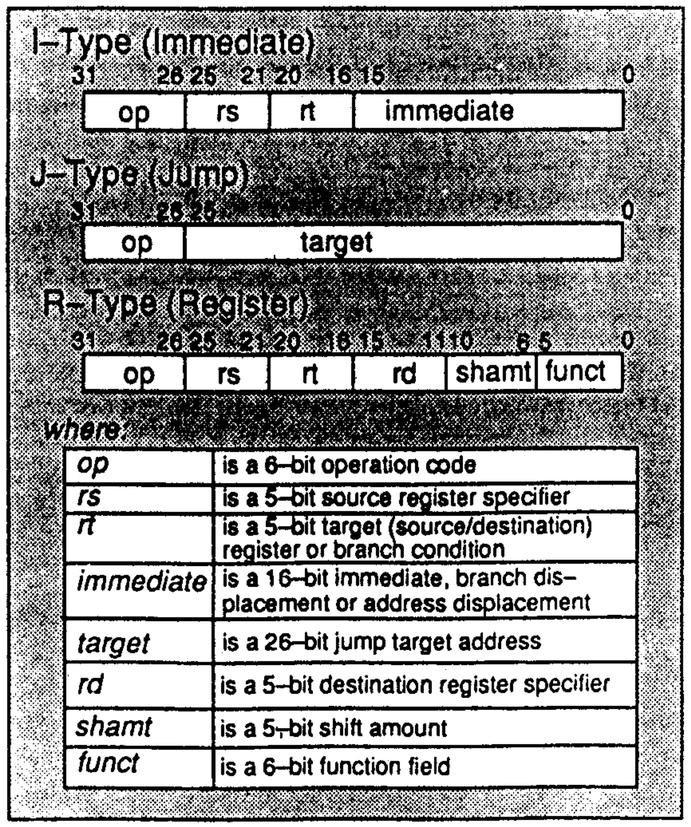
- **Coprocessor** instructions perform operations in the coprocessors. Coprocessor loads and stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats (see the FPU instructions). Coprocessor zero (CP0) instructions manipulate the memory management and exception handling facilities of the processor.

- **Special** instructions perform a variety of tasks, including movement of data between special and general registers, trap, and breakpoint. They are always R-type.

Instruction Formats

Every CPU instruction consists of a single word (32 bits) aligned on a word boundary and the major instruction formats are shown in Figure A–1.



where:

op	is a 6–bit operation code
rs	is a 5–bit source register specifier
rt	is a 5–bit target (source/destination) register or branch condition
immediate	is a 16–bit immediate, branch displacement or address displacement
target	is a 26–bit jump target address
rd	is a 5–bit destination register specifier
shamt	is a 5–bit shift amount
funct	is a 6–bit function field

Figure A–1. CPU Instruction Formats

Instruction Notation Conventions

In this appendix, all variable subfields in an instruction format (such as *rs*, *rt*, *immediate*, etc.) are shown in lower-case names.

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

Figures with the actual bit encoding for all the mnemonics are located at the end of this Appendix, and the bit encoding also accompanies each instruction.

In the instruction descriptions that follow, the *Operation* section describes the operation performed by each instruction using a high-level language notation. Special symbols used in the notation are described in Table A–1.

Table A–1. CPU Instruction Operation Notations

Symbol	Meaning
←	Assignment
||	Bit string concatenation
x^y	Replication of bit value *x* into a *y*-bit string. Note that *x* is always a single-bit value.
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation is always used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+	Two's complement or floating point addition
−	Two's complement or floating point subtraction
*	Two's complement or floating point multiplication
div	Two's complement integer division
mod	Two's complement modulo
/	Floating point division
<	Two's complement less than comparison
and	Bitwise logic AND
or	Bitwise logic OR
xor	Bitwise logic XOR
nor	Bitwise logic NOR
GPR[*x*]	General Register *x*. The content of GPR[0] is always zero. Attempts to alter the content of GPR[0] have no effect.
CPR[*z*,*x*]	Coprocessor unit *z*, general register *x*
CCR[*z*,*x*]	Coprocessor unit *z*, control register *x*
COC[*z*]	Coprocessor unit *z* condition signal.
BigEndianMem	Big-endian mode as configured at reset (0 –> Little, 1 –> Big). Specifies the endianess of the memory interface (see LoadMemory and StoreMemory), and the endianess of Kernel and Supervisor mode execution.
ReverseEndian	Signal to reverse the endianess of load and store instructions. This feature is available in User mode only, and is effected by setting the *RE* bit of the *Status* register. Thus ReverseEndian may be computed as (SR_{25} and User mode). R3000A, R4000 and R6000 only.
BigEndianCPU	The endianess for load and store instructions (0 –> Little, 1 –>Big). In User mode, this endianess may be reversed by setting SR_{25}. Thus BigEndianCPU may be computed as BigEndianMem XOR ReverseEndian.
LL bit	Bit of state to specify synchronization instructions. Set by LL, cleared by RFE, ERET and Invalidate, and read by SC. (R4000/R6000 only)
T+*i*:	Indicates the time steps between operations. Each of the statements within a time step are defined to be executed in sequential order (as modified by conditional and loop constructs). Operations which are marked T+*i*: are executed at instruction cycle *i* relative to the start of execution of the instruction. Thus, an instruction which starts at time *j* executes operations marked T+*i*: at time *i* + *j*. The interpretation of the order of execution between two instructions of two operations which execute at the same time should be pessimistic; the order is not defined.

Instruction Notation Examples

The following examples illustrate the application of some of the instruction notation conventions:

Example #1:

GPR[rt] ← immediate || 0^{16}

Sixteen zero bits are concatenated with an immediate value (typically 16 bits), and the 32–bit string (with the lower 16 bits set to zero) is assigned to General Purpose Register *rt*.

Example #2:

$(immediate_{16})^{16}$ || $immediate_{15..0}$

Bit 15 (the sign bit) of an immediate value is extended for 16 bit positions, and the result is concatenated with bits 15 through 0 of the immediate value to form a 32–bit sign extended value.

Load and Store Instructions

In the R2000/R3000 implementation of the ISA, all loads are implemented with a delay of one instruction. That is, the instruction immediately following a load cannot use the contents of the register which will be loaded with the data being fetched from storage. An exception is the target register for the Load Word Left and Load Word Right instructions, which may be specified as the same register used as the destination of a load instruction that immediately precedes it.

In the R4000/R6000 implementation, the instruction immediately following a load may use the contents of the register loaded. In such cases, the hardware interlocks, requiring additional real cycles, so scheduling load delay slots is still desirable — although not required for functional code.

Two special instructions are provided in the R4000/R6000 implementation of the ISA, Load Linked and Store Conditional. These instructions are used in carefully coded sequences to provide one of several synchronization primitives, including test-and-set, bit-level locks, semaphores, and sequencers/event counts.

In the load/store operation descriptions, the functions listed in Table A-2 are used to summarize the handling of virtual addresses and physical memory.

Table A–2. Load/Store Common Functions

Function	Meaning
AddressTranslation	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the page containing the virtual address is not present in the TLB.
LoadMemory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The low order two bits of the address and the access type field indicates which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.
StoreMemory	Uses the cache, write buffer and main memory to store the word or part of word specified as data in the word containing the specified physical address. The low order two bits of the address and the access type field indicates which of each of the four bytes within the data word should be stored.

The access type field indicates the size of the data item to be loaded or stored as shown in Table A–3. Regardless of access type or byte-numbering order (endianness), the address specifies the byte which has the smallest byte address of the bytes in the addressed field. For a Big-endian machine, this is the leftmost byte and contains the sign for a 2's-complement number; for a Little-endian machine, this is the rightmost byte and contains the lowest precision byte.

Table A–3. Access Type Specifications for Loads/Stores

Access type Mnemonic	Value	Meaning
DOUBLEWORD	7	doubleword (64 bits)
SEPTIBYTE	6	seven bytes (56 bits)
SEXTIBYTE	5	six bytes (48 bits)
QUINTIBYTE	4	five bytes (40 bits)
WORD	3	word (32 bits)
TRIPLEBYTE	2	triple-byte (24 bits)
HALFWORD	1	halfword (16 bits)
BYTE	0	byte (8 bits)

The bytes within the addressed doubleword which are used can be determined directly from the access type and the three low order bits of the address, as shown in Chapter 3.

Jump and Branch Instructions

All jump and branch instructions are implemented with a delay of exactly one instruction. That is, the instruction immediately following a jump or branch (i.e., occupying the delay slot) is always executed while the target instruction is being fetched from storage. It is not valid for a delay slot to be occupied itself by a jump or branch instruction; however, this error is not detected, and the results of such an operation are undefined.

If an exception or interrupt prevents the completion of a legal instruction during a delay slot, the hardware sets the *EPC* register to point at the jump or branch instruction which precedes it. When the code is restarted, both the jump or branch instructions and the instruction in the delay slot are reexecuted.

Because jump and branch instructions may be restarted after exceptions or interrupts, they must be restartable. Therefore, when a jump or branch instruction stores a return link value, register 31 (the register in which the link is stored) may not be used as a source register.

Since instructions must be word-aligned, a Jump Register or Jump and Link Register instruction must use a register whose two low order bits are zero. If these low order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

Coprocessor Instructions

The MIPS architecture provides four coprocessor units, or classes. Coprocessors are alternate execution units, which have separate register files from the CPU. R-Series coprocessors have 2 register spaces, each with 32 32-bit registers. The first space, *coprocessor general* registers, may be directly loaded from memory and stored into memory, and their contents may be transferred between the coprocessor and processor. The second, *coprocessor control* registers, may only have their contents transferred directly between the coprocessor and processor. Coprocessor instructions may alter registers in either space.

Normally, by convention, *Coprocessor Control Register 0* is interpreted as a *Coprocessor Implementation And Revision* register. However, the system control coprocessor (CP0) uses *Coprocessor General Register 15* for the processor/coprocessor revision register. The register's low order byte (bits 7..0) is interpreted as a coprocessor unit revision number. The second byte (bits 15..8) is interpreted as a coprocessor unit implementation descriptor. The revision number is a value of the form *y*. *x* where *y* is a major revision number in bits 7..4 and *x* is a minor revision number in bits 3..0.

The contents of the high order halfword of the register are not defined (currently read as 0 and should be 0 when written).

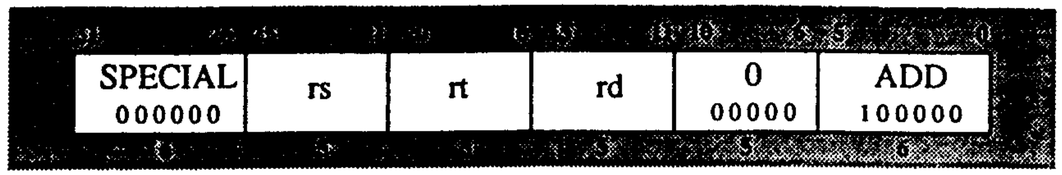
System Control Coprocessor (CP0) Instructions

There are some special limitations imposed on operations involving CP0 that is incorporated within the CPU. Although load and store instructions to transfer data to and from coprocessors and move control to/from coprocessor instructions are generally permitted by the MIPS architecture, CP0 is given a somewhat protected status since it has responsibility for exception handling and memory management. Therefore, the move to/from coprocessor instructions are the only valid mechanism for reading from and writing to the CP0 registers.

Several coprocessor operation instructions are defined for CP0 to directly read, write, and probe TLB entries and to modify the operating modes in preparation for returning to User mode or interrupt-enabled states.

ADD

ADD



Format:

ADD rd,rs,rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form a 32-bit result. The result is placed into general register *rd*.

An overflow exception occurs if the two highest order carry-out bits differ (2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

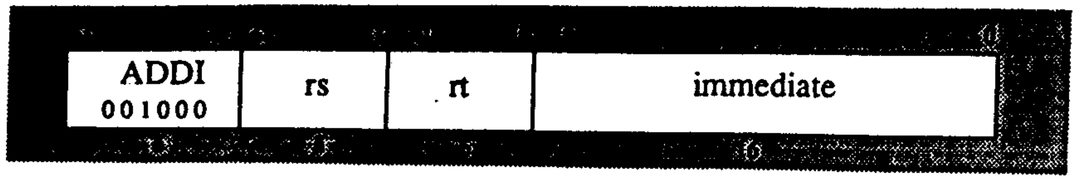
```
T:    GPR[rd] ←GPR[rs] + GPR[rt]
```

Exceptions:

Overflow exception

ADDI

Add Immediate



Format:

ADDI rt,rs,immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form a 32-bit result. The result is placed into general register *rt*.

An overflow exception occurs if the two highest order carry-out bits differ (2's-complement overflow). The destination register *rt* is not modified when an integer overflow exception occurs.

Operation:

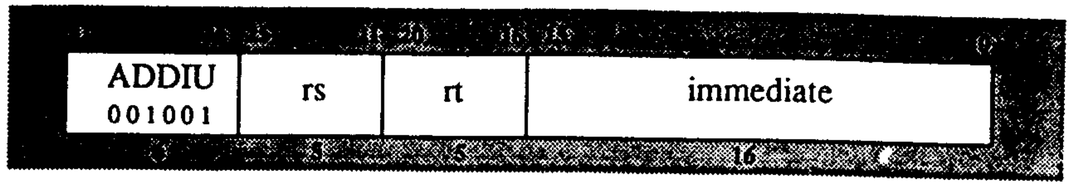
```
T:    GPR [rt] ← GPR[rs] +(immediate15)16 || immediate15..0
```

Exceptions:

Overflow exception

ADDIU

Add Immediate Unsigned



Format:

ADDIU rt,rs,immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form a 32-bit result. The result is placed into general register *rt*. No integer overflow exception occurs under any circumstances.

The only difference between this instruction and the ADDI instruction is that ADDIU never causes an overflow exception.

Operation:

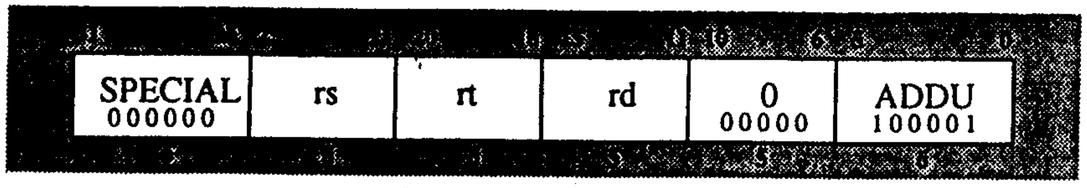
```
T:    GPR [rt] ← GPR[rs] + (immediate15)16 || immediate15..0
```

Exceptions:

None.

ADDU

ADD Unsigned



Format:

ADDU rd,rs,rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form a 32-bit result. The result is placed into general register *rd*.

No overflow exception occurs under any circumstances.

The only difference between this instruction and the ADD instruction is that ADDU never causes an overflow exception.

Operation:

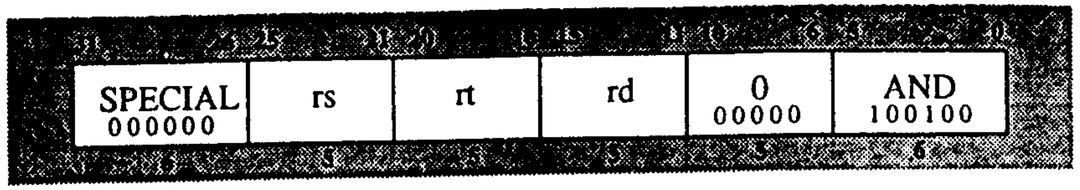
```
T:    GPR[rd] ← GPR[rs] + GPR[rt]
```

Exceptions:

None.

AND

And



Format:

AND rd,rs,rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical AND operation. The result is placed into general register *rd*.

Operation:

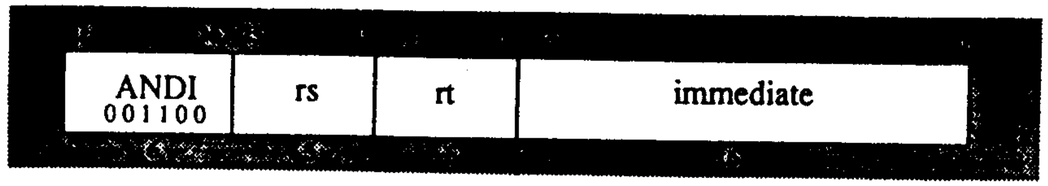
```
T:    GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None.

ANDI

And Immediate



Format:

ANDI rt,rs,immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical AND operation. The result is placed into general register *rt*.

Operation:

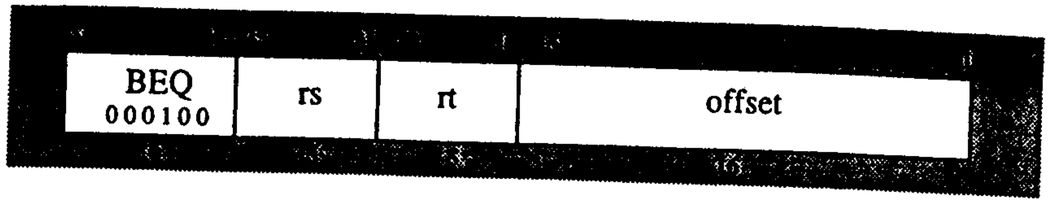
```
T:    GPR[rt] ← 0^16 || (immediate and GPR[rs]15..0)
```

Exceptions:

None.

Branch On Equal

BEQ



Format:

BEQ rs,rt,offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended to 32 bits. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are equal, then the program branches to the target address, with a delay of one instruction.

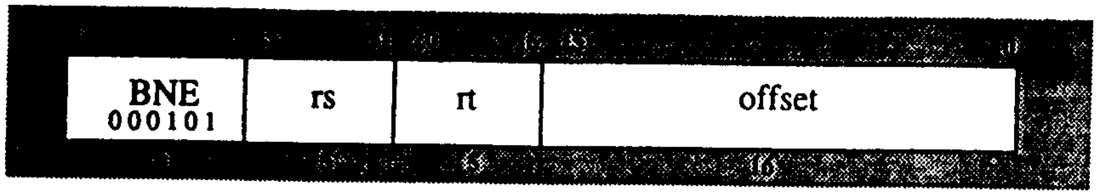
Operation:

```
T:    target ← (offset15)^14 || offset || 0^2
      condition ← (GPR[rs] = GPR[rt])
T+1:  if condition then
            PC ← PC + target
      endif
```

Exceptions:

None.

Branch On Not Equal — BNE



Format:

BNE rs,rt,offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended to 32 bits. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are not equal, then the program branches to the target address, with a delay of one instruction.

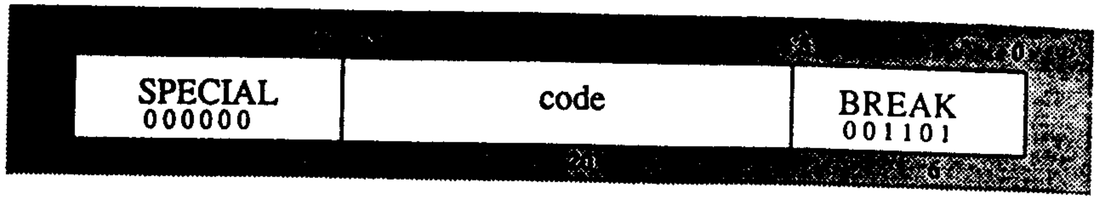
Operation:

```
T:    target ← (offset15)^14 || offset || 0^2
      condition ← (GPR[rs] ≠ GPR[rt])
T+1:  if condition then
        PC ← PC + target
      endif
```

Exceptions:

None.

Breakpoint — BREAK



Format:

BREAK

Description:

A breakpoint trap occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

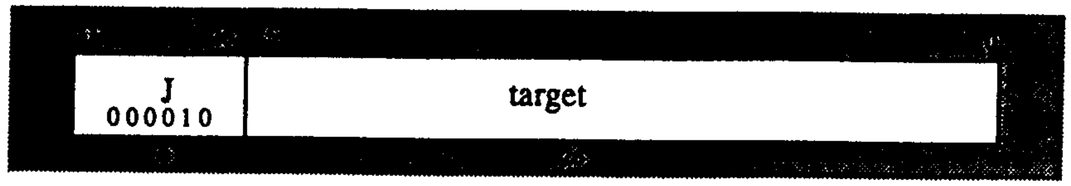
```
T:    BreakpointException
```

Exceptions:

Breakpoint exception

J

Jump



Format:

J target

Description:

The 26-bit target address is shifted left two bits and combined with the high order four bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction.

Operation:

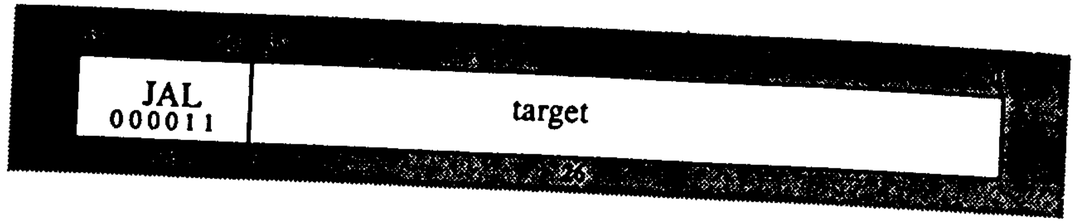
```
T:    temp ← target
T+1:  PC ← PC31..28 || temp || 0²
```

Exceptions:

None.

Jump And Link

JAL



Format:

JAL target

Description:

The 26-bit target address is shifted left two bits and combined with the high order four bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register, *r31*.

Operation:

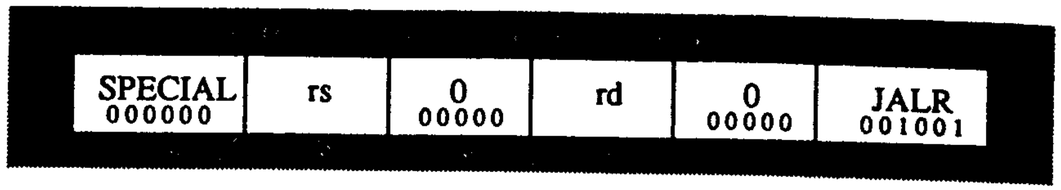
```
T:    temp ← target
      GPR[31] ← PC + 8
T+1:  PC ← PC 31..28 || temp || 0²
```

Exceptions:

None.

JALR

Jump And Link Register



Format:

JALR rs
JALR rd, rs

Description:

The program unconditionally jumps to the address contained in general register *rs*, with a delay of one instruction. The address of the instruction after the delay slot is placed in general register *rd*. The default value of *rd*, if omitted in the assembly language instruction, is 31.

Register specifiers *rs* and *rd* may not be equal, because such an instruction does not have the same effect when reexecuted. However, an attempt to execute this instruction is *not* trapped, and the result of executing such an instruction is undefined.

Since instructions must be word-aligned, a *Jump and Link Register* instruction must specify a target register (rs) whose two low order bits are zero. If these low order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

Operation:

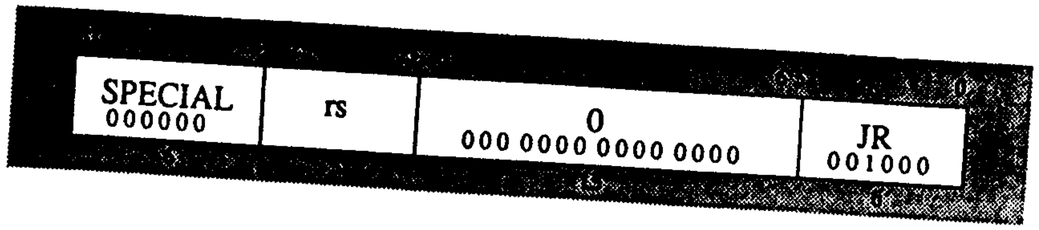
```
T:    temp ← GPR [rs]
      GPR[rd] ← PC + 8
T+1:  PC ← temp
```

Exceptions:

None.

Jump Register

JR



Format:

JR rs

Description:

The program unconditionally jumps to the address contained in general register *rs*, with a delay of one instruction. This instruction is only valid when *rd* = 0.

Since instructions must be word-aligned, a *Jump Register* instruction must specify a target register (*rs*) whose two low order bits are zero. If these low order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

Operation:

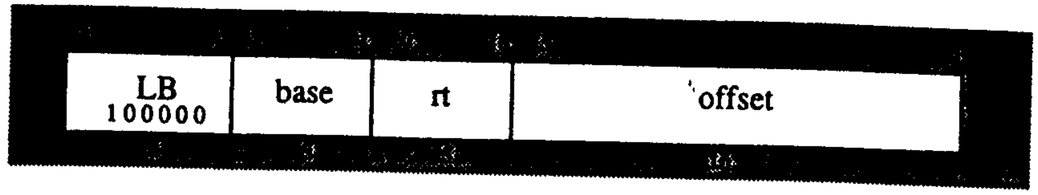
```
T:    temp ← GPR[rs]
T+1:  PC ← temp
```

Exceptions:

None.

LB

Load Byte



Format:

LB rt,offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the byte at the memory location specified by the effective address are sign-extended and loaded into general register *rt*.

In R2000/R3000 implementations, the contents of general register *rt* are undefined for time *T* of the instruction immediately following this load instruction.

R2000/R3000 Operation:

```
T:    vAddr ← ((offset15)16 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
      byte ← vAddr1..0 xor BigEndianCPU2
      GPR[rt] ← undefined
T+1:  GPR[rt] ← (mem7+8*byte)24 || mem7+8*byte..8*byte
```

Load Byte (continued) — LB

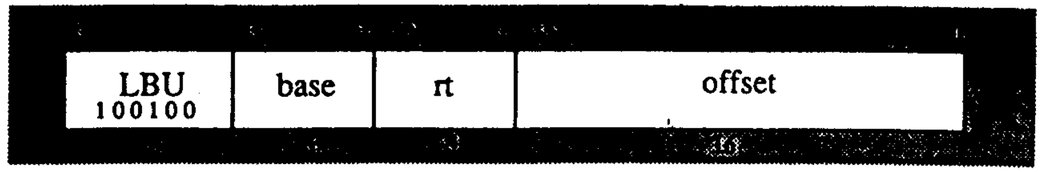
R4000/R6000 Operation:

```
T:   vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     pAddr ← pAddrPSIZE-1..2 || ( pAddr1..0 xor ReverseEndian^2 )
     mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
     byte ← vAddr1..0 xor BigEndianCPU^2
     GPR[rt] ← (mem7+8*byte)^24 || mem7+8*byte..8*byte
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LBU — Load Byte Unsigned



Format:

LBU rt,offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the byte at the memory location specified by the effective address are zero-extended and loaded into general register *rt*. In R2000/R3000 implementations, the contents of general register *rt* are undefined for time *T* of the instruction immediately following this load instruction.

R2000/R3000 Operation:

```
T:   vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
     byte ← vAddr1..0 xor BigEndianCPU^2
     GPR[rt] ← undefined
T+1: GPR[rt] ← 0^24 || mem7+8*byte..8*byte
```

R4000/R6000 Operation:

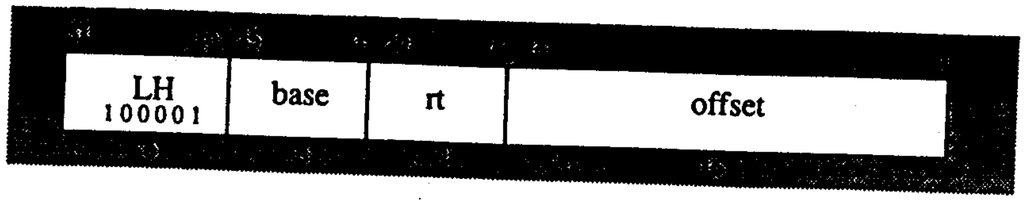
```
T:   vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian^2)
     mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
     byte ← vAddr1..0 xor BigEndianCPU^2
     GPR[rt] ← 0^24 || mem7+8*byte..8*byte
```

Exceptions:

TLB refill exception TLB invalid exception
Bus error exception Address error exception

Load Halfword

LH



Format:

LH rt,offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the halfword at the memory location specified by the effective address are sign-extended and loaded into general register *rt*.

If the least significant bit of the effective address is non-zero, an address error exception occurs.

In R2000/R3000 implementations, the contents of general register *rt* are undefined for time *T* of the instruction immediately following this load instruction.

R2000/R3000 Operation:

```
T:    vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
      byte ← vAddr1..0  xor (BigEndianCPU || 0)
      GPR[rt] ← undefined
T+1:  GPR[rt] ← (mem15+8*byte)^16 || mem15+8*byte..8*byte
```


LH

Load Halfword (continued)

R4000/R6000 Operation:

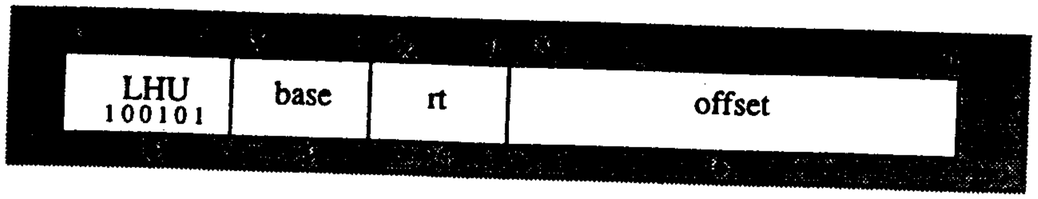
```
T:    vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      pAddr ← pAddrPSIZE-1..2 || ( pAddr1..0  xor (ReverseEndian || 0))
      mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
      byte ← vAddr1..0  xor (BigEndianCPU || 0)
      GPR[rt] ← (mem15+8*byte)^16 || mem15+8*byte..8*byte
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

Load Halfword Unsigned

LHU



Format:

LHU rt,offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the halfword at the memory location specified by the effective address are zero-extended and loaded into general register *rt*.

If the least significant bit of the effective address is non-zero, an address error exception occurs.

In R2000/R3000 implementations, the contents of general register *rt* are undefined for time *T* of the instruction immediately following this load instruction.

R2000/R3000 Operation:

```
T:   vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
     byte ← vAddr1..0  xor (BigEndianCPU || 0)
     GPR[rt] ← undefined
T+1: GPR[rt] ← 0^16 || mem15+8*byte..8*byte
```


R4000/R6000 Operation:

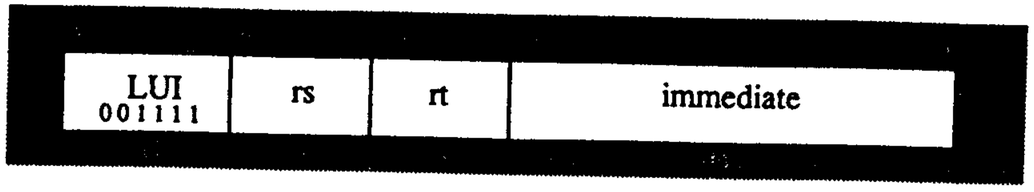
```
T:   vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     pAddr ← pAddrPSIZE-1..2 || ( pAddr1..0  xor (ReverseEndian || 0))
     mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
     byte ← vAddr1..0  xor (BigEndianCPU || 0)
     GPR[rt] ← 0^16 || mem15+8*byte..8*byte
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

Load Upper Immediate

LUI



Format:

LUI rt,immediate

Description:

The 16-bit *immediate* is shifted left 16 bits and concatenated to 16 bits of zeros. The result is placed into general register *rt*.

Operation:

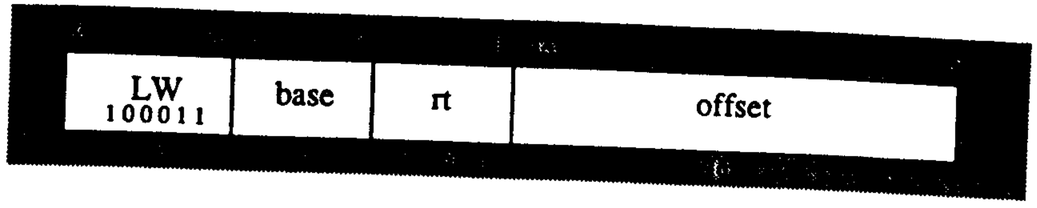
```
T:    GPR[rt] ← immediate || 0^16
```

Exceptions:

None.

LW

Load Word



Format:

LW rt,offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*.

If either of the two least significant bits of the effective address is non-zero, an address error exception occurs.

In R2000/R3000 implementations, the contents of general register *rt* are undefined for time *T* of the instruction immediately following this load instruction.

R2000/R3000 Operation:

```
T:    vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
      GPR[rt] ← undefined
T+1:  GPR[rt] ← mem
```

Load Word (continued) — LW

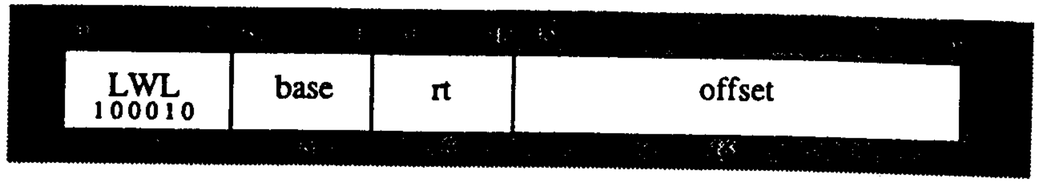
R4000/R6000 Operation:

```
T:   vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
     GPR[rt] ← mem
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LWL — Load Word Left



Format:

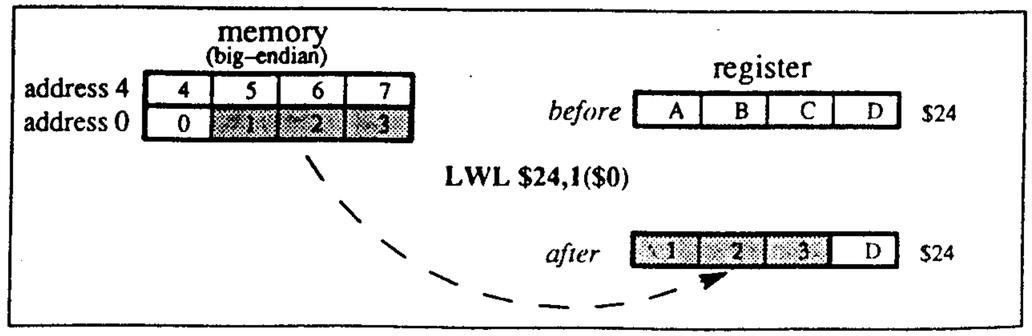
LWL rt,offset(base)

Description:

This instruction can be used in combination with the LWR instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words. LWL loads the left portion of the register from the appropriate part of the high-order word; LWR loads the right portion of the register from the appropriate part of the low order word.

The LWL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified.

Conceptually, it starts at the specified byte in memory and loads that byte into the high order (left-most) byte of the register; then it proceeds toward the low order byte of the word in memory and the low order byte of the register, loading bytes from memory into the register until it reaches the low order byte of the word in memory. The least significant (right-most) byte(s) of the register will not be changed.



Load Word Left (continued) — LWL

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWL (or LWR) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

In R2000/R3000 implementations, the contents of general register *rt* are undefined for time *T* of the instruction immediately following this load instruction.

R2000/R3000 Operation:

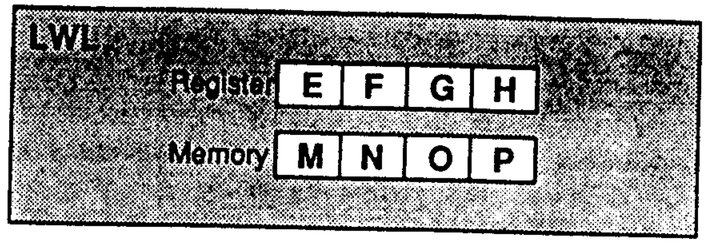
```
T:   vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     byte ← vAddr1..0 xor BigEndianCPU^2
     if BigEndianMem = 0 then
          pAddr ← pAddr31..2 || 0^2
     endif
     mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
T+1: GPR[rt] ← mem7+8*byte..0 || GPR[rt]23-8*byte..0
```

R4000/R6000 Operation:

```
T:   vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian^2)
     byte ← vAddr1..0 xor BigEndianCPU^2
     if BigEndianMem = 0 then
          pAddr ← pAddrPSIZE-1..2 || 0^2
     endif
     mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
     GPR[rt] ← mem7+8*byte..0 || GPR[rt]23-8*byte..0
```


LWL — Load Word Left (continued)

Given a word in a register and a word in memory, the operation of LWL is as follows:



vAddr2..0	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset LEM	Offset BEM	Destination	Type	Offset LEM	Offset BEM
0	P F G H	0	0	3	M N O P	3	0	0
1	O P G H	1	0	2	N O P H	2	0	1
2	N O P H	2	0	1	O P G H	1	0	2
3	M N O P	3	0	0	P F G H	0	0	3

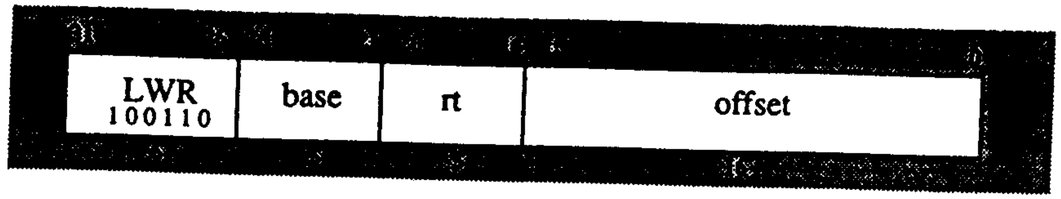
LEM BigEndianMem = 0
BEM BigEndianMem = 1
Type AccessType sent to memory
Offset pAddr2..0 sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LWR

Load Word Right



Format:

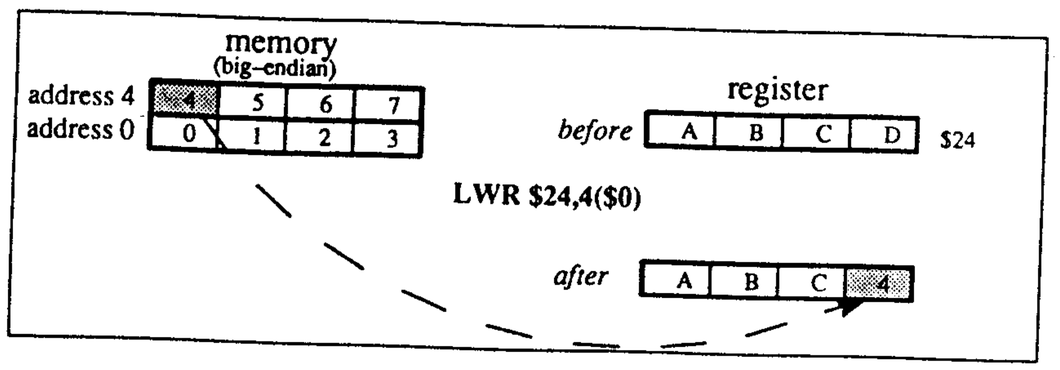
LWR rt,offset(base)

Description:

This instruction can be used in combination with the LWL instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words. LWR loads the right portion of the register from the appropriate part of the low order word; LWL loads the left portion of the register from the appropriate part of the high order word.

The LWR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified.

Conceptually, it starts at the specified byte in memory and loads that byte into the low order (right-most) byte of the register; then it proceeds toward the high order byte of the word in memory and the high order byte of the register, loading bytes from memory into the register until it reaches the high order byte of the word in memory. The most significant (left-most) byte(s) of the register will not be changed.



Load Word Right (continued)

LWR

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWR (or LWL) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

In R2000/R3000 implementations, the contents of general register *rt* are undefined for time T of the instruction immediately following this load instruction.

R2000/R3000 Operation:

```
T:   vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     byte ← vAddr1..0  xor  BigEndianCPU^2
     if BigEndianMem = 1 then
          pAddr ← pAddr31..2 || 0^2
     endif
     mem ← LoadMemory (uncached, WORD–byte, pAddr, vAddr, DATA)
T+1: GPR[rt] ← GPR[rt]31..32–8*byte || mem31..8*byte
```

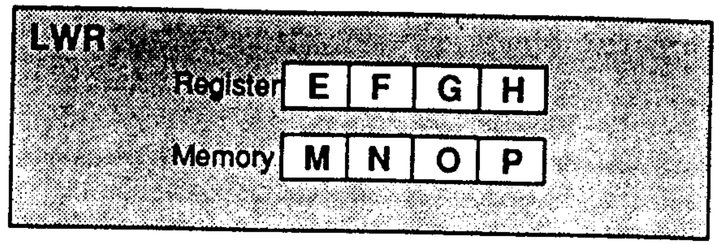
R4000/R6000 Operation:

```
T:   vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     pAddr ← pAddrPSIZE–1..2 || (pAddr1..0  xor  ReverseEndian^2)
     byte ← vAddr1..0  xor  BigEndianCPU^2
     if BigEndianMem = 1 then
     pAddr ← pAddrPSIZE–1..2 || 0^2
     endif
     mem ← LoadMemory (uncached, WORD–byte, pAddr, vAddr, DATA)
     GPR[rt] ← GPR[rt]31..32–8*byte || mem31..8*byte
```

LWR

Load Word Right (continued)

Given a word in a register and a word in memory, the operation of LWR is as follows:



vAddr2..0	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset LEM	Offset BEM	Destination	Type	Offset LEM	Offset BEM
0	M N O P	3	0	0	E F G M	0	3	0
1	E M N O	2	1	0	E F M N	1	2	0
2	E F M N	1	2	0	E M N O	2	1	0
3	E F G M	0	3	0	M N O P	3	0	0

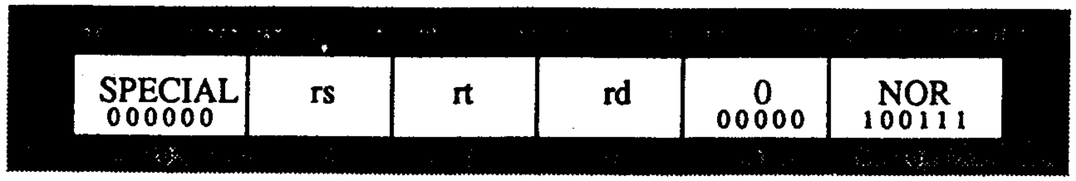
LEM BigEndianMem = 0
BEM BigEndianMem = 1
Type AccessType sent to memory
Offset $pAddr_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

NOR

Nor



Format:

NOR rd,rs,rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical NOR operation. The result is placed into general register *rd*.

Operation:

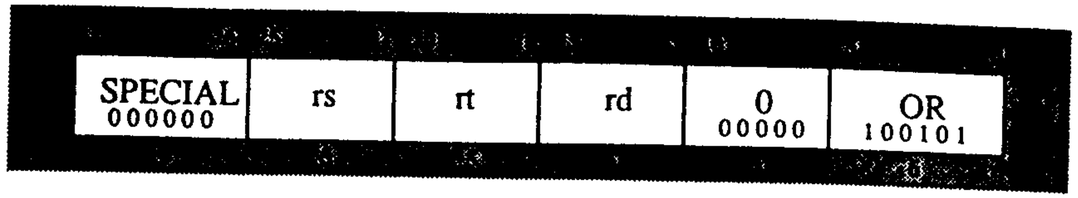
```
T:    GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None.

Or

OR



Format:

OR rd,rs,rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical OR operation. The result is placed into general register *rd*.

Operation:

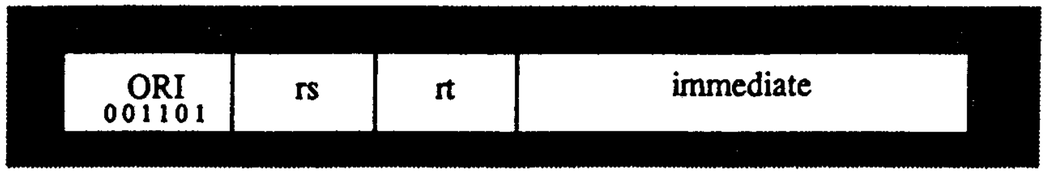
```
T:    GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None.

ORI

Or Immediate



Format:

ORI rt,rs,immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical OR operation. The result is placed into general register *rt*.

Operation:

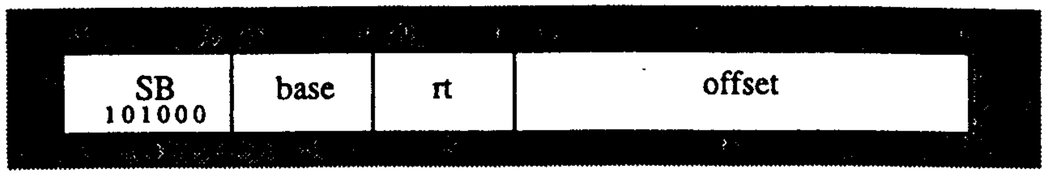
```
T:    GPR[rt] ← GPR[rs]31..16 || (immediate or GPR[rs]15..0)
```

Exceptions:

None.

SB

Store Byte



Format:

SB rt,offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The least significant byte of register *rt* is stored at the effective address.

Operation:

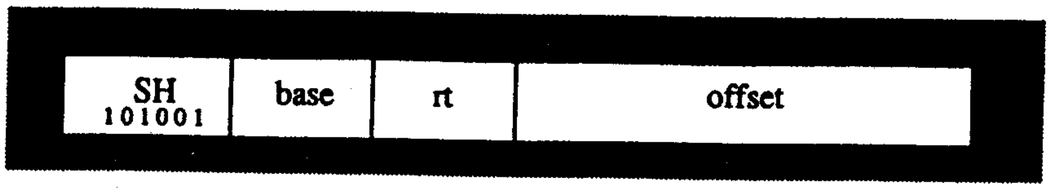
```
T:    vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian^2)
      byte ← vAddr1..0 xor BigEndianCPU^2
      data ← GPR[rt]31-8*byte..0 || 0^8*byte
      StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

Store Halfword

SH



Format:

SH rt,offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a 32-bit unsigned effective address. The least significant halfword of register *rt* is stored at the effective address. If the least significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

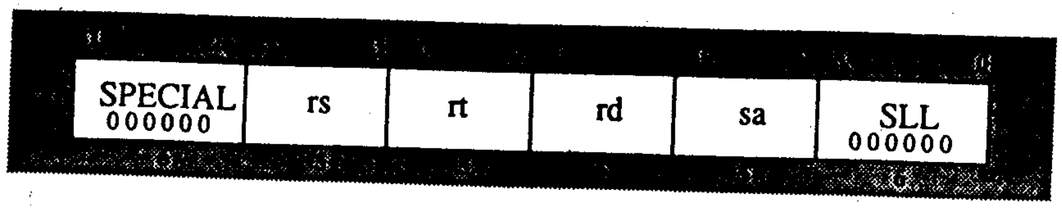
```
T:    vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian^2 || 0))
      byte ← vAddr1..0 xor (BigEndianCPU || 0)
      data ← GPR[rt]31-8*byte..0 || 0^8*byte
      StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SLL

Shift Left Logical



Format:

SLL rd,rt,sa

Description:

The contents of general register *rt* are shifted left by *sa* bits, inserting zeros into the low order bits. The 32-bit result is placed in register *rd*.

Operation:

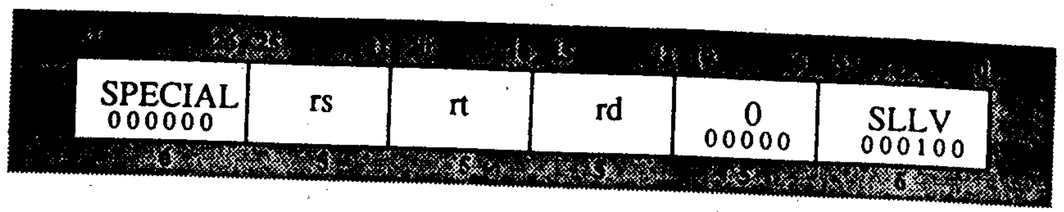
```
T:    GPR[rd] ← GPR[rt]31-sa..0 || 0sa
```

Exceptions:

None.

Shift Left Logical Variable

SLLV



Format:

SLLV rd,rt,rs

Description:

The contents of general register *rt* are shifted left by the number of bits specified by the-low order five bits contained as contents of general register *rs*, inserting zeros into the low order bits. The result is placed in register *rd*.

Operation:

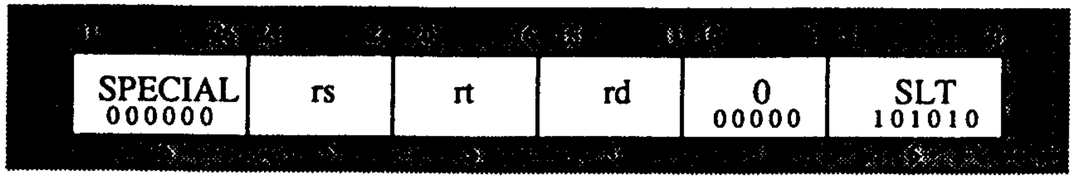
```
T:    s ← GP[rs]4..0
      GPR[rd]← GPR[rt](31-s)..0 || 0s
```

Exceptions:

None.

SLT

Set On Less Than



Format:

SLT rd,rs,rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs*. Considering both quantities as signed 32-bit integers, if the contents of general register *rs* are less than the contents of general register *rt*, the result is set to one, otherwise the result is set to zero. The result is placed into general register *rd*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

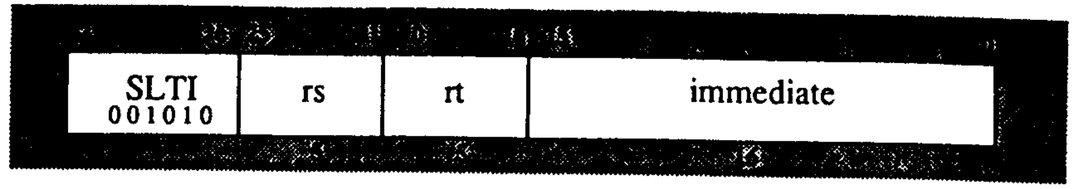
```
T:    if GPR[rs] < GPR[rt] then
          GPR[rd] ← 0^31 || 1
      else
          GPR[rd] ← 0^32
      endif
```

Exceptions:

None.

Set On Less Than Immediate

SLTI



Format:

SLTI rt,rs,immediate

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general register *rs*. Considering both quantities as signed integers, if *rs* is less than the sign-extended immediate, the result is set to one, otherwise the result is set to zero. The result is placed into general register *rt*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

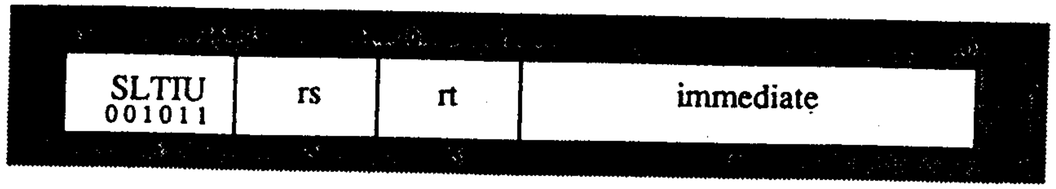
```
T:    if GPR[rs] < (immediate15)^16 || immediate15..0 then
          GPR[rt] ← 0^31 || 1
      else
          GPR[rt] ← 0^32
      endif
```

Exceptions:

None.

SLTIU

Set On Less Than Immediate Unsigned



Format:

SLTIU rt,rs,immediate

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general register *rs*. Considering both quantities as unsigned integers, if *rs* is less than the sign-extended immediate, the result is set to one, otherwise the result is set to zero. The result is placed into general register *rt*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

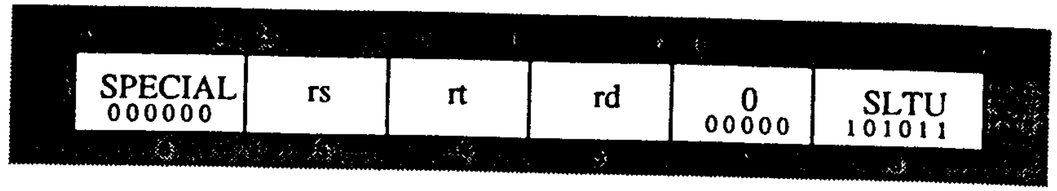
```
T:    if (0 || GPR[rs]) < 0 || ((immediate15)16 || immediate15..0) then
          GPR[rt] ← 031 || 1
      else
          GPR[rt] ← 032
      endif
```

Exceptions:

None.

Set On Less Than Unsigned

SLTU



Format:

SLTU rd,rs,rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are less than the contents of general register *rt*, the result is set to one, otherwise the result is set to zero. The result is placed into general register *rd*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

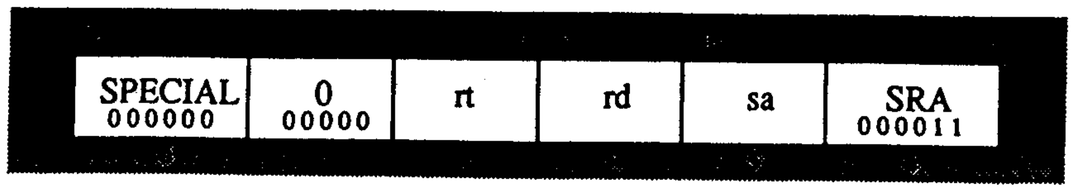
```
T:    if (0 || GPR[rs]) < (0 || GPR[rt]) then
        GPR[rd] ← 031 || 1
      else
        GPR[rd] ← 032
      endif
```

Exceptions:

None.

SRA

Shift Right Arithmetic



Format:

SRA rd,rt,sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, sign-extending the high order bits. The 32-bit result is placed in register *rd*.

Operation:

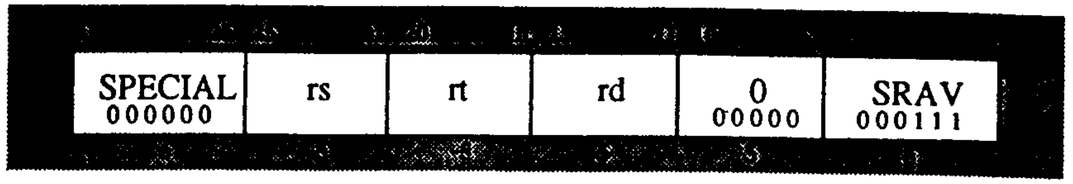
```
T:    GPR[rd] ← (GPR[rt]31)^64 || GPR[rt] 31..64
```

Exceptions:

None.

Shift Right Arithmetic Variable

SRAV



Format:

SRAV rd,rt,rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low order five bits of general register *rs*, sign-extending the high order bits. The result is placed in register *rd*.

Operation:

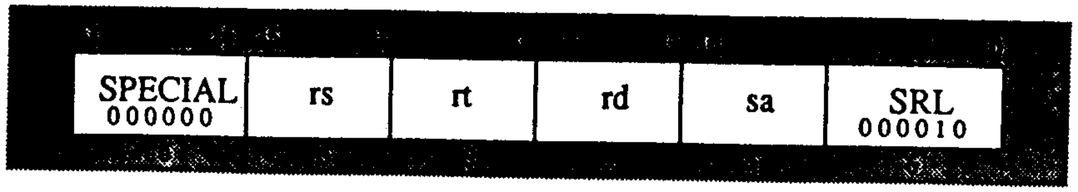
```
T:    s ← GPR[rs]4..0
      GPR[rd] ← (GPR[rt]31)^s || GPR[rt]31..s
```

Exceptions:

None.

SRL

Shift Right Logical



Format:

SRL rd,rt,sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, inserting zeros into the high order bits. The result is placed in register *rd*.

Operation:

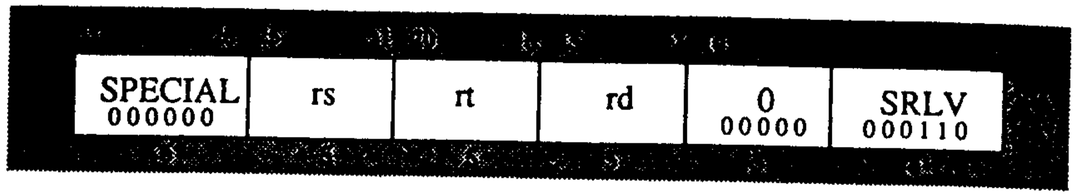
```
T:    GPR[rd] ← 0^sa || GPR[rt]31..sa
```

Exceptions:

None.

Shift Right Logical Variable

SRLV



Format:

SRLV rd,rt,rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low order five bits of general register *rs*, inserting zeros into the high order bits. The 32-bit result is placed in register *rd*.

Operation:

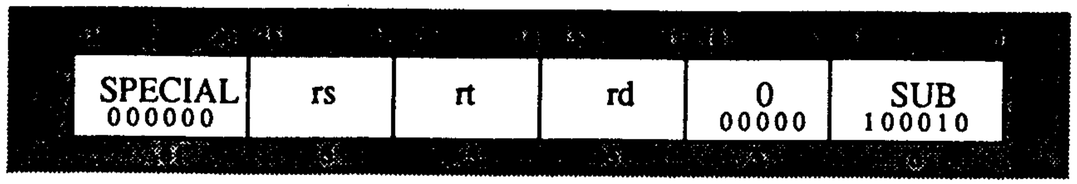
```
T:    s ← GP[rs]4..0
      GPR[rd] ← 0^s || GPR[rt]31..s
```

Exceptions:

None.

SUB

Subtract



Format:

SUB rd,rs,rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*.

The only difference between this instruction and the SUBU instruction is that SUBU never traps on overflow.

An integer overflow exception takes place if the carries out of bits 30 and 31 differ (2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

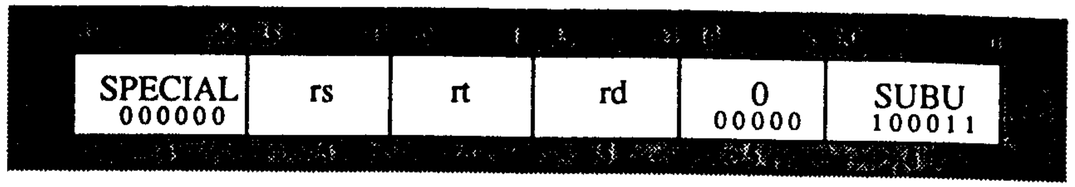
```
T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

Exceptions:

Integer overflow exception

Subtract Unsigned

SUBU



Format:

SUBU rd,rs,rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*.

The only difference between this instruction and the SUB instruction is that SUBU never traps on overflow. No integer overflow exception occurs under any circumstances.

Operation:

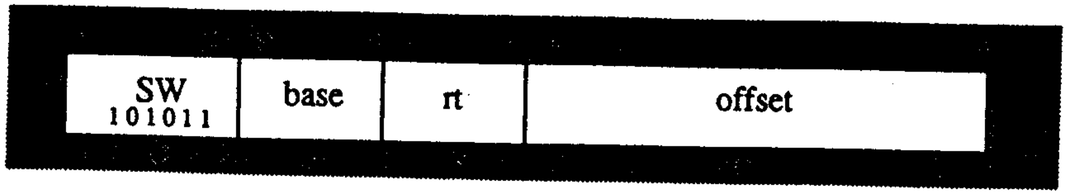
```
T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

Exceptions:

None.

SW

Store Word



Format:

SW rt,offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are stored at the memory location specified by the effective address.

If either of the two least significant bits of the effective address are non-zero, an address error exception occurs.

Operation:

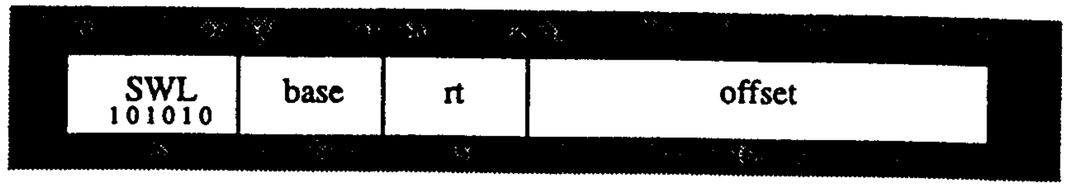
```
T:    vAddr ← ((offset15)^16 || offset15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      data ← GPR[rt]
      StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SWL

Store Word Left



Format:

SWL rt,offset(base)

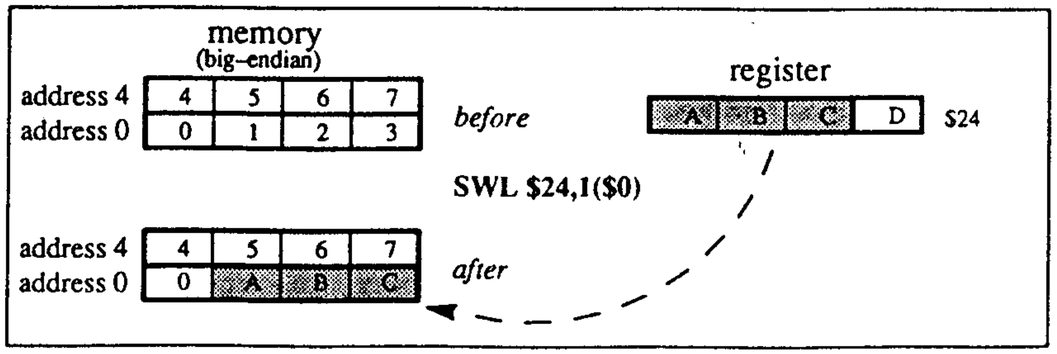
Description:

This instruction can be used with the SWR instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words. SWL stores the left portion of the register into the appropriate part of the high order word of memory; SWR stores the right portion of the register into the appropriate part of the low order word.

The SWL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most significant byte of the register and copies it to the specified byte in memory; then it proceeds toward the low order byte of the register and the low order byte of the word in memory, copying bytes from register to memory until it reaches the low order byte of the word in memory.

No address exceptions due to alignment are possible.

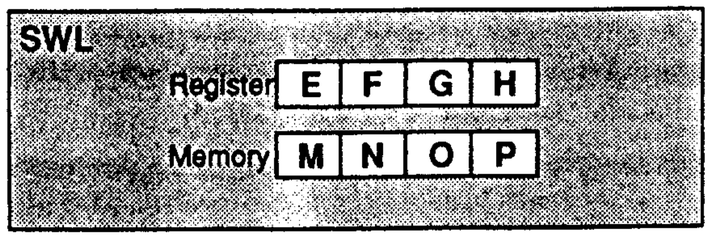


Store Word Left (continued) — SWL

Operation:

```
T:    vAddr ← ((offset15)16 || offset 15..0) + GPR[base]
      (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
      pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
      byte ← vAddr1..0 xor BigEndianCPU2
      If BigEndianMem = 0 then
            pAddr ← pAddrPSIZE-1..2 || 02
      endif
      data ← 024-8*byte || GPR[rt]31..24-8*byte
      StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)
```

Given a word in a register and a word in memory, the operation of SWL is as follows:



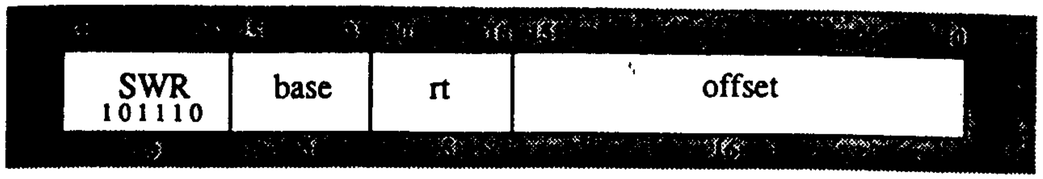
vAddr2..0	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset LEM	Offset BEM	Destination	Type	Offset LEM	Offset BEM
0	M N O E	0	0	3	E F G H	3	0	0
1	M N E F	1	0	2	M E F G	2	0	1
2	M E F G	2	0	1	M N E F	1	0	2
3	E F G H	3	0	0	M N O E	0	0	3

LEM BigEndianMem = 0
BEM BigEndianMem = 1
Type AccessType sent to memory
Offset $pAddr_{2..0}$ sent to memory

Exceptions:

TLB refill exception TLB invalid exception TLB modification exception
Bus error exception Address error exception

Store Word Right — SWR



Format:

SWR rt,offset(base)

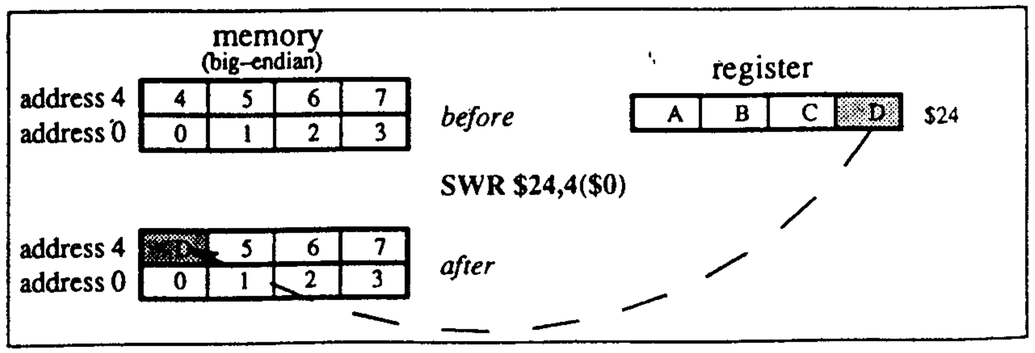
Description:

This instruction can be used with the SWL instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words. SWR stores the right portion of the register into the appropriate part of the low order word; SWL stores the left portion of the register into the appropriate part of the low order word of memory.

The SWR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least significant (rightmost) byte of the register and copies it to the specified byte in memory; then it proceeds toward the high order byte of the register and the high order byte of the word in memory, copying bytes from register to memory until it reaches the high order byte of the word in memory.

No address exceptions due to alignment are possible.

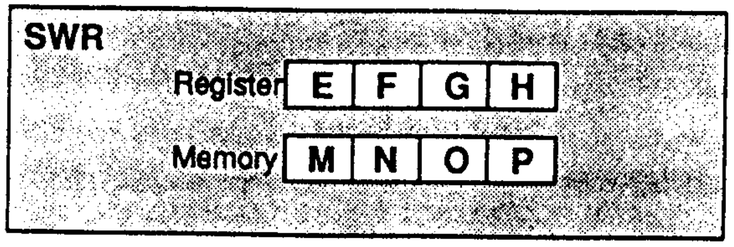


SWR — Store Word Right (continued)

Operation:

```
T:   vAddr ← ((offset15)^16 || offset 15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian^2)
     byte ← vAddr1..0 xor BigEndianCPU^2
     If BigEndianMem = 1 then
          pAddr ← pAddrPSIZE-1..2 || 0^2
     endif
     data ← GPR[rt]31..8*byte..0 || 0^8*byte
     StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
```

Given a word in a register and a word in memory, the operation of SWR is as follows:



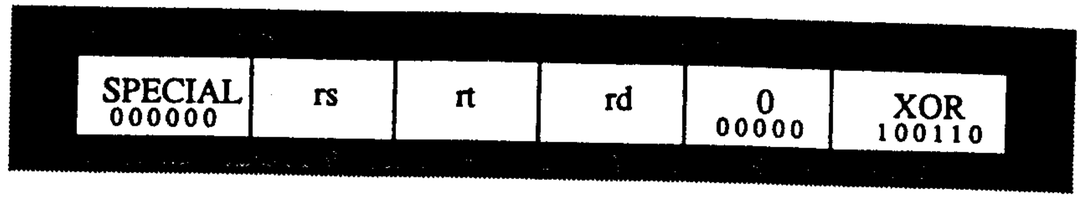
vAddr2..0	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset LEM	Offset BEM	Destination	Type	Offset LEM	Offset BEM
0	E F G H	3	0	0	H N O P	0	3	0
1	F G H P	2	1	0	G H O P	1	2	0
2	G H O P	1	2	0	F G H P	2	1	0
3	H N O P	0	3	0	E F G H	3	0	0

LEM BigEndianMem = 0
BEM BigEndianMem = 1
Type AccessType sent to memory
Offset pAddr2..0 sent to memory

Exceptions:

TLB refill exception TLB invalid exception TLB modification exception
Bus error exception Address error exception

Exclusive Or — XOR



Format:

XOR rd,rs,rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical exclusive OR operation. The result is placed into general register *rd*.

Operation:

```
T:   GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None.

XORI

Exclusive Or Immediate

XORI 001110	rs	rt	immediate

Format:

XORI rt,rs,immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical exclusive-OR operation. The result is placed into general register *rt*.

Operation:

```
T:    GPR[rt] ← GPR[rs] xor (0 16 || immediate)
```

Exceptions:

None.

CPU Instruction Opcode Bit Encoding

The remainder of this Appendix presents the opcode bit encoding for the CPU instruction set (ISA and extensions), as implemented by the R2000/3000 (Figure A–1), R4000 (Figure A–2), and R6000 (Figure A–3).

Opcode (28..26 × 31..29)

31..29 \ 28..26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	COP3	*	*	*	*
3	*	*	*	*	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	SB	SH	SWL	SW	*	*	SWR	*
6	*	LWC1	LWC2	LWC3	*	*	*	*
7	*	SWC1	SWC2	SWC3	*	*	*	*

SPECIAL function (2..0 × 5..3)

5..3 \ 2..0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	*
2	MFHI	MTHI	MFLO	MTLO	*	*	*	*
3	MULT	MULTU	DIV	DIVU	*	*	*	*
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

REGIMM rt (18..16 × 20..19)

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	Y	Y	Y	Y	Y	Y
1	Y	Y	Y	Y	Y	Y	Y	Y
2	BLTZAL	BGEZAL	Y	Y	Y	Y	Y	Y
3	Y	Y	Y	Y	Y	Y	Y	Y

COPz rs (23..21 × 25..24)

25..24 \ 23..21	0	1	2	3	4	5	6	7
0	MF	Y	CF	Y	MT	Y	CT	Y
1	BC	Y	Y	Y	Y	Y	Y	Y
2	CO							
3								

COPz rt (18..16 × 20..19)

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BCF	BCT	Y	Y	Y	Y	Y	Y
1	Y	Y	Y	Y	Y	Y	Y	Y
2	Y	Y	Y	Y	Y	Y	Y	Y
3	Y	Y	Y	Y	Y	Y	Y	Y

Figure A–1. R2000/R3000 Opcode Bit Encoding

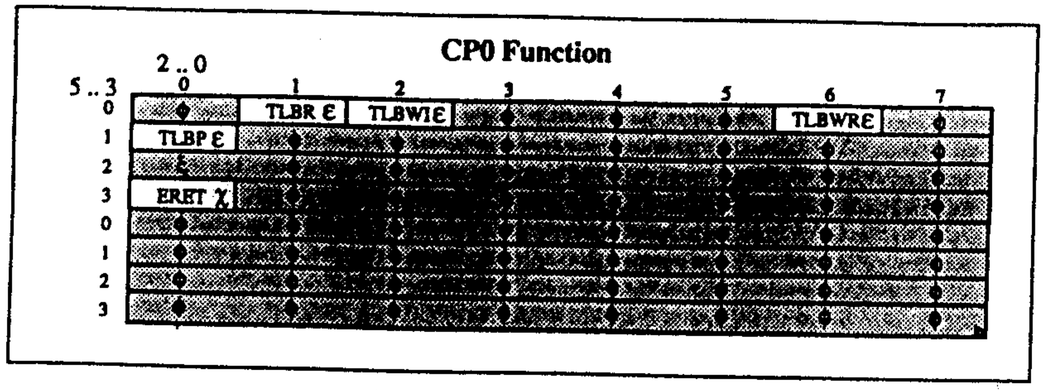


Figure A–2. R4000 Opcode Bit Encoding (cont.)

Key:

* Operation codes marked with an asterisk cause reserved instruction exceptions in all current implementations and are reserved for future versions of the architecture.

α Operation codes marked with an alpha cause reserved instruction exceptions in R2000/R3000 implementations, and are valid for R4000 implementations.

β Operation codes marked with a beta are not valid for R2000/R3000 implementations, and are valid for R4000/R6000 implementations. R2000/R3000 implementations do not take a reserved instruction exception on these opcodes.

γ Operation codes marked with a gamma are not valid for R2000/R3000 implementations, and for R4000 implementations cause a reserved instruction exception. They are reserved for future versions of the architecture.

δ Operation codes marked with a delta are valid only for R4000 processors with CP0 enabled, and cause a reserved instruction exception on other processors.

ε Operation codes marked with an epsilon are valid only for R2000, R3000 and R4000 processors (processors with an on-chip associative TLB), and are not valid on the R6000.

φ Operation codes marked with a phi are invalid but do not cause reserved instruction exceptions in R4000 implementations.

ξ Operation codes marked with a xi are valid on the R2000, R3000 and R6000, but are not valid and cause a reserved instruction exception on R4000 processors.

χ Operation codes marked with a chi are valid only on R4000 processors and cause a reserved instruction exception on the R2000, R3000 and R6000.

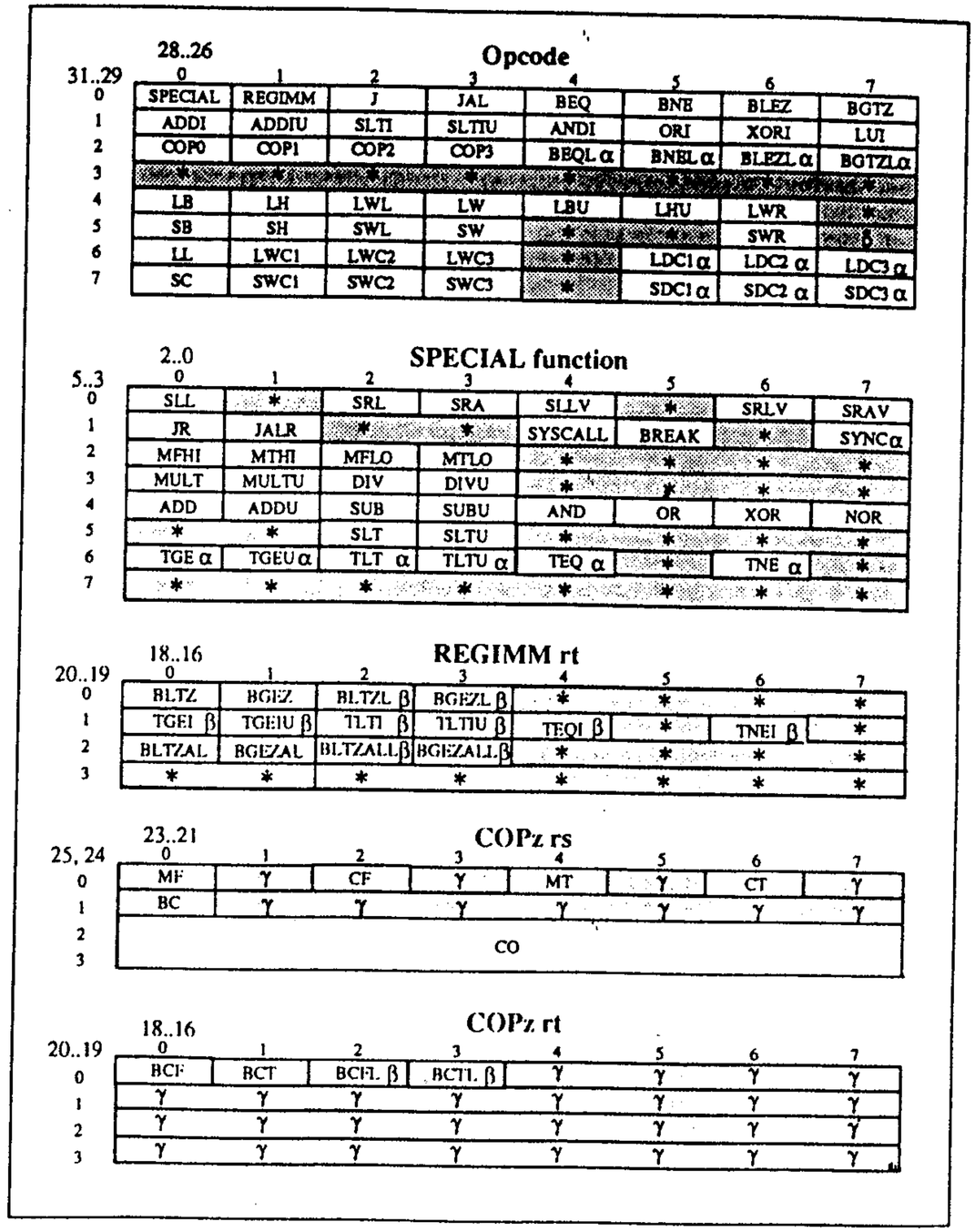


Figure A–3. R6000 Opcode Bit Encoding