpose register contain the 32-bit target address.
Branch instructions calculate their target address by using a signed 16-bit offset relative to the program counter. The Jump-And-Link instruction (JAL) and Branch-And-Link instruction (BAL) save the return address in register r31 (the link register).
Table 2-12 and Table 2-13 list the MIPS jump instructions supported by ALLEGREX™, Table 2-14 and Table 2-15 list the branch instructions, and Table 2-16 and Table 2-17 list the MIPS extended branch instructions.

Table 2-12: MIPS Jump Instructions (J-Type)

Format

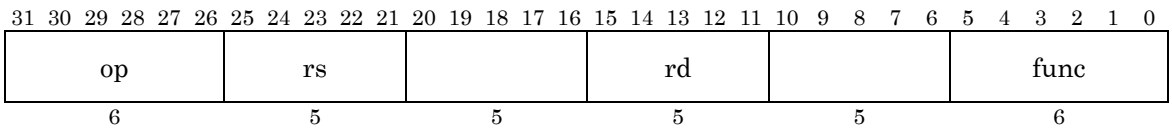


Instructions

Instruction	Format	Description
Jump	J target	The program jumps to the destination address. The destination address is formed by shifting the 26-bit target left two bits and prefixing it with the high-order 4 bits of the PC. The low-order two bits are set to 0. The program jumps after a one instruction delay.
Jump And Link	JAL target	The program jumps to the destination address and stores the address of the instruction following the delay slot in r31 (the link register). The destination address is formed by shifting the 26-bit target left two bits and prefixing it with the high-order 4 bits of the PC. The low-order two bits are set to 0. The program jumps after a one instruction delay.

Table 2-13: MIPS Jump Instructions (R-Type)

Format

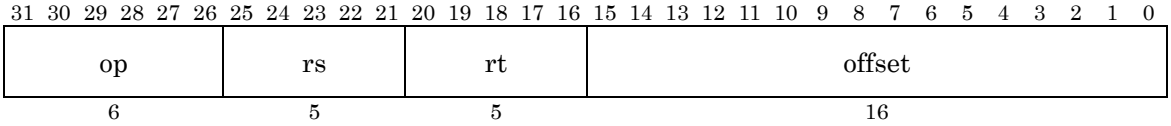


Instructions

Instruction	Format	Description
Jump Register	JR rs	The program jumps to the address in the rs register. The program jumps after a one instruction delay.
Jump And Link Register	JALR rd,rs	The program jumps to the address in the rs register and stores the address of the instruction following the delay slot in register rd (if rd is omitted, r31 is used as the default). The program jumps after a one instruction delay.

Table 2-14: MIPS Branch Instructions (I-Type)

Format



Instructions

Instruction	Format	Description
Branch on Equal	BEQ rs,rt,offset	When the value of register rs is equal to the value of register rt, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.
Branch on Not Equal	BNE rs,rt,offset	When the value of register rs is not equal to the value of register rt, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.
Branch on Less than or Equal to Zero	BLEZ rs,offset	When the value of register rs is less than or equal to zero, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.
Branch on Greater Than Zero	BGTZ rs,offset	When the value of register rs is greater than zero, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.

Table 2-15: MIPS Branch Instructions (REGIMM Class)

Format

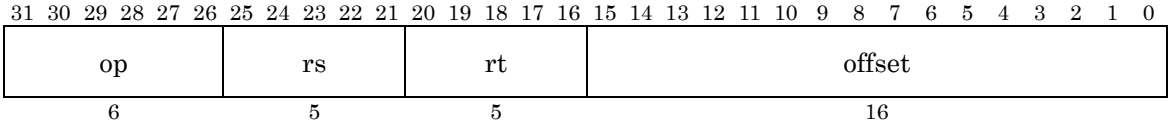
31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
REGIMM	rs	func	offset
6	5	5	16

Instructions

Instruction	Format	Description
Branch on Less Than Zero	BLTZ rs,offset	When the value of register rs is less than zero, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.
Branch on Greater than or Equal to Zero	BGEZ rs,offset	When the value of register rs is greater than or equal to zero, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.
Branch on Less Than Zero And Link	BLTZAL rs,offset	When the value of register rs is less than zero, the program branches to the branch target address and stores the address of the instruction following the delay slot in register r31 (the link register). The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.
Branch on Greater than or Equal to Zero And Link	BGEZAL rs, offset	When the value of register rs is greater than or equal to zero, the program branches to the branch target address and stores the address of the instruction following the delay slot in register r31 (the link register). The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.

Table 2-16: MIPS Extended Branch Instructions (R-Type)

Format

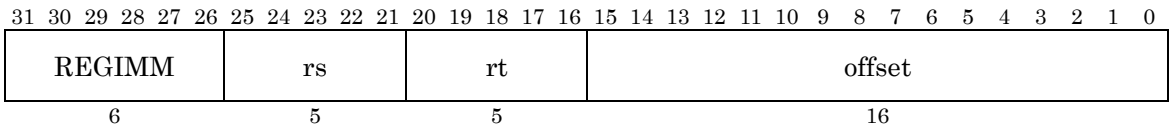


Instructions

Instruction	Format	Description
Branch on Equal Likely	BEQL rs,rt,offset	If the value of register rs is equal to the value of register rt, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.
Branch on Not Equal Likely	BNEL rs,rt,offset	If the value of register rs is not equal to the value of register rt, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.
Branch on Less than or Equal to Zero Likely	BLEZL rs,offset	If the value of register rs is less than or equal to zero, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.
Branch on Greater Than Zero Likely	BGTZL rs,offset	If the value of register rs is greater than zero, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.

Table 2-17: MIPS Extended Branch Instructions (REGIMM Class)

Format



Instructions

Instruction	Format	Description
Branch on Less Than Zero Likely	BLTZL rs,offset	If the value of register rs is less than zero, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.
Branch on Greater than or Equal to Zero Likely	BGEZL rs,offset	If the value of register rs is greater than or equal to zero, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.
Branch on Less Than Zero And Link Likely	BLTZALL rs,offset	If the value of register rs is less than zero, the program branches to the branch target address and stores the address of the instruction following the delay slot in register r31 (the link register). The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.

Instruction	Format	Description
Branch on Greater than or Equal to Zero And Link Likely	BGEZALL rs,offset	If the value of register rs is greater than or equal to zero, the program branches to the branch target address and stores the address of the instruction following the delay slot in register r31 (the link register). The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.

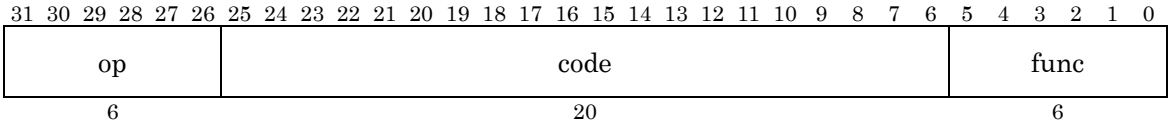
2.5. Special Instructions

2.5.1. MIPS Special Instructions

The MIPS special instructions are used for generating software traps. The instruction formats are all R-type. Table 2-18 summarizes the MIPS special instructions.

Table 2-18: MIPS Special Instructions

Format



Instructions

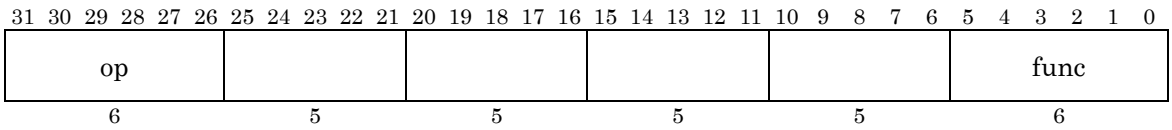
Instruction	Format	Description
System Call	SYSCALL code	A system call exception is generated and control is immediately passed to the exception handler. The code field can be set to an arbitrary value.
Break Point	BREAK code	A breakpoint exception is generated and control is immediately passed to the exception handler. The code field can be set to an arbitrary value.

2.5.2. Special Instructions Unique to ALLEGREX™

There is one special instruction unique to ALLEGREX™. Table 2-19 summarizes that instruction.

Table 2-19: ALLEGREX™ Proprietary Special Instruction

Format



Instruction

Instruction	Format	Description
Synchronize Shared Memory	SYNC	The pipeline is stalled until the cache writeback buffer and non-cached write buffer have emptied. This is done to block the execution of subsequent instructions.

2.6. Coprocessor Instructions

MIPS coprocessor instructions perform operations on coprocessors 1, 2 and 3 of the ALLEGREX™ CPU. On the PSP™ system chip, coprocessor 1 is the ALLEGREX™ FPU and coprocessor 2 is the VFPU.

2.6.1. Instructions Common to All Coprocessors

Coprocessor load/store instructions are I-type instructions. These are summarized in Table 2-20.

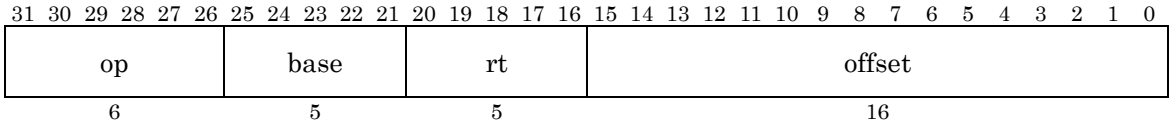
Table 2-21 summarizes the coprocessor transfer instructions.

The format of coprocessor operation instructions depends on the coprocessor. Table 2-22 shows the format of these instructions.

Table 2-23 summarizes the coprocessor branch instructions.

Table 2-20: Coprocessor Load/Store Instructions

Format



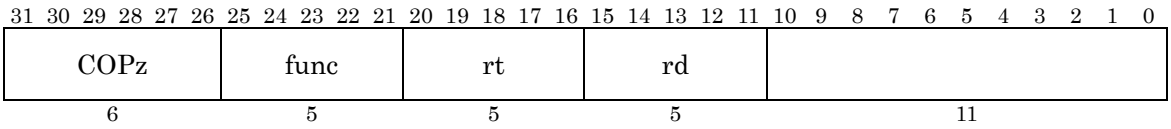
Instructions

Instruction	Format	Description
Load Word to Coprocessor z	LWCz rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The word in memory at that address is loaded into register rt of coprocessor z.

Instruction	Format	Description
Store Word from Coprocessor z	SWCz rt,offset(base)	The 16-bit offset is sign-extended and added to the contents of the base register to generate an address. The contents of register rt of coprocessor z are stored in memory at that address.

Table 2-21: Coprocessor Transfer Instructions

Format

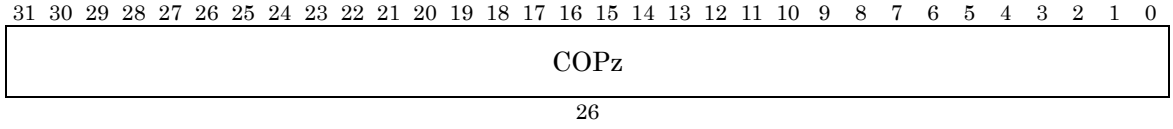


Instructions

Instruction	Format	Description
Move To Coprocessor z	MTCz rt,rd	The contents of CPU general-purpose register rt are transferred to register rd of coprocessor z.
Move From Coprocessor z	MFCz rt,rd	The contents of register rd of coprocessor z are transferred to CPU general-purpose register rt.
Move Control To Coprocessor z	CTCz rt,rd	The contents of CPU general-purpose register rt are transferred to control register rd of coprocessor z.
Move Control From Coprocessor z	CFCz rt,rd	The contents of control register rd of coprocessor z are transferred to CPU general-purpose register rt.

Table 2-22: Coprocessor Operation Instructions

Format

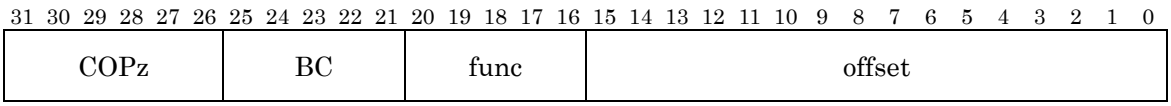


Instructions

Instruction	Format	Description
Coprocessor Operation z	COPz func	The specified operation for coprocessor z is executed. This instruction does not change the CPU state.

Table 2-23: Coprocessor Branch Instructions

Format



Instructions

Instruction	Format	Description
Branch on Coprocessor z True	BCzT offset	If the condition for coprocessor z is true, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.
Branch on Coprocessor z True Likely	BCzTL offset	If the condition for coprocessor z is true, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.
Branch on Coprocessor z False	BCzF offset	If the condition for coprocessor z is false, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.
Branch on Coprocessor z False Likely	BCzFL offset	If the condition for coprocessor z is false, the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.

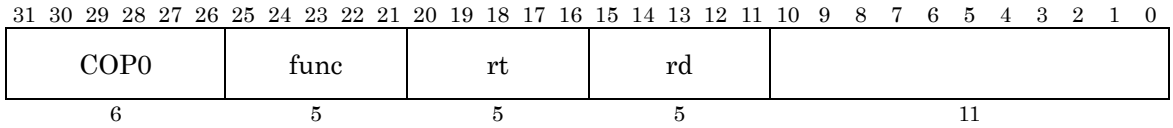
2.6.2. System Control Coprocessor (CP0) Instructions

Coprocessor 0 instructions perform operations on system control coprocessor (CP0) registers and are used for exception handling and memory management. Table 2-24 summarizes the system control coprocessor (CP0) transfer instructions, Table 2-25 summarizes the system control coprocessor (CP0) return instruction, and

Table 2-26 summarizes the system control coprocessor (CP0) cache instruction.

Table 2-24: System Control Coprocessor (CP0) Transfer Instructions

Format

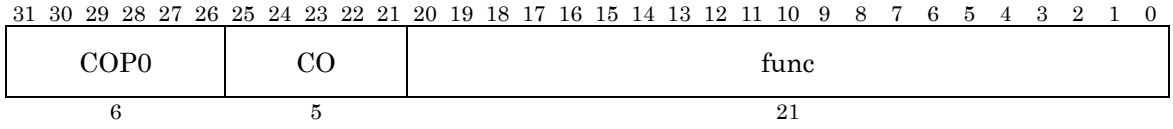


Instructions

Instruction	Format	Description
Move To CP0	MTC0 rt,rd	The contents of CPU register rt are transferred to system control coprocessor (CP0) register rd.
Move From CP0	MFC0 rt,rd	The contents of system control coprocessor (CP0) register rd are transferred to CPU register rt.

Table 2-25: System Control Coprocessor (CP0) Return Instruction

Format

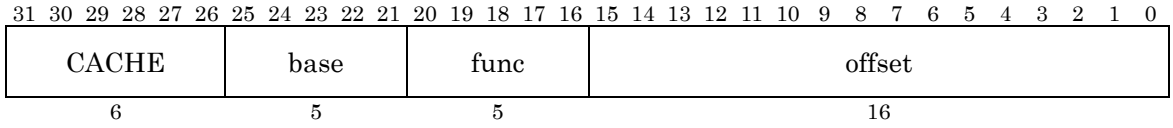


Instructions

Instruction	Format	Description
Exception Return	ERET	This instruction returns from an interrupt, exception, or error.

Table 2-26: System Control Coprocessor (CP0) Cache Instruction

Format



Instructions

Instruction	Format	Description
Cache Operation	CACHE func,offset(base)	The offset is sign-extended and added to the base register to generate an address. The cache operation indicated by func is performed on the cache line at that address.

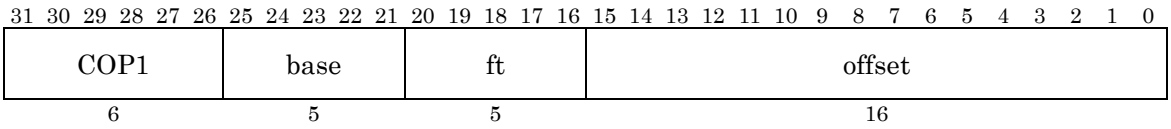
Although the opcode of the CACHE instruction is MIPS III ISA compliant, the func codes and the operational details are proprietary to ALLEGREX™. The operation of the CACHE instruction is explained in Chapter 4.

2.6.3. Single-Precision Floating-Point Unit (FPU) Instructions

This section shows typical instructions that are used to transfer data between the CPU and FPU. These are summarized in Table 2-27, Table 2-28, and Table 2-29. For detailed information on other FPU instructions, please see the “FPU User's Manual.”

Table 2-27: FPU Load and Store Instructions

Format

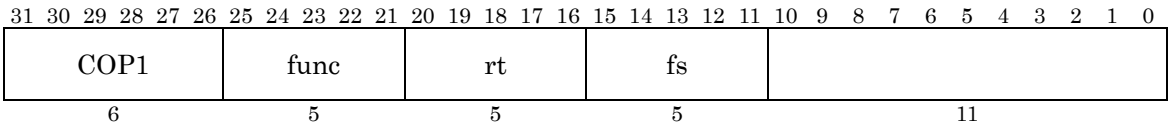


Instructions

Instruction	Format	Description
Load Word to FPU	LWC1 ft,offset(base)	The offset is sign-extended and added to the contents of the base register to generate an address. The word in memory at that address is loaded into FPU register ft.
Store Word from FPU	SWC1 ft,offset(base)	The offset is sign-extended and added to the contents of the base register to generate an address. The contents of FPU register ft are stored in memory at that address.

Table 2-28: FPU Transfer Instructions

Format



Instructions

Instruction	Format	Description
Move To FPU	MTC1 rt,fs	The contents of CPU register rt are transferred to FPU register fs.
Move From FPU	MFC1 rt,fs	The contents of FPU register fs are transferred to CPU register rt.
move Control To FPU	CTC1 rt,fs	The contents of CPU register rt are transferred to FPU control register fs.
move Control From FPU	CFC1 rt,fs	The contents of FPU control register fs are transferred to CPU register rt.

Table 2-29: FPU Branch Instructions

Format

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
COP1	BC	func	offset
6	5	5	16

Instructions

Instruction	Format	Description
Branch on FPU True	BC1T offset	If the FPU condition is true(1), the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.
Branch on FPU True Likely	BC1TL offset	If the FPU condition is true(1), the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.
Branch on FPU False	BC1F offset	If the FPU condition is false(0), the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay.
Branch on FPU False Likely	BC1FL offset	If the FPU condition is false(0), the program branches to the branch target address. The branch target address is calculated by shifting the 16-bit offset left two bits, sign-extending it to a 32 bit value and adding it to the PC. The program branches after a one instruction delay. If the branch is not taken, the instruction in the branch delay slot is nullified.

2.6.4. Vector Floating-Point Unit (VFPU) Instructions

VFPU instructions execute operations on VFPU (CP2) registers and are used to perform vector arithmetic. For more details, please refer to the “VFPU User's Manual.”

3. CPU Pipeline

This chapter explains the ALLEGREX™ pipeline.

3.1. Pipeline Stages

The ALLEGREX™ processor uses a 7-stage pipeline. The pipeline is single scalar and can issue one instruction per cycle. 7 clock cycles are required to execute each instruction. A new instruction is started every clock cycle, and seven independent instructions can be executed in parallel.

Figure 3-1 presents an overview of the pipeline. If all pipelines are filled, seven instructions will be executed simultaneously.

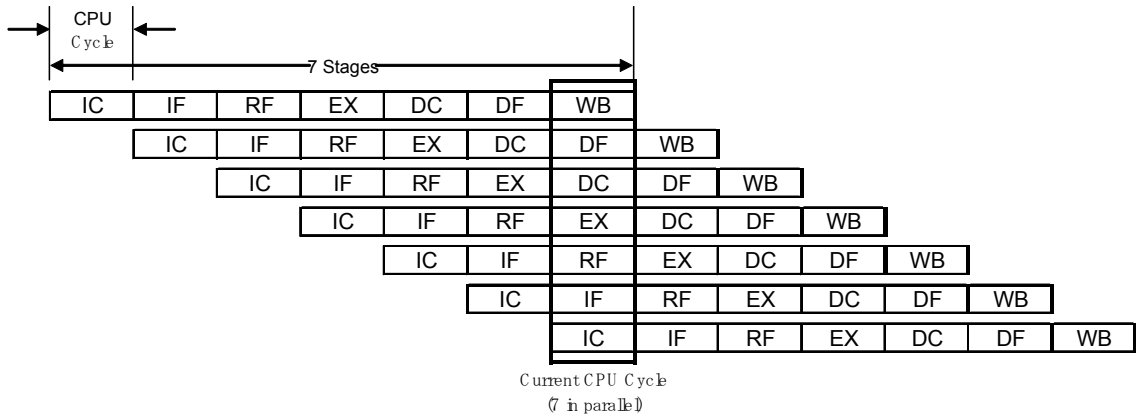


Figure 3-1: ALLEGREX™ Pipeline

The operation of each pipeline stage is described below.

1. IC Stage

Instruction cache access. When the instruction address is in a cached area, the instruction cache is accessed to fetch the instruction.

2. IF Stage

32-bit instruction fetch. The tag is checked for the accessed instruction to make sure the instruction is in the cache. If the instruction fetch is to a non-cached area, main memory is accessed directly.

3. RF Stage

Register fetch. The following operations are performed simultaneously.

(a) Decode the instruction and check the interlock state.
(b) Fetch the operands from the register file.
(c) If the instruction is a jump instruction, generate the jump destination address.

4. EX Stage

Instruction execution. One of the following operations is performed, depending on the type of instruction.

(a) If the instruction is a register-to-register operation, perform arithmetic, logical, and shift operations.
(b) If the instruction is a load/store, calculate the virtual address of the data.
(c) If the instruction is a branch, calculate the virtual address of the branch destination and check the branch condition.

5. DC Stage

Data cache access. When the instruction is a load/store and the data address is in the cached area, the data cache is accessed.

6. DF Stage

32-bit data fetch. One of the following operations is performed, depending on the type of instruction.

(a) If the instruction is a load/store and the data address is in a cached area, the tag for the accessed data is checked to make sure that the data is in the cache. In addition, the fetched data is aligned.
(b) If the instruction is a load and the data address is in a non-cached area, 32-bit data is fetched directly from main memory and aligned.
(c) If the instruction is a MOVE (MFCz), 32-bit data is fetched from the corresponding unit.

7. WB Stage

Write result. The instruction result is written to the register file, the data cache, or a coprocessor register.

Figure 3-2 shows the operation of each pipeline stage in the ALLEGREX™CPU.

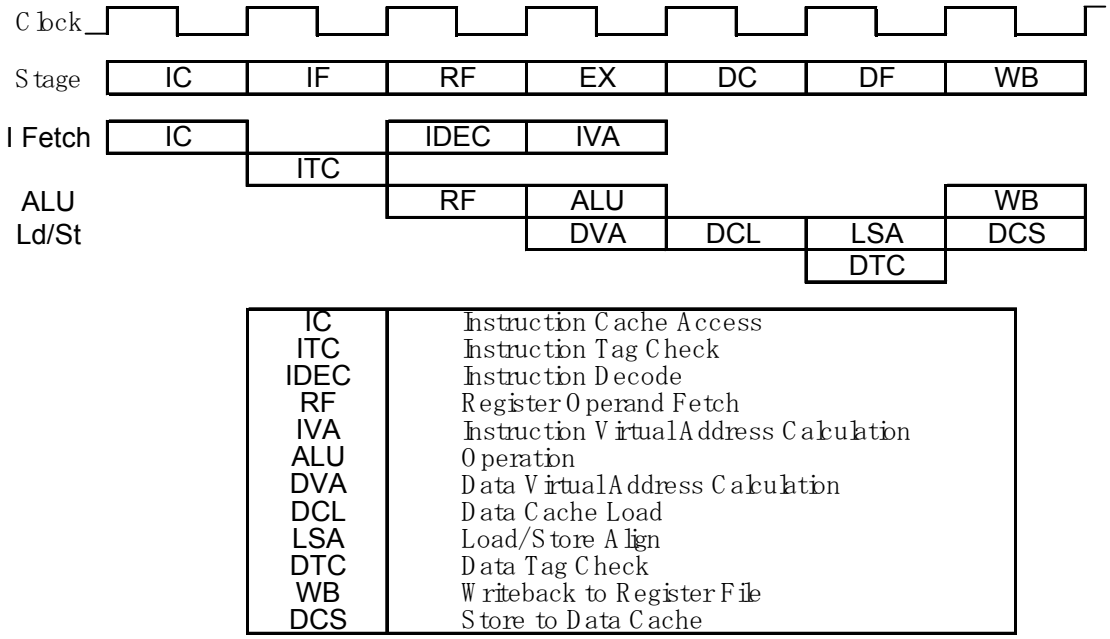


Figure 3-2: Operation of Each ALLEGREX™ Pipeline Stage

3.2. Jump/Branch Delay

ALLEGREX™ does not have a conditional branch prediction unit. As a result, the instruction following the delay slot of a jump or branch instruction always ends up getting fetched, creating a 2-cycle delay for a jump and a 3-cycle delay for a branch.
For a jump instruction (unconditional branch), the jump destination address is calculated in the RF stage, and the jump is performed the next cycle. However, by this time, the instruction immediately following the jump instruction in the delay slot has already been fetched and it is always executed. Figure 3-3 shows this jump delay.

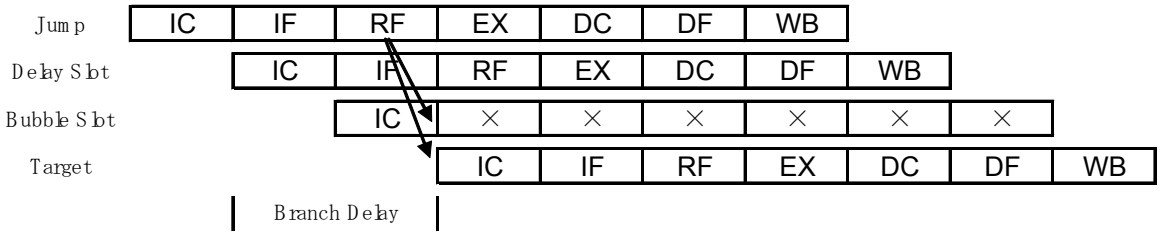
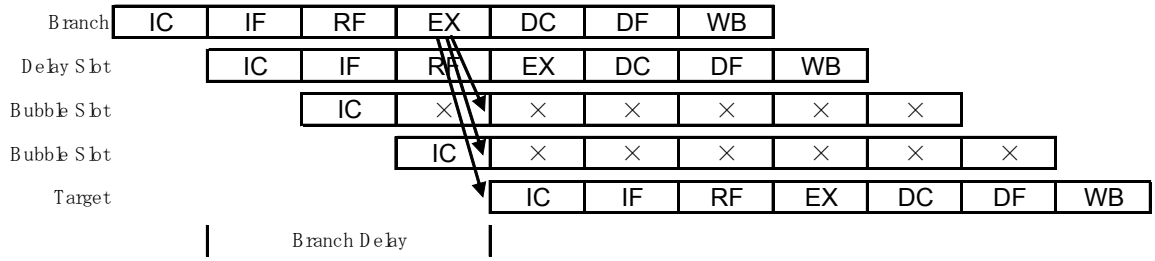


Figure 3-3: Jump Delay (1 Delay Slot + 1 Bubble Slot)

For a branch instruction, the branch condition is checked and the branch destination instruction address is calculated in the EX stage, then the branch is performed the next cycle. By this time, the next three sequential instructions after the branch instruction have already been fetched, and their execution has started. The first of these is the instruction in the branch delay slot and it is always executed. If the branch condition is satisfied and the

branch is taken, the second and third sequential instructions (the instructions immediately after the branch delay slot) are nullified and are not executed. This creates two bubble slots in the pipeline. Although the software sees this as a one-cycle branch delay because of the delay slot, the hardware adds two bubble slots to actually produce a three-cycle branch delay. Figure 3-4 shows this branch delay.



* For software, it will be a one delay slot same as a jump

Figure 3-4: Branch Delay (1 Delay Slot + 2 Bubble Slots)

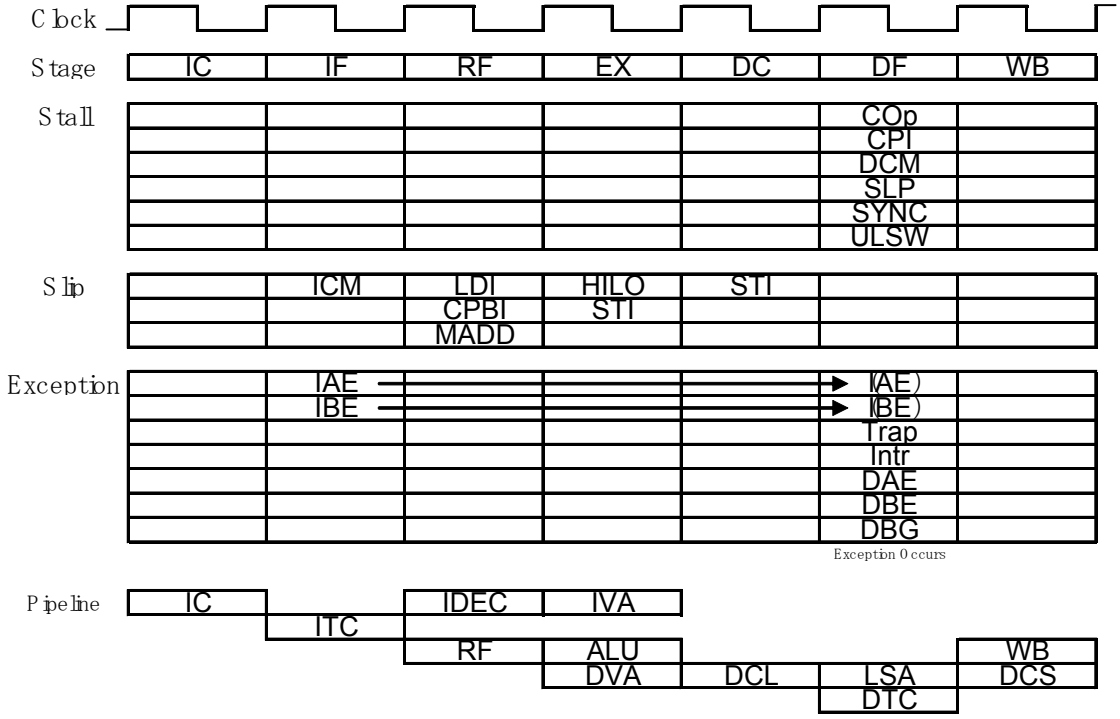
3.3. Interlocks and Exceptions

3.3.1. Detection of Interlocks and Exceptions

Under certain conditions the smooth flow of the pipeline may be interrupted. If some condition like a cache miss occurs in which the hardware is responsible for restoring the pipeline flow, the interruption is known as an interlock. Interlocks are further divided into stalls in which the entire pipeline is stopped while the hardware works to remove the interlocking condition, and sleeps in which only part of the pipeline is stopped and the rest of the pipeline advances.

An interruption in which software performs return processing is called an interrupt.

All interlocks and exception conditions are called faults. Figure 3-5 lists the various faults that can be generated and the pipeline stages when they are checked.



Interlock		Exception	
ICM	Instruction Cache Miss	IAE	Instruction Address Error
LDI	Load Interlock	IBE	Instruction Bus Error
CPBI	Coprocessor Busy	Trap	Synchronous Trap Exception (Sysca
CPI	Coprocessor Interlock		Break, Overflow, Reserved
MADD	Product-Sum Operation Interlock		Instruction Exception, FPU Exception
HILO	HI/LO Register Busy		Coprocessor Unusable Exception)
DCM	Data Cache Miss		
ULSW	Non-Cached Load/Store Weight	Intr	Interrupt
STI	Store Interlock	DAE	Data Address Error
COp	Cache Operation	DBE	Data Bus Error
SYNC	SYNC	DBG	Debug Break
SLP	Sleep		

Figure 3-5: ALLEGREX™ Fault Detection Stages

All exceptions are generated in the DF stage of the pipeline even if they were detected earlier. For example, if an instruction address error exception is detected in the IF stage, the exception is not generated immediately. Instead, the detection result continues to flow on the pipeline along with the execution of the instruction until the instruction enters the DF stage at which point the exception will be generated.

When an exception is generated, the program jumps to the exception vector in the next cycle. At the same time, all instructions currently in the pipeline are nullified and their execution is suppressed. In addition, the address of the instruction in the DF stage that caused the exception is saved in the EPC/ErrorEPC register of CP0. When control returns from the exception, execution is restarted from this instruction.

3.3.2. Pipeline Operation During an Interlock

This section shows pipeline diagrams and timing for the different types of interlocks that can occur in ALLEGREX™.

Load Interlock

A load interlock occurs when a register which is being loaded by a load instruction is referenced immediately by the following instruction. Load interlocks can be alleviated by placing an appropriate instruction between the load instruction and the instruction that uses the loaded data.

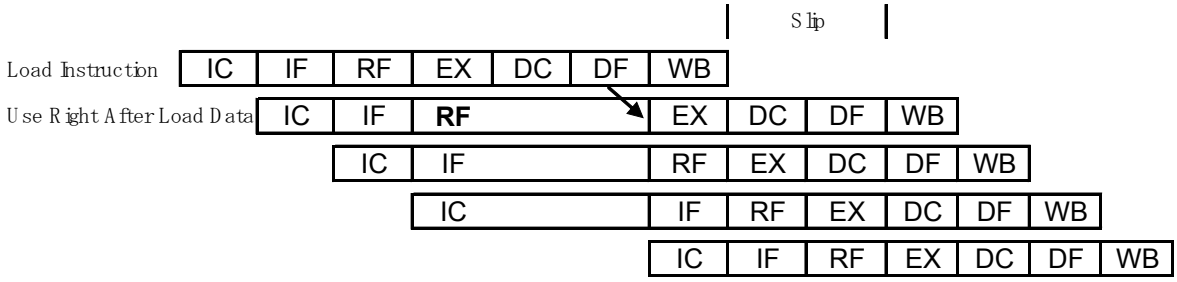


Figure 3-6: Pipeline During Load Interlock (1)

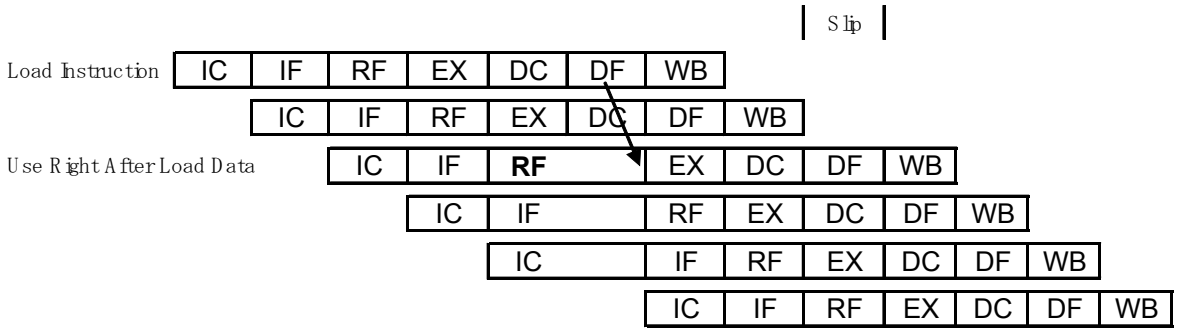


Figure 3-7: Pipeline During Load Interlock (2)

Store Interlock

A store interlock occurs when a load/store instruction is executed immediately after a store instruction. Store interlock can be alleviated by placing an appropriate instruction between the store instruction and the load/store instruction that follows it.

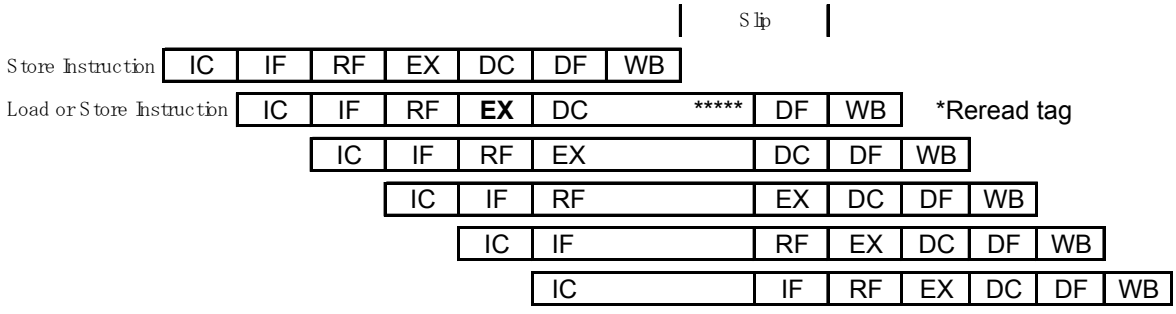


Figure 3-8: Pipeline During Store Interlock (1)

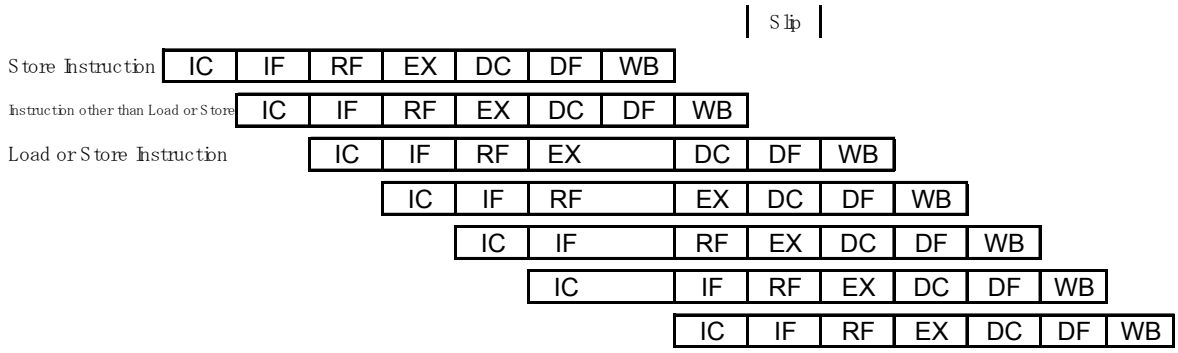


Figure 3-9: Pipeline During Store Interlock (2)

Non-Cached Load/Store Wait

A non-cached load/store wait occurs when a load/store instruction is executed for a non-cached address. However, when free space is available in the non-cached write buffer (16 words), no wait will occur for a store instruction to a non-cached address.

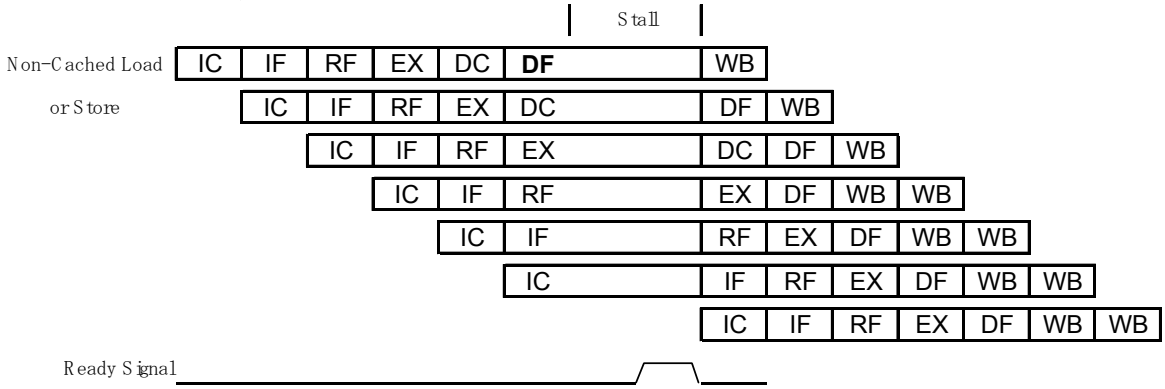


Figure 3-10: Pipeline During a Non-Cached Load/Store Wait

Data Cache Miss Interlock

A data cache miss interlock occurs when a load/store instruction tries to access data at some virtual address but that data is not present in the D-cache. ALLEGREX™ does not perform hit under miss. The data for which the miss occurred is filled starting at the beginning, and the stall is canceled after the cache fill has completed.

Stalls due to data cache misses can be alleviated by prefetching data to the D-cache in advance by using the Fill (D) function of the CACHE instruction, which is described later.

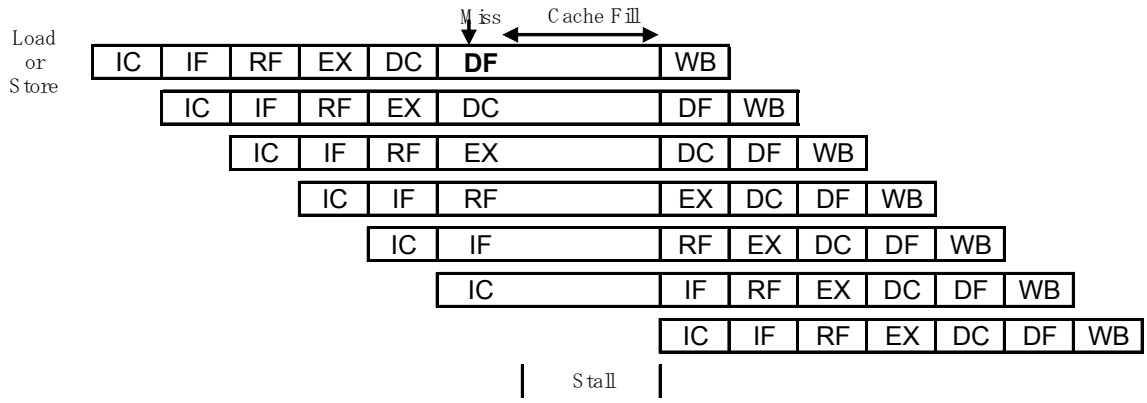


Figure 3-11: Pipeline Operation During a Data Cache Miss

Instruction Cache Miss Interlock

An instruction cache miss interlock occurs when an instruction fetch tries to fetch an instruction at some virtual address but that instruction is not present in the I-cache. ALLEGREX™ sleeps during an instruction cache miss until all replaced lines are filled.

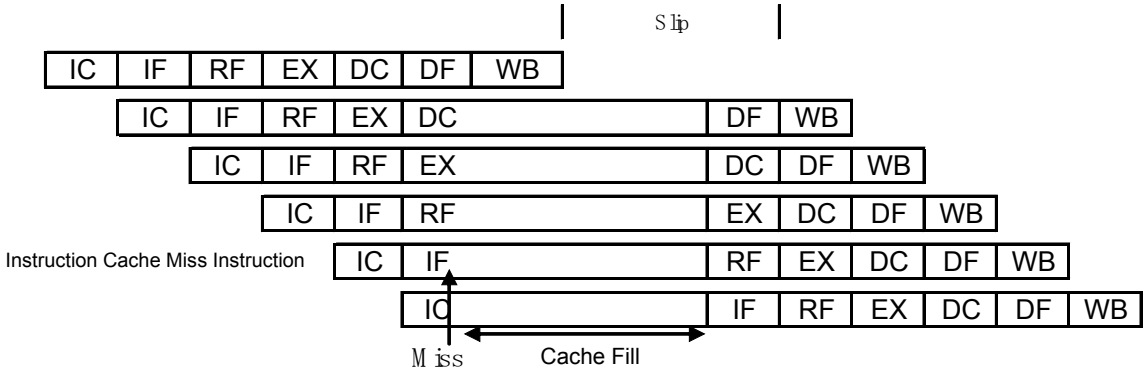


Figure 3-12: Pipeline During an Instruction Cache Miss

Cache Operation Interlock

A cache operation interlock occurs when the CACHE instruction is executed.

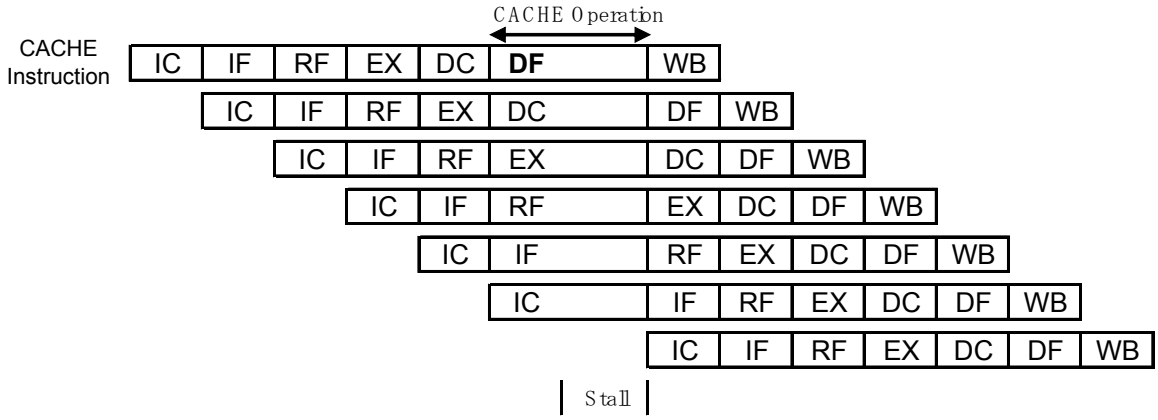


Figure 3-13: Pipeline During a Cache Operation Interlock

Hi/Lo Register Busy Interlock

A Hi/Lo register busy interlock occurs when a MFHI/MFLO instruction, MTHI/MTLO instruction, or a multiply or divide instruction is executed while another multiply, divide, multiply and add, or multiply and subtract instruction is still executing.

Hi/Lo register busy interlocks can be alleviated by scheduling instructions appropriately.

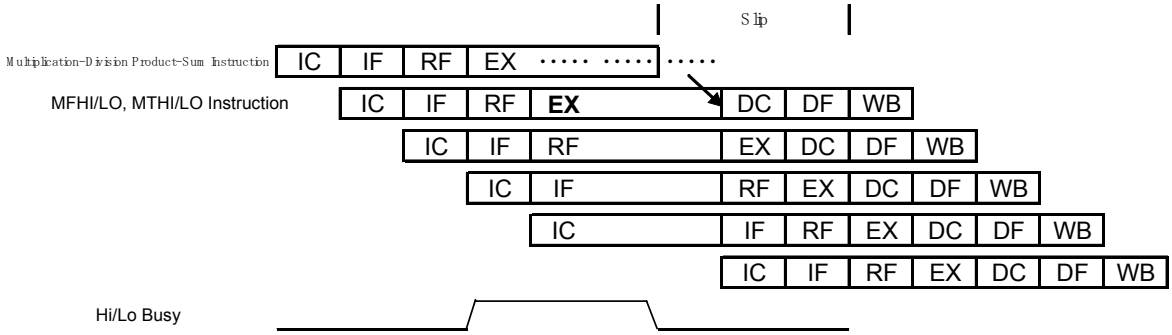


Figure 3-14: Pipeline During a Hi/Lo Register Busy interlock

Multiply and Add Operation Interlock

A multiply and add operation interlock occurs when a multiply and add or a multiply and subtract instruction is executed immediately after another multiply, divide, multiply and add, or multiply and subtract instruction.

Multiply and add operation interlocks can be alleviated by scheduling instructions appropriately.

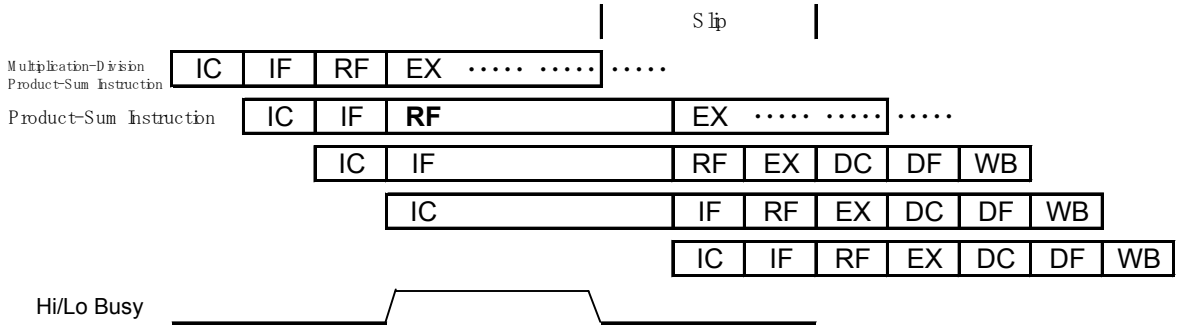


Figure 3-15: Pipeline During a Multiply and Add Interlock

Coprocessor Busy Interlock

A coprocessor busy interlock occurs when the FPU pipeline is interlocked. This interlock can be alleviated by scheduling FPU instructions appropriately.

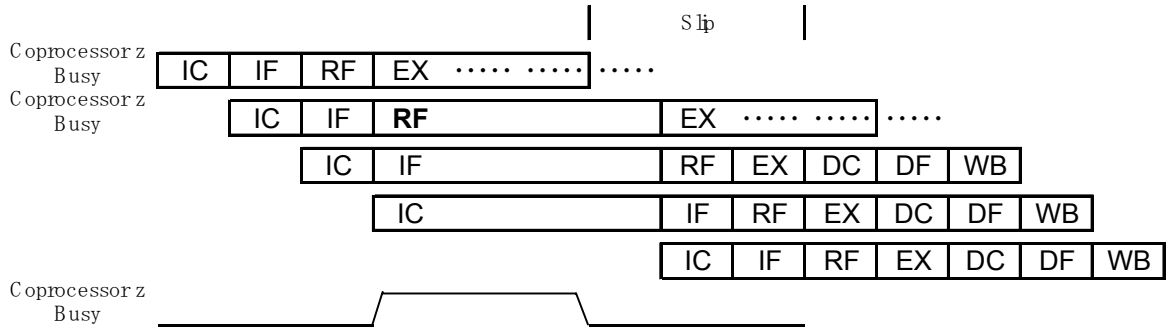


Figure 3-16: Pipeline During a Coprocessor Busy Interlock

Synchronization Interlock

A synchronization interlock occurs when the SYNC instruction is executed. The pipeline stalls until the D-cache writeback buffer or non-cached write buffer have completed write operations.

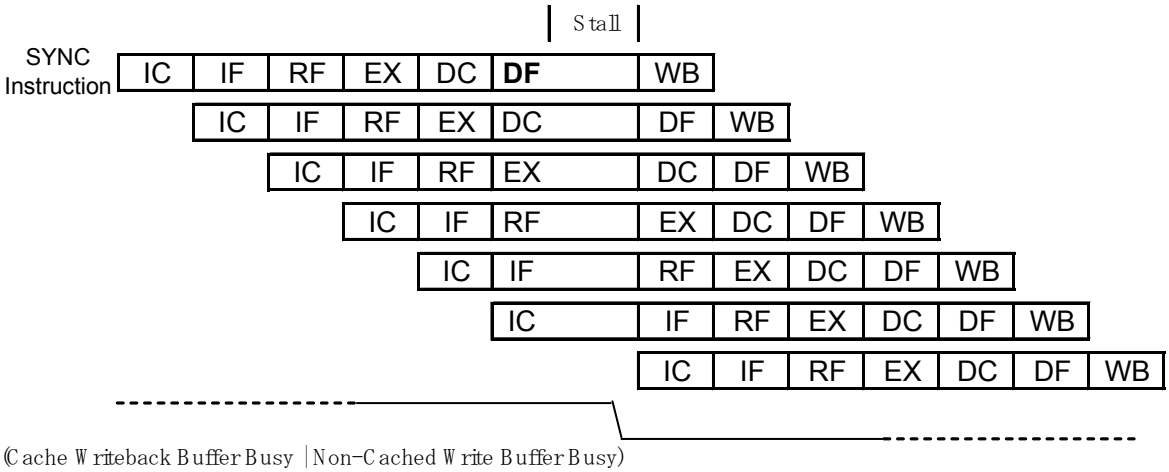


Figure 3-17: Pipeline During a Synchronization Interlock

3.3.3. Pipeline Operation When an Exception Occurs

All ALLEGREX™ exceptions are generated in the DF stage of the pipeline.
When an exception occurs, the address of the instruction in the DF stage is stored in the EPC (or ErrorEPC) register, all instructions that have been fetched and are executing are canceled, all pipeline interlocks are canceled, and the program jumps to the exception vector in the next cycle. In addition, the system control coprocessor (CP0) Status and Cause registers are updated appropriately for the exception.
Figure 3-18 shows the pipeline when an exception occurs.

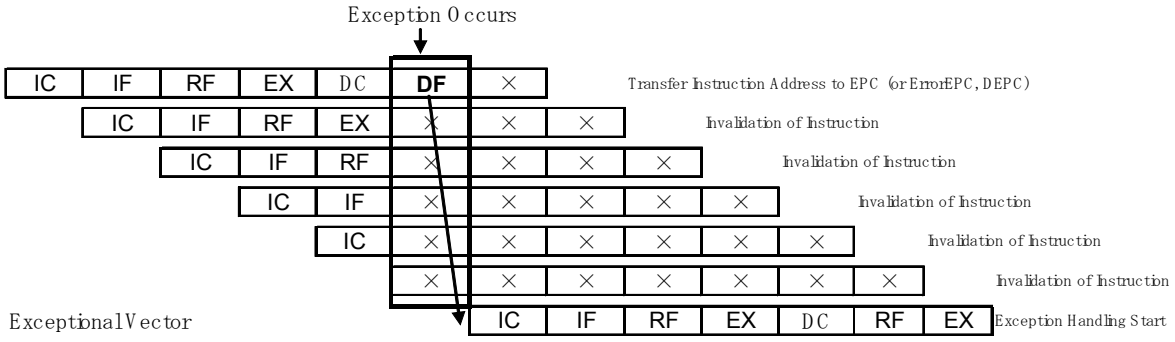


Figure 3-18: Pipeline When an Exception Occurs

Normally, the address that is saved in the EPC (or ErrorEPC) register is the address of the instruction that caused the exception. However, when the instruction that caused the exception is in a branch delay slot, the address of the branch instruction is saved instead. This is because the branch instruction must be re-executed when control returns from the exception. For general exceptions, this case can be identified by checking the BD bit in the Cause register.

3.3.4. Pipeline Operation When Returning From an Exception (when ERET is executed)

When an ERET instruction is decoded in the RF stage, the program jumps to the execution restart address pointed to by the EPC (or ErrorEPC) register in the next cycle, enabling the program to return from the exception. The two instructions immediately following the ERET instruction are nullified and are not executed. Figure 3-19 shows the pipeline when the ERET instruction is executed.

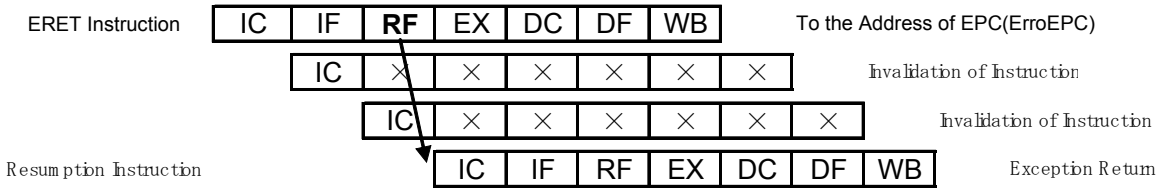


Figure 3-19: Pipeline When Control Returns From an Exception

3.4. Instruction Sequences Requiring nops to be Inserted

To execute the instruction sequences shown in Table 3-1, nops must be inserted between the indicated instructions. For all other instruction sequences, the hardware performs interlock processing so it is not necessary to insert nops.

Table 3-1: Instruction Sequences for Which nops Must be Inserted

Instruction Sequence	Number of nops Required
MTC0 ↓ ERET	2
CTC1 (FPU control register write) ↓ BC1* (FPU branch instruction)	1 (automatically inserted by the compiler)
C.* (FPU condition instruction) ↓ BC1* (FPU branch instruction)	1 (automatically inserted by the compiler)

(*) It is not necessary to insert any nops in a mfhi/mflo→mult/div sequence.

4. Caches

4.1. Cache Configuration

ALLEGREX™ has built-in, independent instruction (I) and data (D) caches. The caches have a data width of 128 bits and are connected to main memory via a 128-bit bus interface, providing high-speed cache fill and writeback processing. The data cache also has a one-line writeback buffer. Figure 4-1 shows the configuration of the cache system.

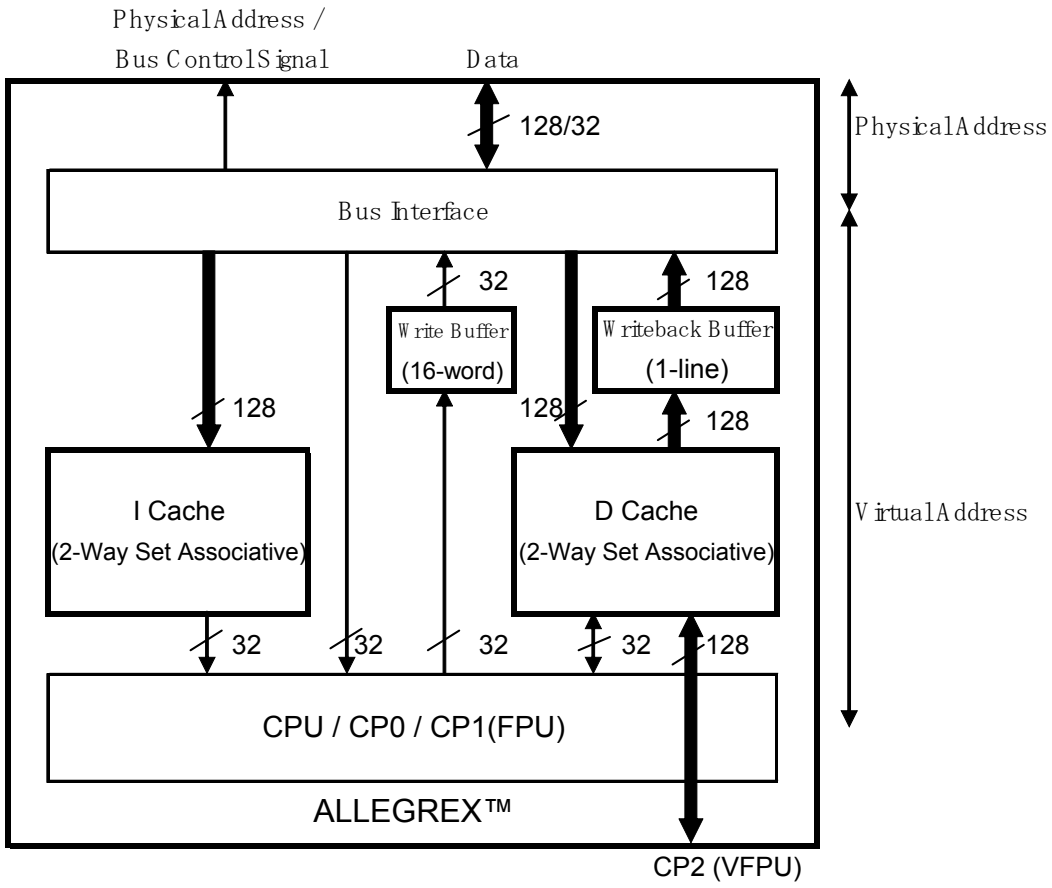


Figure 4-1: ALLEGREX™ Cache System

4.2. Instruction Cache

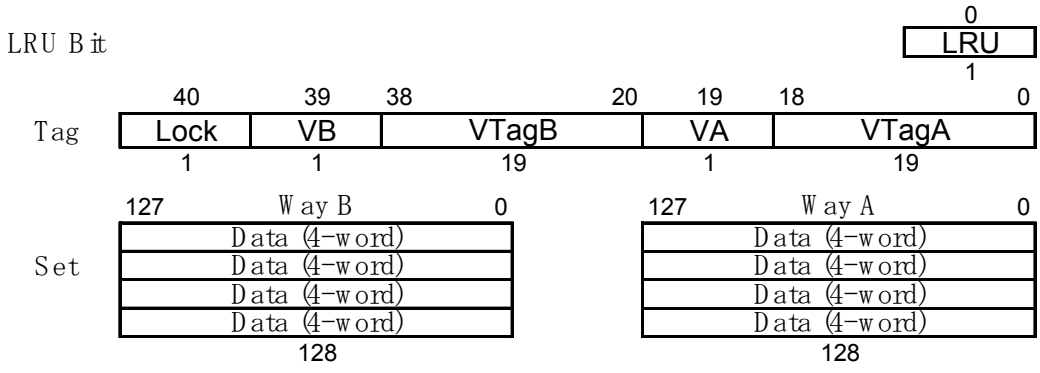
The ALLEGREX™ instruction cache has the following features.

1. Two-way set associative structure
2. Way selection using LRU algorithm
3. Virtual index addresses
4. Tag check using virtual address
5. 16-word (64-byte) block size
6. Locking mechanism
7. Refill from beginning of cache line
8. No snoop mechanism

The size of the instruction cache is 16 KB.

The instruction cache is cleared by hardware when a hardware or software reset occurs.

The instruction cache has a two-way set associative structure. Each set consists of two ways, called WayA and WayB, a tag, and 1 LRU bit. Each way contains a 64-byte data block, which is also known as a cache line. The tag consists of a Lock bit (1 bit), WayA virtual address (19 bits) and Valid bit (1 bit), WayB virtual address (19 bits) and Valid bit (1 bit). Figure 4-2 shows one set in the instruction cache.



VTagA	Virtual Tag of Way A (Bit 31..13 of Virtual Address)
VA	Valid Bit of Way A
VTagB	Virtual Tag of Way B (Bit 31..13 of Virtual Address)
VB	Valid Bit of Way B
Lock	Lock Bit (Holds LRU Bit)

Figure 4-2: Instruction Cache Set

The Valid bit indicates whether the corresponding block is valid. When the Valid bit is 1, the block is valid. When a hardware reset occurs, the tags of all instruction cache lines are marked invalid (the Valid bits are reset to 0).

The LRU bit indicates which way contains the older block (the block least recently

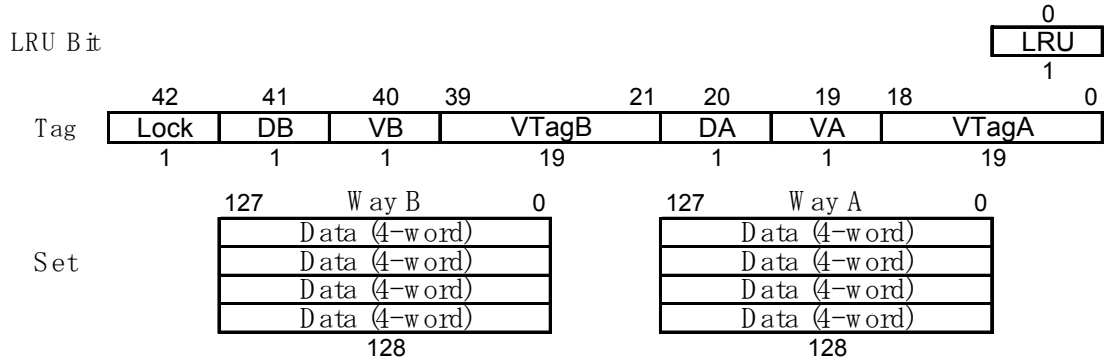
accessed). When a cache index is to be replaced, the LRU bit is used to determine which way to replace. When LRU=0, the block in WayA is replaced. When LRU=1, the block in WayB is replaced. However, if one of the ways is invalid (Valid=0), that way is filled.
By holding the LRU bit constant, the lock bit serves to lock one of the ways so that it is not replaced. When the Lock bit is 0, the LRU bit is updated whenever an instruction is fetched from the cache. The Lock bit is set and reset by using the CACHE instruction, described later.

4.3. Data Cache

The ALLEGREX™ data cache has the following features.

1. Writeback method
2. Two-way set associative structure
3. Way selection using LRU algorithm
4. Virtual index addresses
5. Tag check using virtual address
6. 16-word (64-byte) block size
7. Locking mechanism
8. Four words missed are refilled first on cache miss
9. No snoop mechanism
10. Non-blocking load (prefetch) instruction
11. Non-blocking writeback

The data cache has a two-way set associative structure. Each set consists of two ways, called WayA and WayB, a tag, and 1 LRU bit. Each way contains a 64-byte data block, which is also known as a cache line. The tag consists of a Lock bit (1 bit), WayA virtual address (19 bits), Valid bit (1 bit) and Dirty bit (1 bit), WayB virtual address (19 bits), Valid bit (1 bit) and Dirty bit (1 bit). Figure 4-3 shows one set in the data cache.



VTagA	Virtual Tag of Way A (Bit 31..13 of Virtual Address)
VA	Valid Bit of Way A
DA	Dirty Bit of Way A
VTagB	Virtual Tag of Way B (Bit 31..13 of Virtual Address)
VB	Valid Bit of Way B
DB	Dirty Bit of Way B
Lock	Lock Bit (Holds LRU Bit)

Figure 4-3: Data Cache Set

The ALLEGREX™ data cache is a writeback cache so a store instruction only writes to the cache and does not access main memory directly. Since main memory is only updated when a line is automatically replaced or explicitly written back by the CACHE instruction, main memory accesses are kept to a minimum.

Each way also has a Dirty bit and the data in that way is written back only when the Dirty bit is 1. When the Dirty bit is 0, it means that the data block in that way is the same as the data block in main memory at that address. In other words, when the line is replaced, the block does not need to be written back to main memory. When the data in a block is updated by a store instruction, the Dirty bit is set to 1. When the data is subsequently written back, the Dirty bit is cleared to 0.

The Valid bit indicates whether the corresponding block is valid. When the Valid bit is 1, the block is valid.

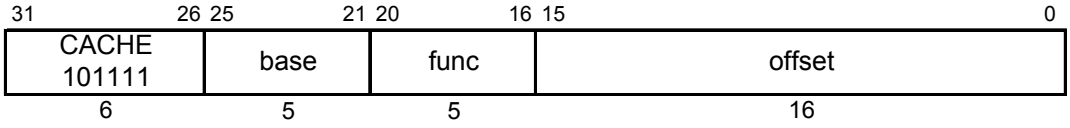
The LRU bit indicates which way contains the older block (the block least recently accessed). When a cache index is to be replaced, the LRU bit is used to determine which way to replace. When LRU=0, the block in WayA is replaced. When LRU=1, the block in WayB is replaced. However, if one of the ways is invalid (Valid=0), that way is filled.

By holding the LRU bit constant, the Lock bit serves to lock one of the ways so that it is not replaced. If the Lock bit is 0, the LRU bit is updated every time data is fetched from the data cache or data is stored to the data cache.The Lock bit is set and reset by using the CACHE instruction, described later. The data cache has a one-line writeback buffer. When a data cache miss occurs and a line is replaced, the old line is moved out to this writeback buffer at the same time that a fill request for the new line is issued to the bus. After the fill

has completed, the old line is written back to main memory from the writeback buffer. This keeps the cache miss penalty that accompanies a writeback to a minimum.

4.4. CACHE Instruction

ALLEGREX™ supports the CACHE instruction for performing explicit operations on the caches. The CACHE instruction opcode is MIPS III ISA compliant, however, the func codes which indicate the cache operation to be performed are unique to ALLEGREX™. Figure 4-4 shows the format of the CACHE instruction. Table 4-1 lists the func codes of the ALLEGREX™ CACHE instruction and their corresponding operations.



Instruction Format: CACHE func, offset(base)

Figure 4-4: CACHE Instruction Format

Table 4-1: CACHE Operations

Func Code	Target	Operation
0	I-cache	-
1	I-cache	-
2	I-cache	-
3	I-cache	-
4	I-cache	Index Invalidate
5	I-cache	-
6	I-cache	Index Unlock
7	I-cache	-
8	I-cache	Hit Invalidate
9	I-cache	-
10	I-cache	Fill
11	I-cache	Fill with Lock
12	I-cache	-
13	I-cache	-
14	I-cache	-
15	I-cache	-
16	D-cache	-
17	D-cache	-
18	D-cache	-
19	D-cache	-
20	D-cache	Index Writeback Invalidate
21	D-cache	-
22	D-cache	Index Unlock

Func Code	Target	Operation
23	D-cache	-
24	D-cache	Create Dirty Exclusive
25	D-cache	Hit Invalidate
26	D-cache	Hit Writeback
27	D-cache	Hit Writeback Invalidate
28	D-cache	Create Dirty Exclusive with Lock
29	D-cache	-
30	D-cache	Fill
31	D-cache	Fill with Lock

4.4.1. CACHE Instruction Details

Index Invalidate (I)

This instruction is used to clear an instruction cache line.

The specified address is used to obtain an index that points to a pair of entries in the cache. The appropriate entry, either WayA or WayB, is selected and invalidated (the Valid bit is cleared to 0) according to the following rules.

1. When both WayA and WayB are valid: Invalidate the oldest entry as determined by the LRU value
2. When either WayA or WayB is valid, but not both: Invalidate the valid entry and clear the Lock bit
3. When both WayA and WayB are invalid: Do nothing

To make sure both WayA and WayB are cleared, issue this instruction twice in a row to the same address (index).

Index Writeback Invalidate (D)

This instruction is used to flush and clear a line in the data cache.

The specified address is used to obtain an index that points to a pair of entries in the cache. The appropriate entry, either WayA or WayB, is selected according to the rules listed below, and if the Dirty bit for the entry is 1, the data block in the entry is written back to main memory and the entry is invalidated (the Valid bit is cleared to 0). If the Dirty bit is 0, the entry is only invalidated and no writeback is performed.

1. When both WayA and WayB are valid: Select the oldest as determined by the value of the LRU bit
2. When either WayA or WayB is valid, but not both: Select the valid entry and clear the Lock bit
3. When both WayA and WayB are invalid: Do nothing

To make sure both WayA and WayB are flushed and cleared, issue this instruction twice in a row to the same address (index).

Hit Invalidate (I/D)

This instruction is used to invalidate a specific line in the instruction or data cache. The tag for the cache index is obtained from the specified address. If the tag indicates that the addressed line is present in the cache (a cache hit), its entry is invalidated (its Valid bit is cleared). If the Lock bit in the tag is 1 and the LRU bit points to the other entry (in other words, the target entry is locked), the Lock bit is cleared (the lock is canceled). If the addressed line is not present in the cache, no operation is performed.

Hit WriteBack (D)

This instruction is used to perform an explicit writeback of a dirty line in the data cache. It is used to explicitly synchronize data in the cache with data in main memory.
The tag for the cache index is obtained from the specified address. If the tag indicates that the addressed line is present in the cache (a cache hit), and if its Dirty bit is 1, a writeback is performed on that cache line and the Dirty bit is 0. cleared. If the addressed line is not present in the cache, no operation is performed.

Hit WriteBack Invalidate (D)

This instruction is used to perform an explicit writeback of a dirty line in the data cache. It is used to explicitly synchronize data in the cache with data in main memory.The cache line is invalidated after it is flushed.
The tag for the cache index is obtained from the specified address. If the tag indicates that the addressed line is present in the cache (a cache hit), and if its Dirty bit is 1, a writeback is performed on that cache line and the line is invalidated. If the Dirty bit is 0, the line will only be invalidated. If the Lock bit in the tag is 1 and the LRU bit points to the other entry (in other words, the target entry is locked), the Lock bit is 0 (the lock is canceled). If the addressed line is not present in the cache, no operation is performed.

Create Dirty Exclusive (D)

This instruction is used to create a dirty line in the data cache. It prevents unnecessary cache fills from being performed on lines that are only going to be written.
An entry in the data cache is created for the line at the specified address, and the Dirty bit in the tag is set to 1. This instruction does not perform a data fill. When the Lock bit is 0 (not locked), the LRU bit is updated to point to the way corresponding to the entry that was not created.
Specifying a non-cache address with the Create Dirty Exclusive (D) instruction will cause indeterminate behavior and is thus prohibited.

Create Dirty Exclusive with Lock (D)

This instruction performs the same operation as Create Dirty Exclusive, except that it also sets the Lock bit to 1. As a result, the LRU bit will be frozen and the corresponding block will be locked, preventing it from being replaced.

Specifying a non-cache address with the Create Dirty Exclusive with Lock (D) instruction will cause indeterminate behavior and is thus prohibited.

Fill (I/D)

This instruction is used to explicitly fill a specific line in the cache, if the line is not already in the cache.

If the cache line at the specified address is not in the cache (a cache miss), the cache line is filled. If the line is already in the cache (a cache hit), no operation is performed. If the Lock bit = 0 (not locked), the LRU bit is updated to point to the way that is not being filled.

During the instruction cache fill, the pipeline is interlocked so no other processing can be performed in parallel. However, with a data cache fill, as long as the following instruction is not a load/store or cache instruction, that instruction can be executed in parallel with the data cache fill (the data cache fill is a non-blocking operation).

Fill with Lock (I/D)

This instruction performs the same operation as Fill, except that it also sets the Lock bit to 1. As a result, the LRU bit will be frozen and the corresponding block will be locked, preventing it from being replaced.

Index Unlock (I/D)

This instruction is used to cancel a lock on a cache line.

The specified address is used to obtain an index that points to a set in the cache. The Lock bit for that set is cleared (the lock on the cache line is canceled).

5. Memory Management System

The ALLEGREX™ processor is equipped with a memory management unit (MMU) that uses direct segment mapping to convert virtual addresses to physical addresses, and provides a memory protection function.

5.1. Operating Modes and Memory Protection

The following three operating modes are provided by the ALLEGREX™ processor.

1. User mode
2. Supervisor mode
3. Kernel mode

The memory space is divided into five segments and segments can be accessed according to mode. This prevents memory that is used by a program running in a higher-level mode from being accessed by a program running in a lower level mode.
Generally, the operating system runs in kernel mode and application programs run in user mode. Supervisor mode, which is an intermediate mode between user mode and kernel mode, provides for a highly secure system.
When an exception occurs, the CPU switches to kernel mode. Later, when an exception return instruction (ERET) is executed, the CPU returns to the mode that was active at the time the exception occurred.

5.1.1. User Mode

As shown in Figure 5-1, a 2 GB (2^{31} byte) virtual address space (kuseg) is always available in user mode. A virtual address is valid in user mode when the most significant bit (MSB) is 0. An address error exception will be generated if an access is made to a virtual address with the MSB set to 1.
In ALLEGREX™, the 2 GB user segment is further divided into a 1 GB segment which uses the cache (kuseg0) and a 1 GB segment which does not use the cache (kuseg1). However, kuseg0 and kuseg1 both reference the same physical memory. In other words, the maximum physical memory space that can be accessed is 1 GB, and whether or not the program uses the cache can be switched by selecting either the kuseg0 or kuseg1 address space when memory is accessed. The CPU is in user mode when system control coprocessor (CP0) Status register bits are KSU=10, EXL=0, and ERL=0.

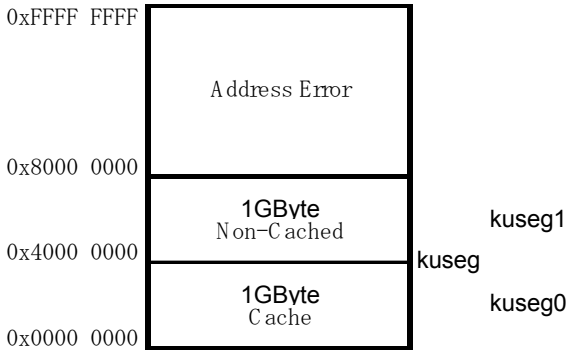


Figure 5-1: User Mode Virtual Address Space

5.1.2. Supervisor Mode

Supervisor mode is provided to create a hierarchy within the operating system. It enables the "true kernel" to run in kernel mode, and the rest of the operating system to run in supervisor mode. The CPU is in supervisor mode when system control coprocessor (CP0) Status register bits KSU=01, EXL=0, and ERL=0.

A program running in supervisor mode can access a 512 MB segment (ksseg) in addition to the user-mode segment, kuseg.

1. kuseg (kuseg0+kuseg1):

kuseg, the 2 GB user-mode address space, can also be accessed in supervisor mode by setting the most significant bit of the virtual address to 0, just like in user mode.

2. ksseg:

ksseg is a 512 MB (2^{29} bytes) supervisor-mode segment which can be accessed by setting the 3 most significant bits of the virtual address to 110. In ALLEGREX™, accessing physical memory with a virtual address in ksseg will always go through the cache.

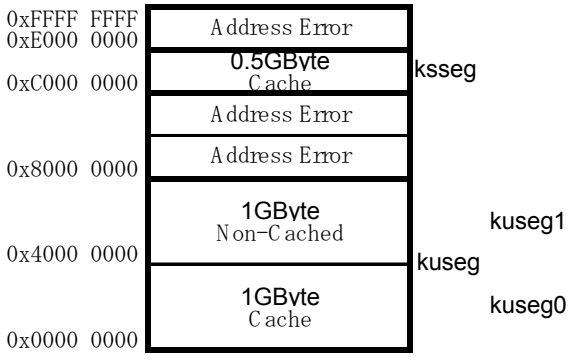


Figure 5-2: Supervisor Mode Virtual Address Space

5.1.3. Kernel Mode

In kernel mode, all segments can be accessed. The CPU is in kernel mode when system control coprocessor (CP0) Status register bits KSU=00, EXL=1, and ERL=1.
Figure 5-3 shows the configuration of kernel mode virtual address space.

1. **kuseg (kuseg0+kuseg1):**

   kuseg, the 2 GB user-mode address space, can also be accessed in kernel mode by setting the most significant bit of the virtual address to 0, just like in user mode and supervisor mode.

2. **kseg0:**

   kseg0 is a 512 MB (2^{29} bytes) kernel-mode segment which can be accessed by setting the 3 most significant bits of the virtual address to 100. In ALLEGREX™, accessing physical memory with a virtual address in kseg0 will always go through the cache.

3. **kseg1:**

   kseg1 is a 512 MB (2^{29} bytes) kernel-mode segment which can be accessed by setting the 3 most significant bits of the virtual address to 101. In ALLEGREX™, using a virtual address in kseg1 will directly access physical memory (or a memory mapped I/O device register) and will not use the cache.

4. **ksseg:**

   ksseg, the 512 MB supervisor-mode segment, can also be referenced in kernel mode by setting the 3 most significant bits of the virtual address to 110, just like in supervisor mode.

5. **kseg3:**

   kseg3 is a 512 MB (2^{29} bytes) kernel-mode segment which can be accessed by setting the 3 most significant bits of the virtual address to 111. In ALLEGREX™, accessing physical memory with a virtual address in kseg3 will always go through the cache.

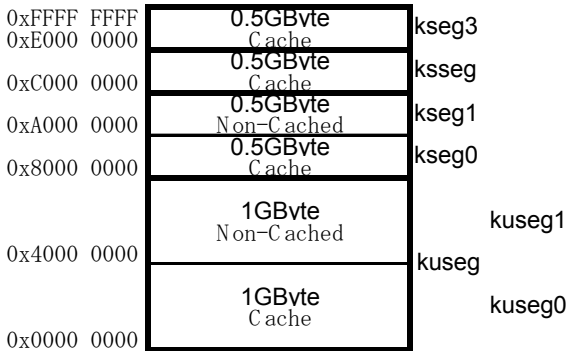


Figure 5-3: Kernel Mode Virtual Address Space

5.2. Converting Virtual Addresses to Physical Addresses

ALLEGREX™ converts virtual addresses to physical addresses using the direct segment mapping method. Direct segment mapping establishes a direct correspondence between virtual addresses and physical addresses.

Figure 5-4 shows how virtual to physical direct segment mapping works in ALLEGREX™.

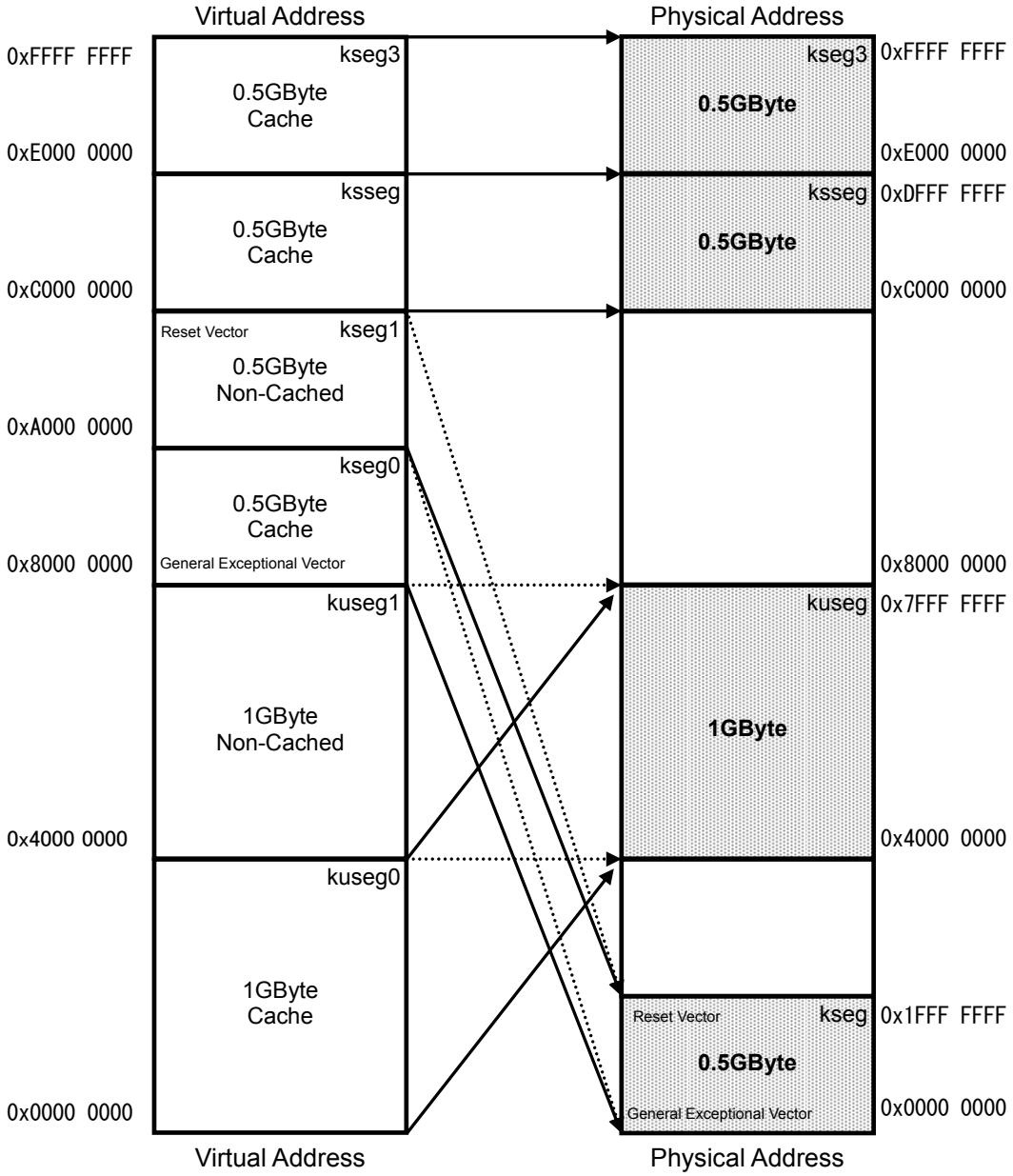


Figure 5-4: ALLEGREX™ Virtual–Physical Direct Segment Mapping

6. CPU Exceptions

This chapter describes the exception handling capabilities of ALLEGREX™ and associated hardware.

ALLEGREX™ supports the same exceptions as those of a MIPS R4000-type processor.

6.1. Exception Handling Overview

ALLEGREX™ supports a number of exception causes such as I/O interrupts, address errors, and system calls.

When an exception occurs, the CPU disables interrupts, transitions to kernel mode, and shifts execution to the exception handler. It also saves the address for restarting execution after exception handling has completed in the EPC register. The address that is saved is the address of the instruction whose execution was interrupted by the exception (note that when the instruction which generated the exception is in the delay slot of a branch instruction, the execution restart address is the address of the branch instruction, not the address of the instruction which caused the exception).

The exception handler saves the processor state, which includes the contents of the program counter which is in a system control coprocessor (CP0) register, the current operating mode, and the interrupt-disabled state. The processor state must be restored after the exception handler has completed, and before control returns to the instruction that was being executed when the exception occurred.

6.2. System Control Coprocessor (CP0) Registers

This section describes the system control coprocessor (CP0) registers. Software can check these registers while handling an exception to identify the cause of the exception and to obtain the CPU state that was active when the exception occurred.

6.2.1. BadVAddr Register

The BadVAddr register is a read-only register that saves the virtual address which caused the last address error exception. The BadVAddr register does not save information when a bus error exception occurs.

Figure 6-1 shows the format of the BadVAddr register.

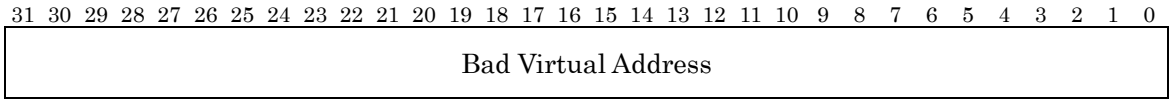


Figure 6-1: BadVaddr Register

6.2.2. Count Register

The Count register is a free-running counter that increments every cycle and is synchronized with the CPU's internal clock, regardless of instruction execution. It is used as a timer by the operating system. The Count register is a read/write register.

Figure 6-2 shows the format of the Count register.

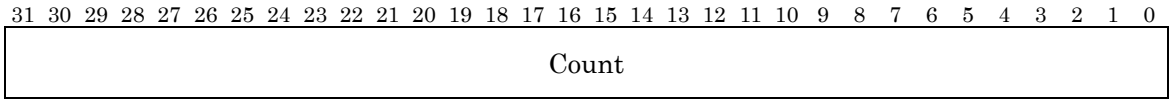


Figure 6-2: Count Register

6.2.3. Compare Register

The Compare register is used together with the Count register to generate internal timer interrupts for the CPU. When the value of the Count register is equal to the value of the Compare register, the IP7 bit of the Cause register is set, and a timer interrupt is generated the next cycle that timer interrupts are enabled. The timer interrupt is cleared when a value is written to the Compare register.

Figure 6-3 shows the format of the Compare register.

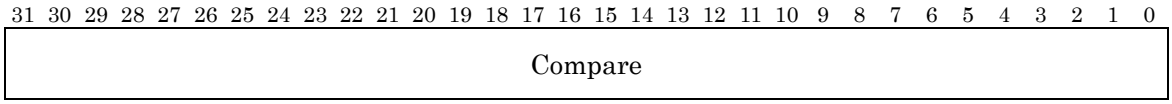


Figure 6-3: Compare Register

6.2.4. Status Register

The Status register is a read/write register that saves the operating mode and the interrupt enabled state.

Figure 6-4 shows the format of the Status register.

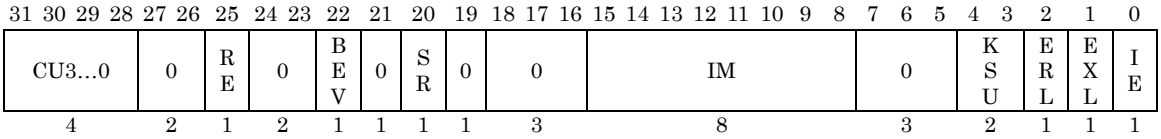


Figure 6-4: Status Register

CU field (Coprocessor Usable)

The CU field of the Status register is a 4-bit field that indicates whether or not each coprocessor is usable. However, in kernel mode, CP0 is always considered to be usable regardless of whether or not the CU0 bit is set.

RE bit (Reverse Endian)

The RE bit of the Status register is an endian reversal bit that is used to reverse the endian in user mode.

With the ALLEGREX™ implementation on the PSP™ system chip, setting RE to 0 selects little-endian in user mode, and setting RE to 1 selects big-endian in user mode. In kernel and supervisor modes, ALLEGREX™ operates in little-endian regardless of the setting of the RE bit.

BEV bit (Bootstrap Exception Vector)

The BEV bit of the Status register specifies the position of the exception vector. It is normally set to 0, and during bootstrapping it is set to 1. It is also set to 1 when a hardware reset occurs.

SR bit (Software Reset)

The SR bit of the Status register is used to distinguish between a hardware and software reset. When a hardware reset is performed, the SR bit is cleared to 0. When a software reset or non-maskable interrupt (NMI) is generated, it is set to 1.

IM field (Interrupt Mask)

The IM field of the Status register is an 8-bit field for controlling the enabling and disabling of 8 interrupts. When an interrupt occurs, if the bit in the IM field corresponding to that interrupt is 1 and the corresponding bit in the IP field of the Cause register is also 1, an interrupt exception will be generated. See also the description of the IP field in the Cause register.

KSU field (Kernel/Supervisor/User)

The KSU field of the Status register is a 2-bit field that indicates the operating mode of the processor when it is in a normal state (EXL=0 and ERL=0).

When the KSU field is 10, it indicates user mode, when it is 01, it indicates supervisor

mode, and when it is 00, it indicates kernel mode.

ERL bit (Error Level)

The ERL bit of the Status register indicates the error state. When this bit is 1, it means that the CPU is in an error state (hardware reset, software reset, or NMI generated). The operating mode at this time is kernel mode regardless of the setting of the KSU bit, and interrupts are disabled regardless of the setting of the IE bit.

EXL bit (Exception Level)

The EXL bit of the Status register indicates the exception state. When this bit is 1, it means that the CPU is in an exception state (state that occurs when a general exception is generated). The operating mode at this time is kernel mode regardless of the setting of the KSU bit, and interrupts are disabled regardless of the setting of the IE bit.

IE bit (Interrupt Enable)

The IE bit of the Status register indicates that interrupts are enabled. When the CPU is in normal mode (ERL=0 and EXL=0), all eight interrupts are collectively enabled (IE=1) or disabled (IE=0).

The operating mode and interrupt enabled state are determined by the combination of the KSU, ERL, EXL, and IE bits of the Status register. Figure 6-5 shows the possible combinations of these bits.

KSU	ERL	EXL	IE	Operating Mode	Interrupts
10	0	0	0	User mode	Disabled
10	0	0	1	User mode	Enabled
01	0	0	0	Supervisor mode	Disabled
01	0	0	1	Supervisor mode	Enabled
00	0	0	0	Kernel mode	Disabled
00	0	0	1	Kernel mode	Enabled
any	0	1	*any*	Kernel mode (general exception)	Disabled
any	1	*any*	*any*	Kernel mode (error exception)	Disabled

Figure 6-5: Operating Modes and Interrupt Enabled State

6.2.5. Cause Register

The Cause register saves the cause of the last exception that occurred. The Cause register consists of read/write and read-only bits.

The 5 bits of the ExcCode field indicate the cause of the exception (see Table 6-2).

Among the 8 IP bits, IP[1:0] are read/write bits that are used as software interrupts. The other bits are read-only bits.

Figure 6-6 shows the format of the Cause register.

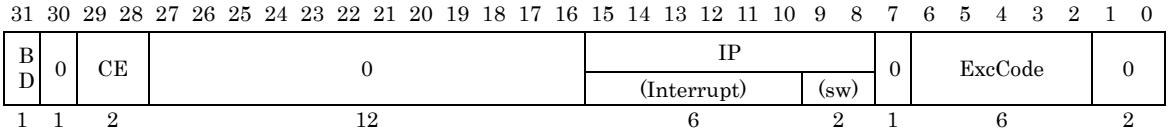


Figure 6-6: Cause Register

BD bit (R) (Branch Delay slot)

The BD bit of the Cause register is set to 1 when the last general exception to occur was generated by an instruction located in a branch delay slot. An error exception will not change the BD bit.

CE bit (R) (Coprocessor Error)

The CE bit of the Cause register indicates the number of the coprocessor that caused a coprocessor unusable exception to occur.

IP field (Interrupt Pending)

The IP field of the Cause register contains bit flags indicating the type of interrupt that occurred. Details are as follows.

Table 6-1: Interrupt Types (Cause.IP)

Bit	R/W	Interrupt
IP[7]	R/-	Internal timer interrupt when Count register = Compare register
IP[6]	R/-	External interrupt[4]
IP[5]	R/-	External interrupt [3]
IP[4]	R/-	External interrupt [2]
IP[3]	R/-	External interrupt [1]
IP[2]	R/-	External interrupt [0]
IP[1]	R/W	Software interrupt[1]
IP[0]	R/W	Software interrupt [0]

IP[6:2] indicate port values [4:0] latched each cycle among the six external interrupts [5:0]. When the external interrupts occur, the bits will be set to 1.
IP[1:0] are read/write bits and when written to 1, enable software interrupts to be generated.

ExcCode field (R)

The ExcCode field of the Cause register indicates the exception code that was generated. Table 6-2 lists the exception codes.

Table 6-2: Exception Codes (Cause.ExcCode)

ExcCode	Mnemonic	Meaning
0	Int	Interrupt
1-3	---	Reserved
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (instruction fetch)
7	DBE	Bus error exception (load)
8	Sys	System call exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor unusable exception
12	Ov	Integer arithmetic overflow exception
13-14	---	Reserved
15	FPE	FPU exception
16-31	---	Reserved

6.2.6. EPC Register

The EPC register is a read/write register that indicates the program restart address that will be used after general exception processing has completed. When the ERET instruction is executed and control returns from a general exception (ERL=0, EXL=1), the CPU loads the program counter with the address indicated by the EPC register.

When a general exception occurs, one of the following addresses is stored in the EPC register.

1. Virtual address of the instruction whose execution was interrupted by the general exception
2. Virtual address of the branch/jump instruction immediately preceding the instruction whose execution was interrupted by the general exception (when the interrupted instruction is in the delay slot of a branch/jump instruction and the BD bit of the Cause register is 1)

Figure 6-7 shows the format of the EPC register.

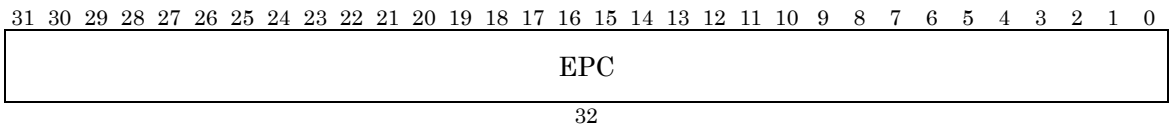


Figure 6-7: EPC Register

6.2.7. EBase Register

The EBase register is a read/write register that stores the virtual address of the exception vector address used as the exception handler for general exceptions, when the BEV bit of

the Status register is 0.

Figure 6-8 shows the format of the EBase register.

For information about determining the exception vector address, see also Table 6-3.

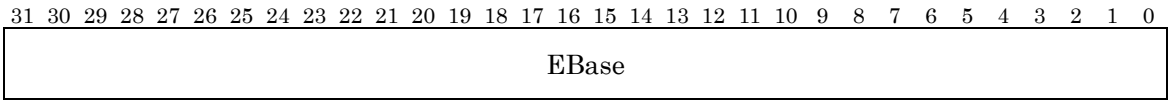


Figure 6-8: EBase Register

6.2.8. ErrorEPC Register

The ErrorEPC register is a 32-bit read/write register that indicates the program restart address that will be used after error exception processing (hardware reset, software reset, or NMI exception) has completed. When the ERET instruction is executed and control returns from an error exception (ERL=1), the CPU loads the program counter with the address indicated by the ErrorEPC register.

When an error exception occurs, one of the following addresses is stored in the ErrorEPC register.

1. Virtual address of the instruction whose execution was interrupted by the error exception
2. Virtual address of the branch/jump instruction immediately preceding the instruction whose execution was interrupted by the error exception

Figure 6-9 shows the format of the ErrorEPC register.

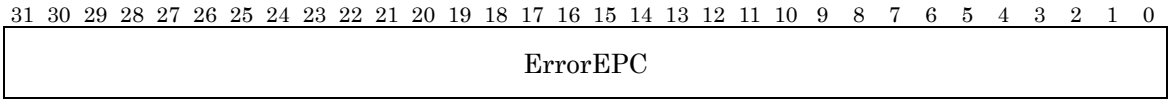


Figure 6-9: ErrorEPC Register

6.2.9. PRId Register

The PRId register stores a value that identifies the version of the ALLEGREX™ Core.

Figure 6-10 shows the format of the PRId register.

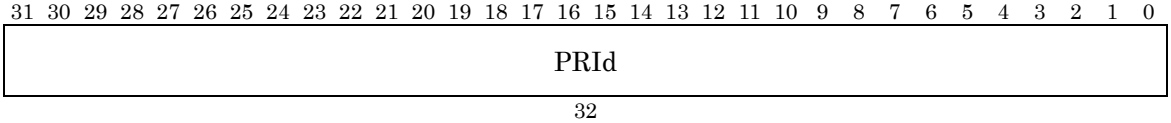


Figure 6-10: PRId Register

6.2.10. Config Register

The Config register is a read-only register for saving various hardware settings of the ALLEGREX™ Core.

Figure 6-11 shows the format of the Config register.

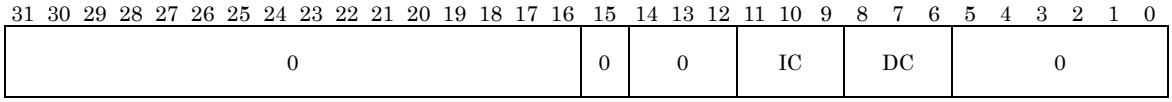


Figure 6-11: Config Register

IC field (R)

The IC field of the Config register indicates the size of the instruction cache. The size of the instruction cache is given by 2^{12+IC} bytes.

DC field (R)

The DC field of the Config register indicates the size of the data cache. The size of the data cache is given by 2^{12+DC} bytes.

6.2.11. TagLo Register

The TagLo register is used for cache control.

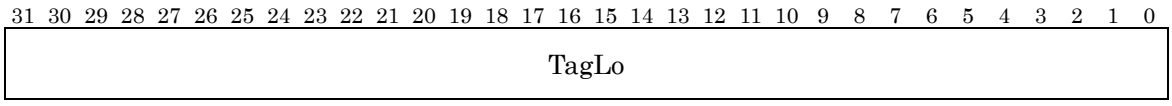


Figure 6-12: TagLo Register

6.2.12. TagHi Register

The TagHi register is used for cache control.

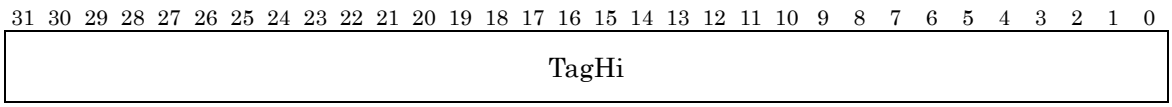


Figure 6-13: TagHi Register

6.3. Exception Operation

6.3.1. Operation When an Exception Occurs

When an exception occurs, the CPU disables interrupts, switches to kernel mode, and starts execution of the exception handler at a predetermined address. When control returns from the exception handler, the program counter (PC) and operating mode are restored and interrupts are enabled. Therefore, the operating system must save this state information when an exception occurs.

In addition to the above, when an exception occurs, the address of the instruction whose execution was interrupted by the exception is saved in the EPC register so it can be used as the address for restarting execution after exception processing has completed (if the interrupted instruction was in the delay slot of a branch instruction, the address of the branch instruction is saved).

The processing performed by the ALLEGREX™ hardware when an exception occurs is described below. The EPC register, ErrorEPC register, and Status register are updated as shown.

Hardware Reset Exception (asynchronous)

ErrorEPC ← PC (saves the address of the execution restart instruction for use when returning from the exception)
PC ← 0xBFC00000 (reset exception vector)
Status.ERL ← 1 (error state)
Status.SR ← 0 (indicates that it is not a software reset or NMI)
Status.BEV ← 1
Bus and cache are reset

Software Reset Exception (synchronous with CPU clock)

ErrorEPC ← PC (saves the address of the execution restart instruction for use when returning from the exception)
PC ← 0xBFC00000 (reset exception vector)
Status.ERL ← 1 (Error state)
Status.SR ← 1 (indicates that it is a software reset)
Status.BEV ← 1
Bus and cache are reset

NMI Exception (synchronous with instruction execution)

ErrorEPC ← PC (saves the address of the execution restart instruction for use when returning from the exception)
PC ← 0xBFC00000 (reset exception vector)
Status.ERL ← 1 (error state)
Status.SR ← 1 (indicates that it is an NMI)
Status.BEV ← 1

General Exception (synchronous with instruction execution)

EPC ← PC (saves the address of the execution restart instruction for use when returning from the exception)
PC ← EBase (when Status.BEV=0 or 0xBFC00200 (when Status.BEV=1)
Status.EXL ← 1 (general exception state)
Cause.BD ← Whether or not the instruction that generated the exception was in the branch delay slot
Cause.ExcCode ← Type of exception that occurred

After the exception handler receives the above information that is provided by the hardware, it should save appropriate information such as the value of the Status register. After that it should return to normal state by clearing the exception state (Status.EXL=0) and clearing the error state (Status.ERL=0). It can then begin handling the exception in the operating mode indicated by Status.KSU. If the Status.IE bit is also set to 1, it can enable multiple interrupts.
After it performs the required processing, the exception handler should restore the information that it saved in advance, return to the general exception state (Status.EXL=1) or error state (Status.ERL=1), and execute the ERET instruction to return from the exception.

6.3.2. Exception Return Instruction (ERET)

The ERET instruction causes a return from an exception by updating the PC and Status registers as shown below depending on the value of the ERL bit of the Status register.

```
if (Status.ERL=1) then
        PC          ← ErrorEPC
        Status.ERL  ← 0
        LLbit       ← 0
else
        PC          ← EPC
        Status.EXL  ← 0
        LLbit       ← 0
endif
```

Although the ERET instruction causes the program to jump to the execution restart address by updating the value of the PC, it differs from a branch or jump instruction in that the instruction immediately following ERET is not executed (the ERET instruction does not have a delay slot).
Also, note that the ERET instruction itself must not be executed in a branch delay slot.
If you use the MTC0 instruction to change the value of the EPC or ErrorEPC register

immediately before executing an ERET, you must insert at least two unrelated instructions (such as 2 nops) between the MTC0 and the ERET instruction.

6.3.3. Exception Vector Address

The exception vector address for hardware/software reset and NMI is always 0xBFC00000 in the kseg1 segment. The address of the general exception vector depends on the value of the BEV bit in the Status register. When BEV=1, the general exception vector is located at 0xBFC00200 in the kseg1 segment. When BEV=0, the address of the general exception vector is the virtual address contained in the EBase register.
Table 6-3 lists the exception vector addresses.

Table 6-3: Determining the Exception Vector Address

Cause	BEV	Virtual Address	Physical Address
Hardware reset, software reset, NMI	Ignored	0xBFC00000 (kseg1: Non-cached)	0x1FC00000
General exception	BEV=0	EBase	EBase converted to a physical address
	BEV=1	0xBFC00200 (kseg1: Non-cached)	0x1FC00200

6.3.4. Exception Priorities

When more than one exception occurs during the same instruction cycle, only one exception is selected as the exception cause. Which exception is selected depends on the priority of the exception, as shown below in Table 6-4.

Table 6-4: Exception Priorities

Priority		Exception Type	Exception Cause
High	A	Hardware reset	Asynchronous XRST signal
↓	B	Software reset	Synchronous XSRST signal
↓	C	NMI	Synchronous XNMI signal
↓	D	Address error	Instruction fetch
↓	E	Bus error	Instruction fetch
↓	F	Integer arithmetic overflow	Instruction execution
↓		System call	Instruction execution
↓		FPU	Instruction execution
↓		Breakpoint	Instruction execution
↓		Reserved instruction	Instruction execution
↓		Coprocessor unusable	Instruction execution
↓		Address error	Data access
↓		Bus error	Data access
Low	G	Interrupt	Latch each instruction cycle

Note that the eight interrupts are collectively handled as a single interrupt exception. When the interrupt exception is selected, multiple interrupts may have been generated. Which interrupts were generated can be determined by checking the IP field of the Cause register.

6.4. Exception Details

For each exception, this section lists the conditions which cause the exception to be generated, operations performed by the hardware, and general information about how the exception handler should respond to the exception.

6.4.1. Hardware Reset Exception

Cause

A hardware reset (cold reset) exception is generated when the hardware reset signal XRST transitions from a low to a high signal value when the CPU clock goes active. This exception is non-maskable.

Hardware Operation

When a hardware reset exception occurs, the CP0 registers are initialized as follows.

- Status.BEV ← 1
- Status.SR ← 0
- Status.ERL ← 1

All instructions being executed and all bus operations are unconditionally aborted, and all instruction cache lines are initialized. Since cache and bus operations are aborted, the contents of cache and memory are indeterminate.
After the above processing, the program jumps to the reset exception vector (0xBFC00000).

6.4.2. Software Reset Exception

Cause

A software reset exception is generated when the software reset signal XSRST transitions from a low to a high signal value when the CPU clock goes active. This exception is non-maskable.

Hardware Operation

When a software reset exception occurs, the CP0 registers are set as follows.

- ErrorEPC ← Execution restart address
- Status.BEV ← 1
- Status.SR ← 1
- Status.ERL ← 1

All instructions being executed and all bus operations are unconditionally aborted, and all instruction cache lines are initialized. Since cache and bus operations are aborted, the contents of cache and memory are indeterminate.

After the above processing, the program jumps to the reset exception vector (0xBFC00000).

6.4.3. Non-Maskable Interrupt (NMI) Exception

Cause

A non-maskable interrupt (NMI) is generated when a falling signal is input to the XNMI port. An NMI exception differs from a general interrupt exception in that it is non-maskable and will be accepted regardless of the state of the EXL, ERL, and IE bits of the Status register. An NMI exception can also be generated while other exceptions are being processed. As a result, a program generally cannot be restarted after NMI exception processing has completed.

The PSP™ system chip uses an NMI to report bus errors during writes that cannot be captured by the bus error exception, or for reporting when a bus access to a protected address has occurred.

Hardware Operation

When an NMI exception occurs, the CP0 registers are set as follows.

- ErrorEPC ← Execution restart address
- Status.BEV ← 1
- Status.SR ← 1
- Status.ERL ← 1

The contents of other registers are saved. The state and contents of the cache and memory are also saved.

After the above processing, the program jumps to the reset exception vector (0xBFC00000).

Note

Unlike a hardware or software reset, the NMI is an interrupt so it can only be generated between instructions, as is also true of other exceptions. Therefore, if an NMI signal is

input while the pipeline is stalled, the NMI will be deferred and the exception will not be generated. Instead, the exception will occur the instant that the stall is canceled.

6.4.4. Address Error Exception

Cause

An address error exception is generated when the following alignment or protected address violations occur.

- When attempting to load or store a word at an address that is not on a word boundary (the low-order 2 bits of the address are not 00)
- When attempting to fetch an instruction from an address that is not on a word boundary (the low-order 2 bits of the address are not 00)
- When attempting to load or store a halfword at an address that is not on a halfword boundary (the least significant bit of the address is not 0)
- When attempting to load or store a quadword at an address that is not on a quadword boundary (the low-order 4 bits of the address are not 0000)
- When attempting to reference kernel address space from user mode
- When attempting to reference supervisor address space from user mode
- When attempting to reference kernel address space from supervisor mode

This exception is non-maskable.

Hardware Operation

When an address error exception occurs, the CP0 registers are set as follows.

- Status.EXL ← 1
- Cause.ExcCode ← AdEL (for an instruction fetch or load instruction) or AdES (for a store instruction)
- EPC ← Execution restart address
- Cause.BD ← 1 when the exception was generated by an instruction in the branch delay slot
- BadVAddr ← Invalid virtual address that was the cause of the exception

After the above processing, the program jumps to the general exception vector.

Software Response

The kernel sends a segment violation signal to the process that is currently executing. Normally, it is impossible to return from this error.

6.4.5. Bus Error Exception

Cause

A bus error exception is generated by an external signal when the specified physical memory address or access type was invalid. This interrupt is non-maskable. ALLEGREX™ will only generate a bus error exception during synchronous reads from the bus such as during cache refills or loads from a non-cached area. Since bus writes are done via the cache writeback buffer or non-cached write buffer, they are non-blocking so the bus error exception isn't used even if a bus error occurs during a write.

The PSP™ system chip uses the NMI exception to report errors during a write operation.

Hardware Operation

When a bus error exception occurs, the CP0 registers are set as follows.

- Status.EXL ← 1
- Cause.ExcCode ← IBE (when the exception was caused by an instruction fetch) or DBE (when the exception was caused by a load instruction)
- EPC ← Execution restart address
- Cause.BD ← 1 when the exception was generated by an instruction in the branch delay slot

After the above processing, the program jumps to the general exception vector.

Software Response

The physical address that caused the bus error can be determined from information in the CP0 registers.

- If IBE is set for Cause.ExcCode (the exception was caused by an instruction fetch), the virtual address of the instruction for which the fetch was attempted is stored in the EPC register.
- If DBE is set for Cause.ExcCode (the exception was caused by a load instruction), the address of the load instruction that caused the exception is stored in the EPC register (however, when the BD bit of the Cause register is set, the address of the load instruction that caused the exception can be found by adding 4 to the value stored in the EPC register).

The virtual address for which the bus error occurred is obtained by decoding the load instruction. The kernel sends a bus error signal to the process that is currently executing. Normally, it is impossible to return from this error and the operating system

cannot re-execute the instruction which caused the exception.

6.4.6. Integer Overflow Exception

Cause

An integer overflow exception is generated when a two's complement overflow occurs during an ADD, ADDI, or SUB instruction (it is not generated by the ADDU, ADDIU, or SUBU instructions). This exception is non-maskable.

Hardware Operation

When an integer overflow exception occurs, the CP0 registers are set as follows.

- Status.EXL ← 1
- Cause.ExcCode ← Ov
- EPC ← Execution restart address
- Cause.BD ← 1 when the exception was generated by an instruction in the branch delay slot

After the above processing, the program jumps to the general exception vector.

Software Response

The kernel sends an integer overflow signal to the process that is currently executing. Normally, it is impossible to return from this error and the operating system cannot re-execute the instruction which caused the exception.

6.4.7. System Call Exception

Cause

A system call exception is generated when a SYSCALL instruction is executed. This exception is non-maskable.

Hardware Operation

When a system call exception occurs, the CP0 registers are set as follows.

- Status.EXL ← 1
- Cause.ExcCode ← Sys
- EPC ← Execution restart address
- Cause.BD ← 1 when the exception was generated by an instruction in the branch delay slot

After the above processing, the program jumps to the general exception vector.

Software Response

The SYSCALL instruction is used as a system call to the operating system. When a system call exception is generated, the operating system switches control to the appropriate system routine for handling the call.

When control returns from a system call exception, if no special action is taken by the software , the SYSCALL instruction will end up getting executed again since the execution restart address is pointing to the SYSCALL instruction. Control can be returned to the instruction following the SYSCALL instruction by adding 4 to the value of the EPC register. However, when the SYSCALL instruction is in a branch delay slot, more complex processing is required.

6.4.8. Breakpoint Exception

Cause

A breakpoint exception is generated when a BREAK instruction is executed. This exception is non-maskable.

Hardware Operation

When a breakpoint exception occurs, the CP0 registers are set as follows.

- Status.EXL ← 1
- Cause.ExcCode ← Bp
- EPC ← Execution restart address
- Cause.BD ← 1 when the exception was generated by an instruction in the branch delay slot

After the above processing, the program jumps to the general exception vector.

Software Response

When a breakpoint exception occurs, control is switched to the appropriate system routine for handling the breakpoint. Additional information can be passed to the system routine by using the unused bits [25:6] of the BREAK instruction opcode. The system routine can obtain this information by reading the data at the address pointed to by the EPC register. (Note that when the BD bit of the Cause register is set, it is necessary to add 4 to the contents of the EPC register to obtain the address of the BREAK instruction.)

When control returns from a breakpoint exception, if no special action is taken by the software, the BREAK instruction will end up getting executed again since the execution restart address is pointing to the BREAK instruction. Control can be returned to the

instruction following the BREAK instruction by adding 4 to the value of the EPC register.

6.4.9. Reserved Instruction Exception

Cause

A reserved instruction exception is generated if an instruction opcode that is not defined by ALLEGREX™ is executed. This exception is non-maskable.

Hardware Operation

When a reserved instruction exception occurs, the CP0 registers are set as follows.

- Status.EXL ← 1
- Cause.ExcCode ← Ri
- EPC ← Execution restart address
- Cause.BD ← 1 when the exception was generated by an instruction in the branch delay slot

After the above processing, the program jumps to the general exception vector.

Software Response

An invalid instruction/reserved operand fault signal is sent to the process that caused this exception. Normally, it is impossible to return from this error.

6.4.10. FPU Exception

Cause

An FPU exception is generated when a floating-point operation exception occurs. FPU exceptions are further classified into the following six types according to the contents of the FPU's Control/Status register (FCR31).

- Inexact exception
- Invalid Operation exception
- Division by Zero exception
- Overflow exception
- Underflow exception
- Unimplemented Instruction exception

All of these exceptions are maskable except for the Unimplemented Instruction.

Hardware Operation

When an FPU exception occurs, the CP0 registers are set as follows.

- Status.EXL ← 1
- Cause.ExcCode ← FPE
- EPC ← Execution restart address
- Cause.BD ← 1 when the exception was generated by an instruction in the branch delay slot

After the above processing, the program jumps to the general exception vector.
Among the FPU exceptions, the inexact exception is a delayed exception which is delayed from the timing of the instruction execution. If ERL or EXL is already 1 when an exception occurs (when another exception occurs during a delay), the program will wait until the exception processing has completed and the ERL or EXL becomes 0. That is, the FPU exceptions will start the exception processing after the ERL or EXL becomes 0.

Software Response

The Cause field of the FPU's Control/Status register (FCR31) indicates the cause of the FPU exception. The FPU exception is cleared by clearing the relevant bit of the Cause field.
In ALLEGREX™, since the FPU and CPU each have a different number of pipeline stages, the EPC register does not precisely point to the address of the instruction which caused the FPU exception. As a result, control cannot return from an FPU exception.
For details about FPU instructions, refer to the "FPU User's Manual."

6.4.11. Coprocessor Unusable Exception

Cause

A coprocessor unusable exception is generated when one of the following occurs.

- A CP0 instruction is executed in user mode or supervisor mode when the CU0 bit of the Status register is 0
- An FPU instruction is executed when the CU1 bit of the Status register is 0
- A VFPU instruction is executed when the CU2 bit of the Status register is 0

This exception is non-maskable.

Hardware Operation

When a coprocessor unusable exception occurs, the CP0 registers are set as follows.

- Status.EXL ← 1
- Cause.ExcCode ← CpU
- EPC ← Execution restart address

- Cause.BD ← 1 when the exception was generated by an instruction in the branch delay slot
- Cause.CE ← Number of the coprocessor that caused the exception

After the above processing, the program jumps to the general exception vector.

Software Response

The coprocessor that the program tried to access when the exception was generated can be determined from the CE field of the Cause register. The operating system can grant the program that caused the exception the right to use the relevant coprocessor and re-execute the instruction.

6.4.12. Interrupt Exception

Cause

An interrupt exception is generated when one of the eight interrupt causes is asserted. Each of the eight interrupts can be masked by clearing the corresponding bit of the IM field in the Status register beforehand. Also, all eight interrupts can be masked at once by clearing the IE bit in the Status register.

Hardware Operation

When an interrupt exception is generated, the CP0 registers are set as follows.

- Status.EXL ← 1
- Cause.ExcCode ← Int
- EPC ← Execution restart address
- Cause.BD ← 1 when the exception was generated by an instruction in the branch delay slot
- Cause.IP7 ← 1 when a timer interrupt occurred
- Cause.IP6-IP2 ← 1 when an external interrupt occurred
- Cause.IP1 ← 1 when a software interrupt occurred
- Cause.IP0 ← 1 when a software interrupt occurred

After the above processing, the program jumps to the general exception vector.

Software Response

If either of the two software interrupts occurred (IP[1:0] of the Cause register), the interrupt can be cleared by clearing the corresponding IP bits in the Cause register.
If a hardware interrupt port signal (INT[5:0]) went active causing one of the six hardware interrupts to occur (IP[7:2] of the Cause register), the interrupt can be cleared by making the appropriate interrupt signal go inactive.