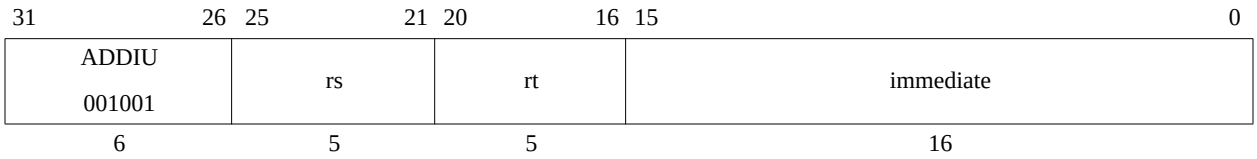
Add Immediate Unsigned Word — ADDIU



Format: `ADDIU rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To add a constant to a 32-bit integer

Description: `rt ← rs + immediate`

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rt.*

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← temp
```

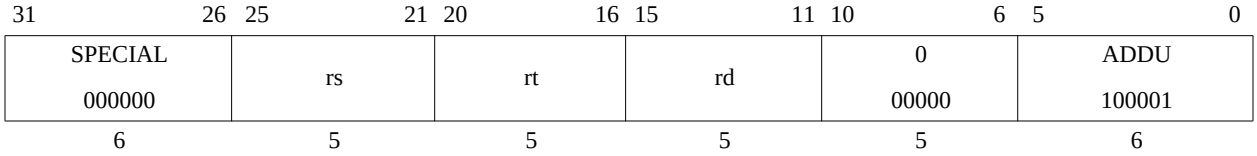
Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

Add Unsigned Word — ADDU



Format: `ADDU rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To add 32-bit integers

Description: `rd ← rs + rt`

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

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

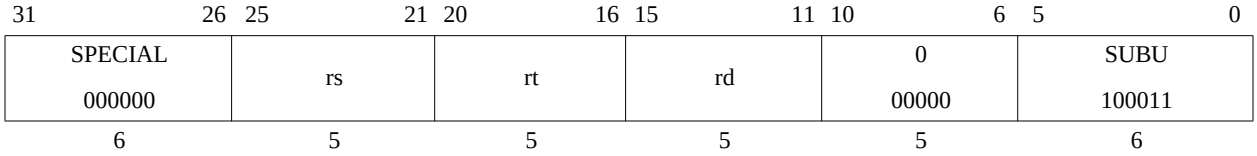
Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

Subtract Unsigned Word SUBU



Format: `SUBU rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To subtract 32-bit integers

Description: `rd ← rs - rt`

The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* and the 32-bit arithmetic result is and placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

```
temp   ← GPR[rs] - GPR[rt]
GPR[rd]← temp
```

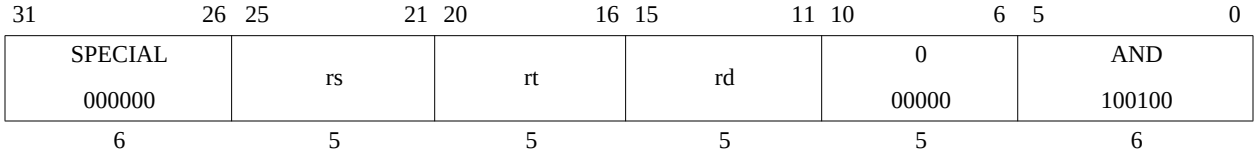
Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. It is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

And AND



Format: `AND rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical AND

Description: `rd ← rs AND rt`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical AND operation. The result is placed into GPR *rd*.

Restrictions:

None

Operation:

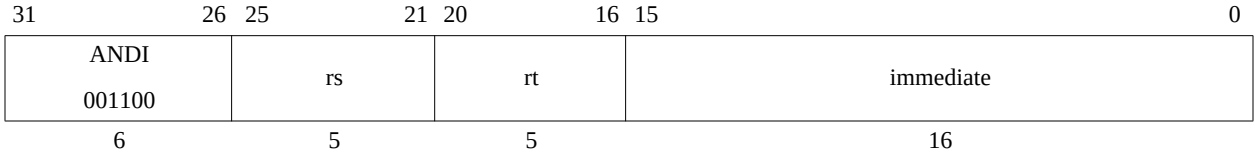
```
GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

And Immediate

ANDI



Format: `ANDI rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical AND with a constant

Description: `rt ← rs AND immediate`

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical AND operation. The result is placed into GPR *rt*.

Restrictions:

None

Operation:

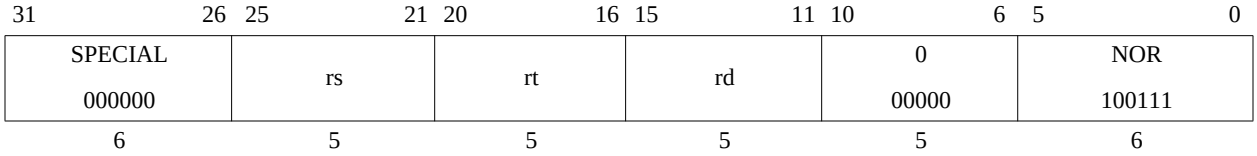
```
GPR[rt] ← GPR[rs] and zero_extend(immediate)
```

Exceptions:

None

Not Or

NOR



Format: `NOR rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical NOT OR

Description: `rd ← rs NOR rt`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical NOR operation. The result is placed into GPR *rd*.

Restrictions:

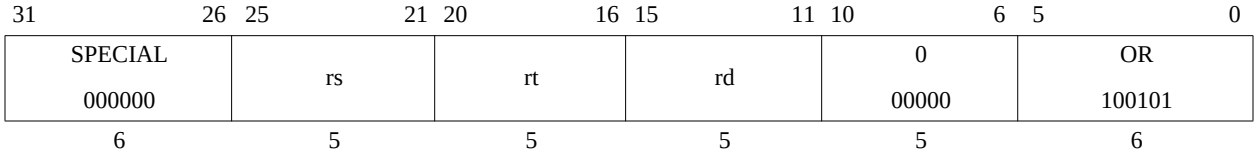
None

Operation:

```
GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None



Format: `OR rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical OR

Description: `rd ← rs or rt`

The contents of GPR *rs* are combined with the contents of GPR *rt* in a bitwise logical OR operation. The result is placed into GPR *rd*.

Restrictions:

None

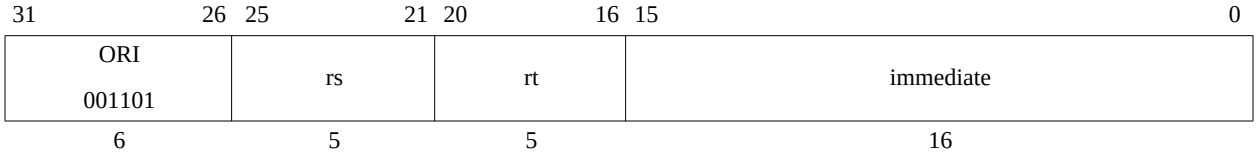
Operation:

```
GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:

None

Or Immediate ORI



Format: `ORI rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical OR with a constant

Description: `rt ← rs or immediate`

The 16-bit *immediate* is zero-extended to the left and combined with the contents of GPR *rs* in a bitwise logical OR operation. The result is placed into GPR *rt*.

Restrictions:

None

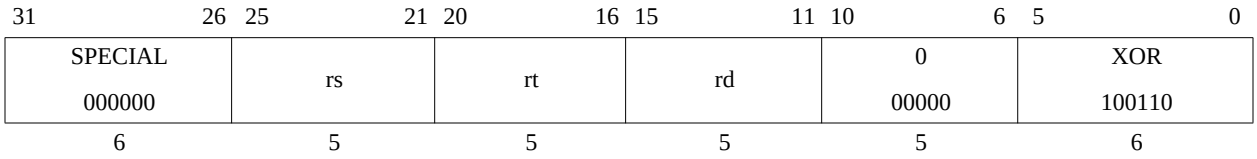
Operation:

```
GPR[rt] ← GPR[rs] or zero_extend(immediate)
```

Exceptions:

None

Exclusive OR — XOR



Format: `XOR rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical Exclusive OR

Description: `rd ← rs XOR rt`

Combine the contents of GPR *rs* and GPR *rt* in a bitwise logical Exclusive OR operation and place the result into GPR *rd.*

Restrictions:

None

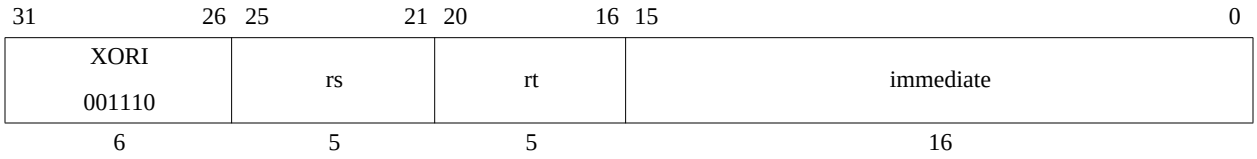
Operation:

```
GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None

Exclusive OR Immediate XORI



Format: `XORI rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To do a bitwise logical Exclusive OR with a constant

Description: `rt ← rs XOR immediate`

Combine the contents of GPR *rs* and the 16-bit zero-extended *immediate* in a bitwise logical Exclusive OR operation and place the result into GPR *rt.*

Restrictions:

None

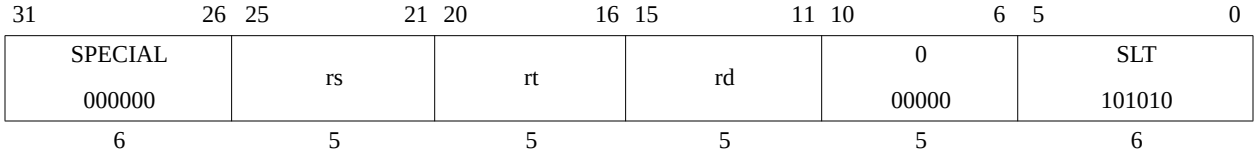
Operation:

```
GPR[rt] ← GPR[rs] xor zero_extend(immediate)
```

Exceptions:

None

Set on Less Than SLT



Format: `SLT rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To record the result of a less-than comparison

Description: `rd ← (rs < rt)`

Compare the contents of GPR *rs* and GPR *rt* as signed integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

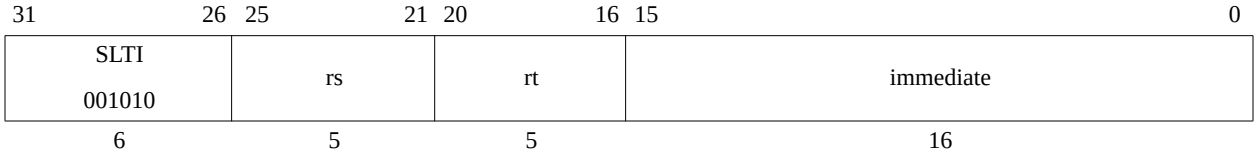
Operation:

```
if GPR[rs] < GPR[rt] then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

Set on Less Than Immediate SLTI



Format: `SLTI rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To record the result of a less-than comparison with a constant

Description: `rt ← (rs < immediate)`

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

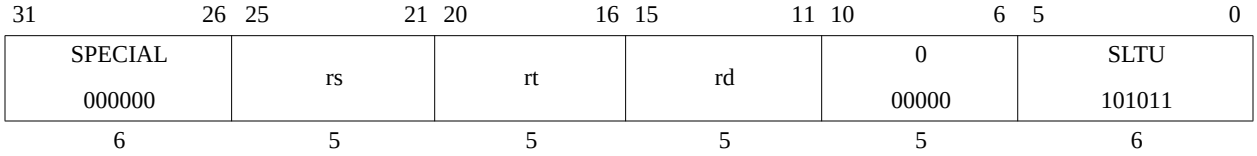
Operation:

```
if GPR[rs] < sign_extend(immediate) then
   GPR[rd] ← 0^(GPRLEN-1)|| 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

Set on Less Than Unsigned — SLTU



Format: `SLTU rd, rs, rt` **MIPS32 (MIPS I)**

Purpose:

To record the result of an unsigned less-than comparison

Description: `rd ← (rs < rt)`

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers and record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

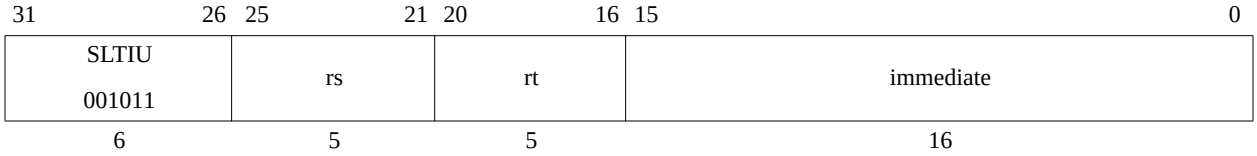
Operation:

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

Set on Less Than Immediate Unsigned — SLTIU



Format: `SLTIU rt, rs, immediate` **MIPS32 (MIPS I)**

Purpose:

To record the result of an unsigned less-than comparison with a constant

Description: `rt ← (rs < immediate)`

Compare the contents of GPR *rs* and the sign-extended 16-bit *immediate* as unsigned integers and record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

Because the 16-bit *immediate* is sign-extended before comparison, the instruction can represent the smallest or largest unsigned numbers. The representable values are at the minimum [0, 32767] or maximum [max_unsigned-32767, max_unsigned] end of the unsigned range.

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

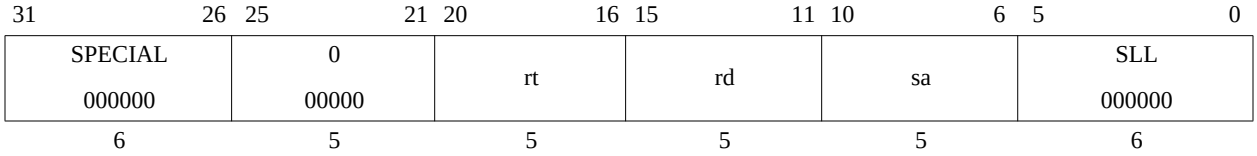
Operation:

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
   GPR[rd] ← 0^(GPRLEN-1) || 1
else
   GPR[rd] ← 0^GPRLEN
endif
```

Exceptions:

None

Shift Word Left Logical — SLL



Format: `SLL rd, rt, sa` **MIPS32 (MIPS I)**

Purpose:

To left-shift a word by a fixed number of bits

Description: `rd ← rt << sa`

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

None

Operation:

```
s      ← sa
temp   ← GPR[rt](31-s)..0 || 0^s
GPR[rd]← temp
```

Exceptions:

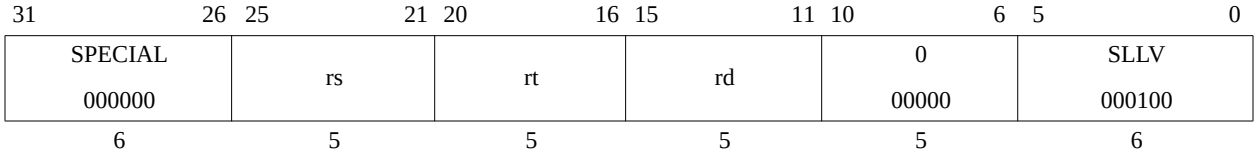
None

Programming Notes:

SLL r0, r0, 0, expressed as NOP, is the assembly idiom used to denote no operation.

SLL r0, r0, 1, expressed as SSNOP, is the assembly idiom used to denote no operation that causes an issue break on superscalar processors.

Shift Word Left Logical Variable — SLLV



Format: `SLLV rd, rt, rs` **MIPS32 (MIPS I)**

Purpose: To left-shift a word by a variable number of bits

Description: `rd ← rt << rs`

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeros into the emptied bits; the result word is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs*.

Restrictions: None

Operation:

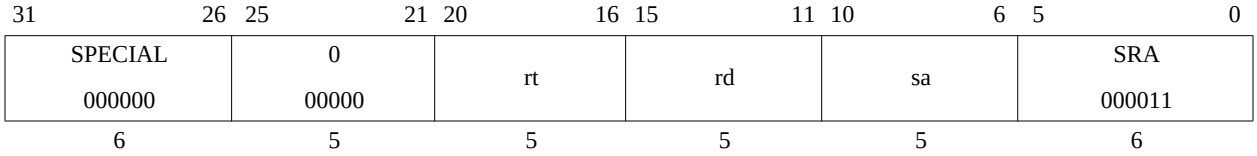
```
s      ← GPR[rs]4..0
temp   ← GPR[rt](31-s)..0 || 0s
GPR[rd]← temp
```

Exceptions: None

Programming Notes:

None

Shift Word Right Arithmetic SRA



Format: `SRA rd, rt, sa` **MIPS32 (MIPS I)**

Purpose:

To execute an arithmetic right-shift of a word by a fixed number of bits

Description: `rd ← rt >> sa (arithmetic)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

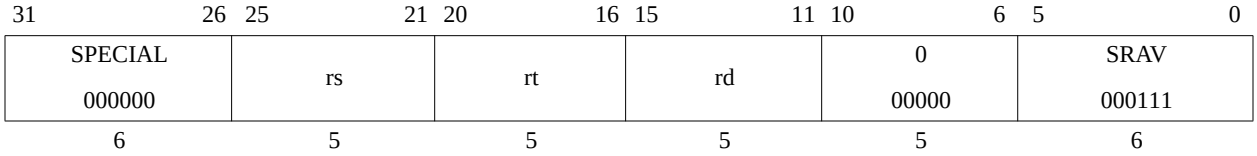
None

Operation:

```
s      ← sa
temp   ← (GPR[rt]31)^s || GPR[rt]31..s
GPR[rd]← temp
```

Exceptions: None

Shift Word Right Arithmetic Variable — SRAV



Format: `SRAV rd, rt, rs` **MIPS32 (MIPS I)**

Purpose:

To execute an arithmetic right-shift of a word by a variable number of bits

Description: `rd ← rt >> rs (arithmetic)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs.*

Restrictions:

None

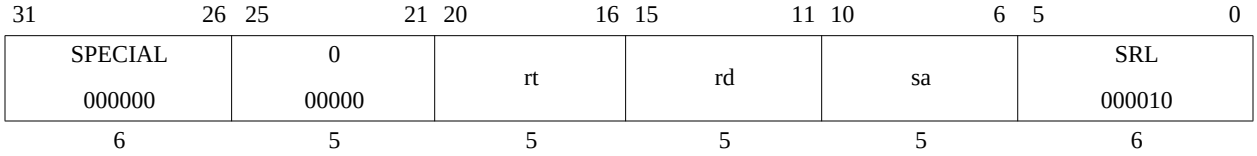
Operation:

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

Exceptions:

None

Shift Word Right Logical SRL



Format: `SRL rd, rt, sa` **MIPS32 (MIPS I)**

Purpose:

To execute a logical right-shift of a word by a fixed number of bits

Description: `rd ← rt >> sa (logical)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa.*

Restrictions:

None

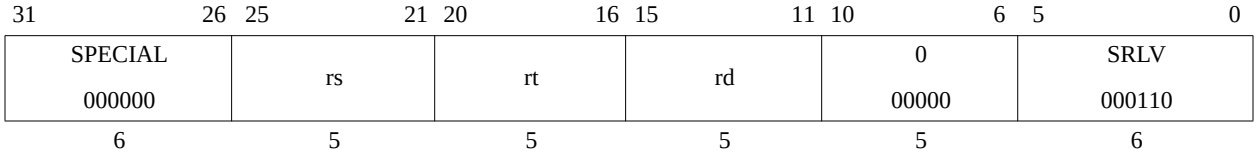
Operation:

```
s      ← sa
temp   ← 0^s || GPR[rt]_31..s
GPR[rd]← temp
```

Exceptions:

None

Shift Word Right Logical Variable — SRLV



Format: `SRLV rd, rt, rs` **MIPS32 (MIPS I)**

Purpose:

To execute a logical right-shift of a word by a variable number of bits

Description: `rd ← rt >> rs (logical)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, inserting zeros into the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by the low-order 5 bits of GPR *rs.*

Restrictions:

None

Operation:

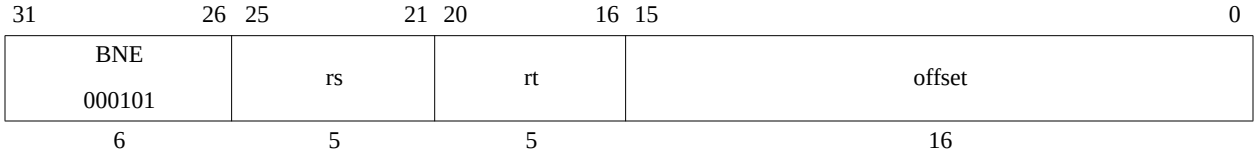
```
s      ← GPR[rs]4..0
temp   ← 0^s || GPR[rt]31..s
GPR[rd]← temp
```

Exceptions:

None

Branch on Not Equal

BNE



Format: `BNE rs, rt, offset` **MIPS32 (MIPS I)**

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if rs ≠ rt 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 GPR *rs* and GPR *rt* are not equal, branch to the effective target address after the instruction in the delay slot is executed.

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 ← (GPR[rs] ≠ GPR[rt])
I+1:   if condition then
          PC ← PC + target_offset
       endif
```

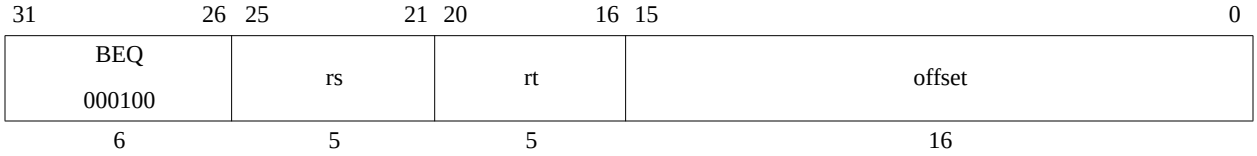
Exceptions:

None

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.

Branch on Equal — BEQ



Format: `BEQ rs, rt, offset` **MIPS32 (MIPS I)**

Purpose:

To compare GPRs then do a PC-relative conditional branch

Description: `if rs = rt 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 GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

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 ← (GPR[rs] = GPR[rt])
I+1:   if condition then
          PC ← PC + target_offset
       endif
```

Exceptions:

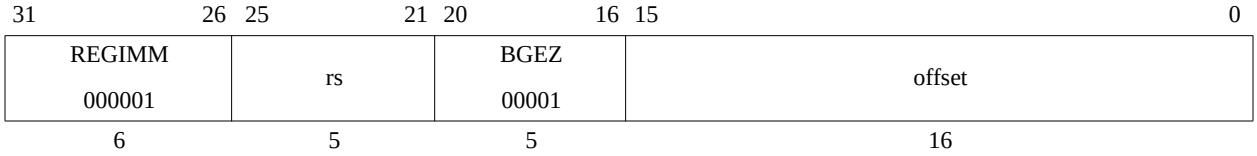
None

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.

BEQ r0, r0 offset, expressed as B offset, is the assembly idiom used to denote an unconditional branch.

Branch on Greater Than or Equal to Zero — BGEZ



Format: `BGEZ rs, offset` **MIPS32 (MIPS I)**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if rs ≥ 0 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 GPR *rs* are greater than or equal to zero (sign bit is 0), branch to the effective target address after the instruction in the delay slot is executed.

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 ← GPR[rs] ≥ 0^GPRLEN
I+1:   if condition then
          PC ← PC + target_offset
       endif
```

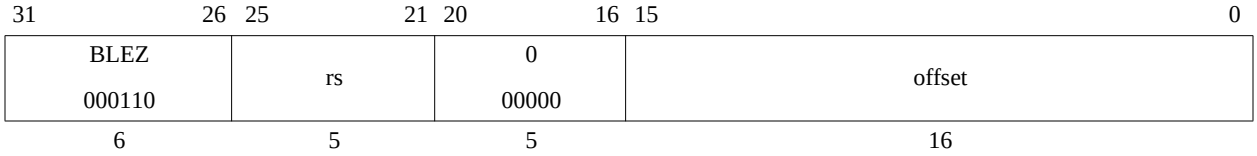
Exceptions:

None

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.

Branch on Less Than or Equal to Zero — BLEZ



Format: `BLEZ rs, offset` **MIPS32 (MIPS I)**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if rs ≤ 0 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 GPR *rs* are less than or equal to zero (sign bit is 1 or value is zero), branch to the effective target address after the instruction in the delay slot is executed.

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 ← GPR[rs] ≤ 0^GPRLEN
I+1:   if condition then
          PC ← PC + target_offset
       endif
```

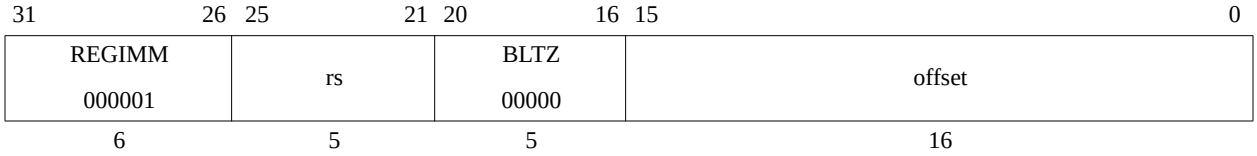
Exceptions:

None

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.

Branch on Less Than Zero — BLTZ



Format: `BLTZ rs, offset` **MIPS32 (MIPS I)**

Purpose:

To test a GPR then do a PC-relative conditional branch

Description: `if rs < 0 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 GPR *rs* are less than zero (sign bit is 1), branch to the effective target address after the instruction in the delay slot is executed.

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 ← GPR[rs] < 0^GPRLEN
I+1:   if condition then
          PC ← PC + target_offset
       endif
```

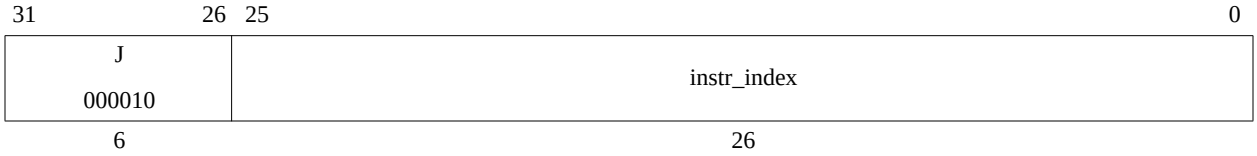
Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump and link (JAL) or jump and link register (JALR) instructions for procedure calls to addresses outside this range.

Jump J



Format: `J target` **MIPS32 (MIPS I)**

Purpose:

To branch within the current 256 MB-aligned region

Description:

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

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:
I+1:PC ← PC_GPRLEN..28 || instr_index || 0^2
```

Exceptions:

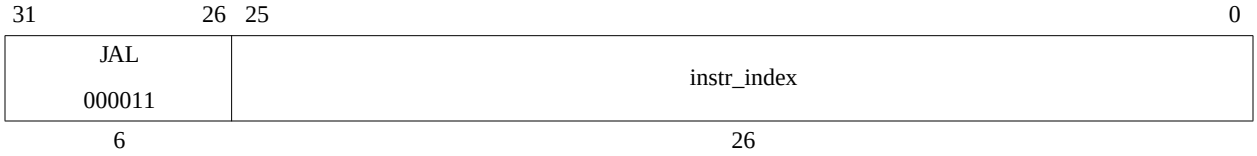
None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the jump instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.

Jump and Link JAL



Format: `JAL target` **MIPS32 (MIPS I)**

Purpose:

To execute a procedure call within the current 256 MB-aligned region

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, at which location execution continues after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the “current” 256 MB-aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself).

Jump to the effective target address. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

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:  GPR[31]← PC + 8
I+1:PC     ← PC_GPRLEN..28 || instr_index || 0^2
```

Exceptions:

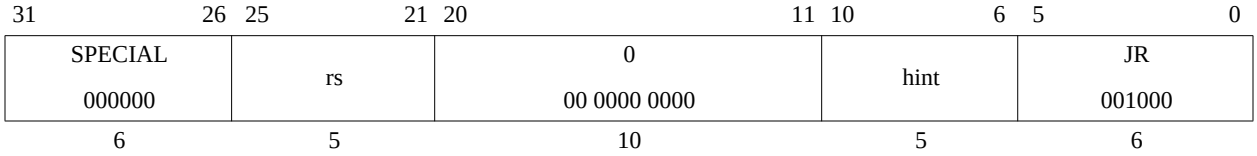
None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch from anywhere in the region to anywhere in the region, an action not allowed by a signed relative offset.

This definition creates the following boundary case: When the branch instruction is in the last word of a 256 MB region, it can branch only to the following 256 MB region containing the branch delay slot.

Jump Register JR



Format: `JR rs` **MIPS32 (MIPS I)**

Purpose:

To execute a branch to an instruction address in a register

Description: `PC ← rs`

Jump to the effective target address in GPR *rs*. Execute the instruction following the jump, in the branch delay slot, before jumping.

For processors that implement the MIPS16 ASE, set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

Restrictions:

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16 ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16 ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

At this time the only defined hint field value is 0, which sets default handling of JR. Future versions of the architecture may define additional hint values.

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:

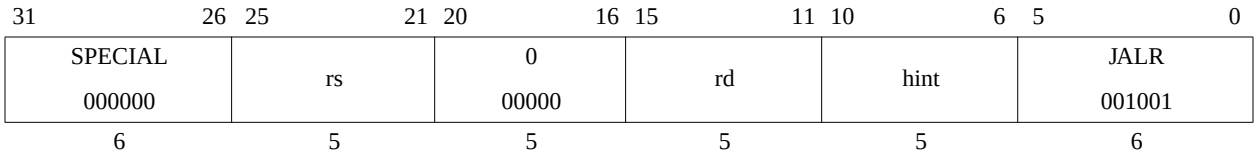
```
I:  temp ← GPR[rs]
I+1:if Config1CA = 0 then
       PC ← temp
    else
       PC ← tempGPRLEN-1..1 || 0
       ISAMode ← temp0
    endif
```

Exceptions:

None

Programming Notes:

Software should use the value 31 for the *rs* field of the instruction word on return from a JAL, JALR, or BGEZAL, and should use a value other than 31 for remaining uses of JR.



Format: `JALR rs (rd = 31 implied)` **MIPS32 (MIPS I)**

`JALR rd, rs` **MIPS32 (MIPS I)**

Purpose:

To execute a procedure call to an instruction address in a register

Description: `rd ← return_addr, PC ← rs`

Place the return address link in GPR *rd*. The return link is the address of the second instruction following the branch, where execution continues after a procedure call.

For processors that do not implement the MIPS16 ASE:

- Jump to the effective target address in GPR *rs*. Execute the instruction that follows the jump, in the branch delay slot, before executing the jump itself.

For processors that do implement the MIPS16 ASE:

- Jump to the effective target address in GPR *rs*. Set the *ISA Mode* bit to the value in GPR *rs* bit 0. Bit 0 of the target address is always zero so that no Address Exceptions occur when bit 0 of the source register is one

At this time the only defined hint field value is 0, which sets default handling of JALR. Future versions of the architecture may define additional hint values.

Restrictions:

Register specifiers *rs* and *rd* must not be equal, because such an instruction does not have the same effect when reexecuted. The result of executing such an instruction is undefined. This restriction permits an exception handler to resume execution by reexecuting the branch when an exception occurs in the branch delay slot.

The effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16 ASE, if either of the two least-significant bits are not zero, an Address Error exception occurs when the branch target is subsequently fetched as an instruction. For processors that do implement the MIPS16 ASE, if bit 0 is zero and bit 1 is one, an Address Error exception occurs when the jump target is subsequently fetched as an instruction.

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:  temp ← GPR[rs]
    GPR[rd] ← PC + 8
I+1:if Config1CA = 0 then
        PC ← temp
    else
        PC ← tempGPRLEN-1..1 || 0
        ISAMode ← temp0
    endif
```

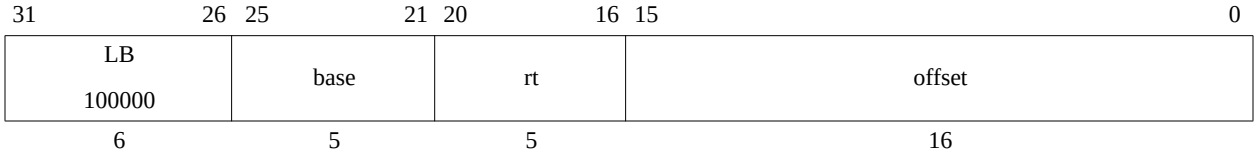
Exceptions:

None

Programming Notes:

This is the only branch-and-link instruction that can select a register for the return link; all other link instructions use GPR 31. The default register for GPR *rd*, if omitted in the assembly language instruction, is GPR 31.

Load Byte LB



Format: `LB rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To load a byte from memory as a signed value

Description: `rt ← memory[base+offset]`

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

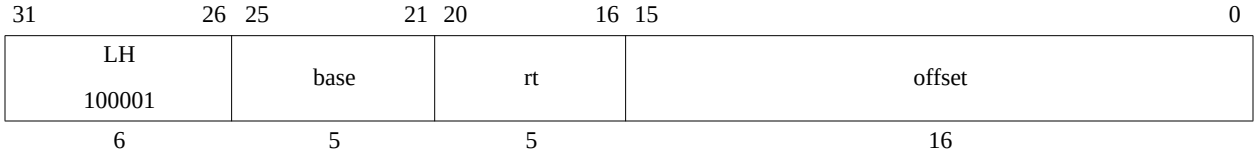
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
memword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte   ← vAddr_1..0 xor BigEndianCPU^2
GPR[rt]← sign_extend(memword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error

Load Halfword LH



Format: `LH rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To load a halfword from memory as a signed value

Description: `rt ← memory[base+offset]`

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, sign-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

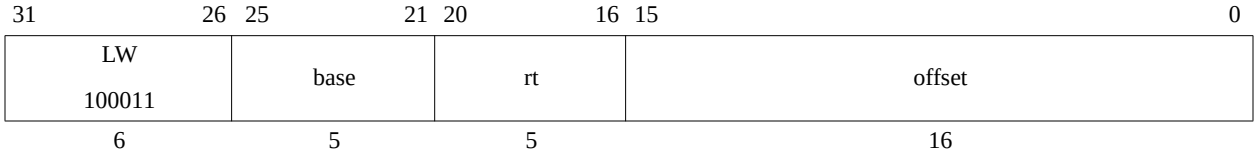
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← vAddr1..0 xor (BigEndianCPU || 0)
GPR[rt] ← sign_extend(memword15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Load Word LW



Format: `LW rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To load a word from memory as a signed value

Description: `rt ← memory[base+offset]`

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 *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

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.

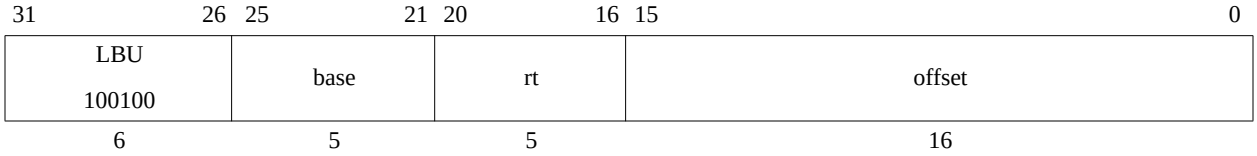
Operation:

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

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

Load Byte Unsigned LBU



Format: `LBU rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To load a byte from memory as an unsigned value

Description: `rt ← memory[base+offset]`

The contents of the 8-bit byte at the memory location specified by the effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

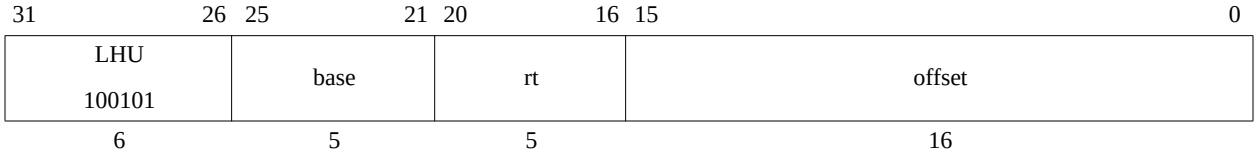
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
memword← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte   ← vAddr_1..0 xor BigEndianCPU^2
GPR[rt]← zero_extend(memword_7+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error

Load Halfword Unsigned LHU



Format: `LHU rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To load a halfword from memory as an unsigned value

Description: `rt ← memory[base+offset]`

The contents of the 16-bit halfword at the memory location specified by the aligned effective address are fetched, zero-extended, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

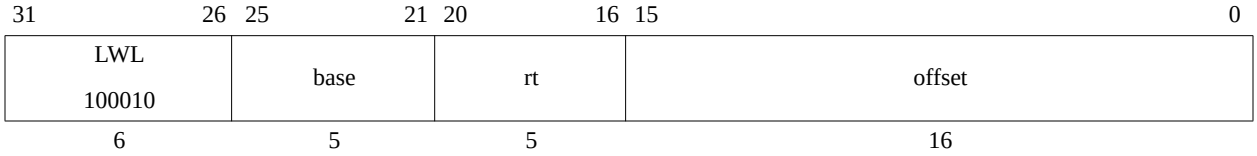
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian || 0))
memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)
byte   ← vAddr1..0 xor (BigEndianCPU || 0)
GPR[rt] ← zero_extend(memword15+8*byte..8*byte)
```

Exceptions:

TLB Refill, TLB Invalid, Address Error

Load Word Left LWL



Format: `LWL rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To load the most-significant part of a word as a signed value from an unaligned memory address

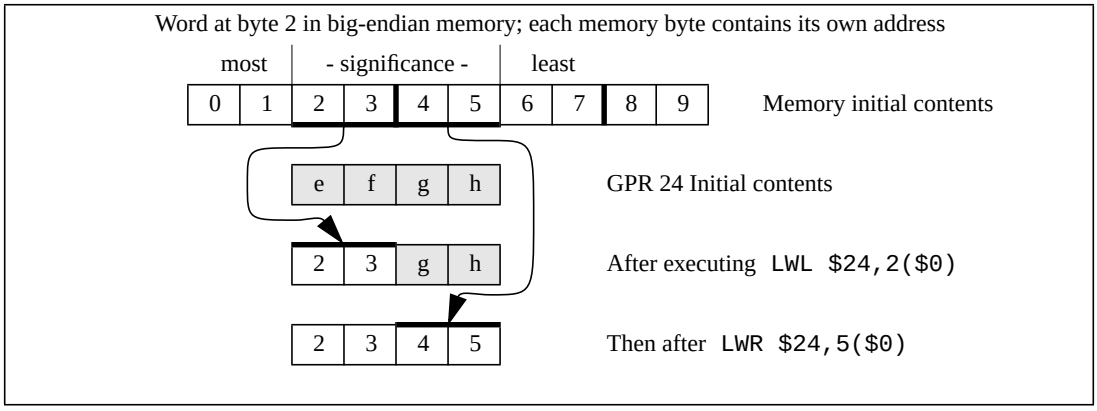
Description: `rt ← rt MERGE memory[base+offset]`

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

The most-significant 1 to 4 bytes of *W* is in the aligned word containing the *EffAddr.* This part of *W* is loaded into the most-significant (left) part of the word in GPR *rt*. The remaining least-significant part of the word in GPR *rt* is unchanged.

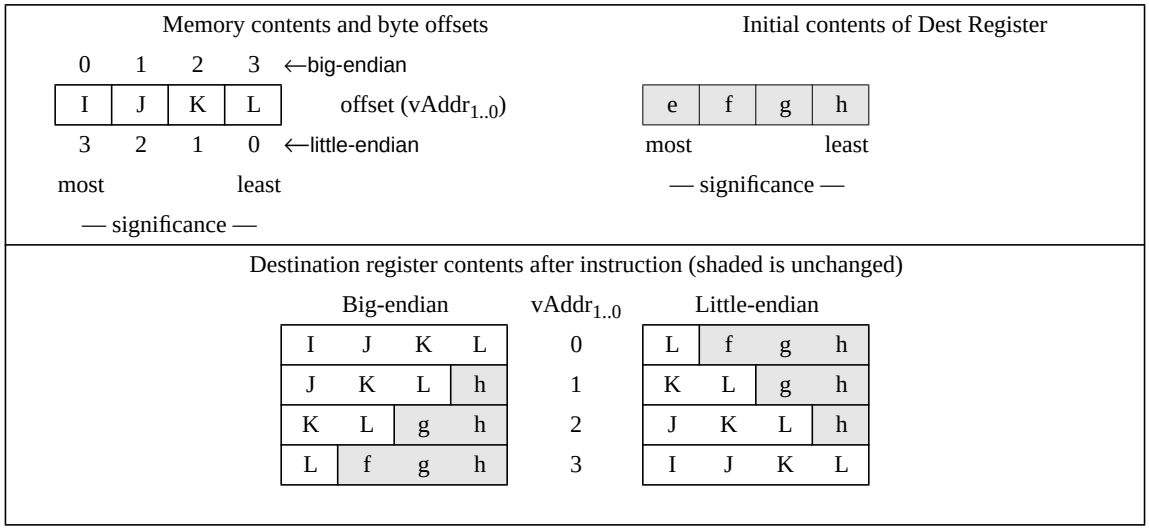
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W,* 2 bytes, is in the aligned word containing the most-significant byte at 2. First, LWL loads these 2 bytes into the left part of the destination register word and leaves the right part of the destination word unchanged. Next, the complementary LWR loads the remainder of the unaligned word

Figure 3-2 Unaligned Word Load Using LWL and LWR



The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ($vAddr_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

Figure 3-3 Bytes Loaded by LWL Instruction



Restrictions:

None

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
if BigEndianMem = 0 then
   pAddr← pAddr_PSIZE-1..2 || 0^2
endif
byte   ← vAddr_1..0 xor BigEndianCPU^2
memword← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp   ← memword_7+8*byte..0 || GPR[rt]_23-8*byte..0
GPR[rt]← temp
```

Exceptions:

None

TLB Refill, TLB Invalid, Bus Error, Address Error

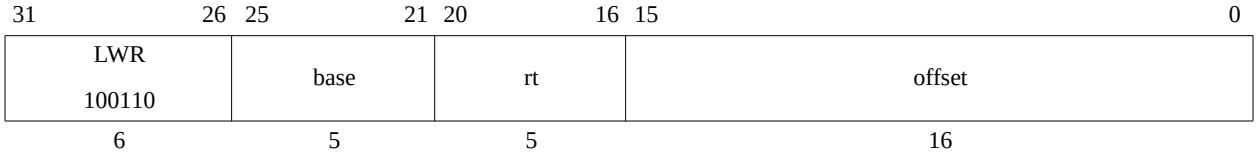
Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

Historical Information

In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.

Load Word Right LWR



Format: `LWR rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To load the least-significant part of a word from an unaligned memory address as a signed value

Description: `rt ← rt MERGE memory[base+offset]`

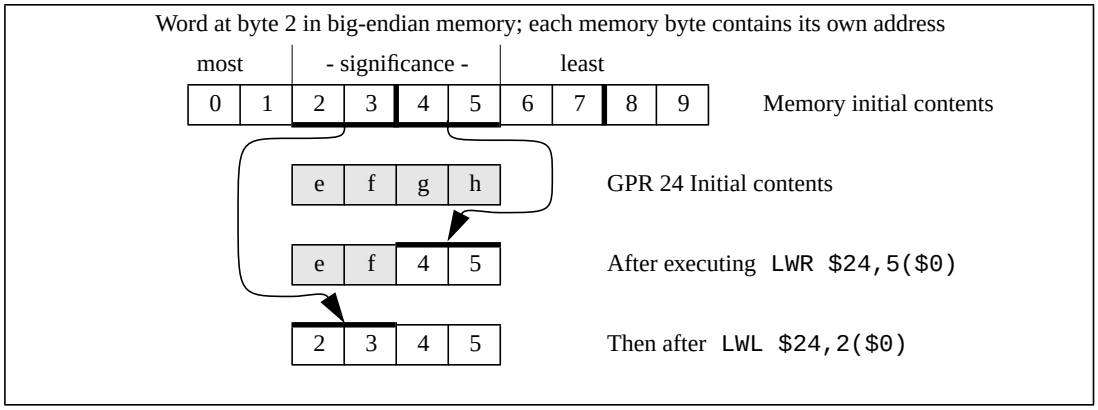
The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

A part of *W*, the least-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. This part of *W* is loaded into the least-significant (right) part of the word in GPR *rt*. The remaining most-significant part of the word in GPR *rt* is unchanged.

Executing both LWR and LWL, in either order, delivers a sign-extended word value in the destination register.

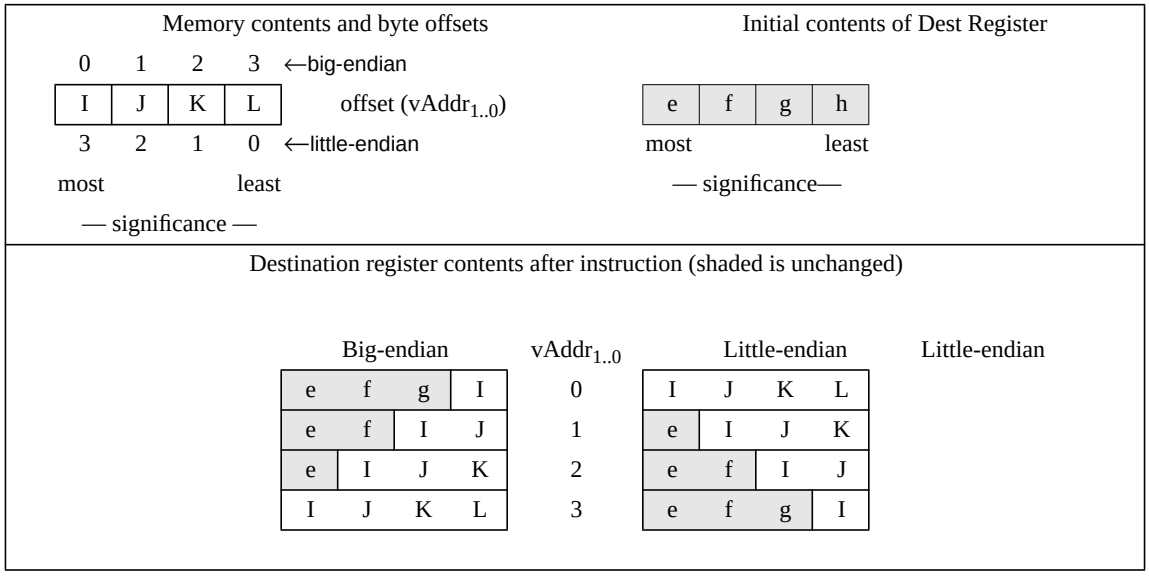
The figure below illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is in the aligned word containing the least-significant byte at 5. First, LWR loads these 2 bytes into the right part of the destination register. Next, the complementary LWL loads the remainder of the unaligned word.

Figure 3-4 Unaligned Word Load Using LWL and LWR



The bytes loaded from memory to the destination register depend on both the offset of the effective address within an aligned word, that is, the low 2 bits of the address ($vAddr_{1..0}$), and the current byte-ordering mode of the processor (big- or little-endian). The figure below shows the bytes loaded for every combination of offset and byte ordering.

Figure 3-5 Bytes Loaded by LWL Instruction



Restrictions:

None

Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, LOAD)
pAddr  ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
if BigEndianMem = 0 then
   pAddr← pAddr_PSIZE-1..2 || 0^2
endif
byte   ← vAddr_1..0 xor BigEndianCPU^2
memword← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp   ← memword_31..32-8*byte || GPR[rt]_31–8*byte..0
GPR[rt]← temp
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error

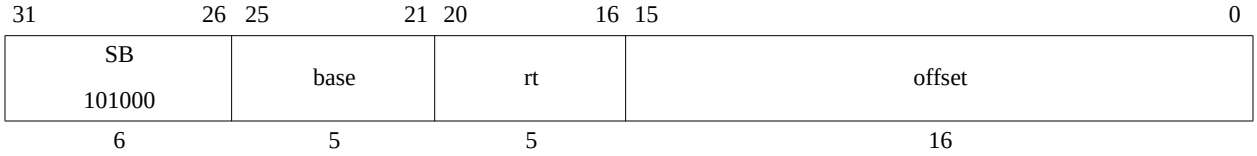
Programming Notes:

The architecture provides no direct support for treating unaligned words as unsigned values, that is, zeroing bits 63..32 of the destination register when bit 31 is loaded.

Historical Information

In the MIPS I architecture, the LWL and LWR instructions were exceptions to the load-delay scheduling restriction. A LWL or LWR instruction which was immediately followed by another LWL or LWR instruction, and used the same destination register would correctly merge the 1 to 4 loaded bytes with the data loaded by the previous instruction. All such restrictions were removed from the architecture in MIPS II.

Store Byte SB



Format: `SB rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To store a byte to memory

Description: `memory[base+offset] ← rt`

The least-significant 8-bit byte of GPR *rt* is stored in memory at the location specified by the effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

None

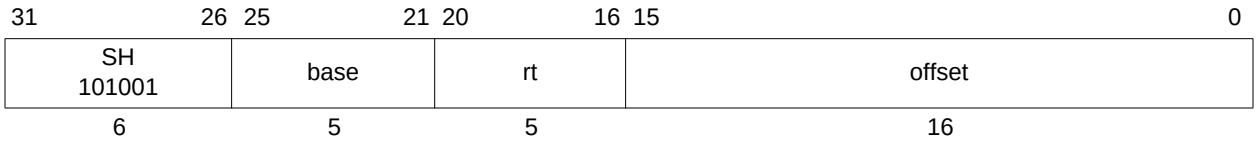
Operation:

```
vAddr     ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr     ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
bytesel   ← vAddr_1..0 xor BigEndianCPU^2
dataword  ← GPR[rt]_31-8*bytesel..0 || 0^8*bytesel
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error

Store Halfword SH



Format: `SH rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To store a halfword to memory

Description: `memory[base+offset] ← rt`

The least-significant 16-bit halfword of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

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

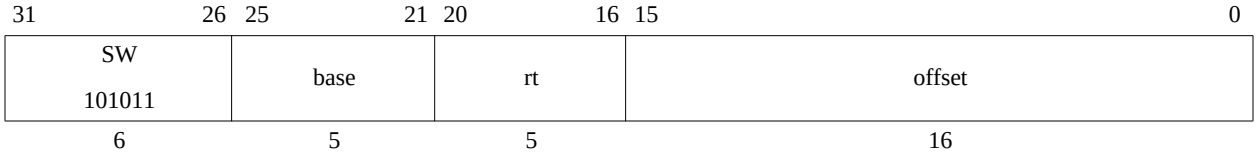
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor (ReverseEndian || 0))
bytesel← vAddr1..0 xor (BigEndianCPU || 0)
dataword← GPR[rt]31–8*bytesel..0 || 08*bytesel
StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Store Word SW



Format: `SW rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To store a word to memory

Description: `memory[base+offset] ← rt`

The least-significant 32-bit word of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

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.

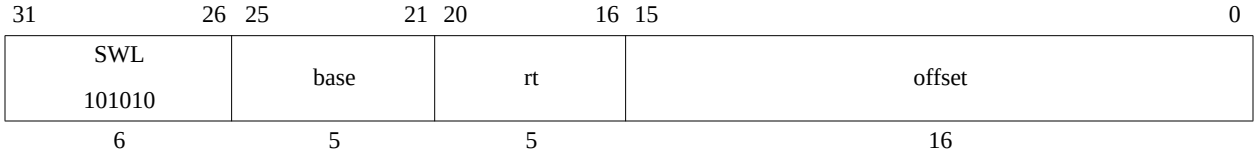
Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
dataword← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error

Store Word Left — SWL



Format: `SWL rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To store the most-significant part of a word to an unaligned memory address

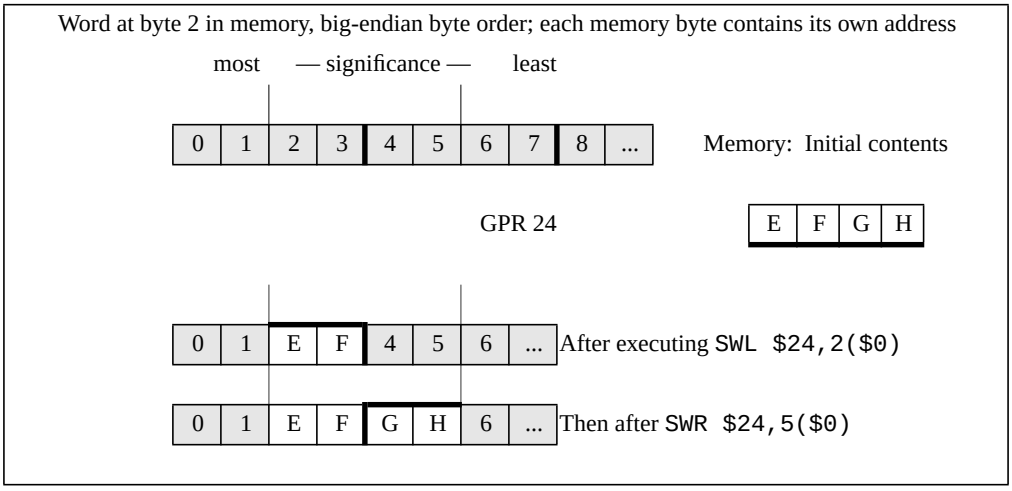
Description: `memory[base+offset] ← rt`

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the most-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

A part of *W*, the most-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. The same number of the most-significant (left) bytes from the word in GPR *rt* are stored into these bytes of *W.*

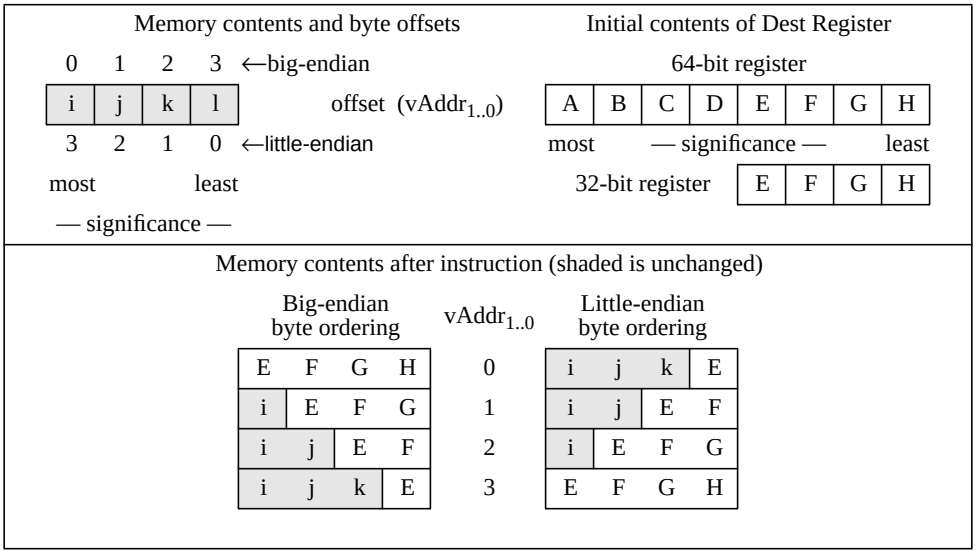
The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is located in the aligned word containing the most-significant byte at 2. First, SWL stores the most-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWR stores the remainder of the unaligned word.

Figure 3-6 Unaligned Word Store Using SWL and SWR



The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (*vAddr1..0*)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte ordering.

Figure 3-7 Bytes Stored by an SWL Instruction



Restrictions:

None

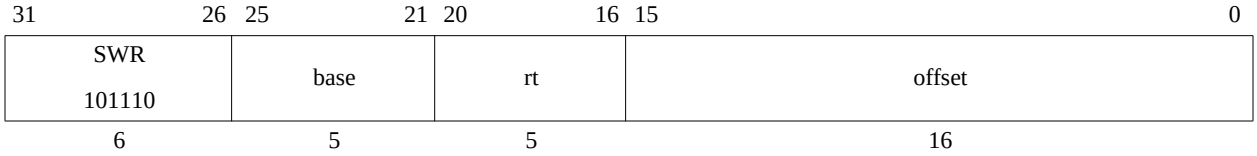
Operation:

```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddrPSIZE-1..2 || (pAddr1..0  xor  ReverseEndian2)
If BigEndianMem = 0 then
   pAddr ← pAddrPSIZE-1..2 || 02
endif
byte   ← vAddr1..0 xor BigEndianCPU2
dataword← 024-8*byte || GPR[rt]31..24-8*byte
StoreMemory(CCA, byte, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error

Store Word Right SWR



Format: `SWR rt, offset(base)` **MIPS32 (MIPS I)**

Purpose:

To store the least-significant part of a word to an unaligned memory address

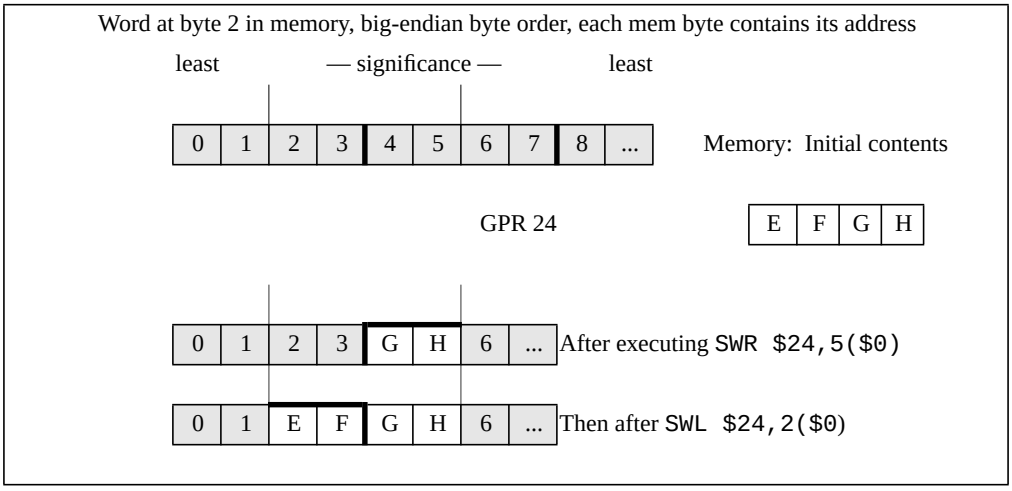
Description: `memory[base+offset] ← rt`

The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address *(EffAddr*). *EffAddr* is the address of the least-significant of 4 consecutive bytes forming a word *(W*) in memory starting at an arbitrary byte boundary.

A part of *W*, the least-significant 1 to 4 bytes, is in the aligned word containing *EffAddr*. The same number of the least-significant (right) bytes from the word in GPR *rt* are stored into these bytes of *W*.

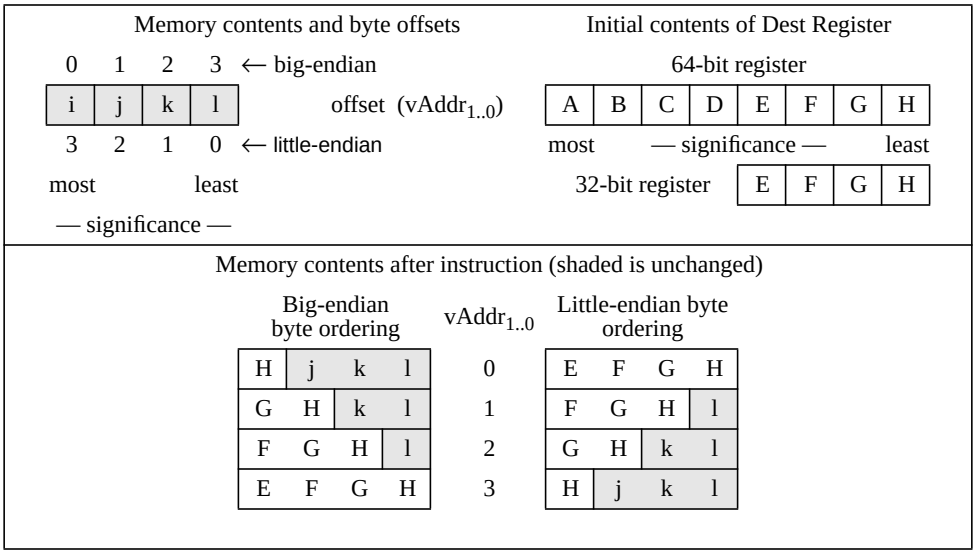
The following figure illustrates this operation using big-endian byte ordering for 32-bit and 64-bit registers. The 4 consecutive bytes in 2..5 form an unaligned word starting at location 2. A part of *W*, 2 bytes, is contained in the aligned word containing the least-significant byte at 5. First, SWR stores the least-significant 2 bytes of the low word from the source register into these 2 bytes in memory. Next, the complementary SWL stores the remainder of the unaligned word.

Figure 3-8 Unaligned Word Store Using SWR and SWL



The bytes stored from the source register to memory depend on both the offset of the effective address within an aligned word—that is, the low 2 bits of the address (*vAddr1..0*)—and the current byte-ordering mode of the processor (big- or little-endian). The following figure shows the bytes stored for every combination of offset and byte-ordering.

Figure 3-9 Bytes Stored by SWR Instruction



Restrictions:

None

Operation:

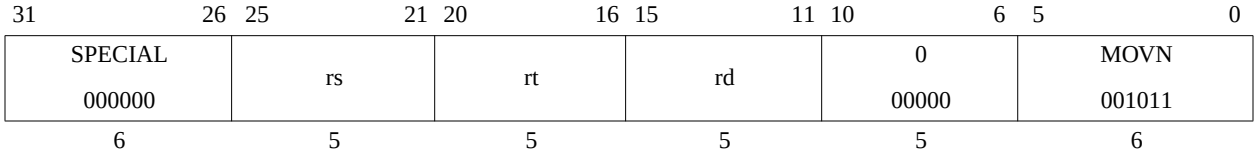
```
vAddr  ← sign_extend(offset) + GPR[base]
(pAddr, CCA)← AddressTranslation (vAddr, DATA, STORE)
pAddr  ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
If BigEndianMem = 0 then
   pAddr ← pAddr_PSIZE-1..2 || 0^2
endif
byte   ← vAddr_1..0 xor BigEndianCPU^2
dataword← GPR[rt]_31-8*byte || 0^8*byte
StoreMemory(CCA, WORD-byte, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Bus Error, Address Error

Move Conditional on Not Zero

MOVN



Format: `MOVN rd, rs, rt` **MIPS32 (MIPS IV)**

Purpose:

To conditionally move a GPR after testing a GPR value

Description: `if rt ≠ 0 then rd ← rs`

If the value in GPR *rt* is not equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
if GPR[rt] ≠ 0 then
   GPR[rd] ← GPR[rs]
endif
```

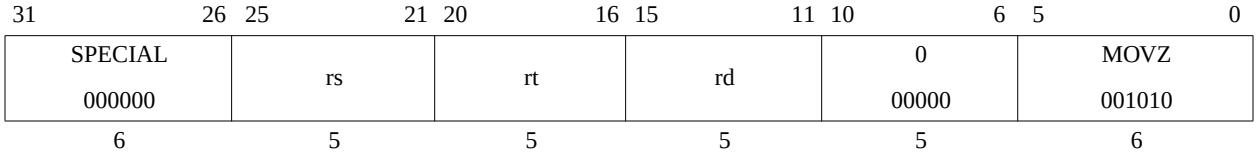
Exceptions:

None

Programming Notes:

The non-zero value tested here is the *condition true* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

Move Conditional on Zero MOVZ



Format: `MOVZ rd, rs, rt` **MIPS32 (MIPS IV**

Purpose:

To conditionally move a GPR after testing a GPR value

Description: `if rt = 0 then rd ← rs`

If the value in GPR *rt* is equal to zero, then the contents of GPR *rs* are placed into GPR *rd*.

Restrictions:

None

Operation:

```
if GPR[rt] = 0 then
   GPR[rd] ← GPR[rs]
endif
```

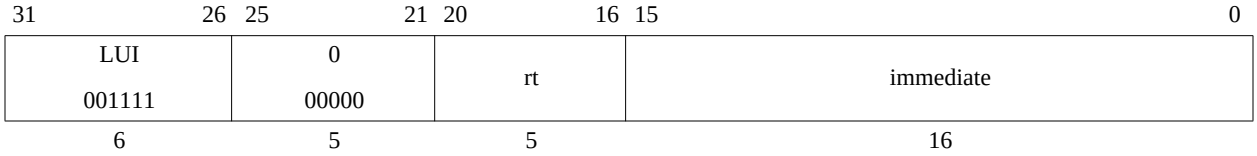
Exceptions:

None

Programming Notes:

The zero value tested here is the *condition false* result from the SLT, SLTI, SLTU, and SLTIU comparison instructions.

Load Upper Immediate — LUI



Format: `LUI rt, immediate` **MIPS32 (MIPS I)**

Purpose:

To load a constant into the upper half of a word

Description: `rt ← immediate || 0^16`

The 16-bit *immediate* is shifted left 16 bits and concatenated with 16 bits of low-order zeros. The 32-bit result is placed into GPR *rt*.

Restrictions:

None

Operation:

```
GPR[rt] ← immediate || 0^16
```

Exceptions:

None