MIPS32® Architecture for Programmers VolumeIV-e: The MIPS® DSP Application-Specific Extension to the MIPS32® Architecture

Document Number: MD00374
Revision 1.00
July 6 2005

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043-1353

Template: B1.14, Built with tags: 2B ARCH MIPS32

Table of Contents

List of Figures

List of Tables

About This Book

The MIPS32® Architecture for Programmers VolumeIV-e comes as a multi-volume set.

- Volume I describes conventions used throughout the document set, and provides an introduction to the MIPS32® Architecture
- Volume II provides detailed descriptions of each instruction in the MIPS32® instruction set
- Volume III describes the MIPS32® Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS32® processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS32® Architecture
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS32® Architecture and is not applicable to the MIPS32® document set
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS32® Architecture
- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture

1.1 Typographical Conventions

This section describes the use of *italic*, **bold** and `courier` fonts in this book.

1.1.1 Italic Text

- is used for *emphasis*
- is used for *bits*, *fields*, *registers*, that are important from a software perspective (for instance, address bits used by software, and programmable fields and registers), and various *floating point instruction formats*, such as *S*, *D*, and *PS*
- is used for the memory access types, such as *cached* and *uncached*

1.1.2 Bold Text

- represents a term that is being **defined**
- is used for **bits** and **fields** that are important from a hardware perspective (for instance, **register** bits, which are not programmable but accessible only to hardware)
- is used for ranges of numbers; the range is indicated by an ellipsis. For instance, **5..1** indicates numbers 5 through 1
- is used to emphasize **UNPREDICTABLE** and **UNDEFINED** behavior, as defined below.

1.1.3 Courier Text

`Courier` fixed-width font is used for text that is displayed on the screen, and for examples of code and instruction pseudocode.

1.2 UNPREDICTABLE and UNDEFINED

The terms **UNPREDICTABLE** and **UNDEFINED** are used throughout this book to describe the behavior of the processor in certain cases. **UNDEFINED** behavior or operations can occur only as the result of executing instructions in a privileged mode (i.e., in Kernel Mode or Debug Mode, or with the CP0 usable bit set in the Status register). Unprivileged software can never cause **UNDEFINED** behavior or operations. Conversely, both privileged and unprivileged software can cause **UNPREDICTABLE** results or operations.

1.2.1 UNPREDICTABLE

UNPREDICTABLE results may vary from processor implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. Software can never depend on results that are **UNPREDICTABLE**. **UNPREDICTABLE** operations may cause a result to be generated or not. If a result is generated, it is **UNPREDICTABLE**. **UNPREDICTABLE** operations may cause arbitrary exceptions.

UNPREDICTABLE results or operations have several implementation restrictions:

- Implementations of operations generating **UNPREDICTABLE** results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode
- **UNPREDICTABLE** operations must not read, write, or modify the contents of memory or internal state which is inaccessible in the current processor mode. For example, **UNPREDICTABLE** operations executed in user mode must not access memory or internal state that is only accessible in Kernel Mode or Debug Mode or in another process
- **UNPREDICTABLE** operations must not halt or hang the processor

1.2.2 UNDEFINED

UNDEFINED operations or behavior may vary from processor implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. **UNDEFINED** operations or behavior may vary from nothing to creating an environment in which execution can no longer continue. **UNDEFINED** operations or behavior may cause data loss.

UNDEFINED operations or behavior has one implementation restriction:

- **UNDEFINED** operations or behavior must not cause the processor to hang (that is, enter a state from which there is no exit other than powering down the processor). The assertion of any of the reset signals must restore the processor to an operational state

1.2.3 UNSTABLE

UNSTABLE results or values may vary as a function of time on the same implementation or instruction. Unlike **UNPREDICTABLE** values, software may depend on the fact that a sampling of an **UNSTABLE** value results in a legal transient value that was correct at some point in time prior to the sampling.

UNSTABLE values have one implementation restriction:

- Implementations of operations generating **UNSTABLE** results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode

1.3 Special Symbols in Pseudocode Notation

In this book, algorithmic descriptions of an operation are described as pseudocode in a high-level language notation resembling Pascal. Special symbols used in the pseudocode notation are listed in Table 1-1.

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
←	Assignment
=, ≠	Tests for equality and inequality
||	Bit string concatenation
x^y	A *y*-bit string formed by *y* copies of the single-bit value *x*
b#n	A constant value *n* in base *b*. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
0bn	A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256).
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+, −	2's complement or floating point arithmetic: addition, subtraction
*, ×	2's complement or floating point multiplication (both used for either)
div	2's complement integer division
mod	2's complement modulo
/	Floating point division
<	2's complement less-than comparison
>	2's complement greater-than comparison
≤	2's complement less-than or equal comparison
≥	2's complement greater-than or equal comparison
nor	Bitwise logical NOR
xor	Bitwise logical XOR
and	Bitwise logical AND
or	Bitwise logical OR
GPRLEN	The length in bits (32 or 64) of the CPU general-purpose registers
GPR[x]	CPU general-purpose register *x*. The content of *GPR[0]* is always zero. In Release 2 of the Architecture, GPR[x] is a short-hand notation for *SGPR[$SRSCtl_{CSS}$, x]*.
SGPR[s,x]	In Release 2 of the Architecture, multiple copies of the CPU general-purpose registers may be implemented. *SGPR[s,x]* refers to GPR set *s*, register *x*.
FPR[x]	Floating Point operand register *x*
FCC[CC]	Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*.
FPR[x]	Floating Point (Coprocessor unit 1), general register *x*

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
CPR[z,x,s]	Coprocessor unit *z*, general register *x*, select *s*
CP2CPR[x]	Coprocessor unit 2, general register *x*
CCR[z,x]	Coprocessor unit *z*, control register *x*
CP2CCR[x]	Coprocessor unit 2, control register *x*
COC[z]	Coprocessor unit *z* condition signal
Xlat[x]	Translation of the MIPS16e GPR number *x* into the corresponding 32-bit GPR number
BigEndianMem	Endian mode as configured at chip reset (0 →Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.
BigEndianCPU	The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the *RE* bit in the *Status* register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR_{RE} and User mode).
LLbit	Bit of **virtual** state used to specify operation for instructions that provide atomic read-modify-write. *LLbit* is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.
I:, **I+n**:, **I-n**:	This occurs as a prefix to *Operation* description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to "execute." Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of **I**. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction **I**, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled **I+1**. The effect of pseudocode statements for the current instruction labelled **I+1** appears to occur "at the same time" as the effect of pseudocode statements labeled **I** for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur "at the same time," there is no defined order. Programs must not depend on a particular order of evaluation between such sections.
PC	The *Program Counter* value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to *PC* during an instruction time. If no value is assigned to *PC* during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the *PC* during the instruction time of the instruction in the branch delay slot. In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. The PC value contains a full 32-bit address all of which are significant during a memory reference.
ISA Mode	In processors that implement the MIPS16e Application Specific Extension, the *ISA Mode* is a single-bit register that determines in which mode the processor is executing, as follows: **Encoding** / **Meaning** 0 — The processor is executing 32-bit MIPS instructions 1 — The processor is executing MIIPS16e instructions In the MIPS Architecture, the ISA Mode value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the ISA Mode into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
PABITS	The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.
FP32RegistersMode	Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR. In MIPS32 implementations, **FP32RegistersMode** is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case **FP32RegisterMode** is computed from the FR bit in the *Status* register. If this bit is a 0, the processor operates as if it had 32 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs. The value of **FP32RegistersMode** is computed from the FR bit in the *Status* register.
InstructionInBranchDelaySlot	Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the *dynamic* state of the instruction, not the *static* state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump.
SignalException(exception, argument)	Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function - the exception is signaled at the point of the call.

1.4 For More Information

Various MIPS RISC processor manuals and additional information about MIPS products can be found at the MIPS URL:

http://www.mips.com

Comments or questions on the MIPS32® Architecture or this document should be directed to

MIPS Architecture Group
MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043

or via E-mail to architecture@mips.com.

Guide to the Instruction Set

This chapter provides a detailed guide to understanding the instruction descriptions, which are listed in alphabetical order in the tables at the beginning of the next chapter.

2.1 Understanding the Instruction Fields

Figure 2-1 shows an example instruction. Following the figure are descriptions of the fields listed below:

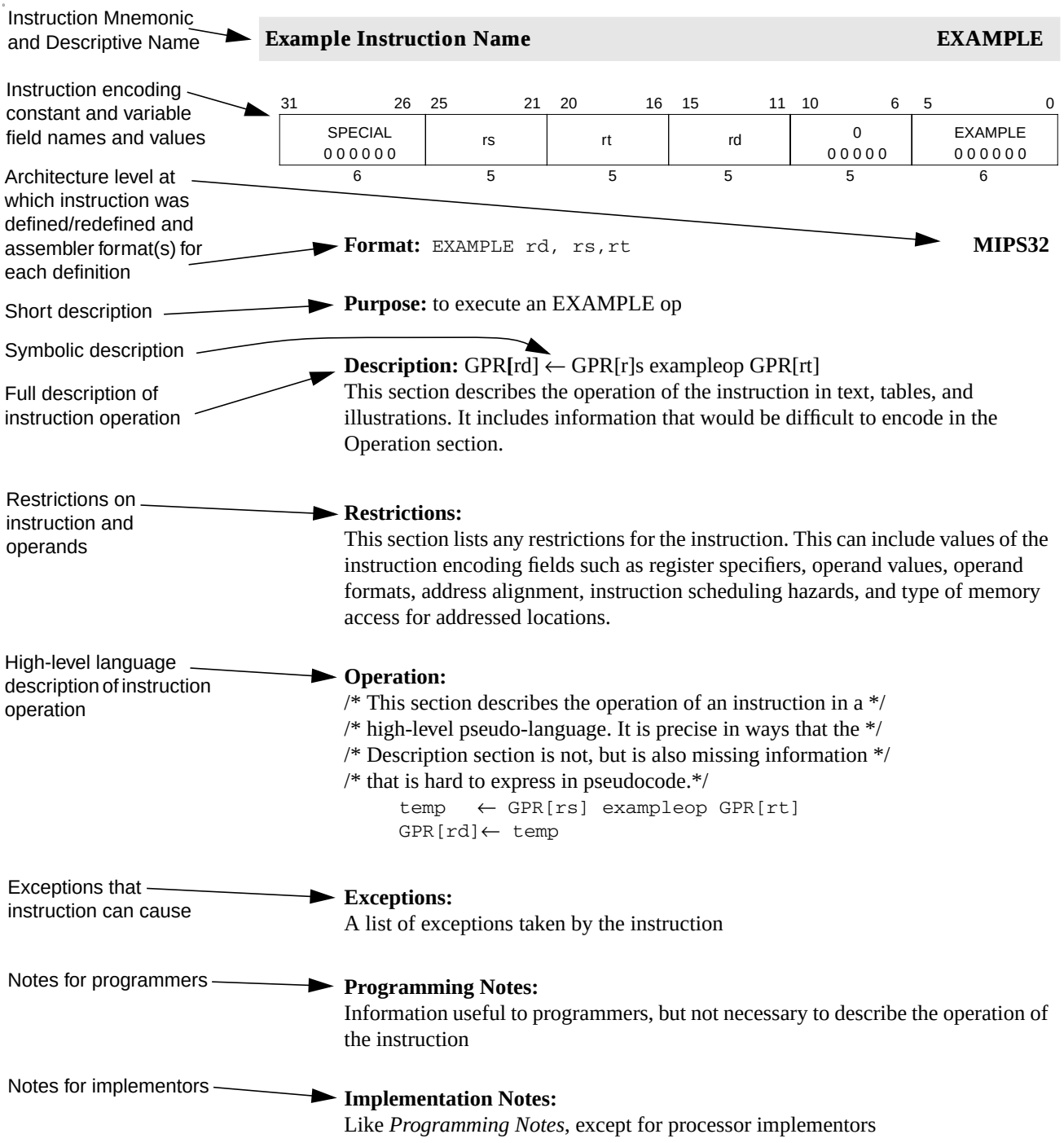


Figure 2-1 Example of Instruction Description

2.1.1 Instruction Fields

Fields encoding the instruction word are shown in register form at the top of the instruction description. The following rules are followed:

- The values of constant fields and the *opcode* names are listed in uppercase (SPECIAL and ADD in Figure 2-2). Constant values in a field are shown in binary below the symbolic or hexadecimal value.
- All variable fields are listed with the lowercase names used in the instruction description (*rs*, *rt* and *rd* in Figure 2-2).
- Fields that contain zeros but are not named are unused fields that are required to be zero (bits 10:6 in Figure 2-2). If such fields are set to non-zero values, the operation of the processor is **UNPREDICTABLE**.

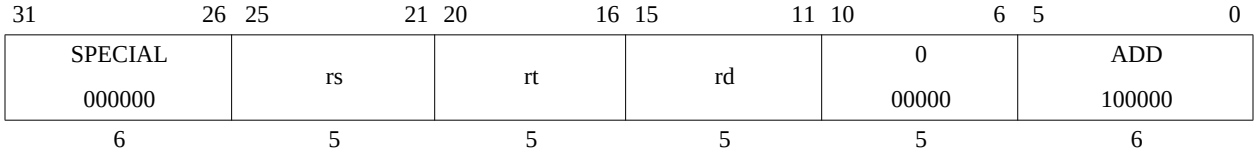


Figure 2-2 Example of Instruction Fields

2.1.2 Instruction Descriptive Name and Mnemonic

The instruction descriptive name and mnemonic are printed as page headings for each instruction, as shown in Figure 2-3.

Add Word **ADD**

Figure 2-3 Example of Instruction Descriptive Name and Mnemonic

2.1.3 Format Field

The assembler formats for the instruction and the architecture level at which the instruction was originally defined are given in the *Format* field. If the instruction definition was later extended, the architecture levels at which it was extended and the assembler formats for the extended definition are shown in their order of extension (for an example, see C.cond.fmt). The MIPS architecture levels are inclusive; higher architecture levels include all instructions in previous levels. Extensions to instructions are backwards compatible. The original assembler formats are valid for the extended architecture.

Format: `ADD rd, rs, rt` **MIPS32**

Figure 2-4 Example of Instruction Format

The assembler format is shown with literal parts of the assembler instruction printed in uppercase characters. The variable parts, the operands, are shown as the lowercase names of the appropriate fields. The architectural level at which the instruction was first defined, for example “MIPS32” is shown at the right side of the page.

There can be more than one assembler format for each architecture level. Floating point operations on formatted data show an assembly format with the actual assembler mnemonic for each valid value of the *fmt* field. For example, the ADD.fmt instruction lists both ADD.S and ADD.D.

The assembler format lines sometimes include parenthetical comments to help explain variations in the formats (once again, see C.cond.fmt). These comments are not a part of the assembler format.

2.1.4 Purpose Field

The *Purpose* field gives a short description of the use of the instruction.

Purpose:

To add 32-bit integers. If an overflow occurs, then trap.

Figure 2-5 Example of Instruction Purpose

2.1.5 Description Field

If a one-line symbolic description of the instruction is feasible, it appears immediately to the right of the *Description* heading. The main purpose is to show how fields in the instruction are used in the arithmetic or logical operation.

Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*

Figure 2-6 Example of Instruction Description

The body of the section is a description of the operation of the instruction in text, tables, and figures. This description complements the high-level language description in the *Operation* section.

This section uses acronyms for register descriptions. "GPR *rt*" is CPU general-purpose register specified by the instruction field *rt*. "FPR *fs*" is the floating point operand register specified by the instruction field *fs*. "CP1 register *fd*" is the coprocessor 1 general register specified by the instruction field *fd*. "*FCSR*" is the floating point *Control /Status* register.

2.1.6 Restrictions Field

The *Restrictions* field documents any possible restrictions that may affect the instruction. Most restrictions fall into one of the following six categories:

- Valid values for instruction fields (for example, see floating point ADD.fmt)
- ALIGNMENT requirements for memory addresses (for example, see LW)
- Valid values of operands (for example, see DADD)
- Valid operand formats (for example, see floating point ADD.fmt)
- Order of instructions necessary to guarantee correct execution. These ordering constraints avoid pipeline hazards for which some processors do not have hardware interlocks (for example, see MUL).
- Valid memory access types (for example, see LL/SC)

Restrictions:

None

Figure 2-7 Example of Instruction Restrictions

2.1.7 Operation Field

The *Operation* field describes the operation of the instruction as pseudocode in a high-level language notation resembling Pascal. This formal description complements the *Description* section; it is not complete in itself because many of the restrictions are either difficult to include in the pseudocode or are omitted for legibility.

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

Figure 2-8 Example of Instruction Operation

See Section 2.2, "Operation Section Notation and Functions" on page 12 for more information on the formal notation used here.

2.1.8 Exceptions Field

The *Exceptions* field lists the exceptions that can be caused by *Operation* of the instruction. It omits exceptions that can be caused by the instruction fetch, for instance, TLB Refill, and also omits exceptions that can be caused by asynchronous external events such as an Interrupt. Although a Bus Error exception may be caused by the operation of a load or store instruction, this section does not list Bus Error for load and store instructions because the relationship between load and store instructions and external error indications, like Bus Error, are dependent upon the implementation.

Exceptions:

Integer Overflow

Figure 2-9 Example of Instruction Exception

An instruction may cause implementation-dependent exceptions that are not present in the *Exceptions* section.

2.1.9 Programming Notes and Implementation Notes Fields

The *Notes* sections contain material that is useful for programmers and implementors, respectively, but that is not necessary to describe the instruction and does not belong in the description sections.

Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

Figure 2-10 Example of Instruction Programming Notes

2.2 Operation Section Notation and Functions

In an instruction description, the *Operation* section uses a high-level language notation to describe the operation performed by each instruction. Special symbols used in the pseudocode are described in the previous chapter. Specific pseudocode functions are described below.

This section presents information about the following topics:

- "Instruction Execution Ordering" on page 12
- "Pseudocode Functions" on page 12

2.2.1 Instruction Execution Ordering

Each of the high-level language statements in the *Operations* section are executed sequentially (except as constrained by conditional and loop constructs).

2.2.2 Pseudocode Functions

There are several functions used in the pseudocode descriptions. These are used either to make the pseudocode more readable, to abstract implementation-specific behavior, or both. These functions are defined in this section, and include the following:

- "Coprocessor General Register Access Functions" on page 12
- "Memory Operation Functions" on page 14
- "Floating Point Functions" on page 17
- "Miscellaneous Functions" on page 20

2.2.2.1 Coprocessor General Register Access Functions

Defined coprocessors, except for CP0, have instructions to exchange words and doublewords between coprocessor general registers and the rest of the system. What a coprocessor does with a word or doubleword supplied to it and how a coprocessor supplies a word or doubleword is defined by the coprocessor itself. This behavior is abstracted into the functions described in this section.

COP_LW

The COP_LW function defines the action taken by coprocessor z when supplied with a word from memory during a load word operation. The action is coprocessor-specific. The typical action would be to store the contents of memword in coprocessor general register *rt*.

```
COP_LW (z, rt, memword)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    memword: A 32-bit word value supplied to the coprocessor

    /* Coprocessor-dependent action */

endfunction COP_LW
```

Figure 2-11 COP_LW Pseudocode Function

COP_LD

The COP_LD function defines the action taken by coprocessor z when supplied with a doubleword from memory during a load doubleword operation. The action is coprocessor-specific. The typical action would be to store the contents of memdouble in coprocessor general register *rt*.

```
COP_LD (z, rt, memdouble)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    memdouble:  64-bit doubleword value supplied to the coprocessor.

    /* Coprocessor-dependent action */

endfunction COP_LD
```

Figure 2-12 COP_LD Pseudocode Function

COP_SW

The COP_SW function defines the action taken by coprocessor z to supply a word of data during a store word operation. The action is coprocessor-specific. The typical action would be to supply the contents of the low-order word in coprocessor general register *rt*.

```
dataword ← COP_SW (z, rt)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    dataword: 32-bit word value

    /* Coprocessor-dependent action */

endfunction COP_SW
```

Figure 2-13 COP_SW Pseudocode Function

COP_SD

The COP_SD function defines the action taken by coprocessor z to supply a doubleword of data during a store doubleword operation. The action is coprocessor-specific. The typical action would be to supply the contents of the low-order doubleword in coprocessor general register *rt*.

```
datadouble ← COP_SD (z, rt)
   z: The coprocessor unit number
   rt: Coprocessor general register specifier
   datadouble: 64-bit doubleword value

   /* Coprocessor-dependent action */

endfunction COP_SD
```

Figure 2-14 COP_SD Pseudocode Function

CoprocessorOperation

The CoprocessorOperation function performs the specified Coprocessor operation.

```
CoprocessorOperation (z, cop_fun)

   /* z:         Coprocessor unit number */
   /* cop_fun:   Coprocessor function from function field of instruction */

   /* Transmit the cop_fun value to coprocessor z */

endfunction CoprocessorOperation
```

Figure 2-15 CoprocessorOperation Pseudocode Function

2.2.2.2 Memory Operation Functions

Regardless of byte ordering (big- or little-endian), the address of a halfword, word, or doubleword is the smallest byte address of the bytes that form the object. For big-endian ordering this is the most-significant byte; for a little-endian ordering this is the least-significant byte.

In the *Operation* pseudocode for load and store operations, the following functions summarize the handling of virtual addresses and the access of physical memory. The size of the data item to be loaded or stored is passed in the *AccessLength* field. The valid constant names and values are shown in Table 2-1. The bytes within the addressed unit of memory (word for 32-bit processors or doubleword for 64-bit processors) that are used can be determined directly from the *AccessLength* and the two or three low-order bits of the address.

AddressTranslation

The AddressTranslation function translates a virtual address to a physical address and its cache coherence algorithm, describing the mechanism used to resolve the memory reference.

Given the virtual address *vAddr*, and whether the reference is to Instructions or Data (*IorD*), find the corresponding physical address (*pAddr*) and the cache coherence algorithm (*CCA*) used to resolve the reference. If the virtual address is in one of the unmapped address spaces, the physical address and *CCA* are determined directly by the virtual address. If the virtual address is in one of the mapped address spaces then the TLB or fixed mapping MMU determines the

physical address and access type; if the required translation is not present in the TLB or the desired access is not permitted, the function fails and an exception is taken.

```
(pAddr, CCA) ← AddressTranslation (vAddr, IorD, LorS)

   /* pAddr: physical address */
   /* CCA:   Cache Coherence Algorithm, the method used to access caches*/
   /*        and memory and resolve the reference */

   /* vAddr: virtual address */
   /* IorD:  Indicates whether access is for INSTRUCTION or DATA */
   /* LorS:  Indicates whether access is for LOAD or STORE */

   /* See the address translation description for the appropriate MMU */
   /* type in Volume III of this book for the exact translation mechanism */

endfunction AddressTranslation
```

Figure 2-16 AddressTranslation Pseudocode Function

LoadMemory

The LoadMemory function loads a value from memory.

This action uses cache and main memory as specified in both the Cache Coherence Algorithm (*CCA*) and the access (*IorD*) to find the contents of *AccessLength* memory bytes, starting at physical location *pAddr*. The data is returned in a fixed-width naturally aligned memory element (*MemElem*). The low-order 2 (or 3) bits of the address and the *AccessLength* indicate which of the bytes within *MemElem* need to be passed to the processor. If the memory access type of the reference is *uncached*, only the referenced bytes are read from memory and marked as valid within the memory element. If the access type is *cached* but the data is not present in cache, an implementation-specific *size* and *alignment* block of memory is read and loaded into the cache to satisfy a load reference. At a minimum, this block is the entire memory element.

```
MemElem ← LoadMemory (CCA, AccessLength, pAddr, vAddr, IorD)

   /* MemElem:  Data is returned in a fixed width with a natural alignment. The */
   /*           width is the same size as the CPU general-purpose register, */
   /*           32 or 64 bits, aligned on a 32- or 64-bit boundary, */
   /*           respectively. */
   /* CCA:      Cache Coherence Algorithm, the method used to access caches */
   /*           and memory and resolve the reference */

   /* AccessLength: Length, in bytes, of access */
   /* pAddr:    physical address */
   /* vAddr:    virtual address */
   /* IorD:     Indicates whether access is for Instructions or Data */

endfunction LoadMemory
```

Figure 2-17 LoadMemory Pseudocode Function

StoreMemory

The StoreMemory function stores a value to memory.

The specified data is stored into the physical location *pAddr* using the memory hierarchy (data caches and main memory) as specified by the Cache Coherence Algorithm (*CCA*). The *MemElem* contains the data for an aligned, fixed-width memory element (a word for 32-bit processors, a doubleword for 64-bit processors), though only the bytes that are

actually stored to memory need be valid. The low-order two (or three) bits of *pAddr* and the *AccessLength* field indicate which of the bytes within the *MemElem* data should be stored; only these bytes in memory will actually be changed.

```
StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)

   /* CCA:       Cache Coherence Algorithm, the method used to access */
   /*            caches and memory and resolve the reference. */
   /* AccessLength: Length, in bytes, of access */
   /* MemElem:   Data in the width and alignment of a memory element. */
   /*            The width is the same size as the CPU general */
   /*            purpose register, either 4 or 8 bytes, */
   /*            aligned on a 4- or 8-byte boundary. For a */
   /*            partial-memory-element store, only the bytes that will be*/
   /*            stored must be valid.*/
   /* pAddr:     physical address */
   /* vAddr:     virtual address */

endfunction StoreMemory
```

Figure 2-18 StoreMemory Pseudocode Function

Prefetch

The Prefetch function prefetches data from memory.

Prefetch is an advisory instruction for which an implementation-specific action is taken. The action taken may increase performance but must not change the meaning of the program or alter architecturally visible state.

```
Prefetch (CCA, pAddr, vAddr, DATA, hint)

   /* CCA:   Cache Coherence Algorithm, the method used to access */
   /*        caches and memory and resolve the reference. */
   /* pAddr: physical address */
   /* vAddr: virtual address */
   /* DATA:  Indicates that access is for DATA */
   /* hint:  hint that indicates the possible use of the data */

endfunction Prefetch
```

Figure 2-19 Prefetch Pseudocode Function

Table 2-1 lists the data access lengths and their labels for loads and stores.

Table 2-1 AccessLength Specifications for Loads/Stores

AccessLength Name	Value	Meaning
DOUBLEWORD	7	8 bytes (64 bits)
SEPTIBYTE	6	7 bytes (56 bits)
SEXTIBYTE	5	6 bytes (48 bits)
QUINTIBYTE	4	5 bytes (40 bits)
WORD	3	4 bytes (32 bits)
TRIPLEBYTE	2	3 bytes (24 bits)
HALFWORD	1	2 bytes (16 bits)
BYTE	0	1 byte (8 bits)

SyncOperation

The SyncOperation function orders loads and stores to synchronize shared memory.

This action makes the effects of the synchronizable loads and stores indicated by *stype* occur in the same order for all processors.

```
SyncOperation(stype)

   /* stype: Type of load/store ordering to perform. */

   /* Perform implementation-dependent operation to complete the */
   /* required synchronization operation */

endfunction SyncOperation
```

Figure 2-20 SyncOperation Pseudocode Function

2.2.2.3 Floating Point Functions

The pseudocode shown in below specifies how the unformatted contents loaded or moved to CP1 registers are interpreted to form a formatted value. If an FPR contains a value in some format, rather than unformatted contents from a load (uninterpreted), it is valid to interpret the value in that format (but not to interpret it in a different format).

ValueFPR

The ValueFPR function returns a formatted value from the floating point registers.

```
value ← ValueFPR(fpr, fmt)

    /* value: The formattted value from the FPR */

    /* fpr:   The FPR number */
    /* fmt:   The format of the data, one of: */
    /*        S, D, W, L, PS, */
    /*        OB, QH, */
    /*        UNINTERPRETED_WORD, */
    /*        UNINTERPRETED_DOUBLEWORD */
    /* The UNINTERPRETED values are used to indicate that the datatype */
    /* is not known as, for example, in SWC1 and SDC1 */

    case fmt of
        S, W, UNINTERPRETED_WORD:
            valueFPR ← FPR[fpr]

        D, UNINTERPRETED_DOUBLEWORD:
            if (FP32RegistersMode = 0)
                if (fpr0 ≠ 0) then
                    valueFPR ← UNPREDICTABLE
                else
                    valueFPR ← FPR[fpr+1]31..0 || FPR[fpr]31..0
                endif
            else
                valueFPR ← FPR[fpr]
            endif

        L, PS:
            if (FP32RegistersMode = 0) then
                valueFPR ← UNPREDICTABLE
            else
                valueFPR ← FPR[fpr]
            endif

        DEFAULT:
            valueFPR ← UNPREDICTABLE

    endcase
endfunction ValueFPR
```

Figure 2-21 ValueFPR Pseudocode Function

The pseudocode shown below specifies the way a binary encoding representing a formatted value is stored into CP1 registers by a computational or move operation. This binary representation is visible to store or move-from instructions. Once an FPR receives a value from the StoreFPR(), it is not valid to interpret the value with ValueFPR() in a different format.

StoreFPR

```
StoreFPR (fpr, fmt, value)

   /* fpr:   The FPR number */
   /* fmt:   The format of the data, one of: */
   /*        S, D, W, L, PS, */
   /*        OB, QH, */
   /*        UNINTERPRETED_WORD, */
   /*        UNINTERPRETED_DOUBLEWORD */
   /* value: The formattted value to be stored into the FPR */

   /* The UNINTERPRETED values are used to indicate that the datatype */
   /* is not known as, for example, in LWC1 and LDC1 */


   case fmt of
      S, W, UNINTERPRETED_WORD:
         FPR[fpr] ← value

      D, UNINTERPRETED_DOUBLEWORD:
         if (FP32RegistersMode = 0)
            if (fpr0 ≠ 0) then
               UNPREDICTABLE
            else
               FPR[fpr]   ← UNPREDICTABLE^32 || value31..0
               FPR[fpr+1] ← UNPREDICTABLE^32 || value63..32
            endif
         else
            FPR[fpr] ← value
         endif

      L, PS:
         if (FP32RegistersMode = 0) then
            UNPREDICTABLE
         else
            FPR[fpr] ← value
         endif

   endcase

endfunction StoreFPR
```

Figure 2-22 StoreFPR Pseudocode Function

The pseudocode shown below checks for an enabled floating point exception and conditionally signals the exception.

CheckFPException

```
CheckFPException()

/* A floating point exception is signaled if the E bit of the Cause field is a 1 */
/* (Unimplemented Operations have no enable) or if any bit in the Cause field */
/* and the corresponding bit in the Enable field are both 1 */

   if (   (FCSR17 = 1) or
          ((FCSR16..12 and FCSR11..7) ≠ 0))   ) then
      SignalException(FloatingPointException)
   endif

endfunction CheckFPException
```

Figure 2-23 CheckFPException Pseudocode Function

FPConditionCode

The FPConditionCode function returns the value of a specific floating point condition code.

```
tf ←FPConditionCode(cc)

   /* tf: The value of the specified condition code */

   /* cc: The Condition code number in the range 0..7 */

   if cc = 0 then
      FPConditionCode ← FCSR23
   else
      FPConditionCode ← FCSR24+cc
   endif

endfunction FPConditionCode
```

Figure 2-24 FPConditionCode Pseudocode Function

SetFPConditionCode

The SetFPConditionCode function writes a new value to a specific floating point condition code.

```
SetFPConditionCode(cc)
   if cc = 0 then
      FCSR ← FCSR31..24 || tf || FCSR22..0
   else
      FCSR ← FCSR31..25+cc || tf || FCSR23+cc..0
endif

endfunction SetFPConditionCode
```

Figure 2-25 SetFPConditionCode Pseudocode Function

2.2.2.4 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

SignalException

The SignalException function signals an exception condition.

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

```
SignalException(Exception, argument)

   /* Exception:    The exception condition that exists. */
   /* argument:     A exception-dependent argument, if any */

endfunction SignalException
```

Figure 2-26 SignalException Pseudocode Function

SignalDebugBreakpointException

The SignalDebugBreakpointException function signals a condition that causes entry into Debug Mode from non-Debug Mode.

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

Figure 2-27 SignalDebugBreakpointException Pseudocode Function

SignalDebugModeBreakpointException

The SignalDebugModeBreakpointException function signals a condition that causes entry into Debug Mode from Debug Mode (i.e., an exception generated while already running in Debug Mode).

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

Figure 2-28 SignalDebugModeBreakpointException Pseudocode Function

NullifyCurrentInstruction

The NullifyCurrentInstruction function nullifies the current instruction.

The instruction is aborted, inhibiting not only the functional effect of the instruction, but also inhibiting all exceptions detected during fetch, decode, or execution of the instruction in question. For branch-likely instructions, nullification kills the instruction in the delay slot of the branch likely instruction.

```
NullifyCurrentInstruction()

endfunction NullifyCurrentInstruction
```

Figure 2-29 NullifyCurrentInstruction PseudoCode Function

JumpDelaySlot

The JumpDelaySlot function is used in the pseudocode for the PC-relative instructions in the MIPS16e ASE. The function returns TRUE if the instruction at *vAddr* is executed in a jump delay slot. A jump delay slot always immediately follows a JR, JAL, JALR, or JALX instruction.

```
JumpDelaySlot(vAddr)

   /* vAddr:Virtual address */

endfunction JumpDelaySlot
```

Figure 2-30 JumpDelaySlot Pseudocode Function

PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

```
PolyMult(x, y)
   temp ← 0
   for i in 0 .. 31
      if x_i = 1 then
         temp ← temp xor (y_(31-i)..0 || 0^i)
      endif
   endfor

   PolyMult ← temp

endfunction PolyMult
```

Figure 2-31 PolyMult Pseudocode Function

2.3 Op and Function Subfield Notation

In some instructions, the instruction subfields *op* and *function* can have constant 5- or 6-bit values. When reference is made to these instructions, uppercase mnemonics are used. For instance, in the floating point ADD instruction, *op*=COP1 and *function*=ADD. In other cases, a single field has both fixed and variable subfields, so the name contains both upper- and lowercase characters.

2.4 FPU Instructions

In the detailed description of each FPU instruction, all variable subfields in an instruction format (such as *fs, ft, immediate*, and so on) are shown in lowercase. The instruction name (such as ADD, SUB, and so on) is shown in uppercase.

For the sake of clarity, an alias is sometimes used for a variable subfield in the formats of specific instructions. For example, *rs=base* in the format for load and store instructions. Such an alias is always lowercase since it refers to a variable subfield.

Bit encodings for mnemonics are given in Volume I, in the chapters describing the CPU, FPU, MDMX, and MIPS16e instructions.

See Section 2.3, "Op and Function Subfield Notation" on page 22 for a description of the *op* and *function* subfields.

The MIPS® DSP Application Specific Extension to the MIPS32® Architecture

3.1 Base Architecture Requirements

The MIPS DSP ASE requires the following base architecture support:

- **MIPS32 Release 2 or MIPS64 Release 2 Architecture:** The MIPS DSP ASE requires a compliant implementation of the MIPS32 Release 2 or MIPS64 Release 2 Architecture.

3.2 Software Detection of the ASE

Software may determine if the MIPS DSP ASE is implemented by checking the state of the DSPP (DSP Present) bit 10 in the *config3* CP0 register.

3.3 Compliance and Subsetting

There are no instruction subsets of the MIPS DSP ASE--all DSP ASE instructions and state must be implemented.

3.4 Introduction to the MIPS® DSP ASE

This document contains a complete specification of the MIPS® DSP Application Specific Extension (ASE), an extension to the MIPS32® architecture. The extension comprises new integer instructions and new state that includes new *HI-LO* accumulator pairs and a *DSPControl* register. The MIPS DSP ASE can be included in either a MIPS32 or MIPS64 architecture implementation. The ASE has been designed to benefit a wide range of DSP, Media, and DSP-like algorithms. The performance increase from this extension can be used for integration of DSP-like functionality on a SOC (System on Chip) onto a MIPS core and therefore reduce overall system cost. The ASE includes many of the typical features found in other integer-based DSP extensions, for example, support for operations on fractional data types and register SIMD (Single Instruction Multiple Data) operations such as add, subtract, multiply, shift, etc. In addition, the extension includes some key features that efficiently address specific problems often encountered in DSP applications. These include for example, support for complex multiply, variable bit insert and extract, implementation and use of virtual circular buffers, etc.

This chapter contains a basic overview of the principles behind DSP application processing and the data types and structures needed to efficiently process such applications. The second chapter contains a list of all the instructions in the ASE by function type. Chapter three describes the position of the new instructions in the MIPS instruction opcode map. The rest of the specification contains a complete list of all the instructions that will comprise the ASE, and serves as a quick reference guide to all the instructions. Finally, various Appendix chapters describe how to implement and use the DSP ASE instructions in some common algorithms and inner loops.

3.5 DSP Applications and their Requirements

The MIPS DSP ASE has been designed specifically to improve the performance of a set of DSP and DSP-like applications. Table 3-1 shows these application areas sorted by the size of the data operands typically preferred by that

application for internal computations. For example, raw audio data is usually signed 16-bit, but 32-bit internal calculations are often necessary for high quality audio. (Internal precision of about 28 bits is often all that is required. This can be obtained using a fractional data type of the appropriate width). There is some cross-over in some cases, which are not explicitly listed here. For example, some hand-held consumer devices may use lower precision internal arithmetic for audio processing, that is, 16-bit internal data formats may be sufficient for the quality required for hand-held devices.

Table 3-1 Data Size of DSP Applications

In/Out Data Size	Internal Data Size	Applications
8 bits	8/16 bits	• Printer image processing. • Still JPEG processing. • Moving video processing
16 bits	16 bits	• Voice Processing. For example, G.723.1, G.729, G.726, echo cancellation, noise cancellation, channel equalization, etc. • Soft modem processing. For example V.92. • General DSP processing. For example, filters, correlation, convolution, etc.
16/24 bits	32 bits	Audio decoding and encoding. For example, MP3, AAC, SRS TruSurround, Dolby Digital Decoder, Pro Logic II, etc.

3.6 Fixed-Point Data Type

Typical DSP processing often requires the use of fixed-point arithmetic. Fixed-point arithmetic, unlike floating-point arithmetic, assumes that the position of the decimal point is fixed with respect to the bits representing the fractional value in the operand. To understand this type of arithmetic further, please consult DSP textbooks or other references that are easily available on the internet.

The fixed-point or fractional data types are often referred to using Q format notation. The general form for this notation is Qm.n, where Q designates that the data is in Q format, m is the number of bits used to designate the twos complement integer portion of the number, and n is the number of bits used to designate the twos complement fractional part of the number. Because the twos complement number is signed, the number of bits required to express a number is m+n+1, where the additional bit is required to denote the sign. In typical usage, it is very common for m to be zero. That is, only fractional bits are represented. In this case, a Q notation of the form Q0.n is abbreviated to Qn.

Using an example, a word (with 32 bits) can be represented using the Q31 format which implies one (left-most) sign bit, followed by the decimal point and then 31 bits of fractional data value. This is a range of -1.0 to +0.999...., the trailing values depending on the number of bits of precision used. Similarly a half-word (16 bits) can be represented using Q15 which implies one sign bit followed by 15 fractional bits that represent a value between -1.0 and +0.999.... Some instructions defined in the MIPS DSP ASE assume the underlying data operand to be a fractional data type. For details check the description under each instruction. The letter “Q” is also used in the instruction mnemonic if it operates on a fractional data type.

DSP applications often, but not always, prefer to saturate the result after an arithmetic operation that causes an overflow. Saturation clamps the result to the smallest negative or largest positive value in the case of underflow and overflow respectively for operations on signed values. On unsigned operands, the lowest value used for clamping is zero.

The interpretation of the 32 bits of the Q31 representation is shown in Table 3-2. Negative values are twos-complement of the equivalent positive value.

Table 3-2 The Value of a Fixed-Point Q31 Number

+ -	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-16}	2^{-17}	2^{-18}	2^{-19}	2^{-20}	2^{-21}	2^{-22}	2^{-23}	2^{-24}	2^{-25}	2^{-26}	2^{-27}	2^{-28}	2^{-29}	2^{-30}	2^{-31}

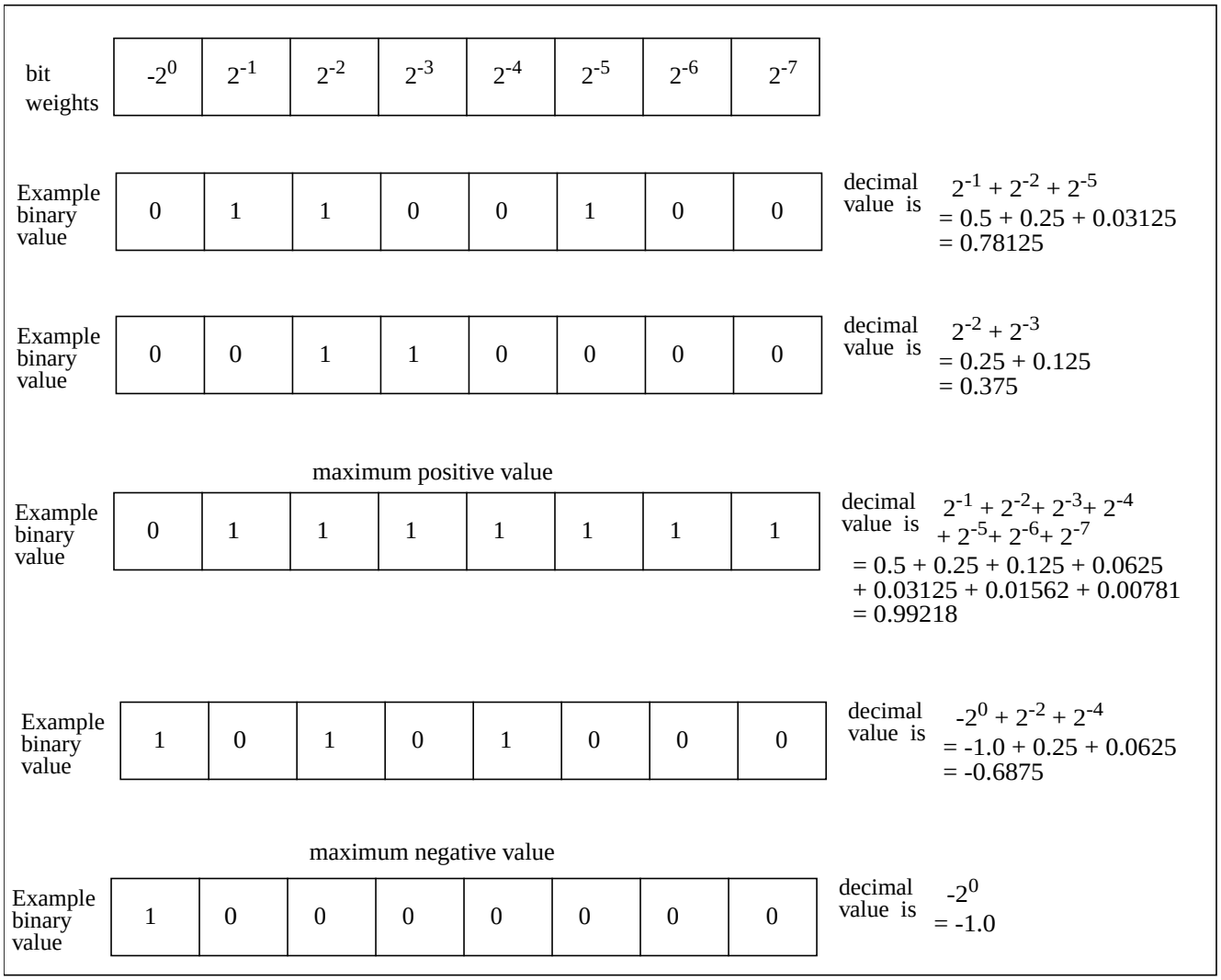
Table 3-3 shows the limits of the Q15 and the Q31 representations for 16 and 32 bits of data respectively. Note from the values in the table that -1.0 can be represented exactly, but +1.0 can not be represented exactly. For practical purposes, 0x7FFFFFFF is used to represent 1.0, but this is not exact. Thus, the multiplication of two values where both are -1 will result in an overflow since there is no representation for +1 in fixed-point format. Saturating instructions must check for this case and prevent the overflow by clamping the result to the maximal representable value.

Table 3-3 The Limits of Q15 and Q31 Representations

Fixed-Point Representation	Definition	Hexadecimal Representation	Decimal Equivalent
Q15 minimum	$-2^{15}/2^{15}$	0x8000	-1.0
Q15 maximum	$(2^{15}-1)/2^{15}$	0x7FFF	0.999969482421875
Q31 minimum	$-2^{31}/2^{31}$	0x80000000	-1.0
Q31 maximum	$(2^{31}-1)/2^{31}$	0x7FFFFFFF	0.99999999534338712692260742187 5

Given a fixed-point representation, we can compute the corresponding decimal value by using bit weights per position as shown in Figure 3-1 for a hypothetical Q7 format number representation with 8 total bits.

Figure 3-1 Computing the Value of a Fixed-Point (Q7) Number



3.7 Saturating Math

Many of the arithmetic instructions specify an optional saturation bit. See the per-page instruction descriptions for details on how the need for saturation checked and how it is performed. Saturation after the underflow or overflow of a fixed-point result requires clamping the value to the closest available fixed-point value for the given number of bits in the result. For unsigned values, the smallest value is zero, and the largest is the maximum positive value for that fixed-point value. For signed values, the two extremes are the maximal negative and the maximal positive values.

3.8 Conventions Used in the Instruction Mnemonics

Many of the instructions in this ASE operate on fractional (or fixed-point) data type operands. These instructions have a **Q** in their mnemonic. There are some instructions like add and subtract where the operation is independent of whether the data is in integer or fractional form. In this situation, the instruction mnemonic will use a **Q**, even though the data value could very well be integer and not fractional and the result of the instruction operation will produce a correct result.

Instructions that use unsigned data are indicated with the letter **U**. This letter appears after the letter **Q** for fractional in the instruction mnemonic. For example, **ADDQU** is an add instruction that does an unsigned add operation on fractional data. Unlike the base instruction set in which the distinction between signed and unsigned arithmetic instructions is whether an overflow trap occurs, an unsigned instruction in the DSP ASE provides a different result than a signed instruction for the case where saturation occurs.

Some instructions have the ability to specify an optional rounding or saturation mode. These instructions use the **_RS**, **_R**, **_S**, or **_SA** in their mnemonic. Note that it is possible to both round and saturate. For example **MULQ_RS** is a multiply instruction (**MUL**) with same size result as the input operand size (indicated by the absence of **E** for expanded result in the mnemonic), that assumes fractional (**Q**) input data operands, and allows optional rounding and saturate (**_RS**) before writing the result in the destination register. The normal saturation check for multiply instructions looks for input operands with a value of -1. Several multiply accumulate (dot-product) instructions use a variant of the saturation flag, **_SA**. This indicates the need to saturate after the accumulate operation in addition to the usual saturation check for -1 input operand values before the multiply.

The size of a general purpose register (GPR) depends on whether the processor implements the MIPS32 or the MIPS64 architecture. In the MIPS64 architecture, a GPR width is 64 bits and in the case of the MIPS32 architecture, the register width is 32 bits. As we noted in the previous discussion on DSP applications, the size of the internal data operands varies from 32 bits for audio to 8 bits for video. Most other general-purpose DSP applications prefer 16-bit data operands.

An obvious way to achieve greater performance and throughput is to use SIMD (Single Instruction Multiple Data) operations that vectorize with the multiple data elements within a given register. That is, a single instruction can implicitly invoke multiple operations on the multiple data operands that is stored within a single register. For example, a 64-bit GPR can store eight 8-bit operands, while a 32-bit GPR can store two 16-bit operands, and so on.

There are three basic formats for the input data operands: signed 32-bit, signed 16-bit, and unsigned 8-bit. The latter is motivated by the fact that video applications typically operate on unsigned 8-bit data.

The instructions provide support for the following data types with the following mnemonics in the MIPS32 architecture:

- W = “Word”, 1x 32-bit
- PH = “Paired Halfword”, 2x16-bit. See Figure 3-2.
- QB = “Quad Byte”, 4x8-bit. See Figure 3-3.

Figure 3-2 A Paired-Half (PH) Representation in a GPR for the MIPS32® Architecture

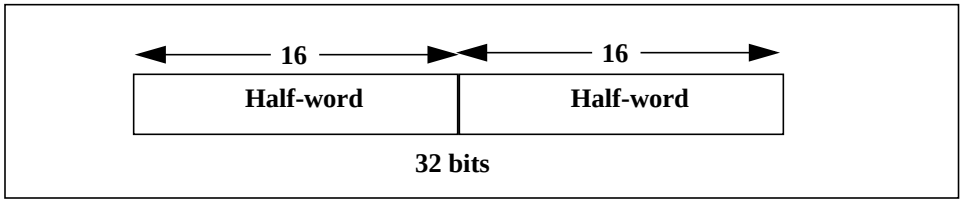
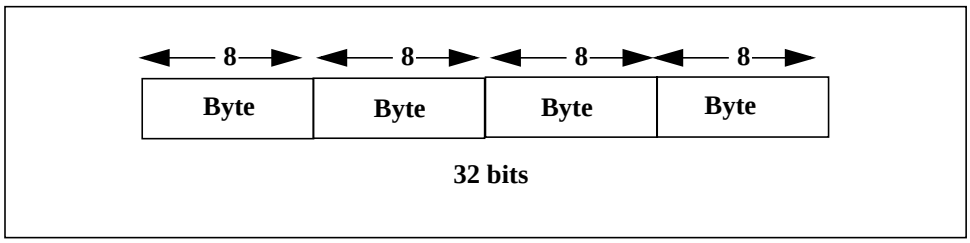


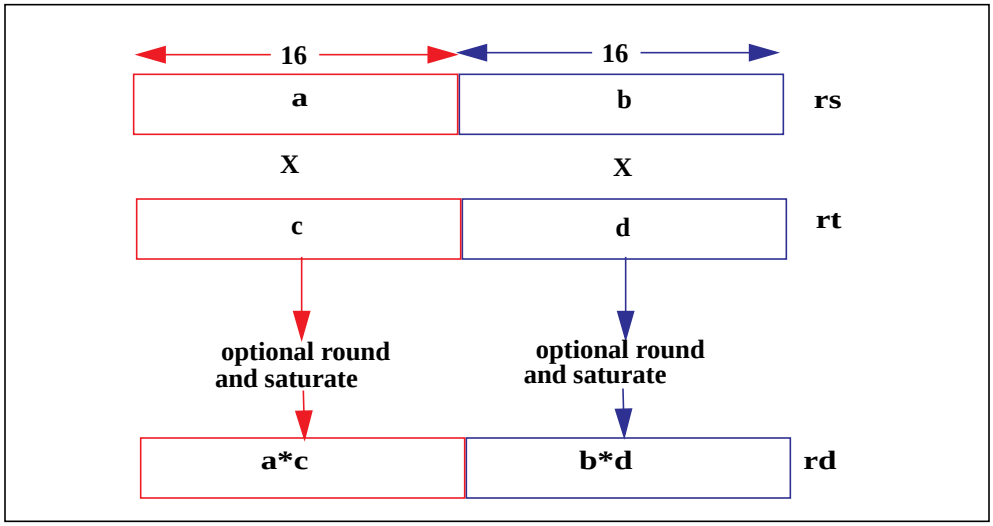
Figure 3-3 A Quad-Byte (QB) Representation in a GPR for the MIPS32® Architecture



For example, **MULQ_RS.PH rd, rs,rt** refers to the multiply instruction (**MUL**) that multiplies two vector elements of type fractional (**Q**) 16 bit (Halfword) data (**PH**) with rounding and saturation (**_RS**). Each source register supplies two data elements and the two results are written into the destination register in the corresponding vector position as shown in Figure 3-4.

When an instruction shows two format types, then the first is the output size and the second is the input size. For example, PRECRQ.PH.W is the (fractional) precision reduction instruction that creates a PH output format and uses W format as input from the two source registers. When the instruction only shows one format then this implies the same source and destination format.

Figure 3-4 Operation of MULQ_RS.PH rd, rs, rt



3.9 Effect of Endian-ness on Register SIMD Data

The order of data in memory and therefore in the register has a direct impact on the algorithm being executed. In order to try to reduce the effort required by the programmer and the development tools to take endian-ness into account, many of the instructions operate on pre-defined bits of a given register. The assembler can be used to map the endian-agnostic names to the actual instructions based on the endian-ness of the processor during the compilation and assembling of the instructions.

When a SIMD vector-type data is loaded into a register or stored back to memory from a register, the endian-ness of the processor and memory has an impact on the view of the data. For example, assume a sequence of eight 8-bit values is aligned in memory on a 64-bit boundary and is loaded together into a 64-bit register using the load double instruction. After the load, the positions of the eight byte values within the register depends on the processor endian-ness. In a big-endian processor, the first byte value in memory (with the smallest address,) is loaded into the left-most (most-significant) 8 bits of the 64-bit register. In a little-endian processor, the same 8-bit value is loaded into the right-most (least-significant) 8 bits of the register.

In general, if we name the 8-bit valued elements 0--7 according to their order in memory, in a big-endian configuration, element 0 is at the most-significant end and element 7 is at the least-significant end. In a little-endian configuration, the order is reversed. This effect applies to all the sizes of data when they are in SIMD format.

To avoid dealing with the endian-ness issue directly, the instructions in the DSP ASE simply refer to the left and right elements of the register when it is required to specify a subset of the elements. This issue can quite easily be dealt with

in the assembler or user code using suitably defined mnemonics that use the appropriate instruction for a given endian-ness of the processor. A description of how to do this is specified in Appendix A.

3.10 Additional Register State for the DSP ASE

This ASE adds four new registers. These registers require the operating system to recognize the presence of the DSP ASE and to include these additional registers in the context save and restore operation.

- Three additional *HI-LO* registers, which together with the existing one would comprise a total of four accumulator registers. Many common DSP computations are accumulate functions, for example, the filter operation, convolution, etc. The *HI-LO* accumulator in the MIPS architecture would be the destination for such instructions. The instructions that target the accumulators use 2 bits to specify the destination accumulator, with the zero value referring to the original accumulator.
- A new control register *DSPControl* used to hold extra state bits needed for efficient support of the new instructions. Figure 3-5 illustrates the bits in this register. Table 3-4 describes the use of the various bits and the instructions that refer the fields. Table 3-5 lists the instructions that affect the ouflag field.

Figure 3-5 MIPS32® DSP ASE Control Register (*DSPControl*) Format

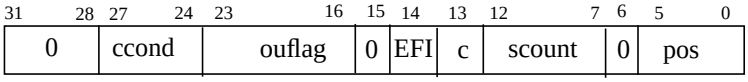


Table 3-4 MIPS® DSP ASE Control Register (*DSPControl*) Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:28	Not used in the MIPS32 architecture, but these are reserved bits since they are used in the MIPS64 architecture. Must be written as zero; returns zero on read.	0	0	Required
ccond	27:24	Condition code bits set after a compare instruction. This instruction will set the right-most bits as required by the number of elements in the vector compare. The bits not set by the instruction remain unchanged.	R/W	0	Required
ouflag	23:16	This field is written by hardware when certain instructions overflow or underflow and may have been saturated if the saturate flag was turned on. See Table 3-5 for a full list of which bits are set by what instructions.	R/W	0	Required
EFI	14	Extract Fail Indicator. This bit is set to 1 when an EXTP, EXTPV, EXTPDP, or EXTPDP fails. These instructions fail when there are insufficient bits to extract, that is, the value of pos in DSPControl is less than the value of size specified in the instruction. This bit is not sticky, that is, each invocation of one of those four instructions will reset the bit depending on whether or not the instruction failed.	R/W	0	Required

Table 3-4 MIPS® DSP ASE Control Register (*DSPControl*) Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	**Bits**				
c	13	Carry bit set and used by a special add instruction used to implement a 64-bit add across two GPRs in a MIPS32 implementation. Instruction ADDSC sets the bit and instruction ADDWC uses this bit.	R/W	0	Required
scount	12:7	This field is for use by the INSV instruction. The value of this field is used to specify the size of the bit field to be inserted.	R/W	0	Required
pos	5:0	This field is used by the variable insert instruction INSV to specify the insert position. It is also used to indicate the extract position for the EXTP, EXTPV, EXTPDP, and EXTPDPV instructions. The decrement pos (DP) variants of these instructions on completion will have decremented the value of pos (by the size amount). The MTHLIP instruction will increment the pos value by 32 after copying the value of LO to HI.	R/W	0	Required
0	15:13	Must be written as zero; returns zero on read.	0	0	Reserved

The bits of the overflow flag ouflag field in the *DSPControl* register are set by a number of instructions. These bits are sticky and can be reset only by an explicit write to these bits in the register (using the WRDSP instruction). The table shows which bits can be set by which instructions and under what conditions.

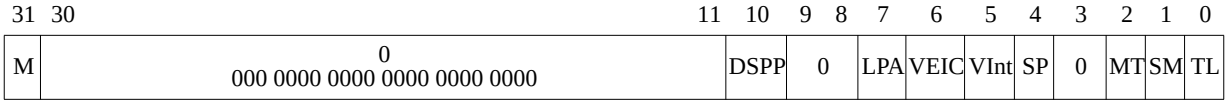
Table 3-5 The Instructions that Set the ouflag bits in *DSPControl*

Bit Number	Which Instruction Sets this Bit
16	When the destination is accumulator (*HI-LO* pair) zero, and an operation overflow or underflow occurs, then this bit is set. Such instructions are: DPAQ_S, DPSQ_S, MULSAQ_S, MAQ_S.
17	Instructions as above, when the destination is accumulator (*HI-LO* pair) one.
18	Instructions as above, when the destination is accumulator (*HI-LO* pair) two.
19	Instructions as above, when the destination is accumulator (*HI-LO* pair) three.
20	Instructions that on a overflow/underflow will set this bit are: ADDQ, ADDQ_S, SUBQ, SUBQ_S, ADDU, ADDU_S, SUBU, SUBU_S, ABSQ_S, and ADDWC.
21	Instructions that on a overflow/underflow will set this bit are: MULQ_RS, MULEQ_S, and MULEU_S.
22	Instructions that on a overflow/underflow will set this bit are: SHLL, SHLLV, SHLL_S, SHLLV_S, PRECRQU_S. PRECRQ_RS.
23	Instructions that on a overflow/underflow will set this bit are: EXTR and variants.

3.11 Software Detection of the DSP ASE

In the *config3* CP0 register bit 10 (DSPP), "DSP Present" is used to indicate the presence of the MIPS DSP ASE in the current implementation as shown in Figure 3-6. Software may check bit 10, DSP Present, (DSPP) of the *config3* CP0 register to check whether this processor has implemented the MIPS DSP ASE. When bit 10 is not set, the attempt to execute instructions in the DSP ASE must cause a Reserved Instruction Exception.

Figure 3-6 Config3 Register Format



Another bit, DSP ASE Enable (DSPEn) is used to enable access to the extra instructions defined by the DSP ASE as well as the MTLO/HI, MFLO/HI that access accumulators ac1, ac2, and ac3. This bit is the MX bit (24) in the CP0 *Status* register, see Figure 3-1. Executing a DSP ASE instruction or the flavor of Move instruction described above with this bit set to zero causes a DSP State Disabled Exception. This uses exception code 26 in the CP0 *Cause* register. This allows the OS to do lazy context-switching. Table 3-6 shows the Cause Register exception code fields. Note that bit 24, the MX field of *Status* is already defined as access bit to the MIPS® MDMX ASE. This use of the bit for enable of the MIPS DSP ASE is an **and** function if an implementation exists with both the MDMX ASE and the DSP ASE. That is, the bit when turned off disables both ASEs, and when turned on enables access to both ASEs, which currently implies that OS context switch code must switch the extra state for both ASEs, if they exist.

Figure 3-7 Status Register Format

Table 3-6 Cause Register ExcCode Field

Exception Code Value		Mnemonic	Description
Decimal	**Hexadecimal**		
26	16#1a	DSPDis	DSP ASE State Disabled Exception

3.12 Exception Table for the DSP ASE

Table 3-7 shows the exceptions caused when a MIPS DSP ASE instruction, MTLO/HI or MFLO/HI, or any other instruction such as an CorExtend instruction attempts to access the new DSP ASE state, that is, ac1, ac2, or ac3, or the DSPControl register, and all other possible exceptions that relate to the DSP ASE.

Table 3-7 Exception Table for the DSP ASE

$Config3_{DSPP}$	$Status_{MX}$	Valid DSP ASE op	Exception
0	x	x	Reserved Instruction
1	0	x	DSP ASE State Disabled
1	1	no	Reserved Instruction
1	1	yes	None

3.13 DSP ASE Instructions that Read and Write the DSPControl Register

There are several instructions that read and write the *DSPControl* register, some explicitly and some implicitly. Like other register resource in the architecture, it is the responsibility of the hardware implementation to ensure that appropriate execution dependency barriers are inserted and the pipeline stalled for read after write dependencies, and other data dependencies that may occur. Table 3-8 shows the instructions that can read and write the DSPControl register and the bits or fields in the register that they read or write.

Table 3-8 Instructions that Read/Write Fields in *DSPControl*

Instruction	Read/Write	*DSPControl* Field
WRDSP	W	All (31:0)
EXTPDP, EXTPDPV, MTHLIP	W	pos (5:0)
ADDSC	W	c (13)
EXTP, EXTPV, EXTPDP, EXTPDPV	W	EFI(14)
See Table 3-5	W	ouflag (23:16)
CMP and CMPU variants	W	ccond (31:24)
RDDSP	R	All (31:0)
BPOSGE32,EXTP, EXTPV, EXTPDP, EXTPDPV, INSV,	R	pos(5:0)
INSV,	R	scount (12:7)
ADDWC	R	c (13)
PICK variants	R	ccond (31:24)

3.14 Arithmetic Exceptions

Under no circumstances do any of the DSP ASE instructions cause an arithmetic exception. Other exceptions are possible, for example, the indexed load instruction could cause an address exception. The specific exceptions caused by the different instructions are available in the per-instruction description page.

MIPS® DSP ASE Instruction Summary

4.1 The MIPS® DSP ASE Instruction Summary

The tables in this chapter list all the instructions in the DSP ASE. For operation details about each instruction, read the per-page descriptions. In each table, the column entitled "Writes GPR / ac / *DSPControl*", indicates the explicit write performed by each instruction. This column indicates the writing of a field in the *DSPControl* register other than the *ouflag* field (which is written by a large number of instructions as a side-effect).

Table 4-1 List of instructions in the MIPS32® **DSP ASE in the *Arithmetic* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data Type	Writes GPR / ac / *DSP Control*	App	Description
ADDQ.PH rd,rs,rt ADDQ_S.PH rd,rs,rt	Pair Q15	Pair Q15	GPR	VoIP SoftM	Add two vectors of 16 bit fractional Q15 data. Without saturation, the result is allowed to wrap around. With saturation, the value is clamped to the maximum and minimum values in the Q15 representation.
ADDQ_S.W rd,rs,rt	Q31	Q31	GPR	Audio	Add with saturation. (Without saturation this instruction is similar to ADDU).
ADDU.QB rd,rs,rt ADDU_S.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Performs element-wise vector unsigned addition of 8-bit values. The saturating version of the instruction saturates to the value of 255.
SUBQ.PH rd,rs,rt SUBQ_S.PH rd,rs,rt	Pair Q15	Pair Q15	GPR	VoIP	Subtract two vectors of 16 bit fractional Q15 data, with or without saturation.
SUBQ_S.W rd,rs,rt	Q31	Q31	GPR	Audio	Subtract with saturation. (Without saturation this instruction is similar to SUBU).
SUBU.QB rd,rs,rt SUBU_S.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Performs element-wise vector subtraction with optional unsigned saturation to 8 bits (that is, saturating to the values of 0 and 255).
ADDSC rd,rs,rt	Signed Word	Signed Word	GPR & *DSPControl*	Audio	Add the two source register and set the carry bit in the *DSPControl* register.
ADDWC rd,rs,rt	Signed Word	Signed Word	GPR	Audio	Add the two source registers with the carry bit from the *DSPControl* register.

Table 4-1 List of instructions in the MIPS32® **DSP ASE in the *Arithmetic* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data Type	Writes GPR / ac / *DSP Control*	App	Description
MODSUB rd,rs,rt	Signed Word	Signed Word	GPR	Misc	This instruction does a modular subtract operation on a given address index value in register rs. Register rt specifies the decrement value in the right-most byte and the maximum value to roll around to when the subtract would yield a negative value. This instruction is expected to be used to address a buffer in a circular fashion, when used in combination with the index load word (LWX) instruction. See Appendix A for instruction details.
RADDU.W.QB rd,rs	Quad Unsigned Byte	Unsigned Word	GPR	Misc	Reduction unsigned add of all the 8-bit elements in the input register and put the 32-bit result in the destination GPR register. The upper bits are zero-filled since this operates on unsigned operands. In the MIPS32 architecture the reduction add is done with the four vector elements. Note that this is unsigned add, so for example if all 4 input values are 0x80 (decimal 128), then the result in rd is 0x200 (decimal 512).
ABSQ_S.PH rd,rt	Pair Q15	Pair Q15	GPR	Misc	This instruction creates the absolute value for the two half-word elements in the register independently. It is saturating, i.e., if the input value is a 1 followed by zeros, the result would be 0 followed by all ones.
ABSQ_S.W rd,rt	Q31	Q31	GPR	Misc	Saturating absolute value for a Q31 data, as specified above. (The non-saturating variant is not needed, since this can be obtained by simply anding the value with a 0 followed by all 1s).
PRECRQ.QB.PH rd,rs,rt	2 Pair Q15	Quad Byte	GPR	Misc	This takes two half-word sized elements from the two source registers, truncates them by dropping the least-significant 8 bits from each, and writing into the destination register. Contents of rs go to the 16-23 and 24-31 bit positions in the destination and the contents of rt go to bit positions 0-7 and 8-15 in rd. This allows an endian-agnostic definition of the instruction.
PRECRQ.PH.W rd,rs,rt PRECRQ_RS.PH.W rd,rs,rt	2 Q31	Pair Half-word	GPR	Misc	Precision reduction instruction is used to take two word-sized values from the source registers, reducing their precision by dropping the least significant 16 bits and writing to the destination register. Contents of rs go to 16-31 of the destination register, and rt goes into bits 0-15 of the rd register, to make the instruction endian-agnostic. The round and saturate version adds 0x8000 to the 32 bit value in rs and rt respectively, saturates on overflow, then truncates to 16 bits and uses that reduced value.
PRECRQU_S.QB.PH rd,rs,rt	2 Pair Q15	Quad Unsigned Byte	GPR	Misc	Takes four fractional Q15 values and reduces them to four unsigned 8-bit values with saturation. That is, a negative value clamps to zero; a positive value greater than 0x7F80 is clamped to 0xFF the maximum positive value, and otherwise, the value is truncated to eight bits after dropping the sign bit.

Table 4-1 List of instructions in the MIPS32® **DSP ASE in the *Arithmetic* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data Type	Writes GPR / ac / *DSP Control*	App	Description
PRECEQ.W.PHL rd,rt PRECEQ.W.PHR rd,rt	Q15	Q31	GPR	Misc	This is the signed version of the precision expand instruction. Note that the half-word is treated as a signed fractional value, hence the expansion operation adds 16 bits in the least-significant position. The expansion is therefore from a Q15 to a Q31 format.
PRECEQU.PH.QBL rd,rt PRECEQU.PH.QBR rd,rt PRECEQU.PH.QBLA rd,rt PRECEQU.PH.QBRA rd,rt	Unsig ned Byte	Q15	GPR	Video	Used to expand a selected subset of unsigned byte elements from the rt operand to a wider precision. A byte from rt is appended with 7 least-significant bits, prepended with a single zero bit and put in the corresponding element position in the destination register. This conversion essentially takes an 8-bit integer to a scaled Q15 format value.
PRECEU.PH.QBL rd,rt PRECEU.PH.QBR rd,rt PRECEU.PH.QBLA rd,rt PRECEU.PH.QBRA rd,rt	Unsig ned Byte	Unsig ned half-w ord	GPR	Video	Used to expand a selected subset of unsigned byte elements from the rt operand to an unsigned half-word. A byte from rt is prepended with eight zeros and put in the corresponding element position in the destination register.

Table 4-2 List of instructions in the MIPS32® **DSP ASE in the *GPR-Based Shift* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data-type	Writes GPR / ac / *DSP Control*	App	Description
SHLL.QB rd, rt, sa SHLLV.QB rd, rt, rs	Quad Unsig ned Byte	Quad Unsig ned Byte	GPR	Misc	Vector left shift operation on unsigned 8-bit elements in rt, inserting zeros into the emptied bits. The shift amount is specified by the low-order 3 bits of sa or the rs register in the variable version. The two upper bits are masked to zero.
SHLL.PH rd, rt, sa SHLLV.PH rd, rt, rs SHLL_S.PH rd, rt, sa SHLLV_S.PH rd, rt, rs	Pair Signe d Half-word	Pair Signe d Half-word	GPR	Misc	Optionally saturating vector left shift operation on the two 16-bit vector elements in rt. If signed overflow occurs during the shift and saturation is specified, then the result is saturated. Supports change of scale with saturation on overflow. Zeros are inserted into the emptied bits. The shift amount is specified by the low-order 4 bits of sa or the rs register in the variable version.
SHLL_S.W rd, rt, sa SHLLV_S.W rd, rt, rs	Signe d Word	Signe d Word	GPR	Misc	Left shift the full 32-bit word putting zeros in the emptied bits. The SLL and SLLV instructions in MIPS32 implement the non-saturating versions of this operation. The shift amount is specified by the low-order 5 bits of sa or the rs register in the variable version.

Table 4-2 List of instructions in the MIPS32® **DSP ASE in the *GPR-Based Shift* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data-type	Writes GPR / ac / *DSP Control*	App	Description
SHRL.QB rd, rt, sa SHRLV.QB rd, rt, rs	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Vector versions of the logical right shift, done element-wise. Zeros are inserted into the emptied bits. The shift amount is specified by the low-order 3 bits of sa or the rs register in the variable version.
SHRA.PH rd, rt, sa SHRAV.PH rd, rt, rs SHRA_R.PH rd, rt, sa SHRAV_R.PH rd, rt, rs	Pair Signed Half-word	Pair Signed Half-word	GPR	Misc	Vector versions of the ordinary arithmetic right shift, done element-wise. The sign bit is duplicated in the emptied bits. The shift amount is specified by the low-order 4 bits of sa or the rs register in the variable version. If rounding is required, then a 1 bit is added to the most significant discarded bit.
SHRA_R.W rd, rt, sa SHRAV_R.W rd, rt, rs	Signed Word	Signed Word	GPR	Video	Arithmetic right shift of a word with rounding as specified above. The shift amount is specified by the low-order 5 bits of sa or the rs register in the variable version.

Table 4-3 List of instructions in the MIPS32® **DSP ASE in the *Multiply* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data-type	Writes GPR / ac / *DSP Control*	App	Description
MULEU_S.PH.QBL rd,rs,rt MULEU_S.PH.QBR rd,rs,rt	Pair Unsigned Byte, Pair Unsigned Halfword,	Pair Unsigned Halfword	GPR	Still Image	This instruction does an unsigned multiply of a pair of bytes with a pair of half-words to produce a pair of half-word results. The two bytes are taken from the left-most or right-most positions of the rs register respectively, and the halfwords come from register rt. Note that this is not a fractional multiply, hence no shifts needed after the multiply. The results of the 8x16 multiply are truncated to 16 bit results and put in the corresponding vector position in the destination register. Saturates to a 16 bit value if the 24-bit result exceeds 0xFFFF.
MULQ_RS.PH rd,rs,rt	Pair Q15	Pair Q15	GPR	Misc	Vector fractional multiply with same-size products with rounding and saturation check. The value is rounded by adding a 1 bit at the position of the most-significant of the bits to be discarded after the multiply. This rounds the result to the nearest value with half-way values rounded up. The saturation checks whether both inputs are -1 and if so saturates to the maximum Q15 value. To stay compliant with the base architecture, this instruction leaves the base HI-LO pair UNPREDICTABLE after the operation. The other DSP ASE accumulators ac1-ac3 must be untouched.

Table 4-3 List of instructions in the MIPS32® **DSP ASE in the *Multiply* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data-type	Writes GPR / ac / *DSP Control*	App	Description
MULEQ_S.W.PHL rd,rs,rt MULEQ_S.W.PHR rd,rs,rt	Pair Q15	Q31	GPR	VoIP	Vector fractional multiply with expansion to full-size products. Note that the products are left-shifted by one bit before being written into the destination register. The result is saturated. Saturation checks for -1 input values, and if so clamps the result to the maximal Q31 representation. To stay compliant with the base architecture, this instruction leaves the base HI-LO pair UNPREDICTABLE after the operation. The other DSP ASE accumulators ac1-ac3 must be untouched.
DPAU.H.QBL DPAU.H.QBR	Pair Bytes	Halfw ord	Acc	Image	Do a dot-product accumulate on a pair of bytes using unsigned multiplication. This is not a fractional multiplication, so there is no shift operation after the multiplies. The two 16 bit multiply results are added and then accumulated into the accumulator.
DPSU.H.QBL DPSU.H.QBR	Pair Bytes	Halfw ord	Acc	Image	Do a dot-product subtract on a pair of bytes using unsigned multiplication. This is not a fractional multiplication, so there is no shift operation after the multiplies. The two 16 bit multiply results are added and then subtracted into the accumulator.
DPAQ_S.W.PH ac,rs,rt	Pair Q15	Q32.3 1	ac	VoIP/ SoftM	Dot product accumulate with fractional 16x16 element-wise multiplication (1 bit left shift after the multiply), with saturate after each multiply to 32 bits. The products are accumulated. The saturation checks whether both inputs are -1, and if so the result of the multiply for that element is clamped to the maximal positive value for Q31 representation. The dot product works on 2 vector elements. The result of the multiplication are Q31 values and these are added into a accumulation with sign-extension to the full width of the accumulator. This implies that with the version that does not saturate, the lower 32 bits of the multiplication result are truncated and used for the addition. Used to compute the imaginary component of 16-bit element multiplication. To get the operands in the right order to compute the imaginary component of a complex multiply, the operands first need to be swapped.
DPSQ_S.W.PH ac,rs,rt	Pair Q15	Q32.3 1	ac	VoIP/ SoftM	Dot product subtract from accumulator. Fractional 16x16 element-wise multiplication, and add all products which are then subtracted from the accumulator. The saturation checks whether both inputs are -1, and if so the result of the multiply for that element is clamped to the maximal positive value for Q31 representation. The dot product works on 2 vector elements.

Table 4-3 List of instructions in the MIPS32® **DSP ASE in the *Multiply* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data-type	Writes GPR / ac / *DSP Control*	App	Description
MULSAQ_S.W.PH ac,rs,rt	Pair Q15	Q32.31	ac	SoftM	Fractional 16x16 multiply the two vector elements, subtract the two results and then accumulate into the accumulator. Used to compute the real component of a complex multiplication. The saturation checks whether both inputs are -1, and if so the result of the multiply for that element is clamped to the maximal positive value for Q31 representation. The operation works on 2 vector elements.
DPAQ_SA.L.W ac,rs,rt	Q31	Q63	ac	Audio	Fractional 32x32 multiplication and accumulate. The result is assumed to be Q63, hence a 1 bit left shift is done before the addition of the elements. The saturation checks for -1 inputs and clamps the result of the multiply, as well as checks if an accumulator overflow or underflow occurs after the add into the accumulator. If it does, then the accumulator is also clamped to the maximal positive or maximal negative value. This extra saturation step after the S, denoted by the A indicates saturate after accumulate (as well as the check on the input -1 values). The dot product works on one element.
DPSQ_SA.L.W ac,rs,rt	Q31	Q63	ac	Audio	Fractional 32x32 multiply to full-size interim results and subtract from the accumulator with saturation on the input values and after the subtract operation on the accumulator. The saturation checks if an accumulator overflow or underflow occurs after the subtract into the accumulator. If it does, then the accumulator is also clamped to the maximal positive or maximal negative value.
MAQ_S.W.PHL ac,rs,rt MAQ_S.W.PHR ac,rs,rt	Q15	Q32.31	ac	SoftM	Vector multiply the one halfword element from each source register and accumulate the result to the specified accumulator. Saturation checks both input values for a -1 value, and if so, clamps to the maximum positive value.
MAQ_SA.W.PHL ac,rs,rt MAQ_SA.W.PHR ac,rs,rt	Q15	Q31	ac	speech	Vector multiply the one halfword element from each source register and accumulate the result to the specified accumulator. Saturation checks both input values for a -1 value, and if so, clamps to the maximum positive value. In addition, this version of the instruction also saturates the result of the accumulate to a Q31 result.

Table 4-4 List of instructions in the MIPS32® **DSP ASE in the *Bit/ Manipulation* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data-type	Writes GPR / ac / *DSP Control*	App	Description
BITREV rd,rt	Unsigned Word	Unsigned Word	GPR	Audio /FFT	Bit reverse the right-most 16 bits in-place. (This instruction would need to be followed by a right shift to bring the relevant FFT butterfly table index to the right-justified position). The upper bits of the register are zeroed.
INSV rt,rs	Unsigned Word	Unsigned Word	GPR	Misc	Like the Release 2 INS instruction, except that the 5 bits for pos and size values are obtained from the *DSPControl* register. size = scount[14:10], and pos = pos[20:16].
REPL.QB rd,imm REPLV.QB rd,rt	Byte	Quad Byte	GPR	Video/ Misc	This instruction is used to replicate a scalar value as vector elements in register rd. If the scalar value is specified in the register rt, then only the least significant 8 bits of rt are used. The immediate form specifies 10 bits, of which the least-significant 8 bits are used. In the MIPS32 architecture, the byte is replicated into 4 vector elements
REPL.PH rd,imm REPLV.PH rd,rt	Signed Half-word	Pair Signed Half-word	GPR	Misc	As above, the least-significant 16 bits are replicated from rt. The immediate form of the instruction uses 10 bits which is right-aligned into the 16-bit field, sign-extending to the full 16 bits of the upper bits. In the MIPS32 architecture, the half-word is replicated into 2 elements.Note that in the immediate form of the instruction, to restore the immediate value to a fractional format, a vector left shift of instruction will be needed after the replicate.

Table 4-5 List of instructions in the MIPS32® **DSP ASE in the *Compare-Pick* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data-type	Writes GPR / ac / *DSP Control*	App	Description
CMPU.EQ.QB rs,rt CMPU.LT.QB rs,rt CMPU.LE.QB rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	*DSPControl*	Video	Vector compare the corresponding elements of rs and rt and record the corresponding boolean result to the 4 least-significant ccond bits in the *DSPControl* register. This is an unsigned comparison.
CMPGU.EQ.QB rd,rs,rt CMPGU.LT.QB rd,rs,rt CMPGU.LE.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Vector compare the corresponding elements of rs and rt and record the corresponding boolean result to the 4 least-significant bits in the *rd* register. This is an unsigned comparison.

Table 4-5 List of instructions in the MIPS32® **DSP ASE in the *Compare-Pick* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data-type	Writes GPR / ac / *DSP Control*	App	Description
CMP.EQ.PH rs,rt CMP.LT.PH rs,rt CMP.LE.PH rs,rt	Pair Signed Half-word	Pair Signed Half-word	*DSPControl*	Misc	Vector compare the corresponding elements and record the boolean result to the 2 least-significant ccond bits in the *DSPControl* register. This is a signed comparison.
PICK.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	This instructions looks at the four least-significant bits of the ccond field and for each bit. if 1, it copies the corresponding element from rs into the corresponding position in the rd register, and if 0, it copies the corresponding value from rt into the rd register position.
PICK.PH rd,rs,rt	Pair Signed Half-word	Pair Signed Half-word	GPR	Misc	Like the above, except that it examines the two least-significant bits of ccond to make the decision.
PACKRL.PH rd,rs,rt	Pair Signed Halfwords	Pair Signed Halfword	GPR	Misc	This is a pack instruction that takes the right halfword from rs and the left half-word from rt and packs them into the left and right positions respectively in the destination register rd.

Table 4-6 List of instructions in the MIPS32® **DSP ASE in the *Accumulator and DSPControl Access* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data-type	Writes GPR / ac / *DSP Control*	App	Description
EXTR.W rt,ac,shift EXTR_R.W rt,ac,shift EXTR_RS.W rt,ac,shift	Q63	Q31	GPR	Misc	Extract bits from the HI-LO pair to a GPR with optional 5 bits of right (logical) shift and with optional rounding and optional saturation on the resulting 32 least significant bits in the HI-LO which are then copied to the destination GPR rt. The shift amount ranges from 0 to +31. The rounding adds one to the most significant discarded bit and can result in a 1 bit overflow in the upper-most 32 bit position. If the _RS option is specified, then this overflow will result in clamping the value to the maximum Q31 value.
EXTR_S.H rt,ac,shift	Q63	Q15	GPR	Misc	This instruction extracts a saturated 16 bit value out of the accumulator with an optional right shift specified as an immediate value.
EXTRV_S.H rt,ac,rs	Q63	Q15	GPR	Misc	This instruction extracts a saturated 16 bit value out of the accumulator with an optional right shift specified in the rs register.

Table 4-6 List of instructions in the MIPS32® **DSP ASE in the *Accumulator and DSPControl Access* sub-class**

Instruction Mnemonics	Input Data-type	Out-put Data-type	Writes GPR / ac / *DSP Control*	App	Description
EXTRV.W rt,ac,rs EXTRV_R.W rt,ac,rs EXTRV_RS.W rt,ac,rs	Q63	Q31	GPR	Misc	The variable version takes the shift value from the rs register. The shift value is the least-significant 5 bits in rs and can range from 0 to +31. The value taken from the accumulator is the right-justified 32 bits.
EXTP rt,ac,size EXTPV rt,ac,rs EXTPDP rt,ac,size EXTPDPV rt,ac,rs	Unsigned DWord	Unsigned Word	GPR/ *DSPControl*	Audio /Video	Extract **size** bits from the accumulator pair HI-LO. The bits are extracted from position **pos** (value in *DSPControl)* and written to the rt register. pos specifies the left-most position of the field to be extracted from the combined HI-LO pair. See Appendix C for details. The size of the field to be extracted is either available as an immediate value or in the register rs in the variable version. Size is 5 bits either specified directly in the instruction or the least-significant 5 bits of rs. The extracted bits are right-justified in the destination GPR and the upper bits of the register are zeroed. In the second version of the instruction, bit-extract-decrement-pos EXTPDP, after extracting the bit field, the value of **pos** in *DSPControl* is decremented by **size**, readying for the next bit extract operation.
SHILO ac,shift SHILOV ac,rs	Unsigned DWord	Unsigned DWord	ac	Misc	Shift the Hi-Lo accumulator pair and leave the result in the accumulator. The instruction uses a 5 bit shift amount to specify a left or right shift. A negative value implies a left shift and a positive value is a right shift, -16 to +15. The variable variant specifies a 6 bit value ranging from -31 to +32.
MTHLIP rs, ac	Unsigned Word	Unsigned Word	ac/ *DSPControl*	Audio /Video	Copy LO to HI, copy rs to LO, and increment the pos field in *DSPcontrol* by 32.
MFHI/MFLO/MTHI/MTLO	Unsigned Word	Unsigned Word	GPR/ac	Misc	Extend these instructions to access the three new HI-LO accumulator pairs.
WRDSP rt,mask	Unsigned Word	Unsigned Word	*DSPControl*	Misc	Used to write specific fields in the *DSPControl* register. This instruction has the ability to only write certain fields in the register based on the value of mask. There is a bit per field, where a mask bit of 1 implies write that field into the *DSPcontrol* register. The fields that correspond to a 0 bit in the mask are unmodified. It is expected that the fields to be written are taken from the corresponding position in the source register. See the per-page instruction description for the correspondence between mask values and fields in the register.

Table 4-6 List of instructions in the MIPS32® **DSP ASE in the *Accumulator and DSPControl Access* sub-class**

Instruction Mnemonics	Input Data-type	Output Data-type	Writes GPR / ac / *DSP Control*	App	Description
RDDSP rt,mask	Unsigned Word	Unsigned Word	GPR	Misc	This instruction is used to read fields in the *DSPControl* register. As above, only certain fields can be read based on the mask value. The read fields are moved to the corresponding position in the destination register. See the per-page instruction description for the correspondence between mask values and fields in the register.

Table 4-7 List of instructions in the MIPS32™ **DSP ASE in the *Indexed-Load* sub-class**

Instruction Mnemonics	Input Data-type	Output Data-type	Writes GPR / ac / *DSP Control*	App	Description
LBUX rd,index(base)	-	Unsigned byte	GPR	Misc	Index byte load, base+(index) address. Loads the byte in the low-order bits of the destination register and zero-extends the result.
LHX rd,index(base)	-	Signed Half-word	GPR	Misc	Index half-word load, base+(index) address. Loads the half-word in the low-order bits of the register and sign-extends the result.
LWX rd, index(base)	-	Signed Word	GPR	Misc	Indexed word load, base+(index) address.

Table 4-8 List of instructions in the MIPS32® **DSP ASE in the *Branch* sub-class**

Instruction Mnemonics	Input Data-type	Output Data-type	Writes GPR / ac / *DSP Control*	App	Description
BPOSGE32 offset	-	-	-	Audio /Video	Branch if the pos value is greater than or equal to integer 32.

Instruction Encoding

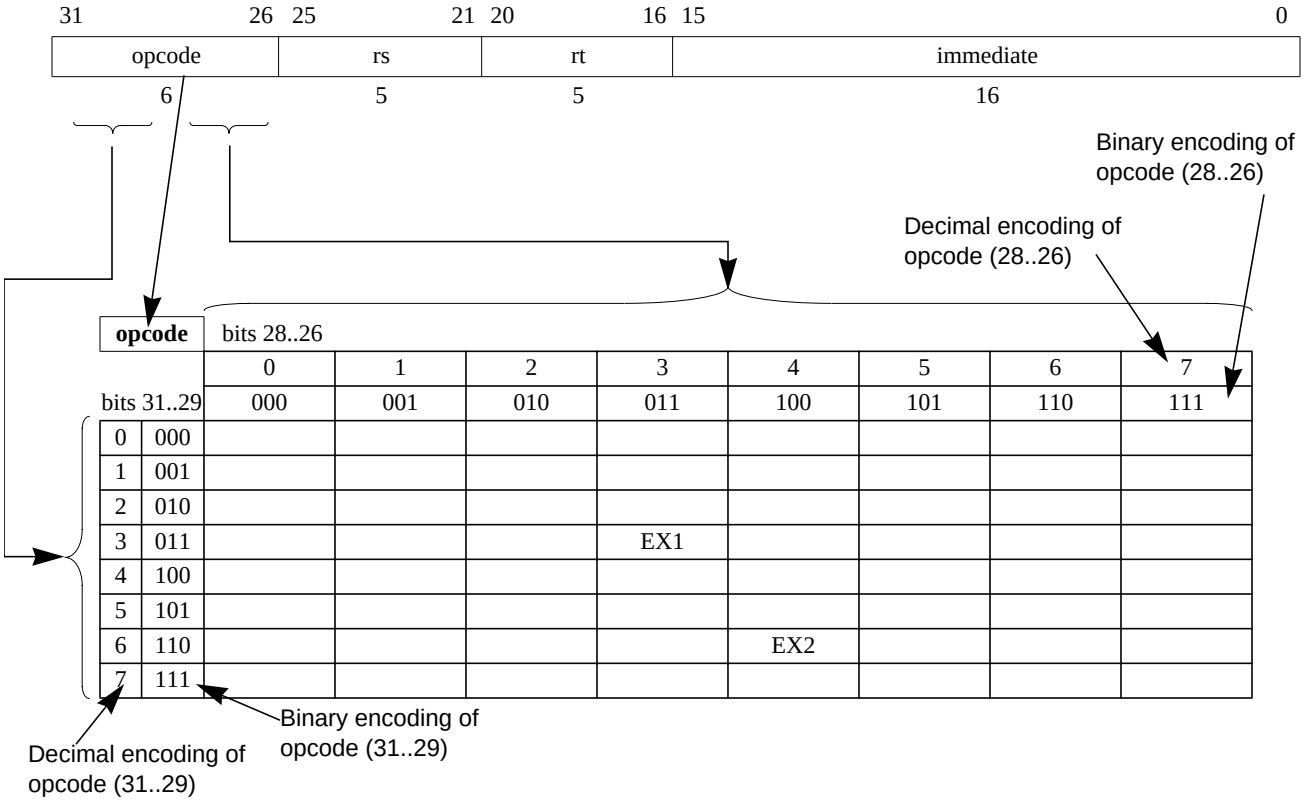
5.1 Instruction Bit Encoding

This chapter describes the bit encoding tables used for the MIPS DSP ASE. Table 5-1 describes the meaning of the symbols used in the tables. These tables only list the instruction encoding for the MIPS DSP ASE instructions. See Volumes I and II of this multi-volume set for a full encoding of all instructions.

Figure 5-1 shows a sample encoding table and the instruction *opcode* field this table encodes. Bits 31..29 of the *opcode* field are listed in the left-most columns of the table. Bits 28..26 of the *opcode* field are listed along the topmost rows of the table. Both decimal and binary values are given, with the first three bits designating the row, and the last three bits designating the column.

An instruction's encoding is found at the intersection of a row (bits 31..29) and column (bits 28..26) value. For instance, the *opcode* value for the instruction labelled EX1 is 33 (decimal, row and column), or 011011 (binary). Similarly, the *opcode* value for EX2 is 64 (decimal), or 110100 (binary).

Figure 5-1 Sample Bit Encoding Table



opcode	bits 28..26	0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000								
1	001								
2	010								
3	011				EX1				
4	100								
5	101								
6	110					EX2			
7	111								

Table 5-1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction must cause a Reserved Instruction Exception.
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encoding if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encoding is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* encoding or coprocessor instruction encoding for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encoding for a coprocessor to which access is not allowed).
σ	Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table.
ε	Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS32 ISA. Software should avoid using these operation or field codes.
⊕	Operation or field codes marked with this symbol are valid for Release 2 implementations of the architecture. Executing such an instruction in a Release 1 implementation must cause a Reserved Instruction Exception.

Table 5-2 MIPS32®DSP ASE Encoding of the Opcode Field

opcode		*bits 28..26*							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000		*REGIMM* δ						
1	001								
2	010								
3	011								*SPECIAL3* δθ
4	100								
5	101								
6	110								
7	111								

The instructions in the MIPS DSP ASE are encoded in the Special3 space under the opcode map as shown in Table 5-2 and Table 5-3. The sub-encoding for individual instructions defined by the MIPS DSP ASE are shown in the following tables in this chapter.

Table 5-3 MIPS32® *SPECIAL3*[1] Encoding of Function Field for DSP ASE Instructions[2]

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000								
1	001			*LX* δ	*	INSV	β	*	*
2	010	*ADDU.QB* δ	*CMPU.EQ.QB* δ	*ABSQ_S.PH* δ	*SHLL.QB* δ	β	β	β	β
3	011	*	*	*	*	*	*	*	*
4	100		*	*	*		*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*DPAQ.W.PH* δ	*	*	*	β	*	*	*
7	111	*EXTR.W* δ	*	*		β	*	*	*

1. Release 2 of the Architecture added the SPECIAL3 opcode. Implementations of Release 1 of the Architecture signaled a Reserved Instruction Exception for this opcode and all function field values shown above.
2. The empty slots in this table are used by Release 2 instructions not shown here, refer to Volume II of this multi-volume specification for these instructions.

Table 5-4 MIPS32® *REGIMM* Encoding of rt Field

rt		*bits 18..16*							
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00					*	*	*	*
1	01						*		*
2	10					*	*	*	*
3	11	*	*	*	*	BPOSGE32	β	*	

Each MIPS DSP ASE instruction sub-class in SPECIAL3 that needs further decoding, is done via the op field as shown in Figure 5-2.

Figure 5-2 SPECIAL3 Encoding for the *ADDU.QB/CMPU.EQ.QB* instruction sub-classes

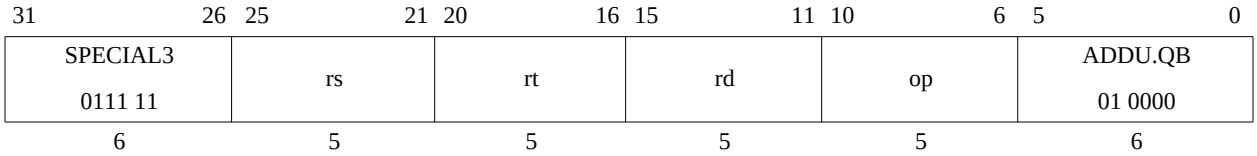


Table 5-5 MIPS32® *ADDU.QB* Encoding of the op Field[1]

op		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	ADDU.QB	SUBU.QB	*	*	ADDU_S.QB	SUBU_S.QB	MULEU_S.PH.QBL	MULEU_S.PH.QBR
1	01	*	*	ADDQ.PH	SUBQ.PH	*	*	ADDQ_S.PH	SUBQ_S.PH
2	10	ADDSC	ADDWC	MODSUB	*	RADDU.W.QB	*	ADDQ_S.W	SUBQ_S.W
3	11	*	*	*	*	MULEQ_S.W.PHL	MULEQ_S.W.PHR	*	MULQ_RS.PH

1. The op field is decoded to identify the final instructions. Entries in this table with no mnemonic are reserved for future use by MIPS Technologies and may or may not cause a Reserved Instruction exception.

Table 5-6 MIPS32® *CMPU.EQ.QB* Encoding of the op Field

op		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	CMPU.EQ.QB	CMPU.LT.QB	CMPU.LE.QB	PICK.QB	CMPGU.EQ.QB	CMPGU.LT.QB	CMPGU.LE.QB	*
1	01	CMP.EQ.PH	CMP.LT.PH	CMP.LE.PH	PICK.PH	PRE-CRQ.QB.PH	*	PACKRL.PH	PRECRQU_S.QB.PH
2	10	*	*	*	*	PRE-CRQ.PH.W	PRECRQ_RS.PH.W	*	*
3	11	*	*	*	*	*	*	*	*

Figure 5-3 SPECIAL3 Encoding for the *ABSQ_S.PH* instruction sub-class without immediate field

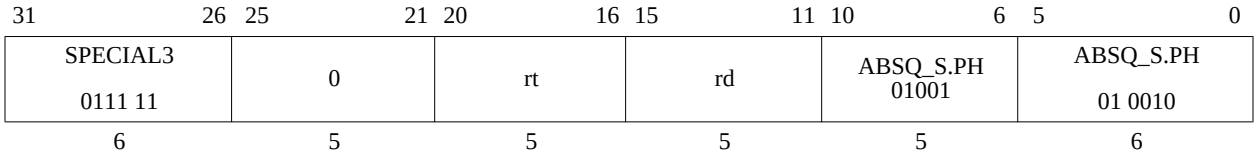


Figure 5-4 SPECIAL3 Encoding for the *ABSQ_S.PH* instruction sub-class with the immediate field

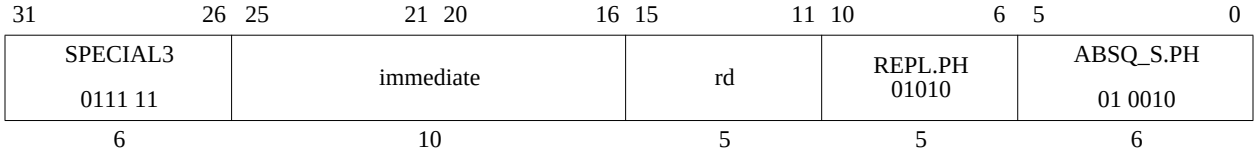


Table 5-7 MIPS32® *ABSQ_S.PH* Encoding of the op Field

op		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	*	*	REPL.QB	REPLV.QB	PRE-CEQU.PH.QBL	PRE-CEQU.PH.QBR	PRE-CEQU.PH.QBLA	PRE-CEQU.PH.QBRA
1	01	*	ABSQ_S.PH	REPL.PH	REPLV.PH	PRE-CEQ.W.PHL	PRE-CEQ.W.PHR	*	*
2	10	*	ABSQ_S.W	*	*	*	*	*	*
3	11	*	*	*	BITREV	PRE-CEU.PH.QBL	PRE-CEU.PH.QBR	PRE-CEU.PH.QBLA	PRE-CEU.PH.QBRA

Figure 5-5 SPECIAL3 Encoding for the *SHLL.QB* instruction sub-class

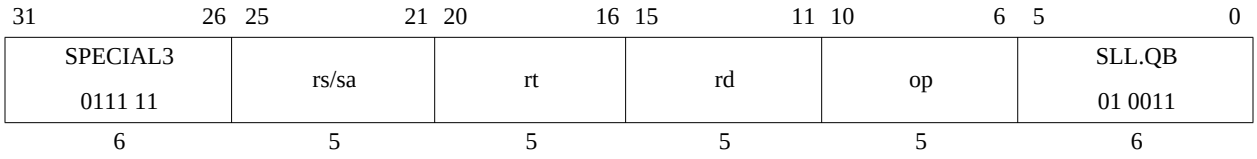


Table 5-8 MIPS32® *SHLL.QB* Encoding of the op Field

op		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	SHLL.QB	SHRL.QB	SHLLV.QB	SHRLV.QB	*	*	*	*
1	01	SHLL.PH	SHRA.PH	SHLLV.PH	SHRAV.PH	SHLL_S.PH	SHRA_R.PH	SHLLV_S.PH	SHRAV_R.PH
2	10	*	*	*	*	SHLL_S.W	SHRA_R.W	SHLLV_S.W	SHRAV_R.W
3	11	*	*	*	*	*	*	*	*

For the LX sub-class of instructions, the format to interpret the op field is similar to the instructions above, with the exception that the rs and rt fields are named to be the base and index fields respectively for the indexed load operation. The instruction format is shown in Figure 5-6.

Figure 5-6 SPECIAL3 Encoding for the *LX* instruction sub-class

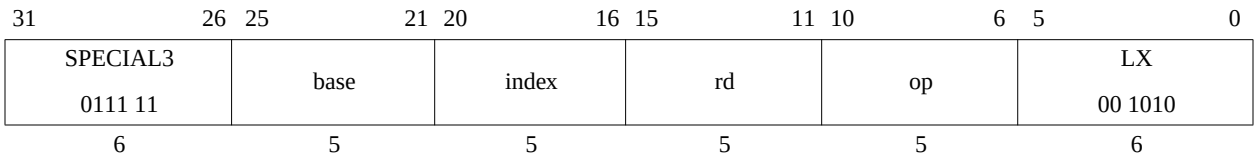


Table 5-9 MIPS32® *LX* Encoding of the op Field

op		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	LWX	*		*	LHX	*	LBUX	*
1	01	β	*	*	*	*	*	*	*
2	10	*	*	*	*	*	*	*	*
3	11	*	*	*	*	*	*	*	*

The sub-class of DPAQ.W.PH instruction targets one of the accumulators for the destination. These instructions use the lower bits of the rd field of the opcode to specify the accumulator number which can range from 0 to 3. This format is shown in Figure 5-7.

Figure 5-7 SPECIAL3 Encoding for the *DPAQ.W.PH* instruction sub-class

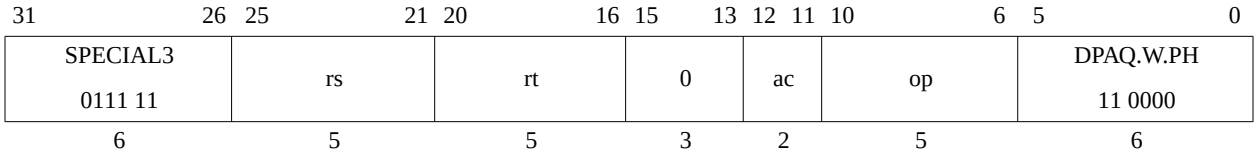


Table 5-10 MIPS32® *DPAQ.W.PH* Encoding of the op Field

op		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	*	*	*	DPAU.H.QBL	DPAQ_S.W.PH	DPSQ_S.W.PH	MULSAQ_S.W.PH	DPAU.H.QBR
1	01	*	*	*	DPSU.H.QBR	DPAQ_SA.L.W	DPSQ_SA.L.W	*	DPSU.H.QBL
2	10	MAQ_SA.W.PHL	*	MAQ_SA.W.PHR	*	MAQ_S.W.PHL	*	MAQ_S.W.PHR	*
3	11	*	*	*	*	*	*	*	*

The *EXTR.W* sub-class is an assortment that has three types of instructions:

1. In the first one, the destination is a GPR and this is specified by the rt field in the opcode, as shown in Figure 5-8. The source is an accumulator and this comes from the right-most 2 bits of the rd field, again, as shown in the figure. When a second source must be specified, then the rs field is used. The second value could be a 5-bit immediate or a variable from a GPR. The first and the second rows of Table 5-11 show this type of instruction.
2. The RDDSP and WRDSP instructions specify one immediate 6 bit mask field and a GPR that holds both the position and size values, as seen in Figure 5-9.
3. The MTHLIP instruction copies the LO part of the specified accumulator to the HI, the GPR contents to LO. In this case, the source rs field is used and the destination is specified by ac, which is both a source and destination, as shown in Figure 5-10. The SHILO and SHILOV instructions which shift the HI-LO pair and leave the result in the HI-LO register pair is a variant that does not use the source rs register. The shift amount can be specified as an immediate value or in the rs register as a variable value.

Figure 5-8 SPECIAL3 Encoding Example for the *EXTR.W* instruction sub-class type 1

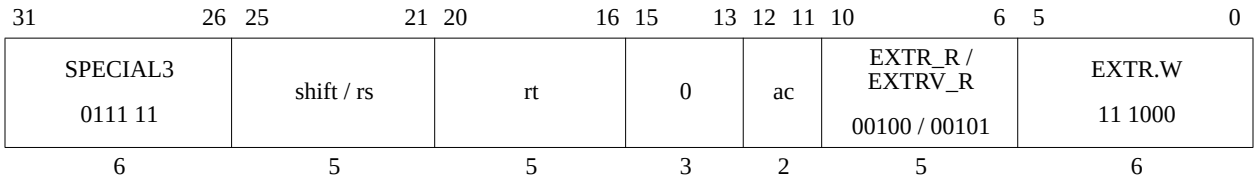


Figure 5-9 SPECIAL3 Encoding Example for the *EXTR.W* instruction sub-class type 2

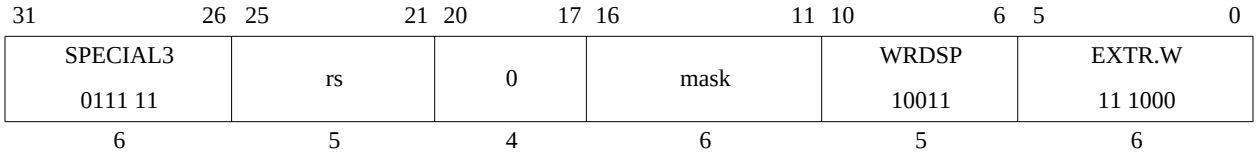


Figure 5-10 SPECIAL3 Encoding Example for the *EXTR.W* instruction sub-class type 3

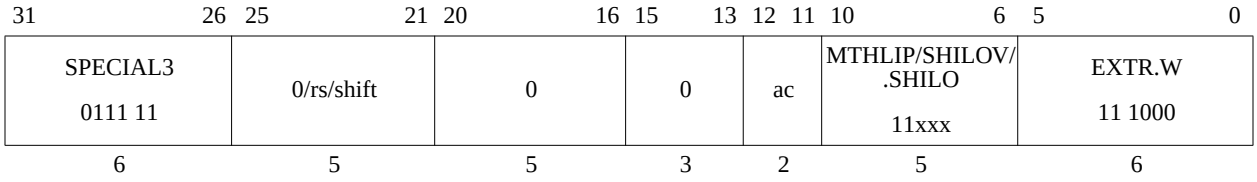


Table 5-11 MIPS32® *EXTR.W* Encoding of the op Field

op		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00	EXTR.W	EXTRV.W	EXTP	EXTPV	EXTR_R.W	EXTRV_R.W	EXTR_RS.W	EXTRV_RS.W
1	01	*	*	EXTPDP	EXTPDPV	*	*	EXTR_S.H	EXTRV_S.H
2	10	*	*	RDDSP	WRDSP	*	*	*	*
3	11	*	*	SHILO	SHILOV	*	*	*	MTHLIP

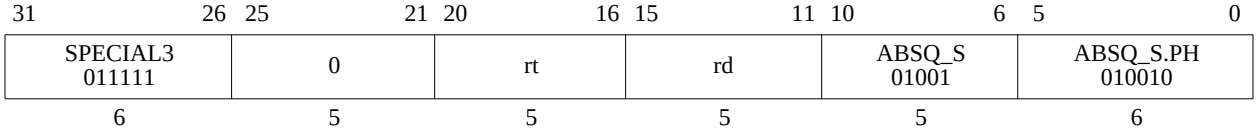
Finally, the opcode change for the MFHI and MTLO instructions requires the specification of the accumulator number. For the MTHI and MTLO instructions, the change will use bits 11 and 12 of the opcode to specify the accumulator, where the value of 0 provides backwards compatibility and refers to the original Hi-Lo pair. For the MFHI and MFLO instructions, the change will use bits 21 and 22 to encode the accumulator, and zero is the original pair as before.

The MIPS® DSP ASE Instruction Set

6.1 Compliance and Subsetting

There are no instruction subsets allowed for the MIPS DSP ASE--all instructions must be implemented with all data format types as shown. The instructions are all listed here in alphabetical order.

Find Absolute Value of Two Fractional Halfwords — ABSQ_S.PH



Format: `ABSQ_S.PH rd, rt` **MIPSDSP**

Purpose:

Find the absolute value of two fractional vector elements with saturation.

Description: `rd ← sat16(abs(rt`$_{31..16}$`)) || sat16(abs(rt`$_{15..0}$`))`

The absolute value of two fractional Q15 vector element values in register *rt* are respectively obtained. For saturation, the input value is checked for a one followed by all zeros (the minimum value of -1). In this case, the result is clamped to the maximum value of a zero followed by all ones before writing to the destination register *rd*. The saturation check is performed independently on both vector elements.

This instruction writes bit 20 in the ouflag field in the *DSPControl* register if saturation is needed for any one of the vector elements.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

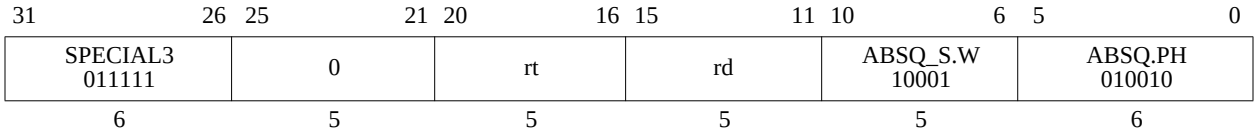
Operation:

```
ABSQ_S.PH
    Setouflag(GPR[rt], satcond1, satcond2)
    if satcond1 then
       temp1 ← 0x7FFF
    else if (temp1_31 = 1) then
       temp1 ← -GPR[rt]_31..16
    else
       temp1 ← GPR[rt]_31..16
    endif
    if satcond2 then
       temp2 ← 0x7FFF
    else if (temp2_31 = 1)
       temp2 ← -GPR[rt]_15..0
    else
       temp2 ← GPR[rt]_15..0
    endif
    GPR[rd] ← temp1_15..0 || temp2_15..0

function Setouflag(rt, satcond1, satcond2)
    satcond1 ← (rt_31..16 = 0x8000)
    satcond2 ← (rt_15..0 = 0x8000)
    if (satcond1 or satcond2) then
       DSPControl_ouflag:20 ← 1
    endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Find Absolute Value of a Fractional Word with Saturation — ABSQ_S.W



Format: `ABSQ_S.W rd, rt` **MIPSDSP**

Purpose:

Find the absolute value of a fractional Q31 value with 32 bit saturation.

Description: `rd ← sat32(abs(rt))`

The absolute value of the fractional Q31 value in register *rt* is obtained. Saturation is needed when the input value is one followed by all zeros (the minimum value of -1). In this case, the result is clamped to the maximum value of a zero followed by all ones before writing to the destination register *rd*.

This instruction writes bit 20 in the ouflag field in the *DSPControl* register if saturation is needed.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

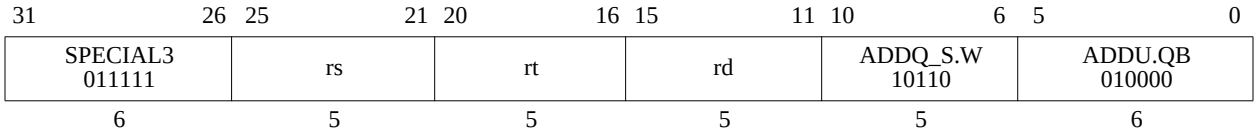
Operation:

```
satcond ← (GPR[rt] = 0x80000000)
if (satcond) then
   DSPControl_ouflag:20 ← 1
   GPR[rd] ← 0x7FFFFFFF
else
   if (GPR[rt]_32 = 1) then
      temp ← -GPR[rt]
   else
      temp ← GPR[rt]
   endif
   GPR[rd] ← temp
endif
```

Exceptions:

Reserved Instruction, DSP Disabled

Add Fractional Word with Saturation — ADDQ_S.W



Format: `ADDQ_S.W rd, rs, rt` **MIPSDSP**

Purpose:

Add two fractional Q31 values with 32 bit saturation.

Description: `rd ← sat32(rs + rt)`

The fractional Q31 value in registers rs and rt are added. Signed saturated arithmetic is performed, where overflow and underflow clamp to the largest (0x7FFFFFFF) or the smallest (0x80000000) representable value respectively before writing the destination register rd.

This instruction, on an overflow or underflow, writes bit 20 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
temp ← (GPR[rs]31 || GPR[rs]31..0) + (GPR[rt]31 || GPR[rt]31..0)
if (temp32 ≠ temp31) then
   DSPControlouflag:20 ← 1
   if (temp32 = 0) then
      temp ← 0x7FFFFFFF
   else
      temp ← 0x80000000
   endif
endif
GPR[rd] ← temp
```

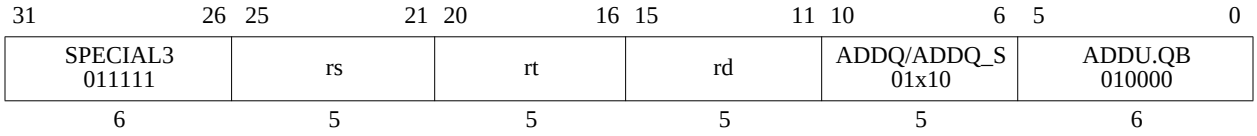
Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

To do a two’s complement add of two words without saturation, use the MIPS32 ADDU instruction.

Add Vector Fractional Half-Words — ADDQ.PH



Format: `ADDQ.PH rd, rs, rt` **MIPSDSP**

`ADDQ_S.PH rd, rs, rt` **MIPSDSP**

Purpose:

Add two vector paired-half fractional Q15 values with optional saturation to obtain a Q15 paired-half result.

Description: `rd ← sat16(rs`$_{31:16}$ `+ rt`$_{31:16}$`) || sat16(rs`$_{15:0}$ `+ rt`$_{15:0}$`)`

The two vector fractional Q15 values in registers rs and rt are added separately. For the saturate version of the instruction, signed saturated arithmetic is performed, where overflow and underflow clamp to the largest (0x7FFF) or the smallest (0x8000) representable value respectively before writing the destination register rd. This saturation is performed separately for each vector operation.

This instruction, on an overflow or underflow of any of the vector operations, writes bit 20 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

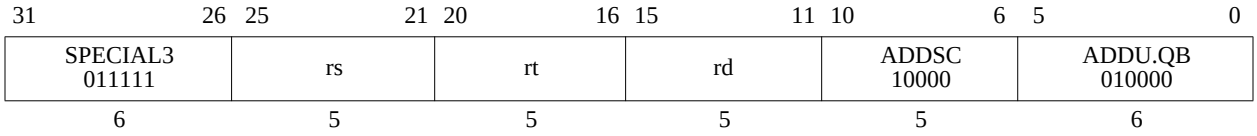
```
ADDQ.PH
    Add(tempw, tempt)
    Setouflag(tempw, tempt)
    GPR[rd] ← tempw15..0 || tempt15..0

ADDQ_S.PH
    Add(tempw, tempt)
    Setouflag(tempw, tempt, wflag, tflag)
    if (wflag) then
        if (tempw16 = 0) then
            tempw ← 0x7FFF
        else
            tempw ← 0x8000
        endif
    endif
    if (tflag) then
        if (tempt16 = 0) then
            tempt ← 0x7FFF
        else
            tempt ← 0x8000
        endif
    endif
    GPR[rd] ← tempw15..0 || tempt15..0

function Add(tempw, tempt)tempw ← (GPR[rs]31 || GPR[rs]31..16) + (GPR[rt]31 ||
GPR[rt]31..16)
    tempt ← (GPR[rs]15 || GPR[rs]15..0) + (GPR[rt]15 || GPR[rt]15..0)
```

```
endfunction Add

function Setouflag(w, t, wf, wt)
   if (w16  ≠ w15) then
      DSPControlouflag:20 ← 1
      wf ← 1
   endif
   if (t16  ≠ t15) then
      DSPControlouflag:20 ← 1
      wt ← 1
   endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Add Unsigned Word and Set Carry Bit — ADDSC



Format: `ADDSC rd, rs, rt` **MIPSDSP**

Purpose:

Add two 32-bit values and set the carry bit in the *DSPControl* register if the addition generates a carry out bit.

Description: *DSPControl*`[c],rd ← rs + rt`

The 32-bit value in GPR rt is added to the 32-bit value in GPR rs and the 32-bit result placed into GPR rd. The carry bit result out of the addition operation is written to bit 13 (the *c* field) of the *DSPControl* register.

This instruction does not modify the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

Operation:

```
temp ← (0 || GPR[rs]31..0) + (0 || GPR[rt]31..0)
DSPControlc ← temp32
GPR[rd] ← temp31..0
```

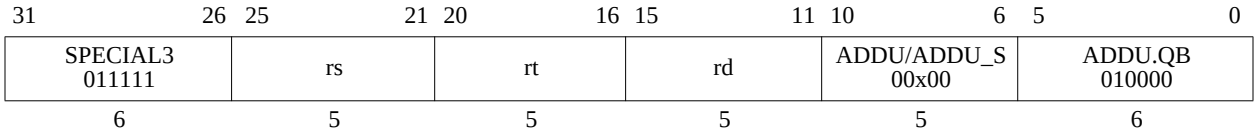
Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

Note that this is really two’s complement (modulo) arithmetic on the two integer values, where the overflow is preserved in architectural state. The ADDWC instruction can be used to do an add using this carry bit. The intended use of these two instructions is to do 64-bit add/subtract using two 32-bit GPRs that hold one 64-bit value, to be added/subtracted to another two GPRs that hold the second 64-bit value.

Unsigned Add Vector Quad-Bytes — ADDU.QB



Format: `ADDU.QB rd, rs, rt` **MIPSDSP**

`ADDU_S.QB rd, rs, rt` **MIPSDSP**

Purpose:

Add unsigned two vectors with byte values with optional saturation.

Description: `rd ← sat8(rs`$_{31:24}$`+rt`$_{31:24}$`) || sat8(rs`$_{23:16}$`+rt`$_{23:16}$`) || sat8(rs`$_{15:8}$`+rt`$_{15:8}$`) || sat8(rs`$_{7:0}$`+rt`$_{7:0}$`)`

Two vector quad-byte values in registers rs and rt are added separately. For the saturate version of the instruction, unsigned saturated arithmetic is performed, where an overflow clamps to the largest (0xFF or decimal 255) representable value, before writing the destination register rd.

This instruction, on an overflow, writes bit 20 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

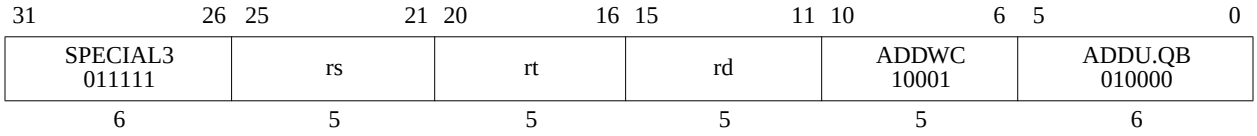
```
ADDU.QB
    Add(tempw, tempr, temps, tempt)
    Setouflag(tempw, tempr, temps, tempt)
    GPR[rd] ← tempw7..0 || tempr7..0 || temps7..0 || tempt7..0

ADDU_S.QB
    Add(tempw, tempr, temps, tempt)
    Setouflag(tempw, tempr, temps, tempt)
    if (tempw8  = 1) then
        tempw ← 0xFF
    endif
    if (tempr8  = 1) then
        tempr ← 0xFF
    endif
    if (temps8  = 1) then
        temps ← 0xFF
    endif
    if (tempt8 = 1) then
        tempt ← 0xFF
    endif
    GPR[rd] ← tempw7..0 || tempr7..0 || temps7..0 || tempt7..0

function Add(tempw, tempr, temps, tempt)
    tempw ← (0 || GPR[rs]31..24) + (0 || GPR[rt]31..24)tempr ← (0 || GPR[rs]23..16) +
(0 || GPR[rt]23..16)
    temps ← (0 || GPR[rs]15..8)  + (0 || GPR[rt]15..8)tempt ← (0 || GPR[rs]7..0) + (0
|| GPR[rt]7..0)
endfunction Add
```

```
function Setouflag(w, r, s, t)
   if ((w8  = 1) or (r8  = 1) or (s8 = 1) or (t8  = 1)) then
      DSPControlouflag:20 ← 1
   endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Add Unsigned Word with Carry Bit — ADDWC



Format: `ADDWC rd, rs, rt` **MIPSDSP**

Purpose:

Add two 32-bit values with the carry bit in the *DSPControl* register.

Description: `rd ← rs + rt + DSPControl[c]`

The 32-bit value in GPR rt is added to the 32-bit value in GPR rs and the carry bit in the *DSPControl* register. The 32-bit result is placed into GPR rd.

On an overflow or underflow, this instruction will set bit 20 of the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

Operation:

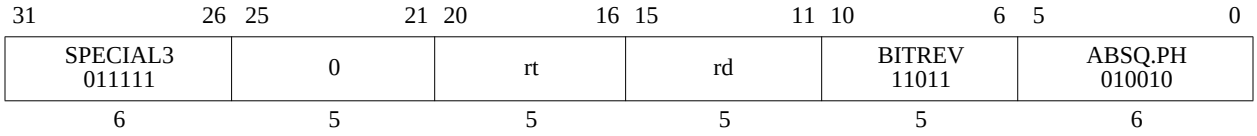
```
temp ← (GPR[rs]31..0 + GPR[rt]31..0 + DSPControlc
if (temp32 ≠ temp31) then
   DSPControlouflag:20 ← 1
endif
GPR[rd] ← temp31..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Bit Reverse a Halfword

BITREV



Format: `BITREV rd, rt` **MIPSDSP**

Purpose:

To reverse the bits of the least significant halfword in a register.

Description: $rd \leftarrow 0^{16} \mid\mid rt_{0..15}$

The least-significant halfword value in register *rt* is bit-reversed into the lower halfword position in the destination register *rd*. And upper 16 bits of the destination register are zero-filled.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

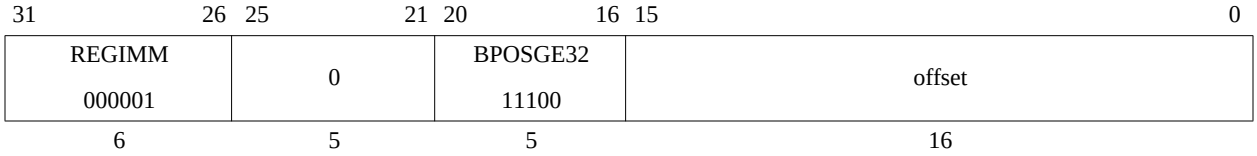
Operation:

```
temp ← GPR[rt]0..15
GPR[rd] ← 016 || temp15..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Branch on Greater Than or Equal to Value 32 in the DSPControl pos Field — BPOSGE32



Format: `BPOSGE32 offset` **MIPSDSP**

Purpose:

To test the *pos* value in *DSPControl* for greater than or equal to 32 then do a PC-relative conditional branch

Description: `if DSPControl[pos] ≥ 32 then branch`

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of the pos field in bits 5..0 of the DSPControl register is greater than or equal to integer value 32, branch to the effective target address after the instruction in the delay slot is executed. This is a signed compare.

Restrictions:

Processor operation is **UNPREDICTABLE** if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:

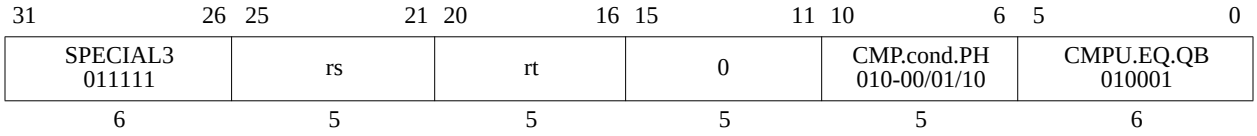
```
I:      target_offset ← sign_extend(offset || 0^2)
        condition ← DSPControl_pos ≥ 32
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.



Format: `CMP.EQ.PH rs, rt` **MIPSDSP**
`CMP.LT.PH rs, rt` **MIPSDSP**
`CMP.LE.PH rs, rt` **MIPSDSP**

Purpose:

Signed compare two vectors of 2 halfwords each and record the boolean results in condition code bits.

Description: $DSPControl[ccond]_{25,24} \leftarrow (rs_{31..16}\ cond\ rt_{31..16})\ ||\ (rs_{15..0}\ cond\ rt_{15..0})$

The two vectors in registers *rs* and *rt* with 2 halfword values are respectively compared as signed 16-bit values, and the resulting 1 bit boolean results are written to the *DSPControl* register's condition code field into the 2 bits in the right-most position.

Restrictions:

No data-dependent exceptions are possible.

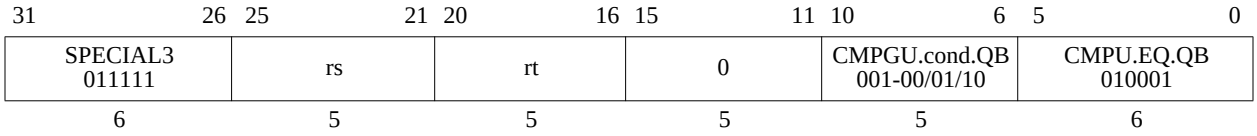
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
cc1 ← GPR[rs]31..16 <signed EQ/LT/LE> GPR[rt]31..16
cc2 ← GPR[rs]15..0 <signed EQ/LT/LE> GPR[rt]15..0
DSPControlcc:25..24 ← cc10 || cc20
DSPControlcc:27..26 ← UNPREDICTABLE
```

Exceptions:

Reserved Instruction, DSP Disabled



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL3 011111	rs	rt	0	CMPGU.cond.QB 001-00/01/10	CMPU.EQ.QB 010001
6	5	5	5	5	6

Format: `CMPGU.EQ.QB rd, rs, rt` **MIPSDSP**
`CMPGU.LT.QB rd, rs, rt` **MIPSDSP**
`CMPGU.LE.QB rd, rs, rt` **MIPSDSP**

Purpose:

Unsigned compare two vectors of 4 bytes each and record the boolean results in condition code bits which are written to the specified destination GPR.

Description: $GPR[rd]_{3..0} \leftarrow (rs_{31..24}\ cond\ rt_{31..24})\ ||\ (rs_{23..16}\ cond\ rt_{23..16})\ ||\ (rs_{15..8}\ cond\ rt_{15..8})\ ||\ (rs_{7..0}\ cond\ rt_{7..0})$

The two vectors in registers *rs* and *rt* with four byte values are respectively compared as unsigned 8-bit values, and the resulting 1 bit boolean results are written to the least significant bits of the *rd* destination register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

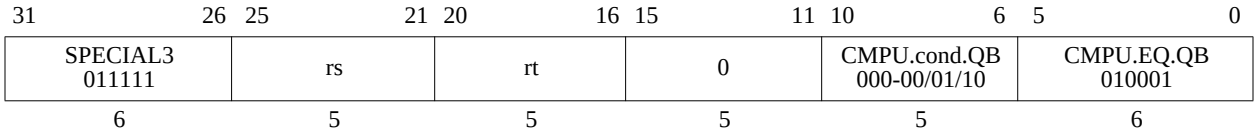
Operation:

```
cc1 ← GPR[rs]31..24 <unsigned EQ/LT/LE> GPR[rt]31..24
cc2 ← GPR[rs]23..16 <unsigned EQ/LT/LE> GPR[rt]23..16
cc3 ← GPR[rs]15..8 <unsigned EQ/LT/LE> GPR[rt]15..8
cc4 ← GPR[rs]7..0 <unsigned EQ/LT/LE> GPR[rt]7..0
GPR[rd]31..0 ← 0^28 || cc1_0 || cc2_0 || cc3_0 || cc4_0
```

Exceptions:

Reserved Instruction, DSP Disabled

Compare Unsigned Vector of Four Bytes — CMPU.cond.QB



Format:

```
CMPU.EQ.QB     rs, rt      MIPSDSP
CMPU.LT.QB     rs, rt      MIPSDSP
CMPU.LE.QB     rs, rt      MIPSDSP
```

Purpose:

Unsigned compare two vectors of 4 bytes each and record the boolean results in condition code bits.

Description: $DSPControl[ccond]_{27..24} \leftarrow (rs_{31..24}\ cond\ rt_{31..24})\ ||\ (rs_{23..16}\ cond\ rt_{23..16})\ ||\ (rs_{15..8}\ cond\ rt_{15..8})\ ||\ (rs_{7..0}\ cond\ rt_{7..0})$

The two vectors in registers *rs* and *rt* with four byte values are respectively compared as unsigned 8-bit values, and the resulting 1 bit boolean results are written to the *DSPControl* register's 4-bit condition code field.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

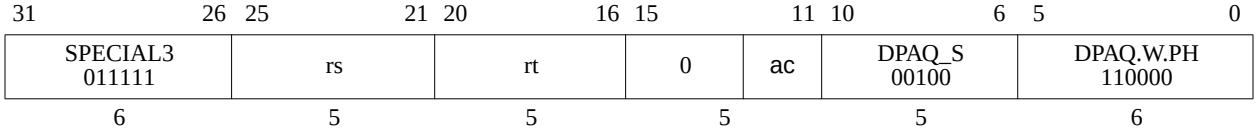
Operation:

```
cc1 ← GPR[rs]31..24 <unsigned EQ/LT/LE> GPR[rt]31..24
cc2 ← GPR[rs]23..16 <unsigned EQ/LT/LE> GPR[rt]23..16
cc3 ← GPR[rs]15..8 <unsigned EQ/LT/LE> GPR[rt]15..8
cc4 ← GPR[rs]7..0 <unsigned EQ/LT/LE> GPR[rt]7..0
DSPControlcc ← cc10 || cc20 || cc30 || cc40
```

Exceptions:

Reserved Instruction, DSP Disabled

Dot Product with Accumulate on Vector Fractional Half-Word Elements — DPAQ_S.W.PH



Format: `DPAQ_S.W.PH ac, rs, rt` **MIPSDSP**

Purpose:

To generate the dot-product of two fractional vector elements using full-size intermediate products and then accumulate into the specified accumulator register.

Description: `ac ← ac + sat32(dot-product(rs[i],rt[i]))`

Four Q15 values from respective vector positions within registers *rt* and *rs* are first multiplied and left-shifted by 1 bit to generate two Q31 results. These products are first added to each other to generate the dot-product result, and then added into the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. This result is a Q32.31 value, i.e., Q31 values are accumulated with 32 integer guard bits that are available in the *HI/LO* register pair. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

To saturate, the instruction checks the multiplicands to see if both are -1, if so, the result for that multiply operation is saturated to the maximum positive value 0x7FFFFFFF. No other saturation is performed.

This instruction can set bits 19..16 of the ouflag field in the *DSPControl* register depending on the *ac* specified. Bit 16 is set for *ac0*, bit 17 for *ac1*, and so on. Note that one of these bits will be set if the operation saturates. Note that the operation overflows if any one of the vector operations overflow.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

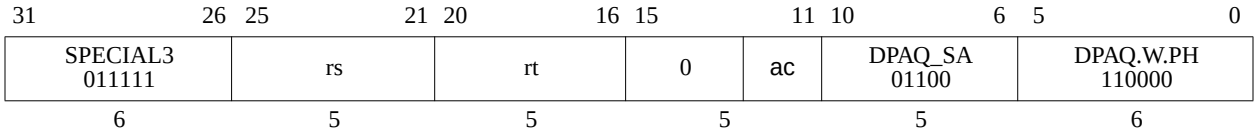
Operation:

```
DPAQ_S.W.PH
    Setouflag(GPR[rs], GPR[rt], satcond1, satcond2)
    if satcond1 then
        temp1[31..0] ← 0x7FFFFFFF
    else
        temp1[31..0] ← (GPR[rs][31..16] * GPR[rt][31..16]) << 1
    endif
    if satcond2 then
        temp2[31..0] ← 0x7FFFFFFF
    else
        temp2[31..0] ← (GPR[rs][15..0] * GPR[rt][15..0]) << 1
    endif
    HI[ac]||LO[ac] ← HI[ac]||LO[ac]+sign_extend(temp1[31..0])+sign_extend(temp2[31..0])

function Setouflag(s, t, satcond1, satcond2)
    satcond1 ← (s[31..16] = 0x8000) and (t[31..16] = 0x8000)
    satcond2 ← (s[15..0] = 0x8000) and (t[15..0] = 0x8000)
    if (satcond1 or satcond2) then
        DSPControl[ouflag:16+ac] ← 1
    endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Dot Product with Accumulate on Fractional Word Element — DPAQ_SA.L.W



Format: `DPAQ_SA.L.W ac, rs, rt` **MIPSDSP**

Purpose:

To multiply two fractional words using full-size intermediate products and then accumulate into the specified accumulator register with saturation.

Description: `ac ← sat64(ac + sat32(rs * rt))`

Two 32 bit Q31 word from registers *rt* and *rs* are first multiplied to a 62-bit product, which is left-shifted by 1 bit to get a Q63 result. This product is then accumulated into the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

To saturate, the instruction checks the multiplicands to see if both are -1, if so, the result for that multiply operation is saturated to the maximum positive value. In addition, the final accumulation result is also saturated to a Q63 result.

This instruction can set bits 19..16 depending on the *ac* specified in the ouflag field in the *DSPControl* register. Bit 16 is set for *ac0*, bit 17 for *ac1*, and so on. Note that one of these bits will be set if the operation saturates.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

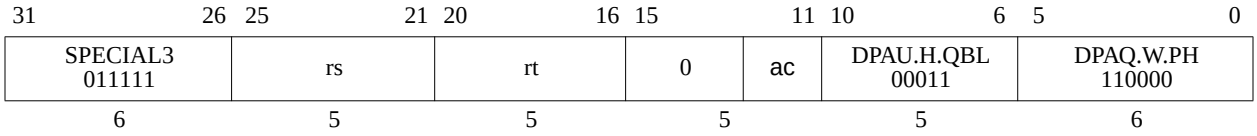
```
DPAQ_SA.L.W
    Setouflag(GPR[rs], GPR[rt], satcond)
    if satcond then
        temp1 ← 0x7FFFFFFFFFFFFFFF
    else
        temp1_63..0 ← (GPR[rs]_31..0 * GPR[rt]_31..0) << 1
    endif
    tempx_63..0 ← HI[ac] || LO[ac]
    temp2_64..0 ← (tempx_63 || tempx_63..0) + (temp1_63 || temp1_63..0)
    if temp2_64 ≠ temp2_63 then
        DSPControl_ouflag:16+ac ← 1
        if temp2_64 = 0 then
            HI[ac]||LO[ac] ← 0x7FFFFFFFFFFFFFFF
        else
            HI[ac]||LO[ac] ← 0x8000000000000000
        endif
    else
        HI[ac]||LO[ac] ← temp2_63..0
    endif

function Setouflag(s, t, satcond)
    satcond ← (s_31..0 = 0x80000000) and (t_31..0 = 0x80000000)
    if satcond then
        DSPControl_ouflag:16+ac ← 1
    endif
```

```
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Dot Product with Accumulate on Vector Byte Elements — DPAU.H.QBL



Format: `DPAU.H.QBL ac, rs, rt` **MIPSDSP**

Purpose:

To generate the dot-product of two bytes from the left-aligned vector positions using full-size intermediate products, followed by a further addition of the resulting dot-product to the specified accumulator register.

Description: `ac ← ac + dot-product(rs[i],rt[i])`

Two unsigned byte values from the left-aligned vector positions within registers *rt* and *rs* are first multiplied using unsigned arithmetic to generate two 16-bit results. These products are first added to each other to generate the dot-product result, and then accumulated into the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

This instructions does not set the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

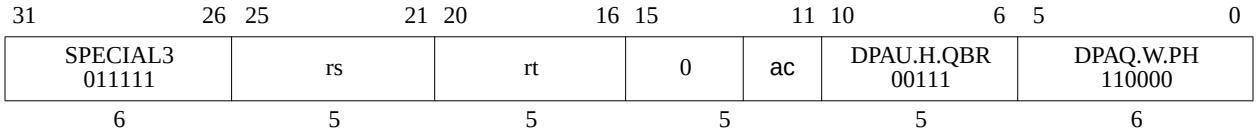
The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
temp1 ← (GPR[rs]31..24 * GPR[rt]31..24)
temp2 ← (GPR[rs]23..16 * GPR[rt]23..16)
HI[ac]||LO[ac] ← HI[ac]||LO[ac] + temp115..0 + temp215..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `DPAU.H.QBR ac, rs, rt` **MIPSDSP**

Purpose:

To generate the dot-product of two bytes from a right-aligned vector position using full-size intermediate products, followed by a further addition of the resulting dot-product to the specified accumulator register.

Description: `ac ← ac + dot-product(rs[i],rt[i])`

Two unsigned byte values from the right-aligned vector positions within registers *rt* and *rs* are first multiplied using unsigned arithmetic to generate two 16-bit results. These products are first added to each other to generate the dot-product result, and then accumulated into the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

This instructions does not set the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

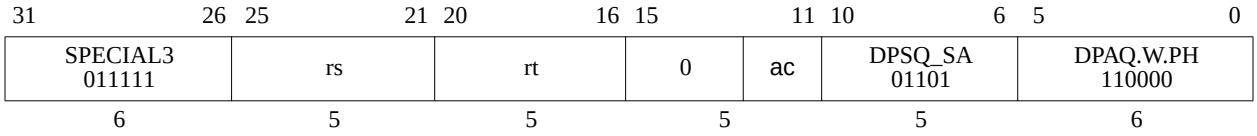
The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
temp1 ← (GPR[rs]15..8 * GPR[rt]15..8)
temp2 ← (GPR[rs]7..0 * GPR[rt]7..0)
HI[ac]||LO[ac] ← HI[ac]||LO[ac] + temp115..0 + temp215..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `DPSQ_SA.L.W ac, rs, rt` **MIPSDSP**

Purpose:

To multiply a fractional word using full-size intermediate products and then accumulate into the specified accumulator register.

Description: `ac ← sat64(ac - sat32(rs * rt))`

Two 32 bit Q31 word from registers *rt* and *rs* are first multiplied to a 62-bit product, which is left-shifted by 1 bit to get a Q63 result. This product is then subtracted from the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

To saturate, the instruction checks both multiplicands for a -1 value, if so, the result for that multiply operation is saturated to the maximum positive value. In addition, the final accumulation result is also saturated to a Q63 result.

This instruction can set bits 19..16 depending on the *ac* specified in the ouflag field in the *DSPControl* register. Bit 16 is set for *ac0*, bit 17 for *ac1*, and so on. Note that one of these bits will be set if the operation saturates.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

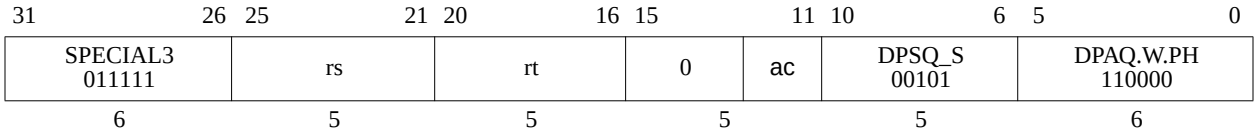
```
DPSQ_SA.L.W
    Setouflag(GPR[rs], GPR[rt], satcond)
    if satcond then
        temp1 ← 0x7FFFFFFFFFFFFFFF
    else
        temp1_63..0 ← (GPR[rs]_31..0 * GPR[rt]_31..0) << 1
    endif
    tempx_63..0 ← HI[ac] || LO[ac]
    temp2_64..0 ← (tempx_63 || tempx_63..0) - (temp1_63 || temp1_63..0)
if temp2_64 ≠ temp2_63 then
        DSPControl_ouflag:16+ac ← 1
        if temp2_64 = 0 then
            HI[ac]||LO[ac] ← 0x7FFFFFFFFFFFFFFF
        else
            HI[ac]||LO[ac] ← 0x8000000000000000
        endif
    else
        HI[ac]||LO[ac] ← temp2_63..0
    endif

function Setouflag(s, t, satcond)
    satcond ← (s_31..0 = 0x80000000) and (t_31..0 = 0x80000000)
    if satcond then
        DSPControl_ouflag:16+ac ← 1
    endif
```

```
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Dot Product with Subtract on Vector Fractional Half-Word Elements — DPSQ_S.W.PH



Format: `DPSQ_S.W.PH ac, rs, rt` **MIPSDSP**

Purpose:

To generate the dot-product of two fractional vector elements using full-size intermediate products and then subtract from the specified accumulator register.

Description: `ac ← ac - sat32(dot-product(rs[i],rt[i])`

Four Q15 values from respective vector positions within registers *rt* and *rs* are first multiplied and left-shifted by 1 bit to generate two Q31 results. These products are first added to each other to generate the dot-product result, and then subtracted from the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. This result is a Q32.31 value, i.e., Q31 values are subtracted with 32 integer guard bits that are available in the *HI/LO* register pair. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

To saturate, the instruction checks if the multiplicands are both -1, if so, the result for that multiply operation is saturated to the maximum positive value 0x7FFFFFFF. No other saturation is performed.

This instruction can set bits 19..16 of the ouflag field in the *DSPControl* register depending on the *ac* specified. Bit 16 is set for *ac0*, bit 17 for *ac1*, and so on. Note that one of these bits will be set if the operation saturates. Note that the operation overflows if any one of the vector operation overflows.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

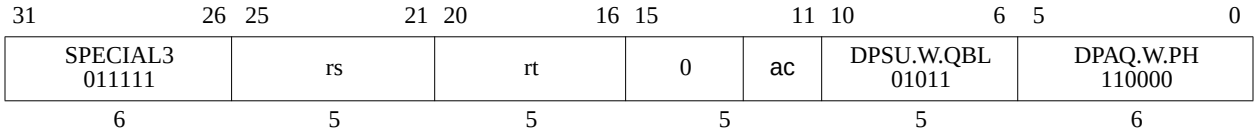
Operation:

```
DPSQ_S.W.PH
    Setouflag(GPR[rs], GPR[rt], satcond1, satcond2)
    if satcond1 then
        temp1[31..0] ← 0x7FFFFFFF
    else
        temp1[31..0] ← (GPR[rs][31..16] * GPR[rt][31..16]) << 1
    endif
    if satcond2 then
        temp2[31..0] ← 0x7FFFFFFF
    else
        temp2[31..0] ← (GPR[rs][15..0] * GPR[rt][15..0]) << 1
    endif
    HI[ac]||LO[ac] ← HI[ac]||LO[ac]-(sign_extend(temp1[31..0])+sign_extend(temp2[31..0]))

function Setouflag(s, t, satcond1, satcond2)
    satcond1 ← (s[31..16] = 0x8000) and (t[31..16] = 0x8000)
    satcond2 ← (s[15..0] = 0x8000) and (t[15..0] = 0x8000)
    if (satcond1 or satcond2) then
        DSPControl[ouflag:16+ac] ← 1
    endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Dot Product with Subtract on Vector Byte Elements — DPSU.H.QBL



Format: `DPSU.W.QBL ac, rs, rt` **MIPSDSP**

Purpose:

To generate the dot-product of two bytes from left-aligned vector positions using full-size intermediate products, followed by subtraction into the specified accumulator register.

Description: `ac ← ac - dot-product(rs[i],rt[i])`

Two unsigned byte values from the left-aligned vector positions within registers *rt* and *rs* are first multiplied using unsigned arithmetic to generate two 16-bit results. These products are first added to each other to generate the dot-product result, and then subtracted from the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

This instructions does not set the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

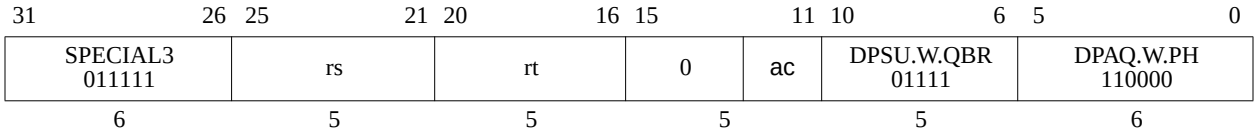
Operation:

```
temp1 ← (GPR[rs]31..24 * GPR[rt]31..24)
temp2 ← (GPR[rs]23..16 * GPR[rt]23..16)
HI[ac]||LO[ac] ← HI[ac]||LO[ac] - (temp115..0 + temp215..0)
```

Exceptions:

Reserved Instruction, DSP Disabled

Dot Product with Accumulate on Vector Byte Elements — DPSU.H.QBR



Format: `DPSU.W.QBR ac, rs, rt` **MIPSDSP**

Purpose:

To generate the dot-product of two bytes from a right-aligned vector position using full-size intermediate products, followed by subtraction into the specified accumulator register.

Description: `ac ← ac - dot-product(rs[i],rt[i])`

Two unsigned byte values from the right-aligned vector positions within registers *rt* and *rs* are first multiplied using unsigned arithmetic to generate two 16-bit results. These products are first added to each other to generate the dot-product result, and then subtracted from the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

This instructions does not set the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

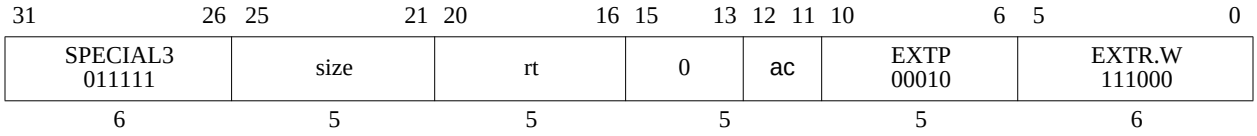
Operation:

```
temp1 ← (GPR[rs]15..8 * GPR[rt]15..8)
temp2 ← (GPR[rs]7..0 * GPR[rt]7..0)
HI[ac]||LO[ac] ← HI[ac]||LO[ac] - (temp115..0 + temp215..0)
```

Exceptions:

Reserved Instruction, DSP Disabled

Extract Bit from an Arbitrary Position in the Accumulator to a GPR — EXTP



Format: `EXTP rt, ac, size` **MIPSDSP**

Purpose:

Extract size+1 bits from an accumulator to a GPR from a position specified in the *DSPControl* register.

Description: $rt \leftarrow 0_{31..size+1} \mid\mid ac_{pos:pos-size}$

The number of bits to be extracted, i.e., size+1 bits are extracted from the specified accumulator *ac*. The 5 bits of the size field specifies a range of 1 to 32 bits to be extracted from the accumulator. The most significant bit position of the size+1 bits to be extracted are specified in the **pos** field in the *DSPControl* register. The result is copied to the destination register *rt* into the least significant position and the upper bits are zero-filled. This version of the instruction leaves the **pos** field in the *DSPControl* register unchanged. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture. After the execution of this instruction, *ac* remains unmodified.

When size = 0, this implies that one bit is to be extracted, where the msb and the lsb are the same. A valid extraction can happen with size=0 for all values of 63 >= pos >= 0. In general, an extraction is valid and said to succeed when pos - (size+1) >= -1, for a given value of size and a given value of pos just before the execution of the instruction. When the extraction is not valid, the destination register is UNPREDICTABLE.

This instruction does not affect the overflow ouflag in the *DSPControl* register. Depending on whether or not the instruction succeeds in extracting the required number of bits, the EFI (Extract Failure Indicator) in the DSPControl register is set appropriately, 1 when it fails, and 0 when it succeeds.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

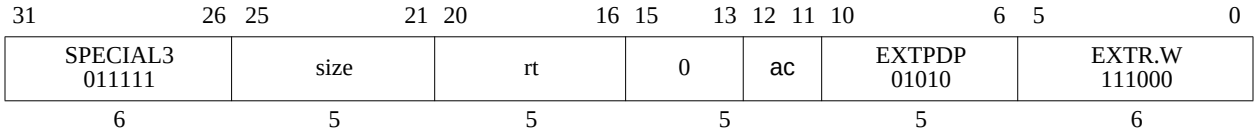
Operation:

```
s ← size
p ← DSPControl_pos:5..0
if (p - (s+1) >= -1) then
   temp ← (HI[ac]||LO[ac])_p..(p-size)
   GPR[rt] ← 0^31-s || temp_s..0
   DSPControl_EFI:14 ← 0
else
   GPR[rt] ← UNPREDICTABLE
   DSPControl_EFI:14 ← 1
endif
```

Exceptions:

Reserved Instruction, DSP Disabled

Extract Bits from an Arbitrary Position in the Accumulator to a GPR and Decrement Pos — EXTPDP



Format: `EXTPDP rt, ac, size` **MIPSDSP**

Purpose:

Extract size+1 bits from an accumulator to a GPR from a position specified in the *DSPControl* register.

Description: $rt \leftarrow 0_{31..size+1} \mid\mid ac_{pos:pos-size}$; *DSPControl*[pos] = pos - (size+1)

The number of bits to be extracted, i.e., size+1 bits are extracted from the specified accumulator *ac*. The 5 bits of the size field specifies a range of 1 to 32 bits to be extracted from the accumulator. The most significant bit position of the size+1 bits to be extracted are specified in the **pos** field in the *DSPControl* register. The result is copied to the destination register *rt* into the least significant position and the upper bits are zero-filled. This version of the instruction leaves the **pos** field in the *DSPControl* register decremented by size+1. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture. After the execution of this instruction, *ac* remains unmodified.

When size = 0, this implies that one bit is to be extracted, where the msb and the lsb are the same. A valid extraction can happen with size=0 for all values of 63 >= pos >= 0. In general, an extraction is valid and said to succeed when pos - (size+1) >= -1, for a given value of size and a given value of pos just before the execution of the instruction. When the extraction is not valid, the destination register is UNPREDICTABLE, and the value of pos is not modified.

This instruction does not affect the overflow ouflag in the *DSPControl* register. Depending on whether or not the instruction succeeds in extracting the required number of bits, the EFI (Extract Failure Indicator) in the DSPControl register is set appropriately, 1 when it fails, and 0 when it succeeds.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

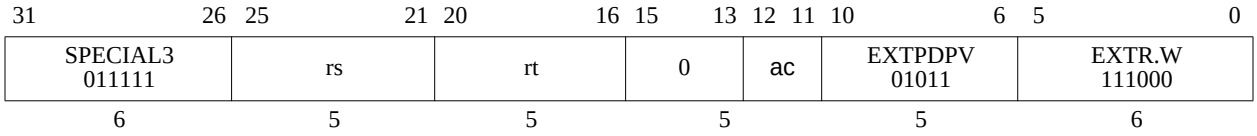
Operation:

```
s ← size
p ← DSPControl_pos:5..0
if (p - (s+1) >= -1) then
   temp ← (HI[ac]||LO[ac])_p..(p-size)
   GPR[rt] ← 0^31-s || temp_s..0
   if (p-(s+1) >= 0) then
      DSPControl_pos:5..0 ← p - (s + 1)
   else if (pos - (s+1) = -1) then
      DSPControl_pos:5..0 ← 63
   endif
   DSPControl_EFI:14 ← 0
else
   GPR[rt] ← UNPREDICTABLE
   DSPControl_EFI:14 ← 1
endif
```

Exceptions:

Reserved Instruction, DSP Disabled

Extract Variable Bits from an Arbitrary Position in the Accumulator to a GPR and Decrement Pos EXTPDPV



Format: `EXTPDPV rt, ac, rs` **MIPSDSP**

Purpose:

Extract variable number of bits from an accumulator to a GPR from a position specified in the *DSPControl* register.

Description: $rt \leftarrow 0_{31..rs[4:0]+1} \,||\, ac_{pos:pos-rs[4:0]}$; *DSPControl*[pos] = pos - (rs[4:0]+1)

The number of bits to be extracted, i.e., size+1 bits are extracted from the specified accumulator *ac*. The value of size is obtained from the 5 least significant bits of the *rs* register and these 5 bits specify a range of 1 to 32 bits to be extracted from the accumulator. The most significant bit position of the size+1 bits to be extracted are specified in the **pos** field in the *DSPControl* register. The result is copied to the destination register *rt* into the least significant position and the upper bits are zero-filled. This version of the instruction leaves the **pos** field in the *DSPControl* register decremented by size+1. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture. After the execution of this instruction, *ac* remains unmodified.

When size = 0, this implies that one bit is to be extracted, where the msb and the lsb are the same. A valid extraction can happen with size=0 for all values of 63 >= pos >= 0. In general, an extraction is valid and said to succeed when pos - (size+1) >= -1, for a given value of size and a given value of pos just before the execution of the instruction. When the extraction is not valid, the destination register is UNPREDICTABLE, and the value of pos is not modified.

This instruction does not affect the overflow ouflag in the *DSPControl* register. Depending on whether or not the instruction succeeds in extracting the required number of bits, the EFI (Extract Failure Indicator) in the DSPControl register is set appropriately, 1 when it fails, and 0 when it succeeds.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

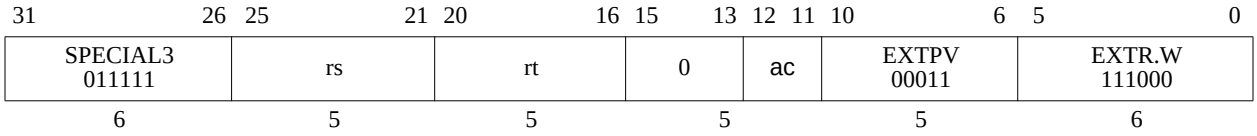
Operation:

```
s ← GPR[rs]4..0
p ← DSPControlpos:5..0
if (p - (s+1) >= -1) then
   temp ← (HI[ac]||LO[ac])p..(p-size)
   GPR[rt] ← 031-s || temps..0
   if (p-(s+1) >= 0) then
      DSPControlpos:5..0 ← p - (s + 1)
   else if (pos -(s+1) = -1) then
      DSPControlpos:5..0 ← 63
   endif
   DSPControlEFI:14 ← 0
else
   GPR[rt] ← UNPREDICTABLE
   DSPControlEFI:14 ← 1
endif
```

Exceptions:

Reserved Instruction, DSP Disabled

Extract Variable Bits from an Arbitrary Position in the Accumulator to a GPR — EXTPV



Format: `EXTPV rt, ac, rs` **MIPSDSP**

Purpose:

Extract variable number of bits from an accumulator to a GPR from a position specified in the *DSPControl* register.

Description: $rt \leftarrow 0_{31..rs[4:0]+1} \,||\, ac_{pos:pos-rs[4:0]}$

The number of bits to be extracted, i.e., size+1 bits are extracted from the specified accumulator *ac*. The value of size is obtained from the 5 least significant bits of the *rs* register and these 5 bits specify a range of 1 to 32 bits to be extracted from the accumulator. The most significant bit position of the size+1 bits to be extracted are specified in the **pos** field in the *DSPControl* register. The result is copied to the destination register *rt* into the least significant position and the upper bits are zero-filled. This version of the instruction leaves the **pos** field in the *DSPControl* register unchanged. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture. After the execution of this instruction, *ac* remains unmodified.

When size = 0, this implies that one bit is to be extracted, where the msb and the lsb are the same. A valid extraction can happen with size=0 for all values of 63 >= pos >= 0. In general, an extraction is valid and said to succeed when pos - (size+1) >= -1, for a given value of size and a given value of pos just before the execution of the instruction. When the extraction is not valid, the destination register is UNPREDICTABLE.

This instruction does not affect the overflow ouflag in the *DSPControl* register. Depending on whether or not the instruction succeeds in extracting the required number of bits, the EFI (Extract Failure Indicator) in the DSPControl register is set appropriately, 1 when it fails, and 0 when it succeeds.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

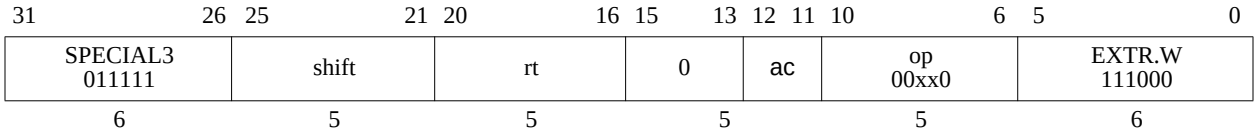
Operation:

```
s ← GPR[rs]4..0
p ← DSPControlpos:5..0
if (p - (s+1) >= -1) then
   temp ← (HI[ac]||LO[ac])p..(p-size)
   GPR[rt] ← 031-s || temps..0
   DSPControlEFI:14 ← 0
else
   GPR[rt] ← UNPREDICTABLE
   DSPControlEFI:14 ← 1
endif
```

Exceptions:

Reserved Instruction, DSP Disabled

Extract a Value with Right Shift from an Accumulator to a GPR — EXTR.W



Format:

```
EXTR.W     rt, ac, shift                    MIPSDSP
EXTR_R.W   rt, ac, shift                    MIPSDSP
EXTR_RS.W  rt, ac, shift                    MIPSDSP
```

Purpose:

Extract a value from an accumulator to a GPR with an optional right shift, rounding and/or saturation.

Description: `rt ← sat32(round(ac[63:0] >> shift))`

The arithmetic right shifted value of the 64 bit value in the accumulator is first obtained, sign-extending the shifted value. The five bits of shift value can range from 0 to +31. The shifted result is assumed to be right justified in fractional format. The lower 32 bits of the result is then copied to the destination register *rt*. The value of *ac* can range from 0 to 3. When ac=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture. This instruction leaves *ac* unmodified.

If rounding is required, then a 1 is added to the most significant discarded bit. When saturation is required after rounding, if the rounded 64-bit result is larger than 0x7FFFFFFF or smaller than 0x80000000, then the value is clamped to 0x7FFFFFFF or 0x80000000 respectively. The resulting word is copied into the GPR *rt*.

All these instructions can set bit 23 of the overflow ouflag in the DSPControl register. For given input operands, all three versions of the instruction give the same ouflag value: the bit will be set if either the rounding or non rounding version of the instruction would overflow or saturate. Thus the ouflag bit is an overflow indication bit, and does not say whether the saturate operation is actually performed or not, since that depends on the exact instruction invoked from this set.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

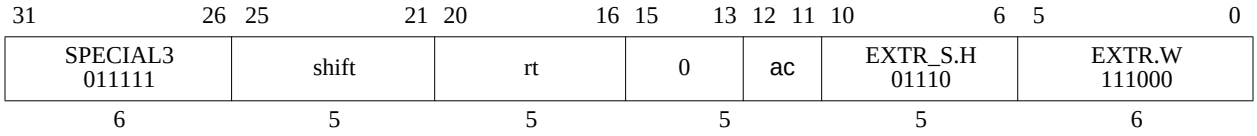
```
EXTR.W
    sa ← shift
    Setouflag(HI[ac], LO[ac], sa, setcond1, setcond2, te)
    temp ← HI[ac]63^sa || (HI[ac]||LO[ac])62..sa
    GPR[rt] ← temp31..0

EXTR_R.W
    sa ← shift
    Setouflag(HI[ac], LO[ac], sa, setcond1, setcond2, te)
    GPR[rt] ← te31..0

EXTR_RS.W
    sa ← shift
    Setouflag(HI[ac], LO[ac], sa, setcond1, setcond2, te)
    if (setcond1) then
        temp ← 0x7FFFFFFF
    else if setcond2
        temp ← 0x80000000
```

```
    else
        temp ← te31..0
    endif
    GPR[rt] ← temp

function Setouflag(h, l, sa, setcond1, setcond2, t)
    if (sa ≠ 0) then
        t ← ((h63^sa || (h||l)63..sa) || lsa-1) + 1#1
        t ← t64..1
    else if (sa = 0) then
        t ← HI[ac] || LO[ac]
    endif
    if ((h63 = 0) and ((h||l)62..sa+31 ≠ 0)) then
        DSPControlouflag:23 ← 1
        setcond1 ← 1
    endif
    if ((h63 = 0) and ((h||l)sa+30..sa-1 = 0xFFFFFFFF)) then
        DSPControlouflag:23 ← 1
        setcond1 ← 1
    endif
    if ((h63 = 1) and (h||l)62..sa+31 ≠ 1^32-sa)) then
        DSPControlouflag:23 ← 1
        setcond2 ← 1
    endif

endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Extract a Halfword with Right Shift and Saturate from an Accumulator to a GPR — EXTR_S.H



Format: `EXTR_S.H rt, ac, shift` **MIPSDSP**

Purpose:

Extract a halfword from an accumulator to a GPR with an optional right shift, and 16-bit saturation.

Description: `rt ← sat16(ac[63:0] >> shift)`

The arithmetic right shifted value of the 64 bit value in the accumulator is first obtained, sign-extending the shifted value. The five bits of shift value can range from 0 to +31. The shifted result is assumed to be right justified in fractional format. The 64 bit value is then saturated to a 16-bit result and written to the destination register *rt*. The value of *ac* can range from 0 to 3. When ac=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture. This instruction leaves *ac* unmodified.

This instruction can set bit 23 of the overflow ouflag in the *DSPControl* register. Note that this bit is set if the operation saturates.

Restrictions:

No data-dependent exceptions are possible.

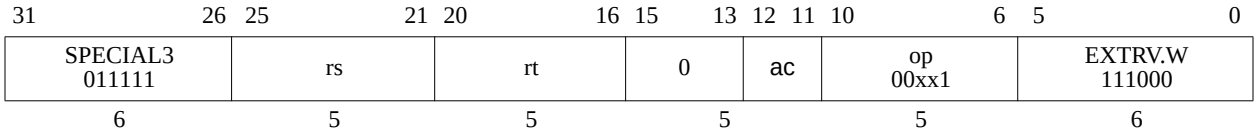
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
EXTR_S.H
    sa ← shift
    temp ← (HI[ac]63^sa || (HI[ac]||LO[ac])62..sa)
    if (temp63..0 > 0x7FFF)then
        temp ← 0x7FFF
        DSPControlouflag:23 ← 1
    else if (temp63..0 < 0xFFFFFFFFFFFF8000)then
        temp ← 0xFFFF8000
        DSPControlouflag:23 ← 1
    endif
    GPR[rt] ← temp31..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format:

```
EXTRV.W     rt, ac, rs                    MIPSDSP
EXTRV_R.W   rt, ac, rs                    MIPSDSP
EXTRV_RS.W  rt, ac, rs                    MIPSDSP
```

Purpose:

Extract a value from an accumulator to a GPR with an optional variable right shift, rounding and/or saturation.

Description: `rt ← sat32(round(ac[63:0] >> rs[4:0]))`

The arithmetic right shifted value of the 64 bit value in the accumulator is first obtained, sign-extending the shifted value. The five bits of shift value can range from 0 to +31. The shifted result is assumed to be right justified in fractional format. The lower 32 bits of the result is copied to the destination register *rt*. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture. This instruction leaves *ac* unmodified.

If rounding is required, then a 1 is added to the most significant discarded bit. When saturation is required after rounding, if the rounded 64-bit result is larger than 0x7FFFFFFF or smaller than 0x800000000, then the value is clamped to 0x7FFFFFFF or 0x80000000 respectively. The resulting word is copied into the GPR *rt*.

All these instructions can set bit 23 of the overflow ouflag in the *DSPControl* register. Note that this bit is set if the operation will saturate if the saturate version is called. That is, these are the overflow indication bits, and not the bits that say whether the saturate operation is actually performed or not, since that depends on the exact instruction invoked from this set. Hence, the ouflag bit may be set for either the rounding or the non-rounding version of the instruction.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

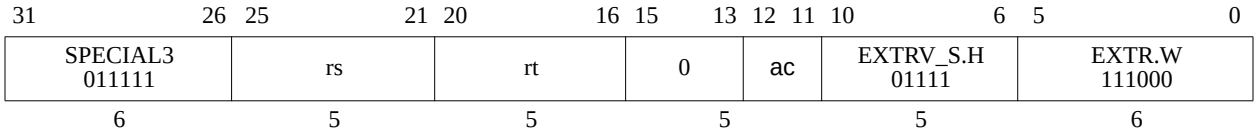
```
EXTRV.W
   sa ← GPR[rs]4..0
   Setouflag(HI[ac], LO[ac], sa, setcond1, setcond2, te)
   temp ← HI[ac]63^sa || (HI[ac]||LO[ac])62..sa
   GPR[rt] ← temp31..0

EXTRV_R.W
   sa ← shift
   Setouflag(HI[ac], LO[ac], sa, setcond1, setcond2, te)
   GPR[rt] ← te31..0

EXTRV_RS.W
   sa ← shift
   Setouflag(HI[ac], LO[ac], sa, setcond1, setcond2, te)
   if (setcond1) then
      temp ← 0x7FFFFFFF
   else if setcond2
      temp ← 0x80000000
```

```
    else
        temp ← te31..0
    endif
    GPR[rt] ← temp

function Setouflag(h, l, sa, setcond1, setcond2, t)
    if (sa ≠ 0) then
        t ← ((h63^sa || (h||l)63..sa) || lsa-1) + 1#1
        t ← t64..1
    else if (sa = 0) then
        t ← HI[ac] || LO[ac]
    endif
    if ((h63 = 0) and ((h||l)62..sa+31 ≠ 0)) then
        DSPControlouflag:23 ← 1
        setcond1 ← 1
    endif
    if ((h63 = 0) and ((h||l)sa+30..sa-1 = 0xFFFFFFFF)) then
        DSPControlouflag:23 ← 1
        setcond1 ← 1
    endif
    if ((h63 = 1) and (h||l)62..sa+31 ≠ 1^32-sa)) then
        DSPControlouflag:23 ← 1
        setcond2 ← 1
    endif

endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Extract a Halfword Variable with Right Shift and Saturate from an Accumulator to a GPR EXTRV_S.H



31–26	25–21	20–16	15–13	12 11	10–6	5–0
SPECIAL3 011111	rs	rt	0	ac	EXTRV_S.H 01111	EXTR.W 111000
6	5	5	5		5	6

Format: `EXTRV_S.H rt, ac, rs` **MIPSDSP**

Purpose:

Extract a halfword from an accumulator to a GPR with an optional variable right shift, and 16-bit saturation.

Description: `rt ← sat16(ac[63:0] >> rs[4:0])`

The arithmetic right shifted value of the 64 bit value in the accumulator is first obtained, sign-extending the shifted value. The five bits of shift value can range from 0 to +31. The shifted result is assumed to be right justified in fractional format. The 64 bit value is then saturated to a 16-bit result and written to the destination register *rt*. The value of *ac* can range from 0 to 3. When ac=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture. This instruction leaves *ac* unmodified.

This instruction can set bit 23 of the overflow ouflag in the *DSPControl* register. Note that this bit is set if the operation saturates.

Restrictions:

No data-dependent exceptions are possible.

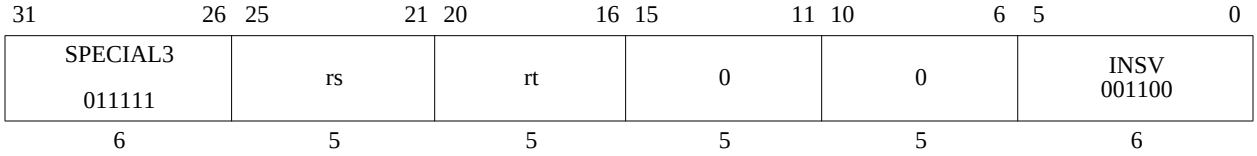
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
EXTRV_S.H
    sa ← GPR[rs]4..0
    temp ← (HI[ac]63^sa || (HI[ac]||LO[ac])62..sa)
    if (temp63..0 > 0x7FFF)then
        temp ← 0x7FFF
        DSPControlouflag:23 ← 1
    else if (temp63..0 < 0xFFFFFFFFFFFF8000)then
        temp ← 0xFFFF8000
        DSPControlouflag:23 ← 1
    endif
    GPR[rt] ← temp31..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `insv rt, rs` **MIPSDSP**

Purpose:

To merge a right-justified bit field from GPR *rs* into a specified field in GPR *rt*.

Description: `rt ← InsertFieldVar(rt, rs, Scount, Pos)`

The DSPControl register provides the *size* value from the *scount* field, and the *pos* value from the pos field. The right-most *size* bits from GPR *rs* are merged into the value from GPR *rt* starting at bit position *pos*. The result is put back in GPR *rt*. These *pos* and *size* values are converted by the instruction into the fields *msb* (the most significant bit of the field), and *lsb* (least significant bit of the field), as follows:

```
pos ← DSPControl5..0
size← DSPControl12..7
msb ← pos+size-1
lsb ← pos
```

The values of *pos* and *size* must satisfy all of the following relations, or the instruction results in UNPREDICTABLE results:

```
0 ≤ pos < 32
0 < size ≤ 32
0 < pos+size ≤ 32
```

Figure 6-1 shows the symbolic operation of the instruction.

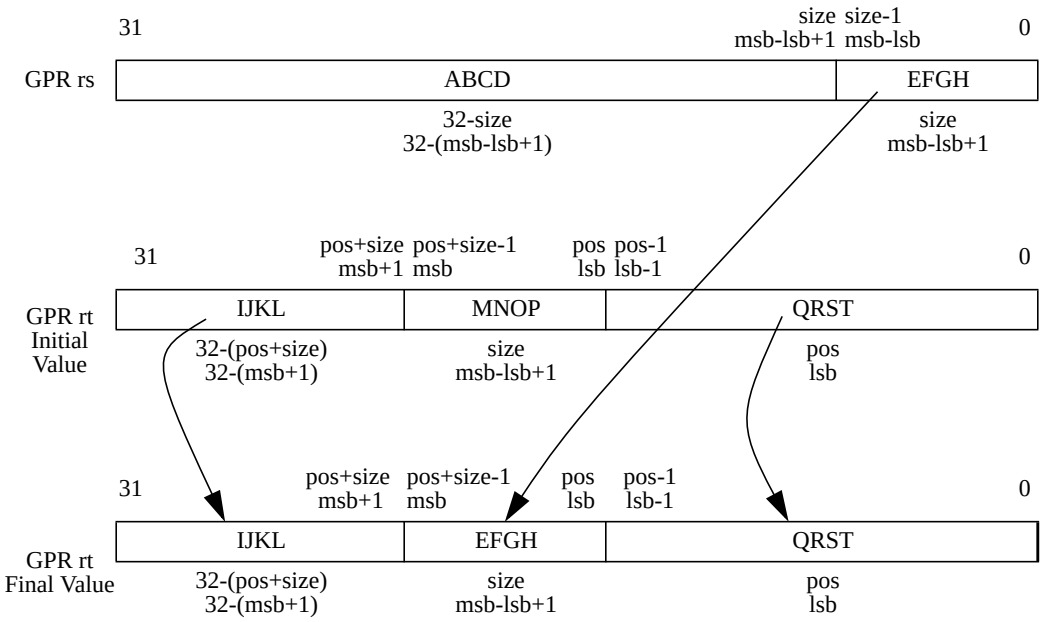


Figure 6-1 Operation of the INSV Instruction

Restrictions:

The operation is **UNPREDICTABLE** if *lsb* > *msb.*

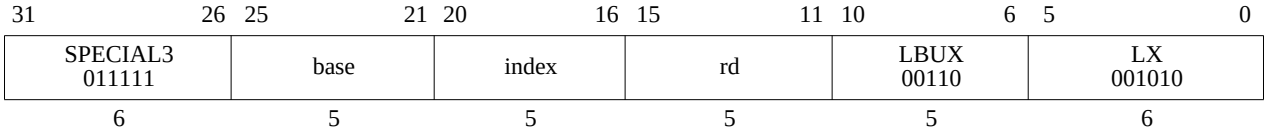
Operation:

```
if (lsb > msb) then
   UNPREDICTABLE
endif
GPR[rt] ← GPR[rt]31..msb+1 || GPR[rs]msb-lsb..0 || GPR[rt]lsb-1..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Load Unsigned Byte Indexed — LBUX



Format: `LBUX rd, index(base)` **MIPSDSP**

Purpose:

To load a byte from memory as an unsigned value, using indexed addressing**.**

Description: `rd ← memory[base+index]`

The contents of GPR *index* is added to the contents of GPR *base* to form an effective address. The contents of the 8-bit byte at the memory location specified by the aligned effective address are fetched, zero-extended to the GPR register length, and placed in GPR *rd*.

Restrictions:

None.

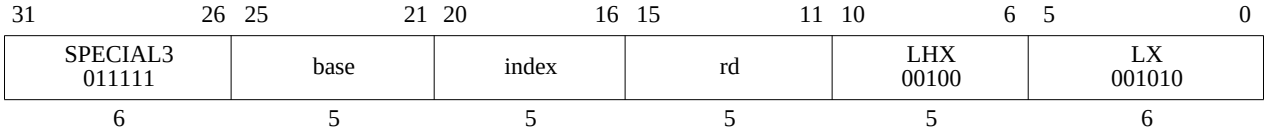
Operation:

```
vAddr ← (GPR[index]29..0) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReserseEndian2)
memword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor BigEndianCPU2
GPR[rd]← zero_extend(memword7+8*byte..8*byte)
```

Exceptions:

Reserved Instruction, DSP Disabled, TLB Refill, TLB Invalid, Address Error, Watch

Load Halfword Indexed — LHX



Format: `LHX rd, index(base)` **MIPSDSP**

Purpose:

To load a halfword from memory as a signed value, using indexed addressing**.**

Description: `rd ← memory[base+index]`

The contents of GPR *index* is added to the contents of GPR *base* to form an effective address. The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rd*.

Restrictions:

The effective address must be naturally-aligned. If the least-significant bit of the address is non-zero, an Address Error exception occurs.

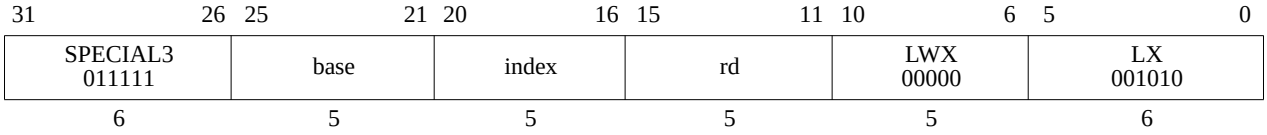
Operation:

```
vAddr ← (GPR[index]29..0) + GPR[base]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rd]← memword
```

Exceptions:

Reserved Instruction, DSP Disabled, TLB Refill, TLB Invalid, Bus Error, Address Error

Load Word Indexed LWX



Format: `LWX rd, index(base)` **MIPSDSP**

Purpose:

To load a word from memory as a signed value, using indexed addressing**.**

Description: `rd ← memory[base+index]`

The contents of GPR *index* is added to the contents of GPR *base* to form an effective address. The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rd*.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

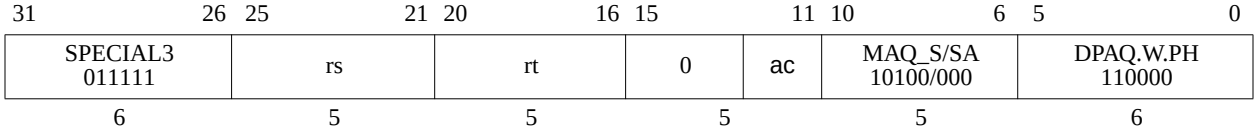
Operation:

```
vAddr ← (GPR[index]29..0) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rd]← memword
```

Exceptions:

Reserved Instruction, DSP Disabled, TLB Refill, TLB Invalid, Bus Error, Address Error

Multiply with Accumulate a Single Left Vector Fractional Half-Word Element — MAQ_S.W.PHL



Format: `MAQ_S.W.PHL ac, rs, rt` **MIPSDSP**

`MAQ_SA.W.PHL ac, rs, rt` **MIPSDSP**

Purpose:

To multiply one set of fractional vector element using full-size intermediate products and then accumulate into the specified accumulator register.

Description: $ac \leftarrow sat32(ac + sat32(rs_{31..16} * rt_{31..16}))$

Two Q15 values from the respective left vector position within registers *rt* and *rs* are first multiplied and left-shifted by 1 bit to generate a Q31 result. This product is then accumulated into the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

Both versions of the instruction check if the input values are both -1, and if so, the result for that multiply operation is saturated to the maximum positive value. The second version of the instruction also performs a saturation check after the accumulate for a value that exceeds a Q31 value in the 64-bit accumulator, and saturates to a maximum positive or maximum negative 32 bit value.

This instruction can set bits 19..16 depending on the *ac* specified, of the ouflag field in the *DSPControl* register. Bit 16 is set for *ac0*, bit 17 for *ac1*, and so on. Note that one of these bits will be set if the operation saturates.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
MAQ_S.W.PHL
    Setouflag(GPR[rs], GPR[rt], satcond)
    if satcond then
        temp ← 0x7FFFFFFF
    else
        temp ← (GPR[rs]31..16 * GPR[rt]31..16) << 1
    endif
    temp2 ← (HI[ac]||LO[ac]) + temp
    HI[ac]||LO[ac] ← temp2

MAQ_SA.W.PHL
    Setouflag(GPR[rs], GPR[rt], satcond)
    if satcond then
        temp ← 0x7FFFFFFF
    else
        temp ← (GPR[rs]31..16 * GPR[rt]31..16) << 1
    endif
    tempx63..0 ← (HI[ac]||LO[ac]) + temp
    if ((tempx63 = 0) and (tempx62..31 ≠ 0)) then
        tempx63..0 ← 0x7FFFFFFF
        DSPControlouflag:16+ac ← 1
```

```
    else if ((tempx63 = 1) and (tempx62..31 ≠ 0xFFFFFFFF)) then
       tempx63..0 ← 0xFFFFFFFF80000000
       DSPControlouflag:16+ac ← 1
    endif
    HI[ac]||LO[ac] ← tempx

function Setouflag(s, t, satcond)
    satcond ← (s31..16 = 0x8000) and (t31..16 = 0x8000)
    if satcond then
       DSPControlouflag:16+ac ← 1
    endif
endfunction Setouflag
```

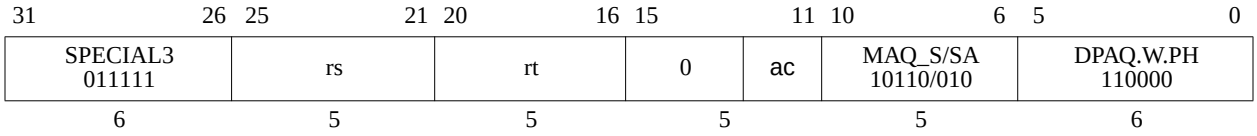
Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

The MAQ_SA version of the instruction is useful for compatibility with some ITU speech processing codecs that require a 32-bit saturation after every multiply-accumulate operation. This is an inherently serial processing method that is not necessarily more bit-accurate, but is required to claim compliance with the standard.

Multiply with Accumulate a Single Right Vector Fractional Half-Word Element — MAQ_S.W.PHR



Format: `MAQ_S.W.PHR ac, rs, rt` **MIPSDSP**

`MAQ_SA.W.PHR ac, rs, rt` **MIPSDSP**

Purpose:

To multiply one set of fractional vector element using full-size intermediate products and then accumulate into the specified accumulator register.

Description: $ac \leftarrow sat32(ac + sat32(rs_{15..0} * rt_{15..0}))$

Two Q15 values from the respective right vector position within registers *rt* and *rs* are first multiplied and left-shifted by 1 bit to generate a Q31 result. This product is then accumulated into the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

Both versions of the instruction check if the input values are both -1, and if so, the result for that multiply operation is saturated to the maximum positive value. The second version of the instruction also performs a saturation check after the accumulate for a value that exceeds a Q31 value in the 64-bit accumulator, and saturates to a maximum positive or maximum negative 32 bit value.

This instruction can set bits 19..16 depending on the *ac* specified, of the ouflag field in the *DSPControl* register. Bit 16 is set for *ac0*, bit 17 for *ac1*, and so on. Note that one of these bits will be set if the operation saturates.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
MAQ_S.W.PHR
    Setouflag(GPR[rs], GPR[rt], satcond)
    if satcond then
        temp ← 0x7FFFFFFF
    else
        temp ← (GPR[rs]15..0 * GPR[rt]15..0) << 1
    endif
    temp2 ← (HI[ac]||LO[ac]) + temp
    HI[ac]||LO[ac] ← temp2

MAQ_S.W.PHR
    Setouflag(GPR[rs], GPR[rt], satcond)
    if satcond then
        temp ← 0x7FFFFFFF
    else
        temp ← (GPR[rs]15..0 * GPR[rt]15..0) << 1
    endif
    tempx63..0 ← (HI[ac]||LO[ac]) + temp
    if ((tempx63 = 0) and (tempx62..32 ≠ 0)) then
        tempx63..0 ← 0x7FFFFFFF
        DSPControlouflag:16+ac ← 1
```

```
        else if ((tempx63 = 1) and (tempx62..31 ≠ 0xFFFFFFFF)) then
            tempx63..0 ← 0xFFFFFFFF80000000
        endif
        HI[ac]||LO[ac] ← tempx

    function Setouflag(s, t, satcond)
        satcond ← (s15..0 = 0x8000) and (t15..0 = 0x8000)
        if satcond then
            DSPControlouflag:16+ac ← 1
        endif
    endfunction Setouflag
```

Exceptions:

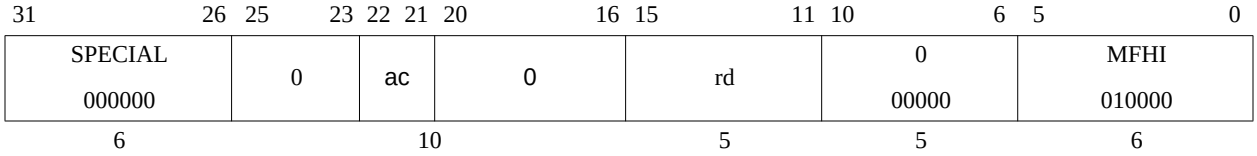
Reserved Instruction, DSP Disabled

Programming Notes:

The MAQ_SA version of the instruction is useful for compatibility with some ITU speech processing codecs that require a 32-bit saturation after every multiply-accumulate operation. This is an inherently serial processing method that is not necessarily more bit-accurate, but is required to claim compliance with the standard.

Move From HI Register

MFHI



Format: `MFHI rd, ac`

MIPS32
MIPSDSP

Purpose:

To copy the special purpose *HI* part of the specified accumulator register to a GPR

Description: `rd ← ac[HI]`

The contents of special register *HI* from accumulator ac is loaded into GPR *rd*. The *HI* part of the ac register is defined to be bits 63..32 of the DSP ASE accumulator register. The value of ac can range from 0 to 3. When ac=0, this refers to the original HI-LO register pair of the MIPS32 architecture.

Restrictions:

None

Operation:

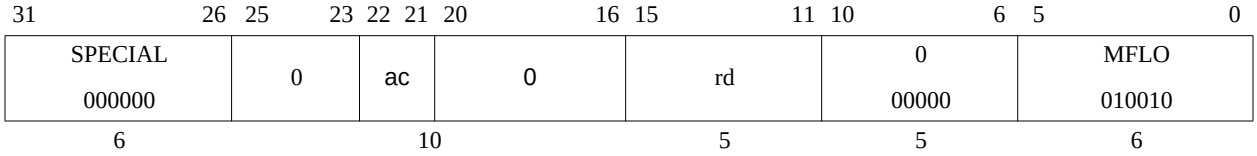
```
if (((ac = 1) or (ac = 2) or (ac = 3)) and (Status_MX = 0)) then
   SignalException(DSPDis)
endif
GPR[rd] ← HI[ac]
```

Exceptions:

DSP Disabled

Move From LO Register

MFLO



Format: `MFLO rd, ac`

MIPS32
MIPSDSP

Purpose:

To copy the *LO* part of the specified special purpose accumulator register to a GPR

Description: `rd ← ac[LO]`

The contents of special register *LO* from accumulator ac is loaded into GPR *rd*. The *LO* part of the ac register is defined to be bits 31..0 of the DSP ASE accumulator register. The value of ac can range from 0 to 3. When ac=0, this refers to the original HI-LO register pair of the MIPS32 architecture.

Restrictions:

None

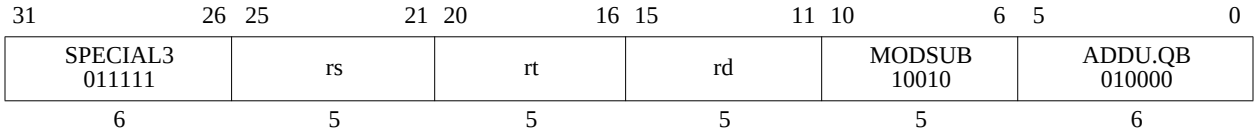
Operation:

```
if (((ac = 1) or (ac = 2) or (ac = 3)) and (Status_MX = 0)) then
   SignalException(DSPDis)
endif
GPR[rd] ← LO[ac]
```

Exceptions:

DSP Disabled

Modular Subtraction on an Index Value — MODSUB



Format: `MODSUB rd, rs, rt` **MIPSDSP**

Purpose:

Do a modular subtraction on a specified index value, using the specified decrement and modular roll-around values.

Description: `rd ← (rs==0) ? rt[23:8] : rs-rt[7:0]`

The 32-bit value in GPR rs is checked for a zero value. If it is zero, then the index value has reached the bottom of the buffer and must be rolled back around to the top of the buffer. The index value of the top element of the buffer is obtained from bits 23..8 in register rt and the destination register rd is set to this value. If the rs value is not zero, then it is simply decremented by the size of the element in the buffer, this size in bytes is obtained from bits 7..0 in register rt.

This instruction does not modify the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

Operation:

```
decr ← GPR[rt]7..0
lastindex ← GPR[rt]23..8
if (GPR[rs] = 0) then
   temp ← lastindex
else
   temp ← GPR[rs] - decr
endif
GPR[rd] ← temp
```

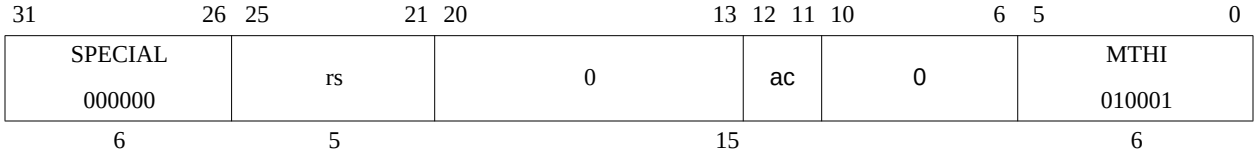
Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

Note that the hardware does not check to verify that a zero value for rs will eventually be reached. If the user does not ensure that the element size is a multiple of the buffer size, then rs will never reach the value of zero and the instruction will not wrap-around to the top of the buffer. Similarly, the user is expected to accurately calculate the index of the last element of the buffer and program the rt register correctly, the hardware cannot verify the correctness of this value for the user. Either one of these errors might eventually lead to an address error exception in the user code when the unexpected index value is used in a load instruction.

Move to HI Register — MTHI



Format: `MTHI rs, ac` **MIPS32 MIPSDSP**

Purpose:

To copy a GPR to the *HI* part of the specified special purpose accumulator register

Description: `ac[HI] ← rs`

The contents of GPR *rs* are loaded into the *HI* part of the specified accumulator ac. The *HI* part of the ac register is defined to be bits 63..32 of the DSP ASE accumulator register. The value of ac can range from 0 to 3. When ac=0, this refers to the original HI-LO register pair of the MIPS32 architecture.

Restrictions:

A computed result written to the *HI*/*LO* pair by DIV, DIVU,MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either *HI* or *LO*.

If an MTHI instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of *LO* are UNPREDICTABLE. The following example shows this illegal situation:

```
MUL    r2,r4  # start operation that will eventually write to HI,LO
...           # code not containing mfhi or mflo
MTHI   r6
...           # code not containing mflo
MFLO   r3     # this mflo would get an UNPREDICTABLE value
```

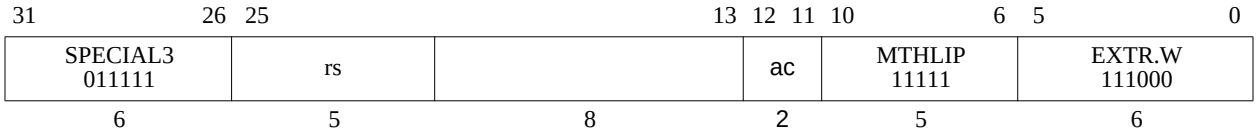
Operation:

```
if (((ac = 1) or (ac = 2) or (ac = 3)) and (Status_MX = 0)) then
   SignalException(DSPDis)
endif
HI[ac] ← GPR[rs]
```

Exceptions:

DSP Disabled

Copy the LO to HI and a GPR value to LO and Increment Pos by 32 — MTHLIP



Format: `MTHLIP rs, ac` **MIPSDSP**

Purpose:

Insert a GPR value into LO of the accumulator, while shifting *LO* up into the *HI* part, and increment the *pos* field in *DSPControl* by 32.

Description: $ac \leftarrow ac_{31..0} \,||\, GPR[rs]_{31..0}$; $DSPControl[pos]_{5..0}$ += 32

The lower 32 bits of the specified accumulator *ac* is copied to the upper 32 bits of the accumulator. The contents of the source register *rs* are copied into the lower 32 bits of *ac*. Valid *ac* values are 0 to 3, where 0 refers to the original *HI-LO* pair of the architecture. This instruction increments the *pos* field, bits 5..0 in the *DSPControl* register by the integer value 32. Before the execution of this instruction, if the value of *pos* is greater than or equal to 32, then the result is UNPREDICTABLE.

This instruction does not modify the overflow ouflag in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

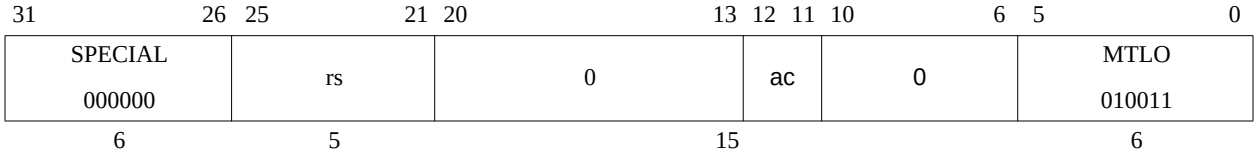
Operation:

```
temp1 ← LO[ac]
temp2 ← GPR[rs]
HI[ac]||LO[ac] ← temp1[31..0] || temp2[31..0]
if ( DSPControl[pos:5..0] >= 32) then
   DSPControl[pos:5..0] ← UNPREDICTABLE
else
   DSPControl[pos:5..0] ← DSPControl[pos:5..0] + 32
endif
```

Exceptions:

Reserved Instruction, DSP Disabled

Move to LO Register MTLO



Format: `MTLO rs, ac` **MIPS32**
MIPSDSP

Purpose:

To copy a GPR to the *LO* part of the specified special purpose accumulator register.

Description: `ac[LO] ← rs`

The contents of GPR *rs* are loaded into the *LO* part of the specified accumulator ac. The *LO* part of the ac register is defined to be bits 31..0 of the DSP ASE accumulator register. The value of ac can range from 0 to 3. When ac=0, this refers to the original HI-LO register pair of the MIPS32 architecture.

Restrictions:

A computed result written to the *HI*/*LO* pair by DIV, DIVU,MULT, or MULTU must be read by MFHI or MFLO before a new result can be written into either *HI* or *LO*.

If an MTHI instruction is executed following one of these arithmetic instructions, but before an MFLO or MFHI instruction, the contents of *LO* are UNPREDICTABLE. The following example shows this illegal situation:

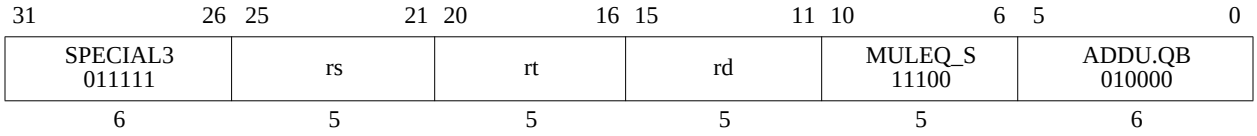
```
MUL    r2,r4  # start operation that will eventually write to HI,LO
...           # code not containing mfhi or mflo
MTHI   r6
...           # code not containing mflo
MFLO   r3     # this mflo would get an UNPREDICTABLE value
```

Operation:

```
if (((ac = 1) or (ac = 2) or (ac = 3)) and (Status_MX = 0)) then
   SignalException(DSPDis)
endif
LO[ac] ← GPR[rs]
```

Exceptions:

DSP Disabled



Format: `MULEQ_S.W.PHL rd, rs, rt` **MIPSDSP**

Purpose:

Multiply two fractional Q15 values to a Q31 result with saturation.

Description: $rd \leftarrow sat32(rs_{31:16} * rt_{31:16})$

Two fractional half-words (Q15) values from registers *rs* and *rt* are multiplied, left-shifted one bit, and written into the destination register *rd* as a Q31 result The two source half-word values are taken from each register from the respective left position of a two half-word vector data type. To do the input saturation, the instruction checks whether both input values are -1, in which case the result is clamped to the maximum Q31 value.

To stay compliant with the base architecture, this instruction leaves the base *HI/LO* pair (accumulator *ac0*) **UNPREDICTABLE** after the operation completes. The other DSP ASE accumulators, *ac1*, *ac2*, and *ac3* must be unmodified.

This instruction, on an overflow or underflow writes bit 21 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
MULEQ_S.W.PHL
    Setouflag(GPR[rs], GPR[rt], satcond)
    if satcond then
        temp ← 0x7FFFFFFF
    else
        temp ← (GPR[rs]31..16 * GPR[rt]31..16) << 1
    endif
    GPR[rd] ← temp
    HI ← UNPREDICTABLE
    LO ← UNPREDICTABLE

function Setouflag(s, t, satcond)
    satcond ← (s31..16 = 0x8000) and (t31..16 = 0x8000)
    if (satcond) then
        DSPControlouflag:21 ← 1
    endif
endfunction Setouflag
```

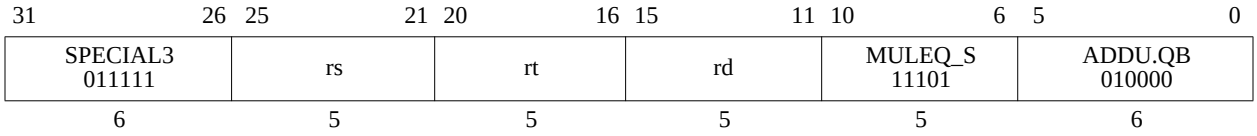
Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

The base MIPS32 architecture states that upon the after a GPR-targeting multiply instruction such as MUL, the contents of HI and LO are **UNPREDICTABLE**. To stay compliant with the base architecture, this multiply instruction states the same requirement. But this requirement does not apply to the new accumulators *ac1-ac3* and hence a programmer must save the value in *ac0* (which is the same as HI and LO) across a GPR-targeting multiply instruction, it

needed, while the values in *ac1-ac3* do not need to be saved.



Format: `MULEQ_S.W.PHR rd, rs, rt` **MIPSDSP**

Purpose:

Multiply two fractional Q15 values to a Q31 result with saturation.

Description: `rd ← sat32(rs`$_{15:0}$ `* rt`$_{15:0}$`)`

Two fractional half-words (Q15) values from registers *rs* and *rt* are multiplied, left-shifted one bit, and written into the destination register *rd* as a Q31 result The two source half-word values are taken from each register from the respective right position of a two half-word vector data type. To do the input saturation, the instruction checks whether both input values are -1, in which case, the result is clamped to the maximum Q31 value.

To stay compliant with the base architecture, this instruction leaves the base *HI/LO* pair (accumulator *ac0*) **UNPREDICTABLE** after the operation completes. The other DSP ASE accumulators, *ac1*, *ac2*, and *ac3* must be unmodified.

This instruction, on an overflow or underflow writes bit 21 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
MULEQ_S.W.PHR
   Setouflag(GPR[rs], GPR[rt], satcond)
   if satcond then
      temp ← 0x7FFFFFFF
   else
      temp ← (GPR[rs]15..0 * GPR[rt]15..0) << 1
   endif
   GPR[rd] ← temp
   HI ← UNPREDICTABLE
   LO ← UNPREDICTABLE

function Setouflag(s, t, satcond)
   satcond ← (s15..0 = 0x8000) and (t15..0 = 0x8000)
   if (satcond) then
      DSPControlouflag:21 ← 1
   endif
endfunction Setouflag
```

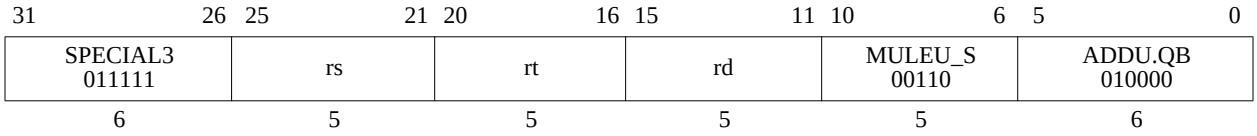
Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

The base MIPS32 architecture states that upon the after a GPR-targeting multiply instruction such as MUL, the contents of HI and LO are **UNPREDICTABLE**. To stay compliant with the base architecture, this multiply instruction

states the same requirement. But this requirement does not apply to the new accumulators *ac1-ac3* and hence a programmer must save the value in *ac0* (which is the same as HI and LO) across a GPR-targeting multiply instruction, it needed, while the values in *ac1-ac3* do not need to be saved.



Format: `MULEU_S.PH.QBL rd, rs, rt` **MIPSDSP**

Purpose:

To multiply a pair of 8-bit vector values to a pair of 16-bit vector values with saturation.

Description: `rd ← sat16(rs31:24 * rt31:16) || sat16(rs23:16 * rt15:0)`

Two left-aligned vector byte values from registers *rs* are multiplied as unsigned integer values with the two 16-bit unsigned values in register *rt*. The result is truncated to the 16 least-significant bits and written back into the respective vector positions in the destination register *rd*. The instruction saturates the result to the maximum positive value 0xFFFF if any of the truncated bits are non-zero.

To stay compliant with the base architecture, this instruction leaves the base *HI/LO* pair (accumulator *ac0*) **UNPREDICTABLE** after the operation completes. The other DSP ASE accumulators, *ac1*, *ac2*, and *ac3* must be unmodified.

This instruction writes bit 21 in the ouflag field in the *DSPControl* register on an overflow of either vector operation.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

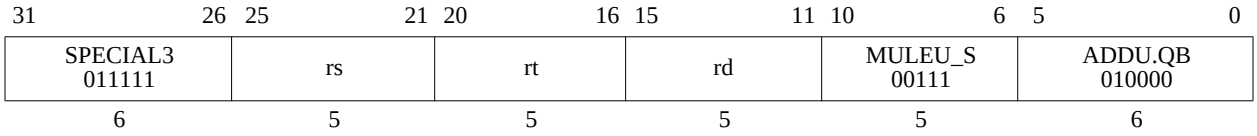
```
MULEU_S.PH.QBL
    Setouflag(GPR[rs], GPR[rt], tempu, tempv, satcondu, satcondv)
    if satcondu then
       tempu ← 0xFFFF
    endif
    if satcondv then
       tempv ← 0xFFFF
    endif
    GPR[rd] ← tempv15..0 || tempu15..0
    HI ← UNPREDICTABLE
    LO ← UNPREDICTABLE

function Setouflag(s, t, tempu, tempv, satcondu, satcondv)
    tempu ← (s23..16 * t15..0)
    tempv ← (s31..24 * t31..16)
    satcondu ← tempu > 0xFFFF
    satcondv ← tempv > 0xFFFF
    if (satcondu or satcondv) then
       DSPControlouflag:21 ← 1
    endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

The base MIPS32 architecture states that upon the after a GPR-targeting multiply instruction such as MUL, the contents of HI and LO are **UNPREDICTABLE**. To stay compliant with the base architecture, this multiply instruction states the same requirement. But this requirement does not apply to the new accumulators *ac1-ac3* and hence a programmer must save the value in *ac0* (which is the same as HI and LO) across a GPR-targeting multiply instruction, it needed, while the values in *ac1-ac3* do not need to be saved.



Format: `MULEU_S.PH.QBR rd, rs, rt` **MIPSDSP**

Purpose:

To multiply a pair of 8-bit vector values to a pair of 16-bit vector values with saturation.

Description: `rd ← sat16(rs`$_{15:8}$ `* rt`$_{31:16}$`) || sat16(rs`$_{7:0}$ `* rt`$_{15:0}$`)`

Two right-aligned vector byte values from registers *rs* are multiplied as unsigned integer values with the two 16-bit unsigned values in register *rt*. The result is truncated to the 16 least-significant bits and written back into the respective vector positions in the destination register *rd*. The instruction saturates the result to the maximum positive value 0xFFFF if any of the truncated bits are non-zero.

To stay compliant with the base architecture, this instruction leaves the base *HI/LO* pair (accumulator *ac0*) **UNPREDICTABLE** after the operation completes. The other DSP ASE accumulators, *ac1*, *ac2*, and *ac3* must be unmodified.

This instruction writes bit 21 in the ouflag field in the *DSPControl* register on an overflow of either vector operation.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

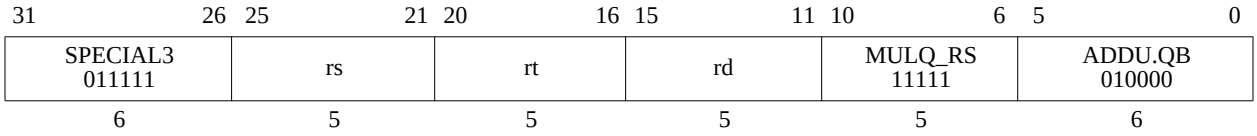
```
MULEU_S.PH.QBR
    Setouflag(GPR[rs], GPR[rt], tempu, tempv, satcondu, satcondv)
    if satcondu then
       tempu ← 0xFFFF
    endif
    if satcondv then
       tempv ← 0xFFFF
    endif
    GPR[rd] ← tempv15..0 || tempu15..0
    HI ← UNPREDICTABLE
    LO ← UNPREDICTABLE

function Setouflag(s, t, tempu, tempv, satcondu, satcondv)
    tempu ← (s7..0 * t15..0)
    tempv ← (s15..8 * t31..16)
    satcondu ← tempu > 0xFFFF
    satcondv ← tempv > 0xFFFF
    if (satcondu or satcondv) then
       DSPControlouflag:21 ← 1
    endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

The base MIPS32 architecture states that upon the after a GPR-targeting multiply instruction such as MUL, the contents of HI and LO are **UNPREDICTABLE**. To stay compliant with the base architecture, this multiply instruction states the same requirement. But this requirement does not apply to the new accumulators *ac1-ac3* and hence a programmer must save the value in *ac0* (which is the same as HI and LO) across a GPR-targeting multiply instruction, it needed, while the values in *ac1-ac3* do not need to be saved.



Format: `MULQ_RS.PH rd, rs, rt` **MIPSDSP**

Purpose:

Multiply two vector fractional Q15 values with rounding and saturation.

Description: `rd ← sat16(rs`$_{31:16}$ `* rt`$_{31:16}$`) || sat16(rs`$_{15:0}$ `* rt`$_{15:0}$`)`

The two vector fractional Q15 values in registers *rs* and *rt* are multiplied separately. The intermediate result is obtained by the most-significant 16 bits of the multiplicand result, which is also bits 30..15. Saturation is done when both input values are -1, resulting in clamping the intermediate result to the maximum Q15 value. To do rounding after the multiply, a 1 bit value is added at the most-significant discarded bit position, which rounds the result to the nearest value with half-way values rounded up. Each multiplicand result thus obtained is written into the respective vector position in the destination register *rd*.

To stay compliant with the base architecture, this instruction leaves the base *HI/LO* pair (accumulator *ac0*) **UNPREDICTABLE** after the operation completes. The other DSP ASE accumulators, *ac1*, *ac2*, and *ac3* must be untouched.

This instruction, on an overflow or underflow of any one of the two vector operation, writes bit 21 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

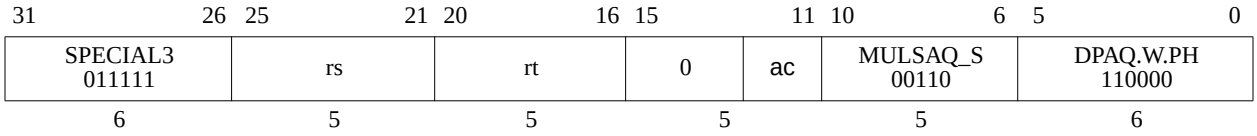
```
MULQ_RS.PH
    Setouflag(GPR[rs], GPR[rt], satcond1, satcond2)
    if satcond1 then
       temp1 ← 0x7FFFFFFF
    else
       temp1 ← ((GPR[rs]31..16 * GPR[rt]31..16) << 1) + 0x8000
    endif
    if satcond2 then
       temp2 ← 0x7FFFFFFF
    else
       temp2 ← ((GPR[rs]15..0 * GPR[rt]15..0) << 1) + 0x8000
    endif
    GPR[rd] ← temp131..16 || temp231..16
    HI ← UNPREDICTABLE
    LO ← UNPREDICTABLE

function Setouflag(s, t, satcond1, satcond2)
    satcond1 ← (s31..16 = 0x8000) and (t31..16 = 0x8000)
    satcond2 ← (s15..0 = 0x8000) and (t15..0 = 0x8000)
    if (satcond1 or satcond2) then
       DSPControl21 ← 1
    endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

The base MIPS32 architecture states that upon the after a GPR-targeting multiply instruction such as MUL, the contents of HI and LO are **UNPREDICTABLE**. To stay compliant with the base architecture, this multiply instruction states the same requirement. But this requirement does not apply to the new accumulators *ac1-ac3* and hence a programmer must save the value in *ac0* (which is the same as HI and LO) across a GPR-targeting multiply instruction, it needed, while the values in *ac1-ac3* do not need to be saved.



Format: `MULSAQ_S.W.PH ac, rs, rt` **MIPSDSP**

Purpose:

To multiply and subtract two fractional vector elements using full-size intermediate products and then accumulate into the specified accumulator register.

Description: $ac \leftarrow ac + (sat32(rs_{31..16} * rt_{31..16}) - sat32(rs_{15..0} * rt_{15..0}))$

Four Q15 values from respective vector positions within registers *rt* and *rs* are first multiplied and left-shifted by 1 bit to generate two Q31 results. These products are first subtracted from each other to generate an intermediate result which is then added to the specified 64 bit *HI/LO* accumulator to generate a 64 bit result. The 64 bit value is Q32.31 since a Q31 value is added with 32 guard bits that are available in the *HI/LO* register pair. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

To saturate, the instruction checks if the multiplicands are both -1, if so, the result for that multiply operation is saturated to the maximum positive value 0x7FFFFFFF. No other saturation is performed.

This instruction can set bits 19..16 of the ouflag field in the *DSPControl* register depending on the *ac* specified. Bit 16 is set for *ac0*, bit 17 for *ac1*, and so on. Note that one of these bits will be set if the operation saturates. Note that the operation overflows if any one of the vector operation overflows.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the result is **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

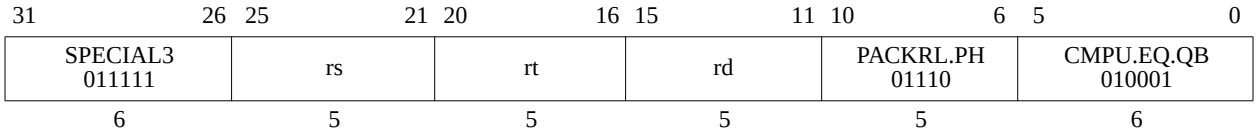
Operation:

```
MULSAQ_S.W.PH
    Setouflag(GPR[rs], GPR[rt], satcond1, satcond2)
    if satcond1 then
       temp1 ← 0x7FFFFFFF
    else
       temp1 ← (GPR[rs]31..16 * GPR[rt]31..16) << 1
    endif
    if satcond2 then
       temp2 ← 0x7FFFFFFF
    else
       temp2 ← (GPR[rs]15..0 * GPR[rt]15..0) << 1
    endif
    HI[ac]||LO[ac] ← (HI[ac]||LO[ac]) + (temp1 - temp2)

function Setouflag(s, t, satcond1, satcond2)
    satcond1 ← (s31..16 = 0x8000) and (t31..16 = 0x8000)
    satcond2 ← (s15..0 = 0x8000) and (t15..0 = 0x8000)
    if (satcond1 or satcond2) then
       DSPControlouflag:16+ac ← 1
    endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Pack a Vector of Two Halfwords from the Right and Left Halfwords of two Sources — PACKRL.PH



Format: `PACKRL.PH rd, rs, rt` **MIPSDSP**

Purpose:

Pick two elements for a halfword-sized vector based on the right halfword and left halfword respectively from the two source registers.

Description: $GPR[rd] \leftarrow rs_{15..0} \,||\, rt_{31..16}$

The right half-word from register *rs* and the left half-word from the register *rt* are packed into the left and right positions of the destination register *rd*.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

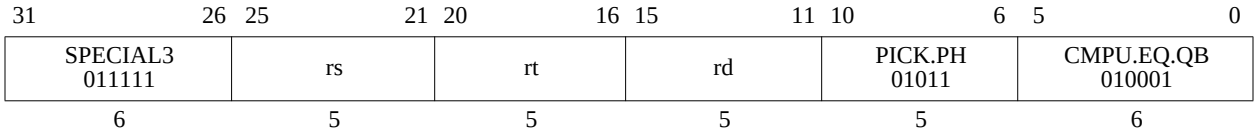
Operation:

```
tempu ← GPR[rs]15..0
tempv ← GPR[rt]31..16
GPR[rd] = tempu15..0 || tempv15..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Pick a Vector of Two Halfwords Based on Condition Code Bits — PICK.PH



Format: `PICK.PH rd, rs, rt` **MIPSDSP**

Purpose:

Pick two elements for a halfword-sized vector based on the corresponding condition code bits.

Description: $GPR[rd] \leftarrow pick(cc_{25}, rs_{31..16}, rt_{31..16}) \ || \ pick(cc_{24}, rs_{15..0}, rt_{15..0})$

The two right-most condition code bits in the *DSPControl* register are used to determine how to pick values from the two vectors in registers *rs* and *rt* each with two halfword values. For each corresponding vector element position, the corresponding condition code bit is checked. If the code value is 1 then the corresponding vector element from the source register *rs* is picked to write into the same vector location in the destination register. If the code is 0, then the source register *rt* supplies the destination value.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

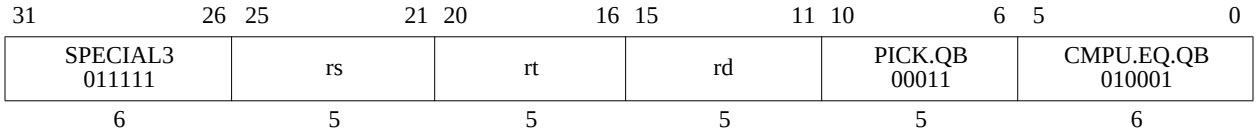
Operation:

```
if (DSPControl_cc:25 = 1) then
   tempu ← GPR[rs]_31..16
else
   tempu ← GPR[rt]_31..16
endif
if (DSPControl_cc:24 = 1) then
   tempv ← GPR[rs]_15..0
else
   tempv ← GPR[rt]_15..0
endif
GPR[rd] = tempu_15..0 || tempv_15..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Pick a Vector of Four Bytes Based on Condition Code Bits — PICK.QB



Format: `PICK.QB rd, rs, rt` **MIPSDSP**

Purpose:

Pick four elements for a byte-sized vector based on the corresponding condition code bits.

Description: $GPR[rd] \leftarrow pick(cc_{27}, rs_{31..24}, rt_{31..24}) \mid\mid pick(cc_{26}, rs_{23..16}, rt_{23..16}) \mid\mid pick(cc_{25}, rs_{15..8}, rt_{15..8}) \mid\mid pick(cc_{24}, rs_{7..0}, rt_{7..0})$

The four condition code bits in the *DSPControl* register are used to determine how to pick values from the two vectors in registers *rs* and *rt* each with four byte values. For each corresponding vector element position, the corresponding condition code bit is checked. If the code value is 1 then the corresponding vector element from the source register *rs* is picked to write into the same vector location in the destination register. If the code is 0, then the source register *rt* supplies the destination value.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

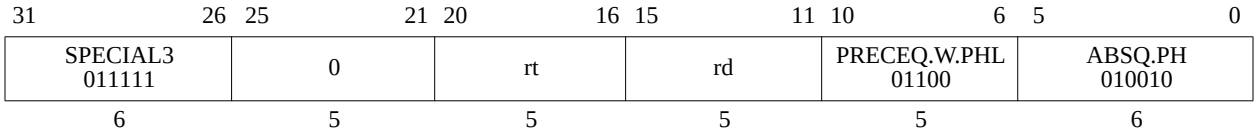
Operation:

```
if (DSPControl_cc:27 = 1) then
   tempu ← GPR[rs]_31..24
else
   tempu ← GPR[rt]_31..24
endif
if (DSPControl_cc:26 = 1) then
   tempv ← GPR[rs]_23..16
else
   tempv ← GPR[rt]_23..16
endif
if (DSPControl_cc:25 = 1) then
   tempw ← GPR[rs]_15..8
else
   tempw ← GPR[rt]_15..8
endif
if (DSPControl_cc:24 = 1) then
   tempx ← GPR[rs]_7..0
else
   tempx ← GPR[rt]_7..0
endif
GPR[rd] = tempu_7..0 || tempv_7..0 || tempw_7..0 || tempx_7..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Precision Expand A Fractional Halfword to a Fractional Word Value — PRECEQ.W.PHL



Format: `PRECEQ.W.PHL rd, rt` **MIPSDSP**

Purpose:

Expand the precision of a left-aligned fractional Q15 value to a fractional Q31 value

Description: `rd ← expand_prec(rt`$_{31..16}$`))`

The left-aligned vector fractional Q15 value in register *rt* is taken and expanded to a fractional Q31 value. The result thus obtained is then written into the destination register *rd*. The expansion is done by appending 16 least-significant zeros to generate the 32 bit fractional value.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

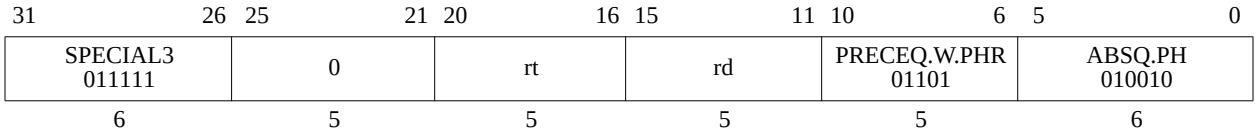
Operation:

```
temp ← GPR[rt]31..16 || 0^16
GPR[rd] ← temp31..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Precision Expand A Fractional Halfword to a Fractional Word Value — PRECEQ.W.PHR



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL3 011111	0	rt	rd	PRECEQ.W.PHR 01101	ABSQ.PH 010010
6	5	5	5	5	6

Format: `PRECEQ.W.PHR rd, rt` **MIPSDSP**

Purpose:

Expand the precision of a right-aligned fractional Q15 value to a fractional Q31 value

Description: `rd ← expand_prec(rt`$_{16..0}$`)`

The right-aligned vector fractional Q15 value in register *rt* is taken and expanded to a fractional Q31 value. The result thus obtained is then written into the destination register *rd*. The expansion is done by appending 16 least-significant zeros to generate the 32 bit fractional value.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

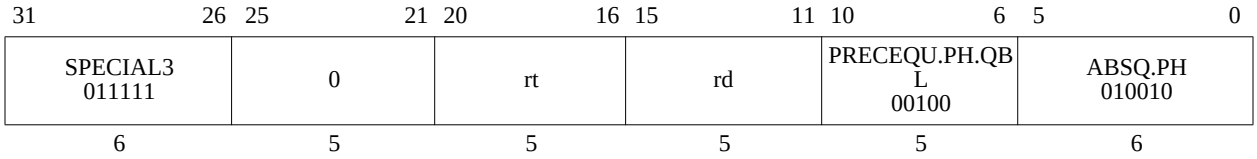
Operation:

```
temp ← GPR[rt]15..0 || 016
GPR[rd] ← temp31..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Precision Expand Two Unsigned Bytes to Fractional Halfword Values — PRECEQU.PH.QBL



Format: `PRECEQU.PH.QBL rd, rt` **MIPSDSP**

Purpose:

Expand the precision of two unsigned bytes from a left-aligned position in a vector to fractional Q15 values

Description: `rd ← expand_prec(rt`$_{31..24}$`)) || expand_prec(rt`$_{23..16}$`))`

The two left-aligned vector byte values in register *rt* are taken, and each expanded to a fractional Q15 value. The two results thus obtained are then written into the destination register *rd* in the corresponding vector location. The expansion is done by appending 7 least-significant zeros and pre-pending a single 0 bit (for positive sign) to generate the 16 bit fractional value.

Restrictions:

No data-dependent exceptions are possible.

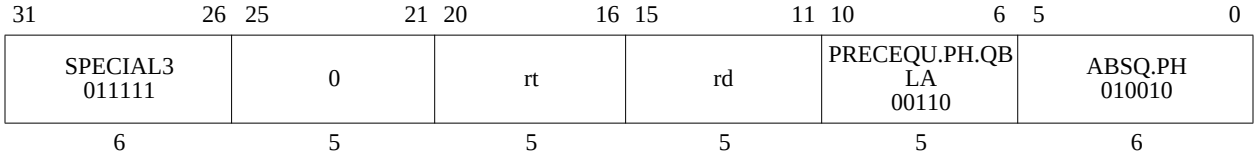
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
tempu ← 0^1 || GPR[rt]31..24 || 0^7
tempv ← 0^1 || GPR[rt]23..16 || 0^7
GPR[rd] ← tempu15..0 || tempv15..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `PRECEQU.PH.QBLA rd, rt` **MIPSDSP**

Purpose:

Expand the precision of two unsigned bytes from a left-alternate aligned position in a vector to fractional Q15 values

Description: `rd ← expand_prec(rt`$_{31..24}$`)) || expand_prec(rt`$_{15..8}$`))`

The two left-alternate aligned vector byte values in register *rt* are taken, and each expanded to a fractional Q15 value. The two results thus obtained are then written into the destination register *rd* in the corresponding vector location. The expansion is done by appending 7 least-significant zeros and pre-pending a single 0 bit (for positive sign) to generate the 16 bit fractional value.

Restrictions:

No data-dependent exceptions are possible.

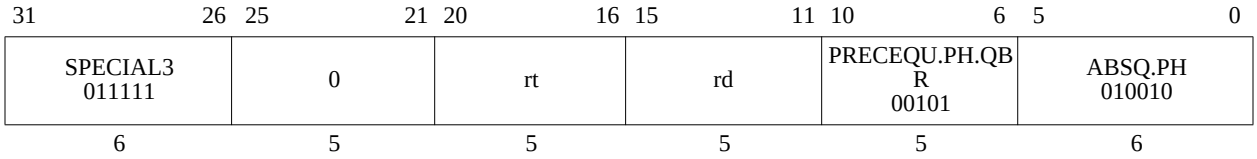
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
tempu ← 0^1 || GPR[rt]31..24 || 0^7
tempv ← 0^1 || GPR[rt]15..8 || 0^7
GPR[rd] ← tempu15..0 || tempv15..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `PRECEQU.PH.QBR rd, rt` **MIPSDSP**

Purpose:

Expand the precision of two unsigned bytes from a right-aligned position in a vector to fractional Q15 values

Description: `rd ← expand_prec(rt`$_{15..8}$`)) || expand_prec(rt`$_{7..0}$`))`

The two right-aligned vector byte values in register *rt* are taken, and each expanded to a fractional Q15 value. The two results thus obtained are then written into the destination register *rd* in the corresponding vector location. The expansion is done by appending 7 least-significant zeros and pre-pending a single 0 bit (for positive sign) to generate the 16 bit fractional value.

Restrictions:

No data-dependent exceptions are possible.

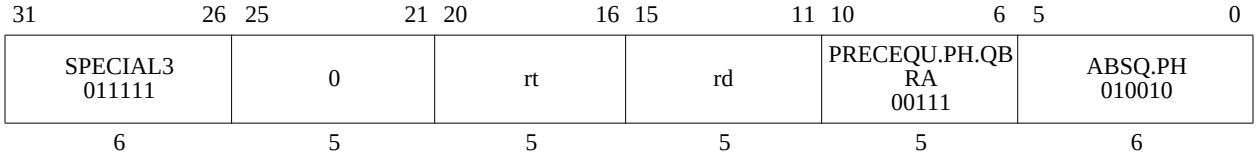
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
tempu ← 0^1 || GPR[rt]15..8 || 0^7
tempv ← 0^1 || GPR[rt]7..0 || 0^7
GPR[rd] ← tempu15..0 || tempv15..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `PRECEQU.PH.QBRA rd, rt` **MIPSDSP**

Purpose:

Expand the precision of two unsigned bytes from a right-alternate aligned position in a vector to fractional Q15 values

Description: `rd ← expand_prec(rt`$_{23..16}$`)) || expand_prec(rt`$_{7..0}$`))`

The two right-alternate aligned vector byte values in register *rt* are taken, and each expanded to a fractional Q15 value. The two results thus obtained are then written into the destination register *rd* in the corresponding vector location. The expansion is done by appending 7 least-significant zeros and pre-pending a single 0 bit (for positive sign) to generate the 16 bit fractional value.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

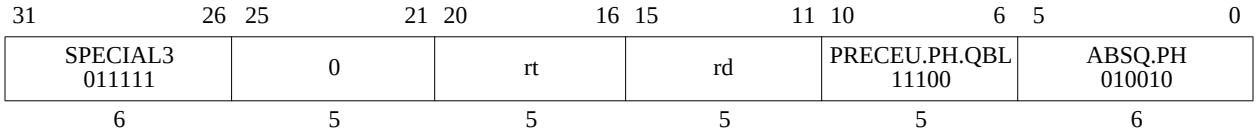
Operation:

```
tempu ← 0^1 || GPR[rt]23..16 || 0^7
tempv ← 0^1 || GPR[rt]7..0 || 0^7
GPR[rd] ← tempu15..0 || tempv15..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Precision Expand Two Unsigned Bytes to Unsigned Halfwords — PRECEU.PH.QBL



Format: `PRECEU.PH.QBL rd, rt` **MIPSDSP**

Purpose:

Expand the precision of two unsigned bytes from a left-aligned position in a vector to unsigned integer halfwords

Description: `rd ← expand_prec(rt`$_{31..24}$`)) || expand_prec(rt`$_{23..16}$`))`

The two left-aligned vector byte values in register *rt* are taken, and each expanded to a 16-bit unsigned integer value. The two results thus obtained are then written into the destination register *rd* in the corresponding vector location. The expansion is done by pre-pending 8 most-significant zeros to generate the 16 bit unsigned integer value.

Restrictions:

No data-dependent exceptions are possible.

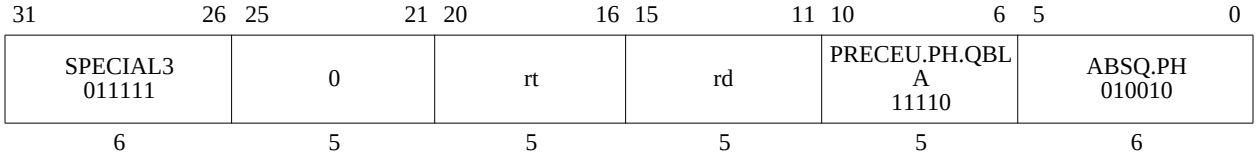
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
tempu ← 0^8 || GPR[rt]31..24
tempv ← 0^8 || GPR[rt]23..16
GPR[rd] ← tempu15..0 || tempv15..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `PRECEU.PH.QBLA rd, rt` **MIPSDSP**

Purpose:

Expand the precision of two unsigned bytes from a left-alternate aligned position in a vector to unsigned halfword integer values

Description: `rd ← expand_prec(rt`$_{31..24}$`)) || expand_prec(rt`$_{15..8}$`))`

The two left-alternate aligned vector byte values in register *rt* are taken, and each expanded to an unsigned halfword integer value. The two results thus obtained are then written into the destination register *rd* in the corresponding vector location. The expansion is done by pre-pending eight most-significant zeros to generate the 16 bit unsigned value.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

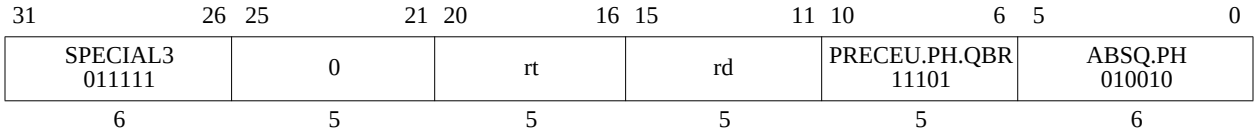
Operation:

```
tempu ← 0^8 || GPR[rt]31..24
tempv ← 0^8 || GPR[rt]15..8
GPR[rd] ← tempu15..0 || tempv15..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Precision Expand Two Unsigned Bytes to Unsigned Integer Halfwords — PRECEU.PH.QBR



Format: `PRECEU.PH.QBR rd, rt` **MIPSDSP**

Purpose:

Expand the precision of two unsigned bytes from a right-aligned position in a vector to unsigned integer halfword values

Description: `rd ← expand_prec(rt`$_{15..8}$`)) || expand_prec(rt`$_{7..0}$`))`

The two right-aligned vector byte values in register *rt* are taken, and each expanded to an unsigned integer halfword value. The two results thus obtained are then written into the destination register *rd* in the corresponding vector location. The expansion is done by pre-pending 8 most-significant zeros to generate the 16 bit integer halfword value.

Restrictions:

No data-dependent exceptions are possible.

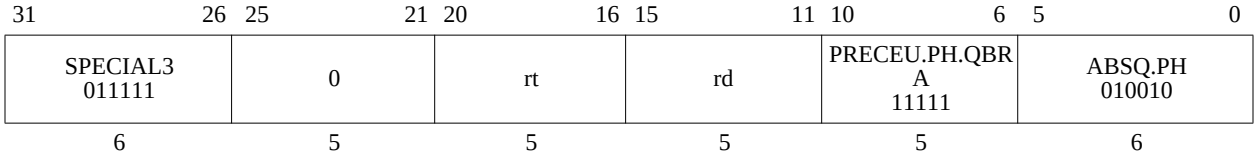
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
tempu ← 0^8 || GPR[rt]15..8
tempv ← 0^8 || GPR[rt]7..0
GPR[rd] ← tempu15..0 || tempv15..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `PRECEU.PH.QBRA rd, rt` **MIPSDSP**

Purpose:

Expand the precision of two unsigned bytes from a right-alternate aligned position in a vector to unsigned integer halfword values

Description: `rd ← expand_prec(rt`$_{23..16}$`)) || expand_prec(rt`$_{7..0}$`))`

The two right-alternate aligned vector byte values in register *rt* are taken, and each expanded to an unsigned integer halfword value. The two results thus obtained are then written into the destination register *rd* in the corresponding vector location. The expansion is done by pre-pending 8 most-significant zeros to generate the 16 bit unsigned value.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

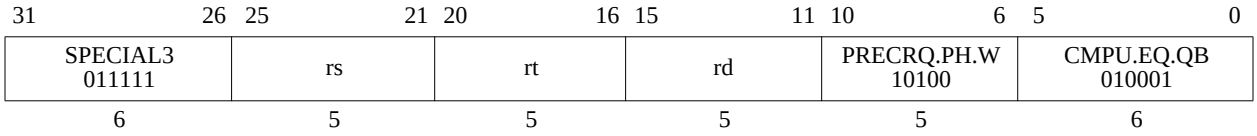
Operation:

```
tempu ← 0^8 || GPR[rt]23..16
tempv ← 0^8 || GPR[rt]7..0
GPR[rd] ← tempu15..0 || tempv15..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Precision Reduce Two Fractional Words to Two Half-Words — PRECRQ.PH.W



Format: `PRECRQ.PH.W rd, rs, rt` **MIPSDSP**

Purpose:

Reduce the precision of two fractional Q31 values to two Q15 values.

Description: $rd \leftarrow rs_{31..16} \mid\mid rt_{31..16}$

The fractional Q31 values in registers *rs* and *rt* are taken, and the 16 least significant bits are truncated to produce two Q15 values, which are then written to the *rd* register. The value thus obtained from *rs* is put in the most significant 16 bits of *rd* and the value obtained from *rt* is put in the least significant 16 bits of *rd*.

Restrictions:

No data-dependent exceptions are possible.

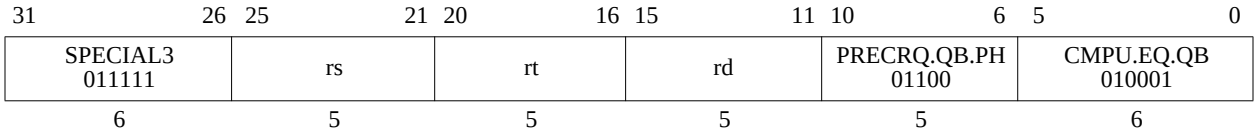
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
tempu ← GPR[rs]31..16
tempv ← GPR[rt]31..16
GPR[rd] ← tempu15..0 || tempv15..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `PRECRQ.QB.PH rd, rs, rt` **MIPSDSP**

Purpose:

Reduce the precision of four fractional halfwords to four byte values.

Description: $rd \leftarrow rs_{31..24} \mid\mid rs_{15..8} \mid\mid rt_{31..24} \mid\mid rt_{15..8}$

The four fractional Q15 values in registers *rs* and *rt* are taken, and the 8 least significant bits are dropped from each value to produce four byte values, which are then written to the *rd* register. The values thus obtained from *rs* are put in the most significant 16 bits of *rd* and the values obtained from *rt* are put in the least significant 16 bits of *rd*.

Restrictions:

No data-dependent exceptions are possible.

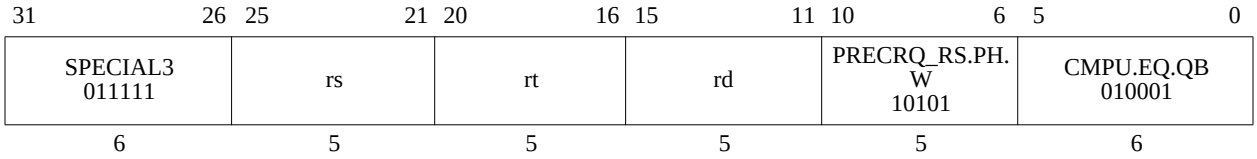
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
tempu ← GPR[rs]31..24
tempv ← GPR[rs]15..8
tempw ← GPR[rt]31..24
tempx ← GPR[rt]15..8
GPR[rd] ← tempu7..0 || tempv7..0 || tempw7..0 || tempx7..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `PRECRQ_RS.PH.W rd, rs, rt` **MIPSDSP**

Purpose:

Reduce the precision of two fractional Q31 values to two Q15 values with rounding and saturation.

Description: `rd ← truncq15(sat32(round16((rs))) || truncq15(sat32(round16((rt)))`

The fractional Q31 values in registers *rs* and *rt* are taken, and precision reduced with round and saturate to produce two Q15 values, which are then written to the destination *rd* register. The value obtained from *rs* is put in the most significant 16 bits of *rd* and the value obtained from *rt* is put in the least significant 16 bits of *rd*. The rounded and saturated precision reduction procedure is as follows. The 32-bit value from the respective source register is added with 0x8000 to round up even. If this addition causes an overflow, the value is saturated to the maximum value. After rounding and saturation, the least significant 16 bits are truncated and the most significant 16 bits are written to the destination register in the appropriate position as already described.

This instruction, on overflow of either source value, sets bit 22 of the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

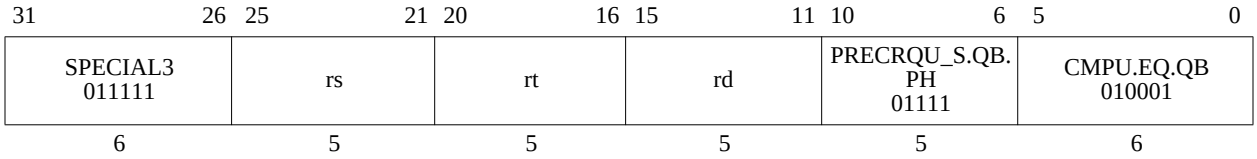
The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
temp1 ← GPR[rs] + 0x8000
if temp1_32 ≠ temp1_31 then
   DSPControl_ouflag:22 ← 1
   temp2 ← 0x7FFF
else
   temp2 ← temp1_31..16
endif

temp3 ← GPR[rt] + 0x8000
if temp3_32 ≠ temp3_31 then
   DSPControl_ouflag:22 ← 1
   temp4 ← 0x7FFF
else
   temp4 ← temp3_31..16
endif

GPR[rd] ← temp2_15..0 || temp4_15..0
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `PRECRQU_S.QB.PH rd, rs, rt` **MIPSDSP**

Purpose:

Reduce the precision of four fractional Q15 values to four unsigned 8-bit values with saturation.

Description: `rd ← sat(reduce_prec(rs`$_{31:16}$`)) || sat(reduce_prec(rs`$_{15:0}$`)) || sat(reduce_prec(rt`$_{31:16}$`)) || sat(reduce_prec(rt`$_{15:0}$`))`

The vector fractional Q15 values in registers *rs* and *rt* are taken and the corresponding unsigned saturated 8 bit values are generated, which are then written to the *rd* register. The values thus obtained from *rs* are put in the most significant 16 bits of *rd* and the values obtained from *rt* are put in the least significant 16 bits of *rd*. To obtain the unsigned saturated result, first the value is checked for its sign, if it is positive, and the value is less than or equal to 0x7F80, then the sign bit is dropped and the next 8 bits are taken to be the result. If the value is positive and greater than 0x7F80, then the value is clamped to the maximum unsigned 8-bit value, i.e., 0xFF. If the sign bit indicates that the value is negative, then the result is obtained by clamping the result to the least positive value, which is 8 zeroes.

This instruction, on underflow of any of the source values, sets bit 22 of the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

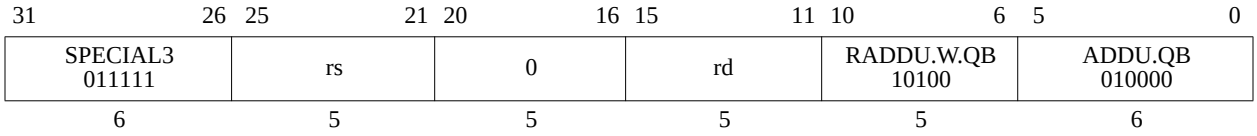
```
satcond1 ← (GPR[rs]31 ≠ 0)
satcond2 ← (GPR[rs]15 ≠ 0)
satcond3 ← (GPR[rt]31 ≠ 0)
satcond4 ← (GPR[rt]15 ≠ 0)
satcond5 ← (GPR[rs]31 = 0) and (GPR[rs]31..16 > 0x7F80)
satcond6 ← (GPR[rs]15 = 0) and (GPR[rs]15..0 > 0x7F80)
satcond7 ← (GPR[rt]31 = 0) and (GPR[rt]31..16 > 0x7F80)
satcond8 ← (GPR[rt]15 = 0) and (GPR[rt]15..0 > 0x7F80)
if (satcond1 or satcond2 or satcond3 or satcond4 or satcond5 or satcond6 or
satcond7 or satcond8) then
   DSPControlouflag:22 ← 1
endif
if satcond1 then
   tempu ← 0
else if satcond5 then
   tempu ← 0xFF
else
   tempu ← GPR[rs]30..23
endif
if satcond2 then
   tempv ← 0
else if satcond6 then
   tempv ← 0xFF
else
```

```
    tempv ← GPR[rs]14..7
endif
if satcond3 then
    tempw ← 0
else if satcond7 then
    tempw ← 0xFF
else
    tempw ← GPR[rt]30..23
endif
if satcond4 then
    tempx ← 0
else if satcond8 then
    tempx ← 0xFF
else
    tempx ← GPR[rt]14..7
endif
GPR[rd] ← tempu7..0 || tempv7..0 || tempw7..0 || tempx7..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Unsigned Reduction Add Vector Quad-Bytes RADDU.W.QB



Format: `RADDU.W.QB rd, rs` **MIPSDSP**

Purpose:

Reduction add of four unsigned byte values in a vector register to produce a word result.

Description: `rd ← zero-extend(rs31:24 + rs23:16 + rs15:8 + rs7:0)`

Vector quad-byte values in register *rs* are added together as unsigned 8-bit values and zero extended to a word size before writing the result to the destination register *rd*.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

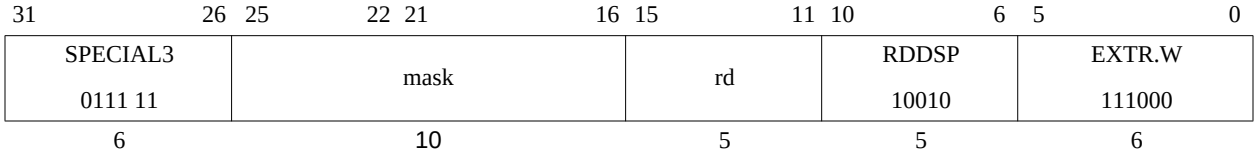
Operation:

```
temp ← GPR[rs]31..24  + GPR[rs]23..16  + GPR[rs]15..8  + GPR[rs]7..0
GPR[rd] ← zero_extend(temp)
```

Exceptions:

Reserved Instruction, DSP Disabled

Read Fields From DSPControl Register to a GPR — RDDSP



Format: `RDDSP rd, mask` **MIPSDSP**
`RDDSP rd` **Assembly Idiom**

Purpose:

To copy selectable fields from the special purpose *DSPControl* register to the specified GPR.

Description: `rd ← select[mask, DSPControl]`

Certain fields from the special register *DSPControl* register selected by the bits in the specified mask field in the instruction are loaded into the destination GPR *rd*. Each bit in the mask value corresponds to a specific field in the *DSPControl* register. If the bit corresponding to a field is 1, then this implies that the field is to be read into the destination register. Each field in the *DSPControl* register is read into the same bit positions in the destination register. For example, if the *scount* field is to be read, and this field is at bit positions 12..7 in *DSPControl*, then after the execution of this instruction the bits of the *scount* field are available in bits 12..7 of the destination register *rd*. If the mask bit is 0, then after the execution of this instructions, the bits in the destination register corresponding to the position of that field in the *DSPControl* register are zeros. Bits 21..16 of the mask field, correspond to fields *EFI*, *ccond*, *ouflag*, *c*, *scount*, and *pos*, respectively. The hardware ignores the upper un-used bits of the mask field.

The one-operand version of the instruction provides a convenient assembly idiom that allows the programmer to read all the allowable fields of the DSPControl register into the destination GPR.

Restrictions:

None

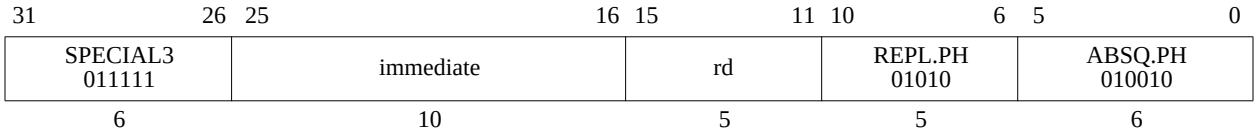
Operation:

```
temp31..0 ← 0
if (mask0 = 1) then
    temp5..0 ← DSPControlpos:5..0
endif
if (mask1 = 1) then
    temp12..7 ← DSPControlscount:12..7
endif
if (mask2 = 1) then
    temp13 ← DSPControlc:13
endif
if (mask3 = 1) then
    temp23..16 ← DSPControlouflag:23..16
endif
if (mask4 = 1) then
    temp27..24 ← DSPControlccond:27..24
endif
if (mask5 = 1) then
    temp14 ← DSPControlEFI:14
endif

GPR[rd] ← temp
```

Exceptions:

Reserved Instruction, DSP Disabled

Replicate a Fixed Halfword into all Vector Element Positions — REPL.PH



Format: `REPL.PH rd, immediate` **MIPSDSP**

Purpose:

Replicate a fixed value into two halfword vector elements.

Description: `rd ← sign_extend(immediate) || sign_extend(immediate)`

The specified 10-bit immediate field is sign-extended to a 16 bit value and replicated into the two halfword vector positions in the destination register *rd*.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

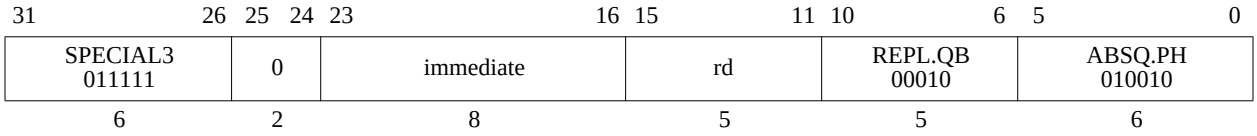
Operation:

```
temp ← sign_extend(immediate)
GPR[rd] ← temp15..0 || temp15..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Replicate a Fixed Byte into all Vector Element Positions — REPL.QB



Format: `REPL.QB rd, immediate` **MIPSDSP**

Purpose:

Replicate a fixed byte value into four byte vector elements.

Description: `rd ← immediate || immediate || immediate|| immediate`

The specified 8-bit immediate field is replicated into the four byte vector positions in the destination register *rd*.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

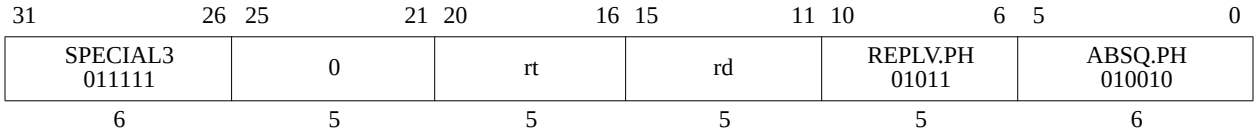
Operation:

```
temp ← immediate
GPR[rd] ← temp7..0 || temp7..0 || temp7..0 || temp7..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Replicate a Halfword into all Vector Element Positions — REPLV.PH



Format: `REPLV.PH rd, rt` **MIPSDSP**

Purpose:

Replicate a variable halfword into two halfword vector elements.

Description: $rd \leftarrow rt_{15..0} \mid\mid rt_{15..0}$

The least-significant halfword value in register *rt* is replicated into the two halfword vector positions in the destination register *rd*.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

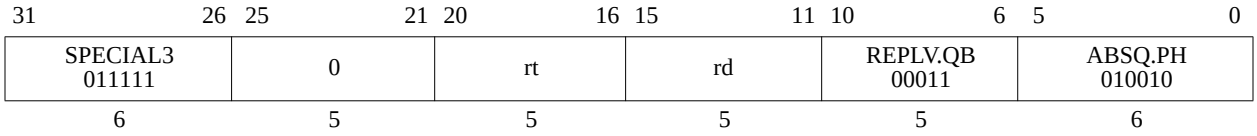
Operation:

```
temp ← GPR[rt]15..0
GPR[rd] ← temp15..0 || temp15..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Replicate a Byte into all Vector Element Positions — REPLV.QB



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL3 011111	0	rt	rd	REPLV.QB 00011	ABSQ.PH 010010
6	5	5	5	5	6

Format: `REPLV.QB rd, rt` **MIPSDSP**

Purpose:

Replicate a variable byte value into four byte vector elements.

Description: $rd \leftarrow rt_{7..0} \,||\, rt_{7..0} \,||\, rt_{7..0} \,||\, rt_{7..0}$

The least-significant bye value in register *rt* is replicated into the four byte vector positions in the destination register *rd*.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

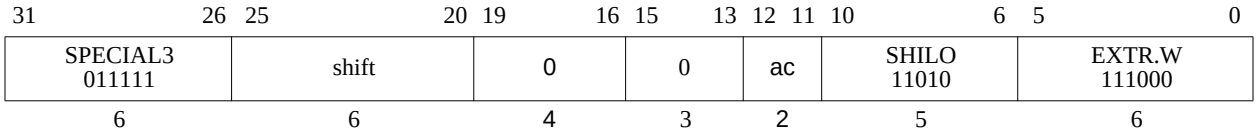
Operation:

```
temp ← GPR[rt]7..0
GPR[rd] ← temp7..0 || temp7..0 || temp7..0 || temp7..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Shift an Accumulator Value Leaving the Result in the Same Accumulator — SHILO



Format: `SHILO ac, shift` **MIPSDSP**

Purpose:

Shift the *HI/LO* paired value in an accumulator either left or right, leaving the result in the same accumulator.

Description: `ac[63:0] ← (shift>=0) ? (ac[63:0] >> shift) : (ac[63:0] << -shift)`

The *HI/LO* paired value is considered as a single 64 bit value in the accumulator, which is shifted left or right logically by the given shift amount. The six bits of shift is assumed to be a signed value that can range from -32 to +31. A negative value implies a left shift and a positive value is a right shift. The result of the shift is left in the same accumulator. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

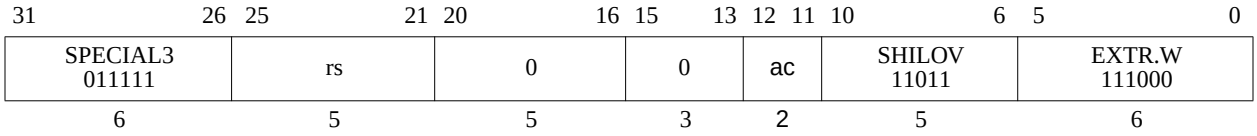
Operation:

```
sa ← shift4..0
sign ← shift5
if (sign = 0)
   temp ← 0^sa || (HI[ac]||LO[ac])63..sa
else
   sa ← -shift5..0
   temp ← (HI[ac]||LO[ac])(63-sa)..0 || 0^sa
endif
(HI[ac]||LO[ac]) ← temp
```

Exceptions:

Reserved Instruction, DSP Disabled

Variable Shift an Accumulator Value Leaving the Result in the Same Accumulator — SHILOV



31–26	25–21	20–16	15–13	12–11	10–6	5–0
SPECIAL3 011111	rs	0	0	ac	SHILOV 11011	EXTR.W 111000
6	5	5	3	2	5	6

Format: `SHILOV ac, rs` **MIPSDSP**

Purpose:

Shift the *HI/LO* paired value in an accumulator either left or right, leaving the result in the same accumulator.

Description: `ac[63:0] ← (rs[5:0]>=0) ? (ac[63:0] >> rs[5:0]) : (ac[63:0] << -rs[5:0])`

The *HI/LO* paired value is considered as a single 64 bit value in the accumulator, which is shifted left or right logically by the given shift amount. The least significant 6 bits of the *rs* register is assumed to be a signed value that can range from -32 to +31. (The upper bits of the rs register are ignored). A negative value implies a left shift and a positive value is a right shift. The value obtained after the shift is left in the same accumulator. The value of *ac* can range from 0 to 3. When *ac*=0, this refers to the original *HI/LO* register pair of the MIPS32 architecture.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

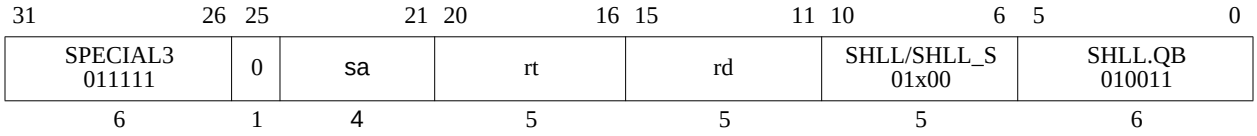
Operation:

```
shift ← GPR[rs]5..0
sa ← shift4..0
sign ← shift5
if (sign = 0)
   temp ← 0^sa || (HI[ac]||LO[ac])63..sa
else
   sa ← -shift
   temp ← (HI[ac]||LO[ac])(63-sa)..0 || 0^sa
endif
(HI[ac]||LO[ac]) ← temp
```

Exceptions:

Reserved Instruction, DSP Disabled

Shift Left Logical Vector of Paired Half-Words — SHLL.PH



Format: `SHLL.PH rd, rt, sa` **MIPSDSP**

`SHLL_S.PH rd, rt, sa` **MIPSDSP**

Purpose:

To execute a left shift of two independent halfwords in a vector data type by a fixed number of bits.

Description: `rd ← (rt`$_{31:16}$ `<< sa) || (rt`$_{15:0}$ `<< sa)`

Two halfword values in register *rt* are separately shifted left, inserting zeros into the emptied bits, before writing the two values thus obtained into respective vector positions in the destination register *rd*. The 4 bits of the *sa* field specifies the shift value.

If saturation is desired, then a signed overflow on the left shift sets the result of the shift to either the maximum positive or maximum negative value. This saturation is done independently for each vector element operation.

This instruction, on an overflow of any of the left shift operations, writes bit 22 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

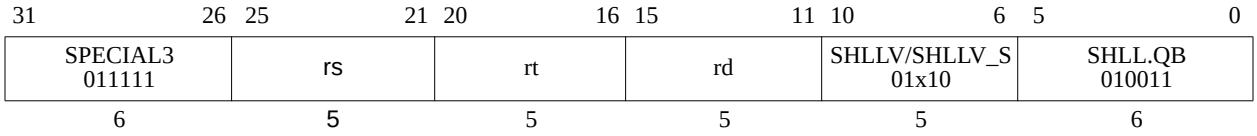
```
SHLL.PH
   SetouflagAndValues(GPR[rt], sa, setcond1, setcond2, setcond3, setcond4, tu, tv)
   GPR[rd]← tv || tu

SHLL_S.PH
   SetouflagAndValues(GPR[rt], sa, setcond1, setcond2, setcond3, setcond4, tu, tv)
   if (sa ≠ 0) then
      if setcond1 then
         tv ← 0xFFFF
      else if setcond2 then
         tv ← 0x8000
      endif
   endif
   if (sa ≠ 0) then
      if setcond3 then
         tu ← 0xFFFF
      else if setcond4 then
         tu ← 0x8000
      endif
   endif
   GPR[rd]← tv || tu

function SetouflagAndValues(t, sa, setcond1, setcond2, setcond3, setcond4, t1, t2)
   setcond1 ← setcond2 ← setcond3 ← setcond4 ← 0
   if (sa ≠ 0) then
```

```
            setcond1 ← ((t31 = 0) and t30..(31-sa) ≠ 0)
            setcond2 ← ((t31 = 1) and t30..(31-sa) ≠ 1sa)
            setcond3 ← ((t15 = 0) and t14..(15-sa) ≠ 0)
            setcond4 ← ((t15 = 1) and t14..(15-sa) ≠ 1sa)
            if (setcond1 or setcond2 or setcond3 or setcond4) then
               DSPControlouflag:22 ← 1
            endif
         endif
         t1 ← GPR[rt](15-sa)..0 || 0sa
         t2 ← GPR[rt](31-sa)..16 || 0sa
      endfunction SetouflagAndValues
```

Exceptions:

Reserved Instruction, DSP Disabled

Shift Left Logical Variable Vector of Paired Half-Words — SHLLV.PH



Format: `SHLLV.PH rd, rt, rs` **MIPSDSP**

`SHLLV_S.PH rd, rt, rs` **MIPSDSP**

Purpose:

To execute a left shift of two independent halfwords in a vector data type by a variable number of bits.

Description: `rd ← (rt`$_{31:16}$ `<< rs) || (rt`$_{15:0}$ `<< rs)`

Two halfword values in register *rt* are separately shifted left, inserting zeros into the emptied bits, before writing the two values thus obtained into respective vector positions in the destination register *rd*. The least-significant 4 bits of the *rs* register specifies the shift value.

If saturation is desired, then a signed overflow on the left shift sets the result of the shift to either the maximum positive or maximum negative value. This saturation is done independently for each vector element operation.

This instruction, on an overflow of any of the left shift operations, writes bit 22 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

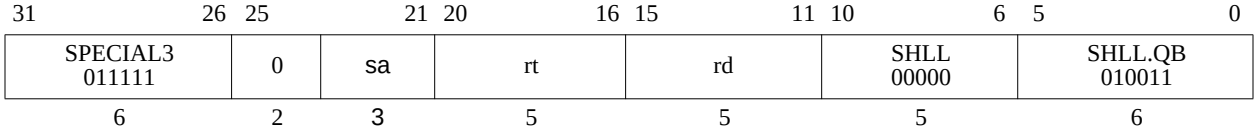
```
SHLLV.PH
    sa ← GPR[rs]3..0
    SetouflagAndValues(GPR[rt], sa, setcond1, setcond2, setcond3, setcond4, tu, tv)
    GPR[rd]← tv || tu

SHLLV_S.PH
    sa ← GPR[rs]3..0
    SetouflagAndValues(GPR[rt], sa, setcond1, setcond2, setcond3, setcond4, tu, tv)
    if (sa ≠ 0) then
        if setcond1 then
            tv ← 0xFFFF
        else if setcond2 then
            tv ← 0x8000
        endif
    endif
    if (sa ≠ 0) then
        if setcond3 then
            tu ← 0xFFFF
        else if setcond4 then
            tu ← 0x8000
        endif
    endif
    GPR[rd]← tv || tu

function SetouflagAndValues(t, sa, setcond1, setcond2, setcond3, setcond4, t1, t2)
```

```
    setcond1 ← setcond2 ← setcond3 ← setcond4 ← 0
    if (sa ≠ 0) then
       setcond1 ← ((t31 = 0) and t30..(31-sa) ≠ 0)
       setcond2 ← ((t31 = 1) and t30..(31-sa) ≠ 1^sa)
       setcond3 ← ((t15 = 0) and t14..(15-sa) ≠ 0)
       setcond4 ← ((t15 = 1) and t14..(15-sa) ≠ 1^sa)
       if (setcond1 or setcond2 or setcond3 or setcond4) then
          DSPControlouflag:22 ← 1
       endif
    endif
    t1 ← GPR[rt](15-sa)..0 || 0^sa
    t2 ← GPR[rt](31-sa)..16 || 0^sa
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Shift Left Logical Vector Quad-Bytes — SHLL.QB



Format: `SHLL.QB rd, rt, sa` **MIPSDSP**

Purpose:

To execute a left shift of four independent bytes in a vector data type by a fixed number of bits.

Description: $rd \leftarrow (rt_{31:24} << sa) \,||\, (rt_{23:16} << sa) \,||\, (rt_{15:8} << sa) \,||\, (rt_{7:0} << sa)$

Four byte values in register *rt* are separately shifted left, inserting zeros into the emptied bits, before writing the four values thus obtained into respective vector positions in the destination register *rd*. The 3 bits of the *sa* field specifies the shift value of 0 to 7 bit positions.

This instruction, on an overflow of any of the left shift operations, writes bit 22 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

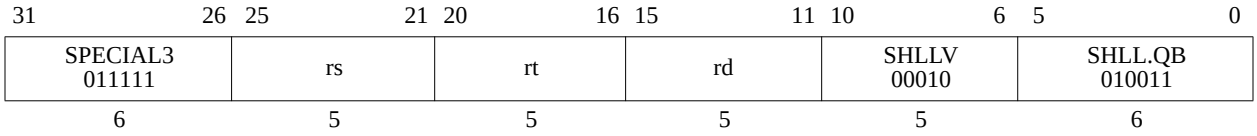
Operation:

```
setcond1 ← setcond2 ← setcond3 ← setcond4 ← 0
if (sa ≠ 0) then
   setcond1 ← (GPR[rt]7..(7-(sa-1)) ≠ 0)
   setcond2 ← (GPR[rt]15..(15-(sa-1)) ≠ 0)
   setcond3 ← (GPR[rt]23..(23-(sa-1)) ≠ 0)
   setcond4 ← (GPR[rt]31..(31-(sa-1)) ≠ 0)
   if (setcond1 or setcond2 or setcond3 or setcond4) then
      DSPControlouflag:22 ← 1
   endif
endif
temp1 ← GPR[rt](7-sa)..0 || 0sa
temp2 ← GPR[rt](15-sa)..8 || 0sa
temp3 ← GPR[rt](23-sa)..16 || 0sa
temp4 ← GPR[rt](31-sa)..24 || 0sa
GPR[rd]← temp4 || temp3 || temp2 || temp1
```

Exceptions:

Reserved Instruction, DSP Disabled

Shift Left Logical Variable Vector Quad-Bytes — SHLLV.QB



Format: `SHLLV.QB rd, rt, rs` **MIPSDSP**

Purpose:

To execute a left shift of four independent bytes in a vector data type by a variable number of bits.

Description: $rd \leftarrow (rt_{31:24} << rs) \,||\, (rt_{23:16} << rs) \,||\, (rt_{15:8} << rs) \,||\, (rt_{7:0} << rs)$

Four byte values in register *rt* are separately shifted left, inserting zeros into the empty bits, before writing the four values thus obtained into respective vector positions in the destination register *rd*. The 3 right-most bits of register *rs* are used as the shift value. The upper bits of *rs* are ignored in this case.

This instruction, on an overflow of any of the left shift operations, writes bit 22 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

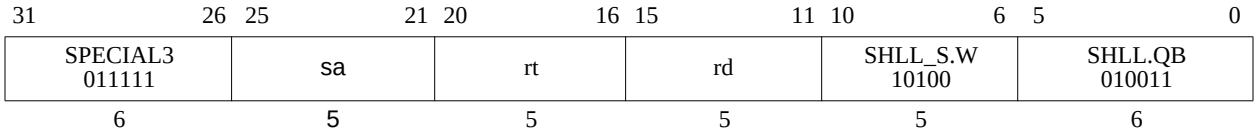
Operation:

```
sa ← GPR[rs]2..0
setcond1 ← setcond2 ← setcond3 ← setcond4 ← 0
if (sa ≠ 0) then
   setcond1 ← (GPR[rt]7..(7-(sa-1)) ≠ 0)
   setcond2 ← (GPR[rt]15..(15-(sa-1)) ≠ 0)
   setcond3 ← (GPR[rt]23..(23-(sa-1)) ≠ 0)
   setcond4 ← (GPR[rt]31..(31-(sa-1)) ≠ 0)
   if (setcond1 or setcond2 or setcond3 or setcond4) then
      DSPControlouflag:22 ← 1
   endif
endif
temp1← GPR[rt](7-s)..0 || 0s
temp2← GPR[rt](15-s)..8 || 0s
temp3← GPR[rt](23-s)..16 || 0s
temp4← GPR[rt](31-s)..24 || 0s
GPR[rd]← temp4 || temp3 || temp2 || temp1
```

Exceptions:

Reserved Instruction, DSP Disabled

Shift Left Logical Word with Saturation — SHLL_S.W



Format: `SHLL_S.W rd, rt, sa` **MIPSDSP**

Purpose:

To execute a left shift of a word with saturation by a fixed number of bits.

Description: `rd ← sat32(rt << sa)`

The word in register *rt* is shifted left, inserting zeros into the emptied bits, and saturated on a signed overflow before writing the result thus obtained into the destination register *rd*. The 5 bits of the *sa* field specifies the shift value.

The saturation operation, on a signed overflow on the left shift sets the result of the shift to either the maximum positive or maximum negative value.

This instruction, on an overflow of the left shift operation, writes bit 22 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

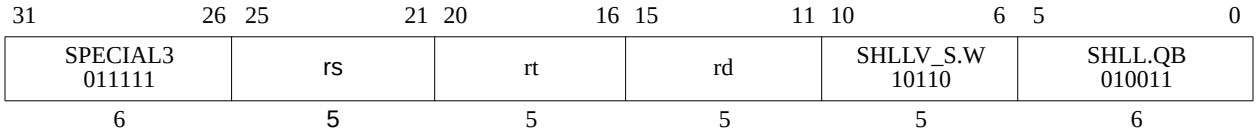
```
setcond1 ← setcond2 ← 0
if (sa ≠ 0) then
    setcond1 ← ((t31 = 0) and t30..(31-sa) ≠ 0)
    setcond2 ← ((t31 = 1) and t30..(31-sa) ≠ 1^sa)
    if (setcond1 or setcond2) then
        DSPControlouflag:22 ← 1
    endif
endif
temp ← GPR[rt](31-sa)..0 || 0^sa
if (sa ≠ 0) then
    if setcond1 then
        temp ← 0xFFFFFFFF
    else if setcond2 then
        temp ← 0x80000000
    endif
endif
GPR[rd]← temp
```

Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

To do a logical left shift of a word in a register without saturation, use the MIPS32 SLL instruction.



Format: `SHLLV_S.W rd, rt, rs` **MIPSDSP**

Purpose:

To execute a left shift of a word with saturation by a variable number of bits.

Description: `rd ← sat32(rt << rs)`

The word in register *rt* is shifted left, inserting zeros into the emptied bits, and saturated on a signed overflow before writing the result thus obtained into the destination register *rd*. The least-significant 5 bits of the *rs* register specifies the shift value.

The saturation operation, on a signed overflow on the left shift sets the result of the shift to either the maximum positive or maximum negative value. The logical variable left shift of a word value without the saturation operation can be obtained by using the MIPS32 SLLV instruction.

This instruction, on an overflow of the left shift operation, writes bit 22 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
sa ← GPR[rs]4..0
setcond1 ← setcond2 ← 0
if (sa ≠ 0) then
   setcond1 ← ((t31 = 0) and t30..(31-sa) ≠ 0)
   setcond2 ← ((t31 = 1) and t30..(31-sa) ≠ 1sa)
   if (setcond1 or setcond2) then
      DSPControlouflag:22 ← 1
   endif
endif
temp ←GPR[rt](31-sa)..0 || 0sa
if (sa ≠ 0) then
   if setcond1 then
      temp ←0xFFFFFFFF
   else if setcond2 then
      temp ←0x80000000
   endif
endif
GPR[rd]← temp
```

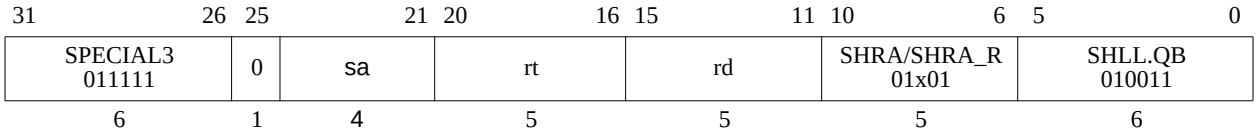
Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

To do a variable logical left shift of a word in a register without saturation, use the MIPS32 SLLV instruction.

Shift Right Arithmetic Vector of Paired Half-Words — SHRA.PH



Format: `SHRA.PH rd, rt, sa` **MIPSDSP**

`SHRA_R.PH rd, rt, sa` **MIPSDSP**

Purpose:

To execute an arithmetic right shift on two independent halfwords in a vector data type by a fixed number of bits.

Description: `rd ← (rt`$_{31:16}$ `>> sa) || (rt`$_{15:0}$ `>> sa)`

Two halfword values in register *rt* are separately shifted right, duplicating the sign bit of the respective half-word in the emptied bits, before writing the two values thus obtained into respective vector positions in the destination register *rd*. The bit-shift amount is specified by the 4 bits of the *sa* field.

If rounding is desired, then after the shifting is done, a 1 bit is added to the most significant discarded bit of the result. This rounding is done independently for each halfword vector element.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

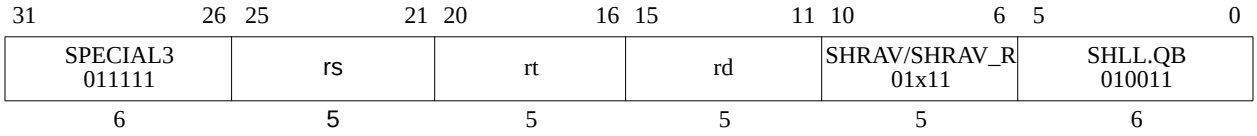
Operation:

```
SHRA.PH
    tempu ← (GPR[rt]31)^sa || GPR[rt]31..16+sa
    tempv ← (GPR[rt]15)^sa || GPR[rt]15..sa
    GPR[rd]← tempu || tempv

SHRA_R.PH
    if (sa ≠ 0) then
        tempu ← ((GPR[rt]31)^sa || GPR[rt]30..16+sa || GPR[rt]16+sa-1) + 1#1
        tempv ← ((GPR[rt]15)^sa || GPR[rt]14..sa || GPR[rt]sa-1) + 1#1
    else
        tempu ← (GPR[rt]31)^sa || GPR[rt]31..16+sa
        tempv ← (GPR[rt]15)^sa || GPR[rt]15..sa
    endif
    GPR[rd]← tempu || tempv
```

Exceptions:

Reserved Instruction, DSP Disabled

Shift Right Arithmetic Variable Vector of Paired Half-Words — SHRAV.PH



Format: `SHRAV.PH rd, rt, rs` **MIPSDSP**

`SHRAV_R.PH rd, rt, rs` **MIPSDSP**

Purpose:

To execute an arithmetic right shift on two independent halfwords in a vector data type by a variable number of bits.

Description: $rd \leftarrow (rt_{31:16} >> rs) \ || \ (rt_{15:0} >> rs)$

Two halfword values in register *rt* are separately shifted right, duplicating the sign bit of the respective half-word in the emptied bits, before writing the two values thus obtained into respective vector positions in the destination register *rd*. The bit-shift amount is specified by the least-significant 4 bits of the *rs* source register.

If rounding is desired, then after the shifting is done, a 1 bit is added to the most significant discarded bit of the result. This rounding is done independently for each halfword vector element.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

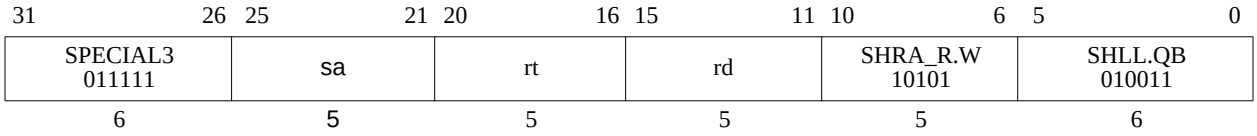
Operation:

```
SHRA.PH
    sa ← GPR[rs]3..0
    tempu ← (GPR[rt]31)^sa || GPR[rt]31..16+sa
    tempv ← (GPR[rt]15)^sa || GPR[rt]15..sa
    GPR[rd]← tempu || tempv

SHRA_R.PH
    sa ← GPR[rs]3..0
    if (sa ≠ 0) then
        tempu ← ((GPR[rt]31)^sa || GPR[rt]30..16+sa || GPR[rt]16+sa-1) + 1#1
        tempv ← ((GPR[rt]15)^sa || GPR[rt]14..sa || GPR[rt]sa-1) + 1#1
    else
        tempu ← (GPR[rt]31)^sa || GPR[rt]31..16+sa
        tempv ← (GPR[rt]15)^sa || GPR[rt]15..sa
    endif
    GPR[rd]← tempu || tempv
```

Exceptions:

Reserved Instruction, DSP Disabled

Shift Right Arithmetic Word with Rounding — SHRA_R.W



Format: `SHRA_R.W rd, rt, sa` **MIPSDSP**

Purpose:

To execute an arithmetic right shift on a word by a fixed number of bits and round.

Description: `rd ← rnd32(rt`$_{31:0}$ `>> sa)`

The word in register *rt* is shifted right, duplicating the sign bit, i.e., bit 31 into the emptied bits. This shifted result is then rounded by adding a 1 bit to the most significant discarded bit before writing the value thus obtained to the destination register *rd*. The bit-shift amount is specified by the 5 bits of the *sa* field.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
if (sa ≠ 0) then
   temp ← ((GPR[rt]31)^sa || GPR[rt]30..sa || GPR[rt]sa-1) + 1#1
else
   temp ← (GPR[rt]31)^sa || GPR[rt]31..sa
endif
GPR[rd]← temp
```

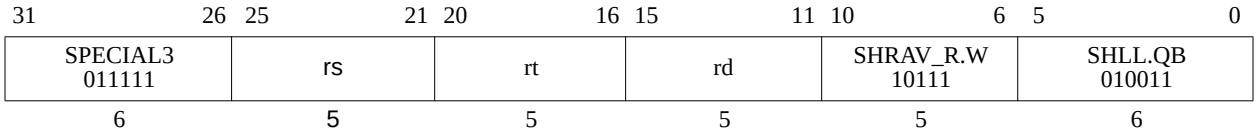
Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

To do an arithmetic right shift of a word in a register without rounding, use the MIPS32 SRA instruction.

Shift Right Arithmetic Word Variable with Rounding — SHRAV_R.W



Format: `SHRAV_R.W rd, rt, rs` **MIPSDSP**

Purpose:

To execute an arithmetic right shift on a word by a variable number of bits and round.

Description: `rd ← rnd32(rt`$_{31:0}$ `>> rs)`

The word in register *rt* is shifted right, duplicating the sign bit, i.e., bit 31 into the emptied bits. This shifted result is then rounded by adding a 1 bit to the most significant discarded bit before writing the value thus obtained to the destination register *rd*. The bit-shift amount is specified by the least significant 5 bits of the *rs* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
sa ← GPR[rs]4..0
if (sa ≠ 0) then
   temp ← ((GPR[rt]31)^sa || GPR[rt]30..sa || GPR[rt]sa-1) + 1#1
else
   temp ← (GPR[rt]31)^sa || GPR[rt]31..sa
endif
GPR[rd]← temp
```

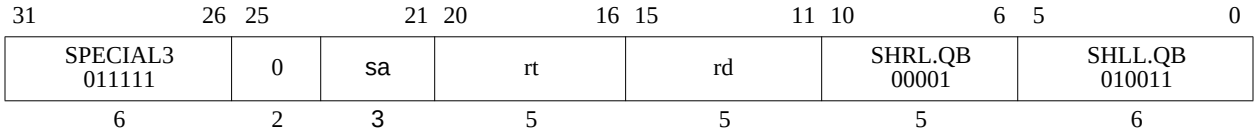
Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

To do a variable arithmetic right shift of a word in a register without rounding, use the MIPS32 SRAV instruction.

Shift Right Logical Vector Quad-Bytes — SHRL.QB



Format: `SHRL.QB rd, rt, sa` **MIPSDSP**

Purpose:

To execute a right shift of four independent bytes in a vector data type by a fixed number of bits.

Description: `rd ← (rt31:24 >> sa) || (rt23:16 >> sa) || (rt15:8 >> sa) || (rt7:0 >> sa)`

Four byte values in register *rt* are separately shifted right, inserting zeros into the emptied bits, before writing the four values thus obtained into respective vector positions in the destination register *rd*. The 3 bits of the *sa* field specifies the shift value.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

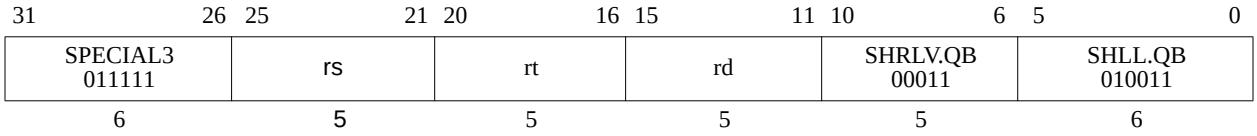
Operation:

```
temp1 ← 0^sa || GPR[rt]7..sa
temp2 ← 0^sa || GPR[rt]15..sa+8
temp3 ← 0^sa || GPR[rt]23..sa+16
temp4 ← 0^sa || GPR[rt]31..sa+24
GPR[rd]← temp4 || temp3 || temp2 || temp1
```

Exceptions:

Reserved Instruction, DSP Disabled

Shift Right Logical Vector Quad-Bytes — SHRLV.QB



Format: `SHRLV.QB rd, rt, rs` **MIPSDSP**

Purpose:

To execute a right shift of four independent bytes in a vector data type by a variable number of bits.

Description: `rd ← (rt31:24 >> rs) || (rt23:16 >> rs) || (rt15:8 >> rs) || (rt7:0 >> rs)`

Four byte values in register *rt* are separately shifted right, inserting zeros into the emptied bits, before writing the four values thus obtained into respective vector positions in the destination register *rd*. The least-significant 3 bits of the *rs* register specifies the shift value.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

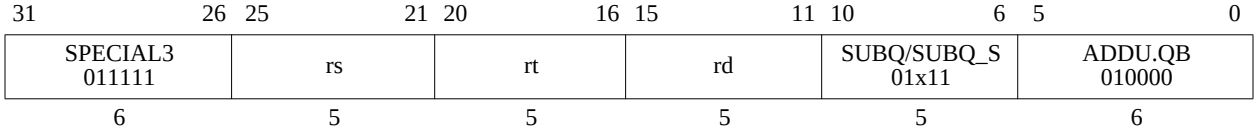
Operation:

```
s   ← GPR[rs]2..0
temp1  ← 0^s || GPR[rt]7..s
temp2← 0^s || GPR[rt]15..s+8
temp3← 0^s || GPR[rt]23..s+16
temp4← 0^s || GPR[rt]31..s+24
GPR[rd]← temp4 || temp3 || temp2 || temp1
```

Exceptions:

Reserved Instruction, DSP Disabled

Subtract Vector Fractional Half-Words SUBQ.PH



Format: `SUBQ.PH rd, rs, rt` **MIPSDSP**

`SUBQ_S.PH rd, rs, rt` **MIPSDSP**

Purpose:

Subtract two vector paired-half fractional Q15 values with optional saturation to obtain a Q15 paired-half result.

Description: `rd ← sat16(rs`$_{31:16}$ `- rt`$_{31:16}$`) | sat16(rs`$_{15:0}$ `- rt`$_{15:0}$`)`

The two vector fractional Q15 values in registers rs and rt are subtracted separately. For the saturate version of the instruction, signed saturated arithmetic is performed, where overflow and underflow clamp to the largest (0x7FFF) or the smallest (0x8000) representable value respectively before writing the destination register rd. This saturation is performed separately for each vector operation.

This instruction, on an overflow or underflow of any of the vector operations, writes bit 20 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

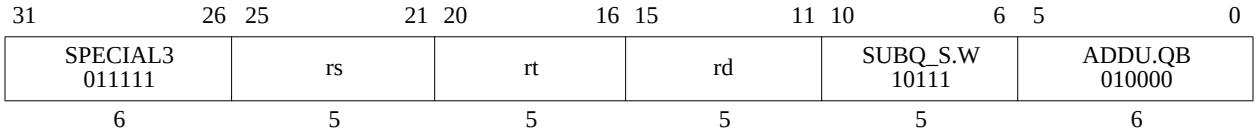
```
SUBQ.PH
    Subtract(tempw, tempt)
    Setouflag(tempw, tempt)
    GPR[rd] ← tempw15..0 || tempt15..0

SUBQ_S.PH
    Subtract(tempw, tempt)
    Setouflag(tempw, tempt, wflag, tflag)
    if (wflag) then
        if (tempw16 = 0) then
            tempw ← 0x7FFF
        else
            tempw ← 0x8000
        endif
    endif
    if (tflag) then
        if (tempt16 = 0) then
            tempt ← 0x7FFF
        else
            tempt ← 0x8000
        endif
    endif
    GPR[rd] ← tempw15..0 || tempt15..0

function Subtract(tempw, tempt)
    tempw ← (GPR[rs]31 || GPR[rs]31..16) - (GPR[rt]31 || GPR[rt]31..16)
    tempt ← (GPR[rs]15 || GPR[rs]15..0) - (GPR[rt]15 || GPR[rt]15..0)
```

```
endfunction Subtract

function Setouflag(w, t, wf, tf)
   if (w16  ≠ w15) then
      DSPControlouflag:20 ← 1
      wf ← 1
   endif
   if (t16  ≠ t15) then
      DSPControlouflag:20 ← 1
      wt ← 1
   endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled

Subtract Fractional Word with Saturation — SUBQ_S.W



Format: `SUBQ_S.W rd, rs, rt` **MIPSDSP**

Purpose:

Subtract two fractional Q31 values with 32 bit saturation.

Description: `rd ← sat32(rs - rt)`

The fractional Q31 value in registers rs and rt are subtracted. Signed saturated arithmetic is performed, where overflow and underflow clamp to the largest (0x7FFFFFFF) or the smallest (0x80000000) representable value respectively before writing the destination register rd.

This instruction, on an overflow or underflow, writes bit 20 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

Operation:

```
temp ← (GPR[rs]31 || GPR[rs]31..0) - (GPR[rt]31 || GPR[rt]31..0)
if (temp32 ≠ temp31) then
   DSPControlouflag:20 ← 1
   if (temp32 = 0) then
      temp ← 0x7FFFFFFF
   else
      temp ← 0x80000000
   endif
endif
GPR[rd] ← temp
```

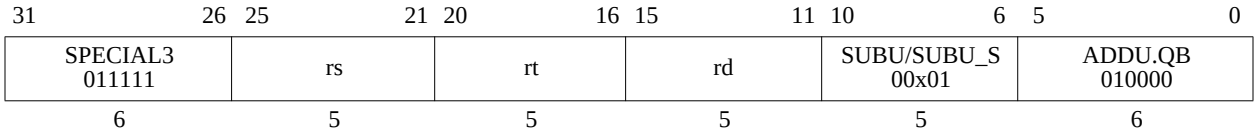
Exceptions:

Reserved Instruction, DSP Disabled

Programming Notes:

To do a two's complement subtract of two words without saturation, use the MIPS32 SUBU instruction.

Unsigned Subtract Vector Quad-Bytes SUBU.QB



Format: `SUBU.QB rd, rs, rt` **MIPSDSP**

`SUBU_S.QB rd, rs, rt` **MIPSDSP**

Purpose:

Unsigned subtract two vectors with byte values with optional saturation.

Description: `rd ← sat8(rs`$_{31:24}$`-rt`$_{31:24}$`) | sat8(rs`$_{23:16}$`-rt`$_{23:16}$`) | sat8(rs`$_{15:8}$`-rt`$_{15:8}$`) | sat8(rs`$_{7:0}$`-rt`$_{7:0}$`)`

Two vector quad-byte values in registers rs and rt are subtracted separately. For the saturate version of the instruction, unsigned saturated arithmetic is performed, where an underflow clamps to the smallest (0x00 or decimal 0) representable value, before writing the destination register rd. This saturation is performed separately for each vector operation.

This instruction, on an overflow of any of the vector operations, writes bit 20 in the ouflag field in the *DSPControl* register.

Restrictions:

No data-dependent exceptions are possible.

The operands must be a value in the specified format. If they are not, the results are **UNPREDICTABLE** and the values of the operand vectors become **UNPREDICTABLE**.

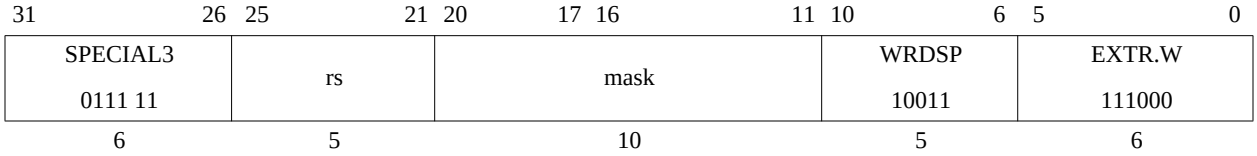
Operation:

```
SUBU.QB
    Subtract(tempw, tempr, temps, tempt)
    Setouflag(tempw, tempr, temps, tempt)
    GPR[rd] ← tempw7..0 || tempr7..0 || temps7..0 || tempt7..0

SUBU_S.QB
    Subtract(tempw, tempr, temps, tempt)
    Setouflag(tempw, tempr, temps, tempt)
    if (tempw8 ≠ tempw7) then
       tempw ← 0x00
    endif
    if (tempr8 ≠ tempr7) then
       tempr ← 0x00
    endif
    if (temps8 ≠ temps7) then
       temps ← 0x00
    endif
    if (tempt8 ≠ tempt7) then
       tempt ← 0x00
    endif
    GPR[rd] ← tempw7..0 || tempr7..0 || temps7..0 || tempt7..0

function Subtract(tempw, tempr, temps, tempt)
    tempw ← (GPR[rs]31 || GPR[rs]31..24) - (GPR[rt]31 || GPR[rt]31..24)
    tempr ← (GPR[rs]23 || GPR[rs]23..16) - (GPR[rt]23 || GPR[rt]23..16)
    temps ← (GPR[rs]15 || GPR[rs]15..8) - (GPR[rt]15 || GPR[rt]15..8)
```

```
    tempt ← (GPR[rs]7 || GPR[rs]7..0) - (GPR[rt]7 || GPR[rt]7..0)
endfunction Subtract

function Setouflag(w, r, s, t)
    if ((w8 ≠ w7) or (r8 ≠ r7) or (s8 ≠ s7) or (t8 ≠ t7)) then
        DSPControlouflag:20 ← 1
    endif
endfunction Setouflag
```

Exceptions:

Reserved Instruction, DSP Disabled



Format: `WRDSP rs, mask` **MIPSDSP**
`WRDSP rs` **Assembly Idiom**

Purpose:

To write selected fields from a GPR to the special purpose *DSPControl* register.

Description: `rd ← select[mask, DSPControl]`

Certain fields from the special register *DSPControl* register selected by the bits in the specified mask field in the instruction are written using corresponding bits from the source GPR *rs*. Each bit in the mask value corresponds to a specific field in the *DSPControl* register. If the bit corresponding to a field is 1, then this implies that the field is to be written. Each field in the *DSPControl* register is written using bits from the same bit positions in the source register. For example, if the *scount* field is to be written, and this field is at bit positions 12..7 in *DSPControl*, then after the execution of this instruction the bits of the *scount* field have been overwritten by bits 12..7 from the source register *rs*. If the mask bit is 0, then after the execution of this instruction, the bits in the *DSPControl* register corresponding to that field remain unmodified. Bits 16..11 of the mask field correspond to fields *EFI*, *ccond*, *ouflag*, *c*, *scount*, and *pos*, respectively. The hardware ignores the upper un-used bits of the mask field.

The one-operand version of the instruction provides a convenient assembly idiom that allows the programmer to write all the allowable fields in the DSPControl register from the source GPR.

Restrictions:

None

Operation:

```
new31..0 ← 0
overwrite31..0 ← 0xFFFFFFFF
if (mask0 = 1) then
    new5..0 ← GPR[rs]pos:5..0
    overwrite5..0 ← 0
endif
if (mask1 = 1) then
    new12..7 ← GPR[rs]scount:12..7
    overwrite12..7 ← 0
endif
if (mask2 = 1) then
    new13 ← GPR[rs]c:13
    overwrite13 ← 0
endif
if (mask3 = 1) then
    new23..16 ← GPR[rs]ouflag:23..16
    overwrite23..16 ← 0
endif
if (mask4 = 1) then
    new27..24 ← GPR[rs]ccond:27..24
    overwrite27..24 ← 0
endif
if (mask5 = 1) then
```

```
        new14 ← GPR[rs]EFI:14
        overwrite14 ← 0
    endif

    DSPControl ← DSPControl and overwrite31..0
    DSPControl ← DSPControl or new31..0
```

Exceptions:

Reserved Instruction, DSP Disabled

Endian-Agnostic Reference to Register Elements

A.1 Using Endian-Agnostic Instruction Names

Certain instructions being proposed in the ASE only operate on a subset of the operands in the register. In most cases, this is simply the left (**L**) or right (**R**) half of the register. Some instructions refer to the left alternating (**LA**) or right alternating (**RA**) elements of the register. But this type of reference does not take the endian-ness of the processor and memory into account. Since the DSP ASE instructions do not take the endian-ness into account and simply use the left or right part of the register, this section describes a method by which users can take advantage of user-defined macros to translate the given instruction to the appropriate one for a given processor endian-ness.

An example is given below that uses actual element numbers in the mnemonics to be endian-agnostic.

In the MIPS32 architecture, the following conventions could be used:

- PH0 refers to halfword element 0 (from a pair in the specified register).
- PH1 refers to halfword element 1.
- QB01 refers to byte elements 0 and 1 (from a quad in the specified register).
- QB23 refers to byte elements 2 and 3.
- QB02 refers to (even) byte elements 0 and 2.
- QB13 refers to (odd) byte elements 1 and 3.

The even and odd subsets are mainly used in storing, computing on, and loading complex numbers that have a real and imaginary part. If the real and imaginary parts of a complex number are stored in consecutive memory locations, then computations that involve only the real or only the imaginary parts must first extract these to a different register. This can most effectively be done using the even and odd formats of the relevant operations.

Note that these mnemonics are translated by the assembler to underlying real instructions that operate on absolute element positions in the register based on the endian-ness of the processor.

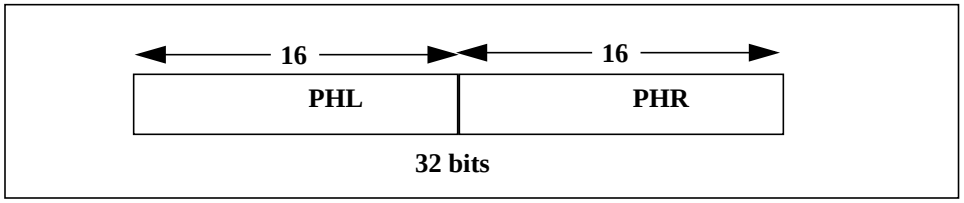
A.2 Mapping Endian-Agnostic Instruction Names to DSP ASE Instructions

To illustrate this process, we will use one instruction as an example. This can be repeated for all the relevant instructions in the ASE.

The **MULEQ_S** instruction multiplies fractional data operands to expanded full-size results in a destination register with optional saturation. Since the result occupies twice the width of the input operands, only half the operands from the source registers are operated on at a time. So the complete instruction mnemonic would be given as **MULEQ_S.W.PH0 rd, rs, rt** where the second part (after the first dot) indicates the size of the result, and the third part (after the second dot) indicates the element of the source register being used, which in this example is the 0^{th} element. The real instructions that the hardware implements are **MULEQ_S.W.PHL** and **MULEQ_S.W.PHR** which operate on the left halfword element and the right halfword element respectively, of the given source registers, as shown in Figure A-1. The user can

map the user instruction (with **.PH0**) to the **MULEQ_S.W.PHL** real instruction if the processor is big-endian or to the real instruction **MULEQ_S.W.PHR** if the processor is little-endian.

Figure A-1 The Endian-Independent PHL and PHR Elements in a GPR for the MIPS32® Architecture



Then **MULEQ_S.W.PH1 rd, rs, rt** instruction can be mapped to **MULEQ_S.W.PHR** if the processor is big-endian (see Figure A-2), and to **MULEQ_S.W.PHL** real instruction if the processor is little-endian (see Figure A-3).

Figure A-2 The Big-Endian PH0 and PH1 Elements in a GPR for the MIPS32® Architecture

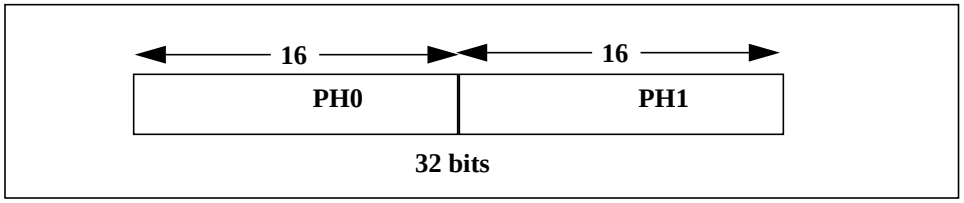
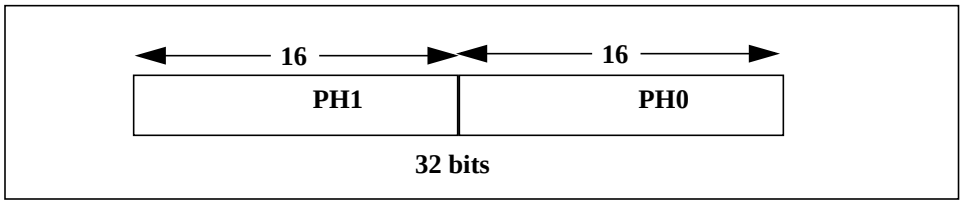


Figure A-3 The Little-Endian PH0 and PH1 Elements in a GPR for the MIPS32® Architecture



To specify the even and odd type operations, a user instruction (to use odd elements) such as **PRECEQ_S.PH.QB02** (which precision expands the values) would be mapped to **PRECEQ_S.PH.QBLA** or **PRECEQ_S.PH.QBRA** depending on whether the endian-ness of the processor was big or little, respectively. (**LA** stands for left-alternating and **RA** for right-alternating).

Figure A-4 The Endian-Independent QBL and QBR Elements in a GPR for the MIPS32® Architecture

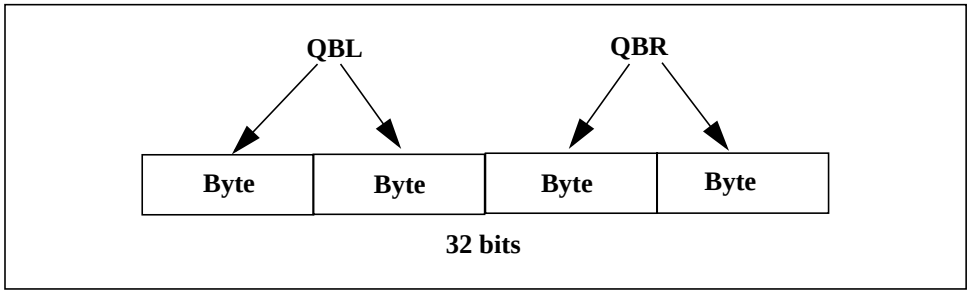
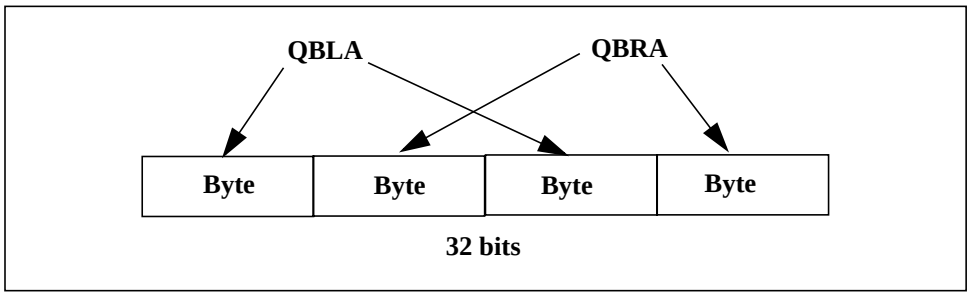


Figure A-5 The Endian-Independent QBLA and QBRA Elements in a GPR for the MIPS32® Architecture



Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself.

Version	Date	Comments
1.00	6 July, 2005	Initial revision