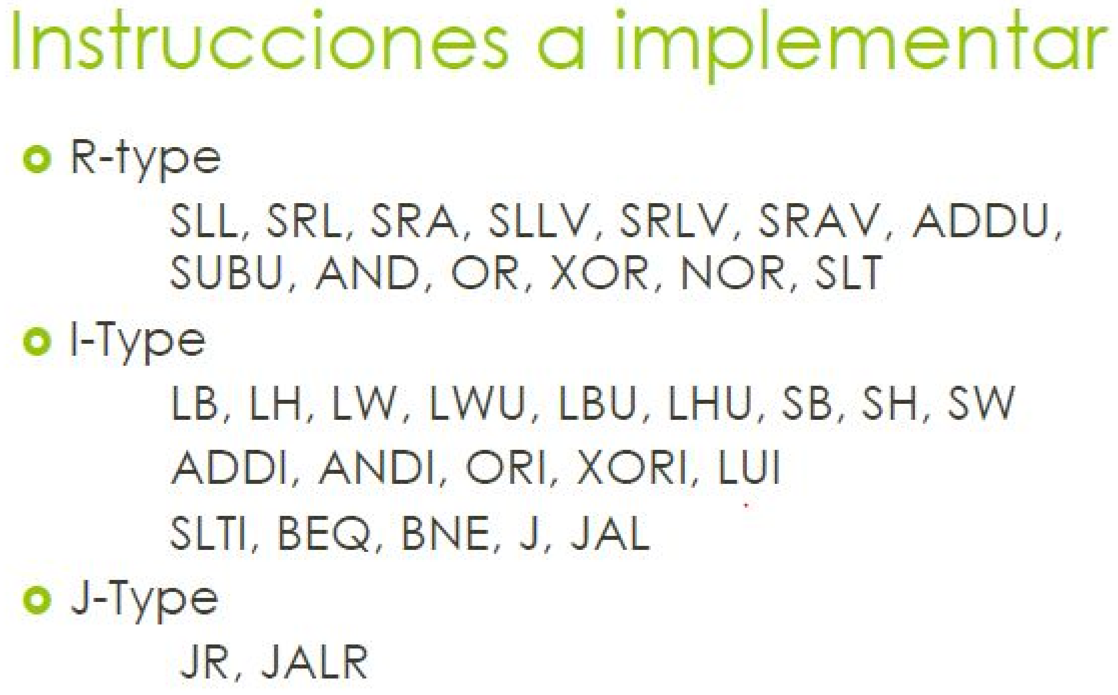
Resumen de las instrucciones del MIPS



Instrucciones a implementar

- R-type
  - SLL, SRL, SRA, SLLV, SRLV, SRAV, ADDU, SUBU, AND, OR, XOR, NOR, SLT
- I-Type
  - LB, LH, LW, LWU, LBU, LHU, SB, SH, SW
  - ADDI, ANDI, ORI, XORI, LUI
  - SLTI, BEQ, BNE, J, JAL
- J-Type
  - JR, JALR

Significados:

R-Type:

1. SLL: Shift Word Left Logical.
2. SRL: Shift Word Right Logical.
3. SRA: Shift Word Right Arithmetic.
4. SLLV: Shift Word Left Logical Variable.

5. SRLV: Shift Word Right Logical Variable.
6. SRAV: Shift Word Right Arithmetic Variable.
7. ADDU: Add Unsigned Word.
8. SUBU: Subtract Unsigned Word.
9. AND: and.
10. OR: or.
11. XOR: exclusive or.
12. NOR: nor.
13. SLT: Set on Less Than.

J-Type:

1. JR: Jump Register.
2. JALR: Jump and Link Register.

I-Type:

1. LB: Load byte.
2. LH: Load Halfword.
3. LW: Load Word.
4. LWU: Load Word unsigned.
5. LBU: Load byte unsigned.
6. LHU: Load Halfword Unsigned.
7. SB: Store byte.
8. SH: Store HalfWord.
9. SW: Store Word.
10. ADDI: Add Immediate Word.
11. ANDI: And Immediate.
12. ORI: Or Immediate.
13. XORI: Exclusive Or Immediate.
14. LUI: Load Upper Immediate.
15. SLTI: Set on Less Than Immediate.
16. BEQ: Branch on Equal.
17. BNE: Branch on Not Equal.
18. J: Jump.

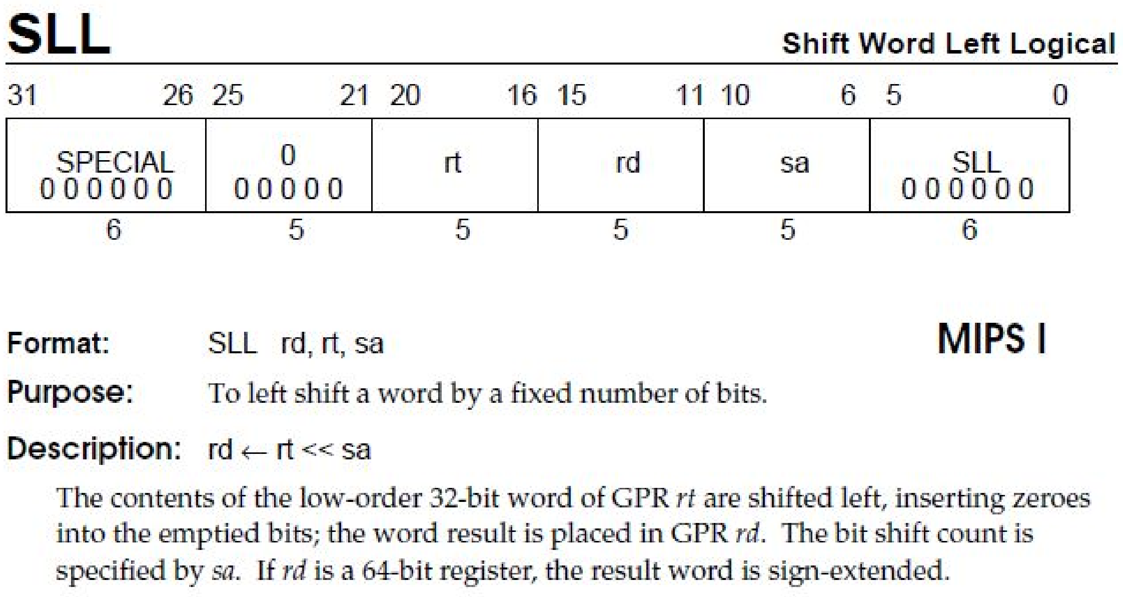
19. JAL: Jump and Link.

Sizes

Halfword: 2 bytes.
Word: 4 bytes.

Descripción de instrucciones R-Type:

1. SLL: Shift Word Left Logical.



SLL **Shift Word Left Logical**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SLL 000000
6	5	5	5	5	6

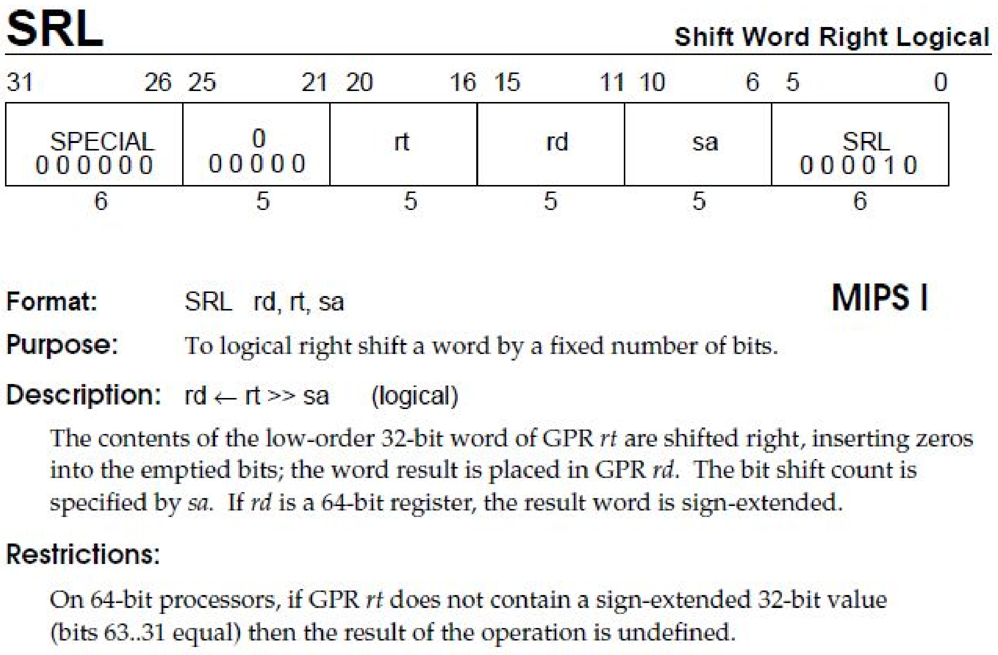
Format: SLL rd, rt, sa **MIPS I**

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 zeroes into the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by *sa*. If *rd* is a 64-bit register, the result word is sign-extended.

2. SRL: Shift Word Right Logical.



SRL

Shift Word Right Logical

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRL 000010
6	5	5	5	5	6

Format: SRL rd, rt, sa **MIPS I**

Purpose: To logical right shift a word by a fixed number of bits.

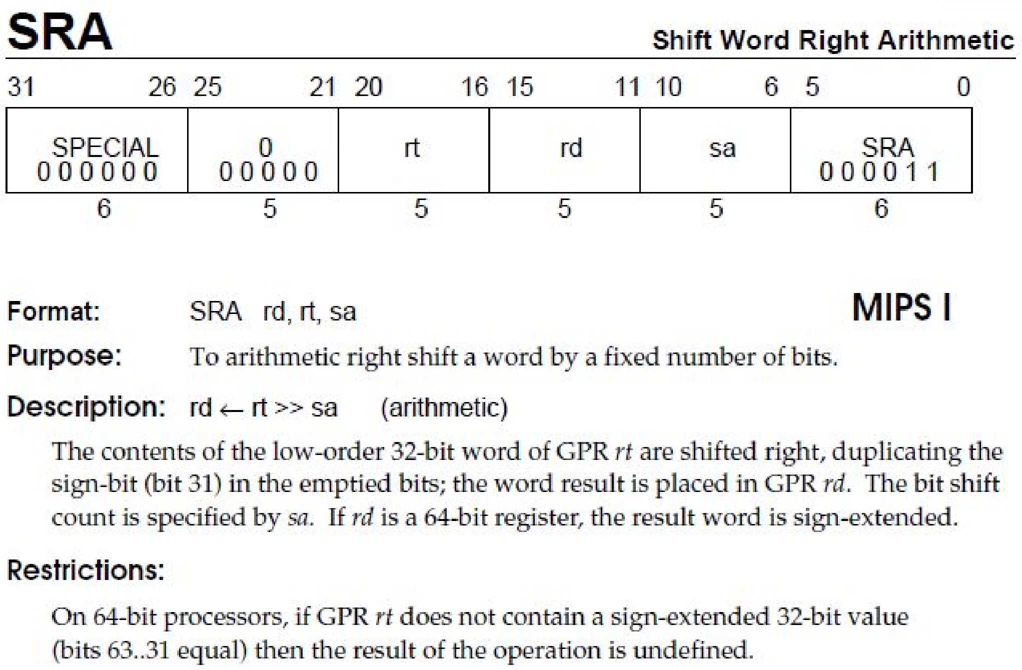
Description: rd ← rt >> sa (logical)

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 count is specified by *sa*. If *rd* is a 64-bit register, the result word is sign-extended.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

3. SRA: Shift Word Right Arithmetic.



SRA

Shift Word Right Arithmetic

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRA 000011
6	5	5	5	5	6

Format: SRA rd, rt, sa **MIPS I**

Purpose: To arithmetic right shift a word by a fixed number of bits.

Description: rd ← rt >> sa (arithmetic)

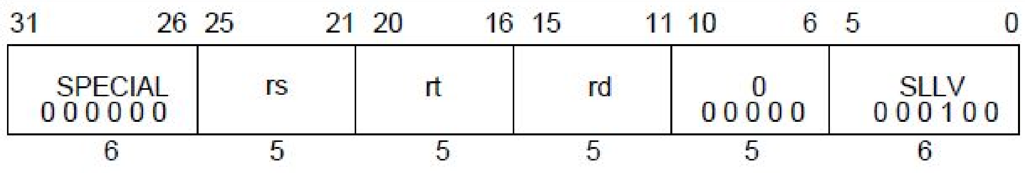
The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit shift count is specified by *sa*. If *rd* is a 64-bit register, the result word is sign-extended.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

4. SLLV: Shift Word Left Logical Variable.

Shift Word Left Logical Variable **SLLV**



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SLLV 000100
6	5	5	5	5	6

Format: SLLV rd, rt, rs **MIPS I**

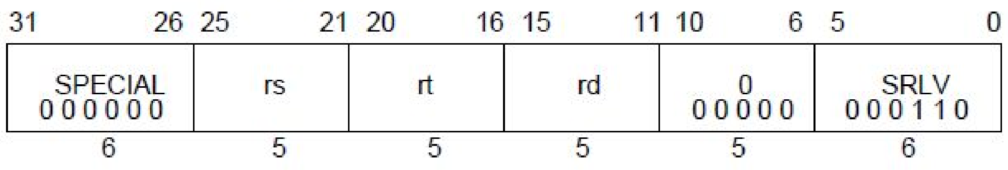
Purpose: To left shift a word by a variable number of bits.

Description: rd ← rt << rs

The contents of the low-order 32-bit word of GPR *rt* are shifted left, inserting zeroes into the emptied bits; the result word is placed in GPR *rd*. The bit shift count is specified by the low-order five bits of GPR *rs*. If *rd* is a 64-bit register, the result word is sign-extended.

5. SRLV: Shift Word Right Logical Variable.

Shift Word Right Logical Variable **SRLV**



31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	SRLV 000110
6	5	5	5	5	6

Format: SRLV rd, rt, rs **MIPS I**

Purpose: To logical right shift 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 count is specified by the low-order five bits of GPR *rs*. If *rd* is a 64-bit register, the result word is sign-extended.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

6. SRAV: Shift Word Right Arithmetic Variable.

Shift Word Right Arithmetic Variable **SRAV**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SRAV 000111
6	5	5	5	5	6

Format: SRAV rd, rt, rs **MIPS I**

Purpose: To arithmetic right shift 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 count is specified by the low-order five bits of GPR *rs*. If *rd* is a 64-bit register, the result word is sign-extended.

Restrictions:

On 64-bit processors, if GPR *rt* does not contain a sign-extended 32-bit value (bits 63..31 equal) then the result of the operation is undefined.

7. ADDU: Add Unsigned Word.

Add Unsigned Word **ADDU**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001
6	5	5	5	5	6

Format: ADDU rd, rs, rt **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:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp    ←GPR[rs] + GPR[rt]
GPR[rd]← sign_extend(temp$_{31..0}$)
```

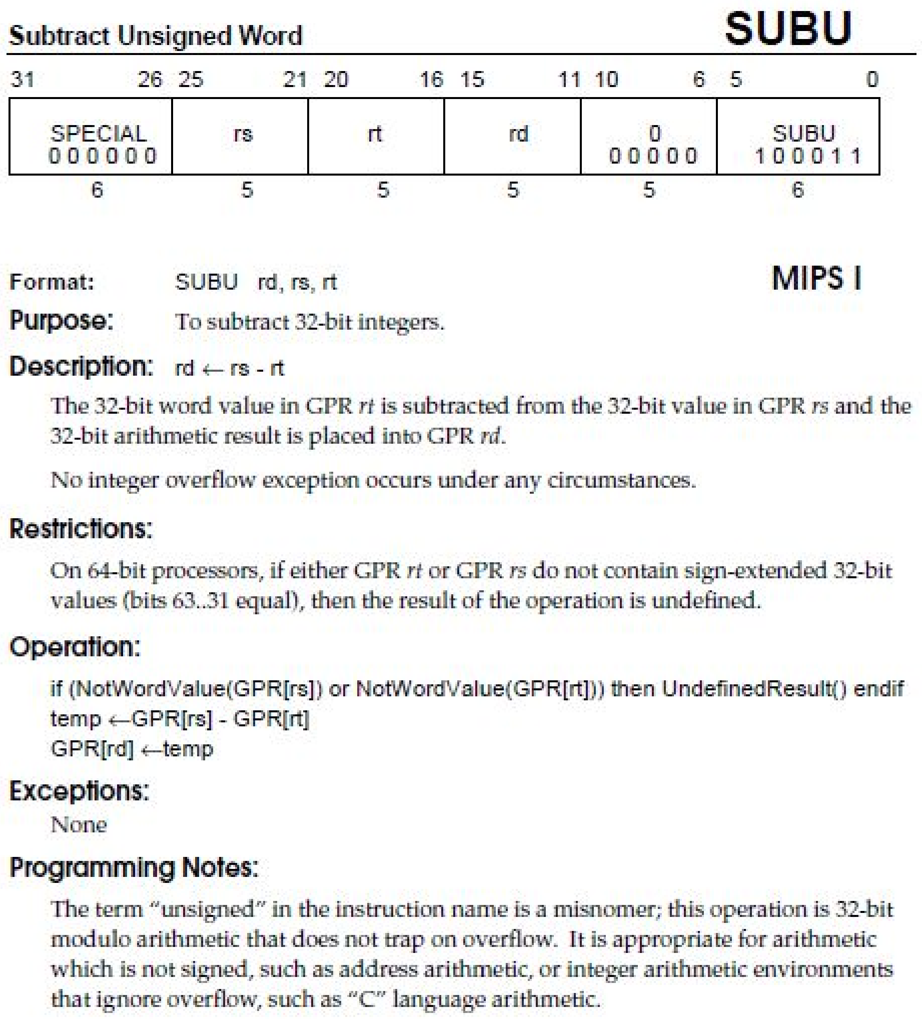
Exceptions:

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 arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as “C” language arithmetic.

8. SUBU: Subtract Unsigned Word.



Subtract Unsigned Word **SUBU**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUBU 100011
6	5	5	5	5	6

Format: SUBU rd, rs, rt **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 placed into GPR *rd*.

No integer overflow exception occurs under any circumstances.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp ←GPR[rs] - GPR[rt]
GPR[rd] ←temp
```

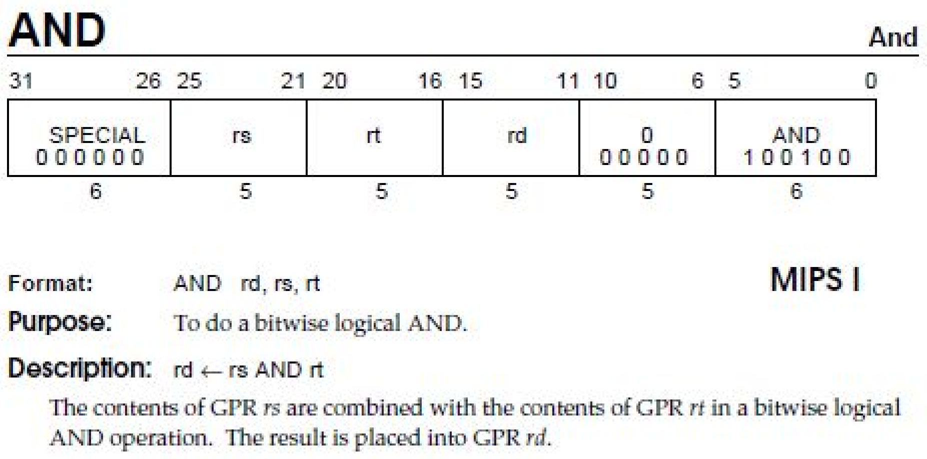
Exceptions:

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 arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as "C" language arithmetic.

9. AND: and.



AND **And**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	AND 100100
6	5	5	5	5	6

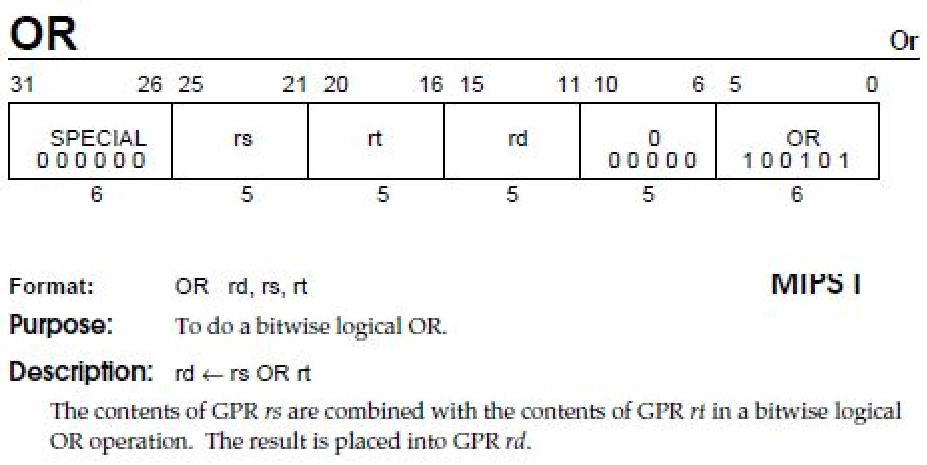
Format: AND rd, rs, rt **MIPS I**

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*.

10. OR: or.



OR Or

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101
6	5	5	5	5	6

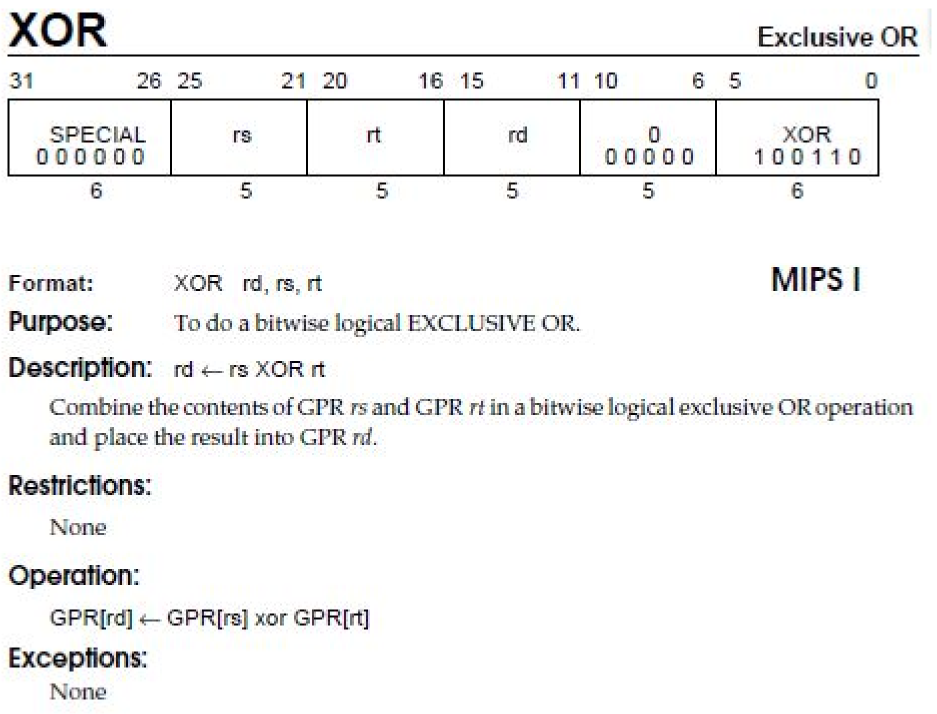
Format: OR rd, rs, rt **MIPS I**

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*.

11. XOR: exclusive or.



XOR Exclusive OR

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110
6	5	5	5	5	6

Format: XOR rd, rs, rt **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

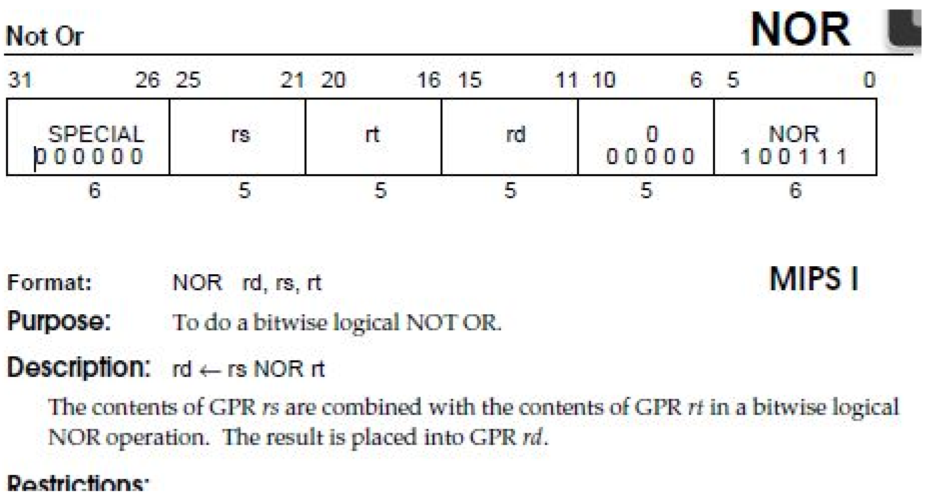
Operation:

GPR[rd] ← GPR[rs] xor GPR[rt]

Exceptions:

None

12. NOR: nor.



Not Or **NOR**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111
6	5	5	5	5	6

Format: NOR rd, rs, rt **MIPS I**

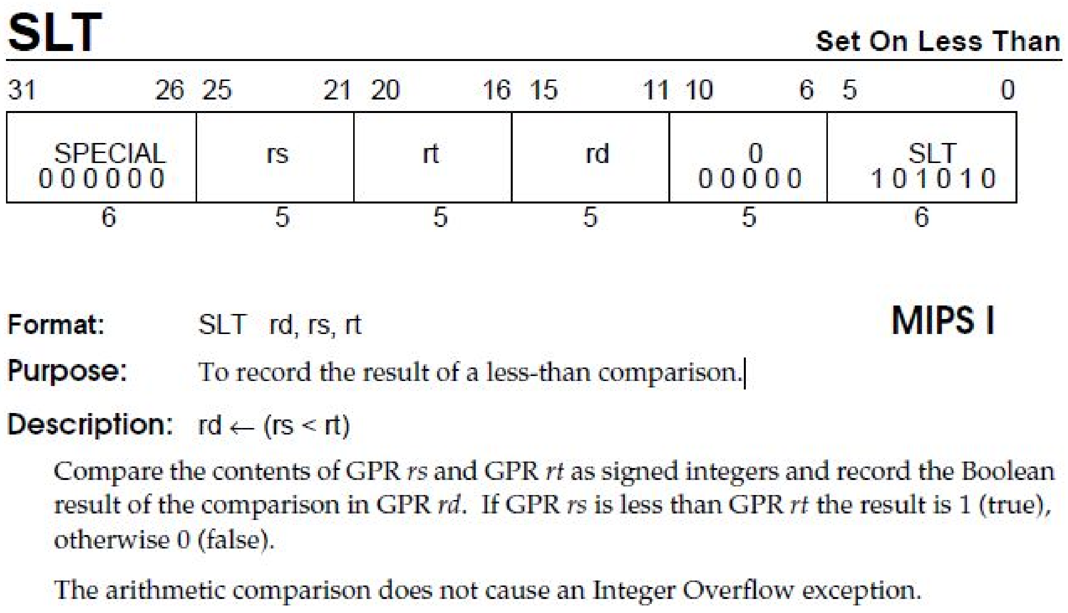
Purpose: To do a bitwise logical NOT OR.

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:

13. SLT: Set on Less Than.



SLT **Set On Less Than**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010
6	5	5	5	5	6

Format: SLT rd, rs, rt **MIPS I**

Purpose: To record the result of a less-than comparison.

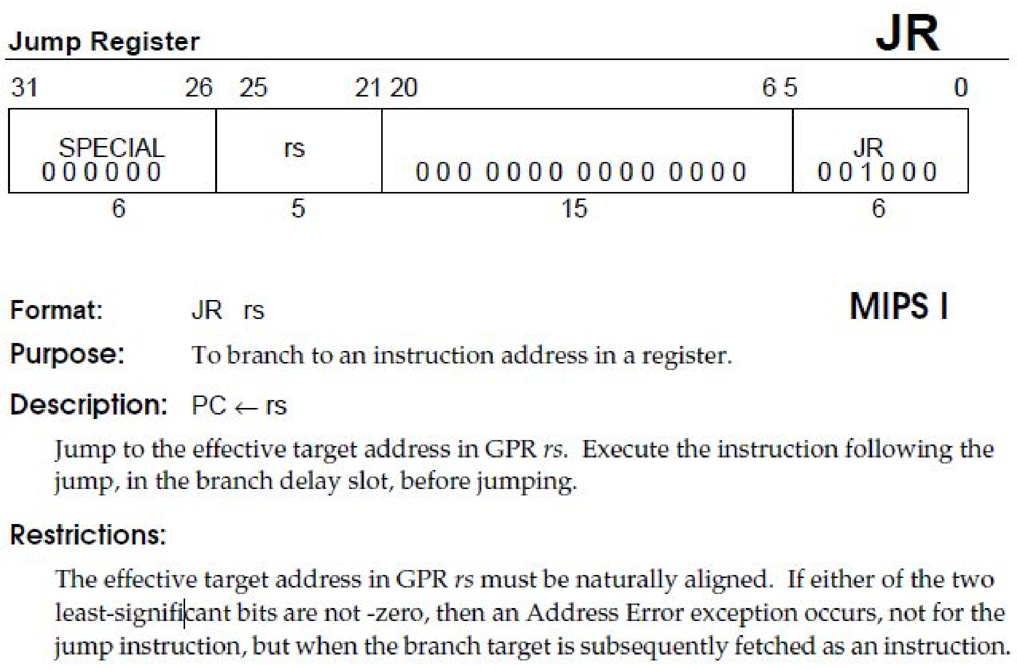
Description: rd ← (rs < rt)

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 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Descripción de instrucciones J-Type:

1. JR:



Jump Register **JR**

31–26	25–21	20–6	5–0
SPECIAL 000000	rs	000 0000 0000 0000	JR 001000
6	5	15	6

Format: JR rs **MIPS I**

Purpose: To branch to an instruction address in a register.

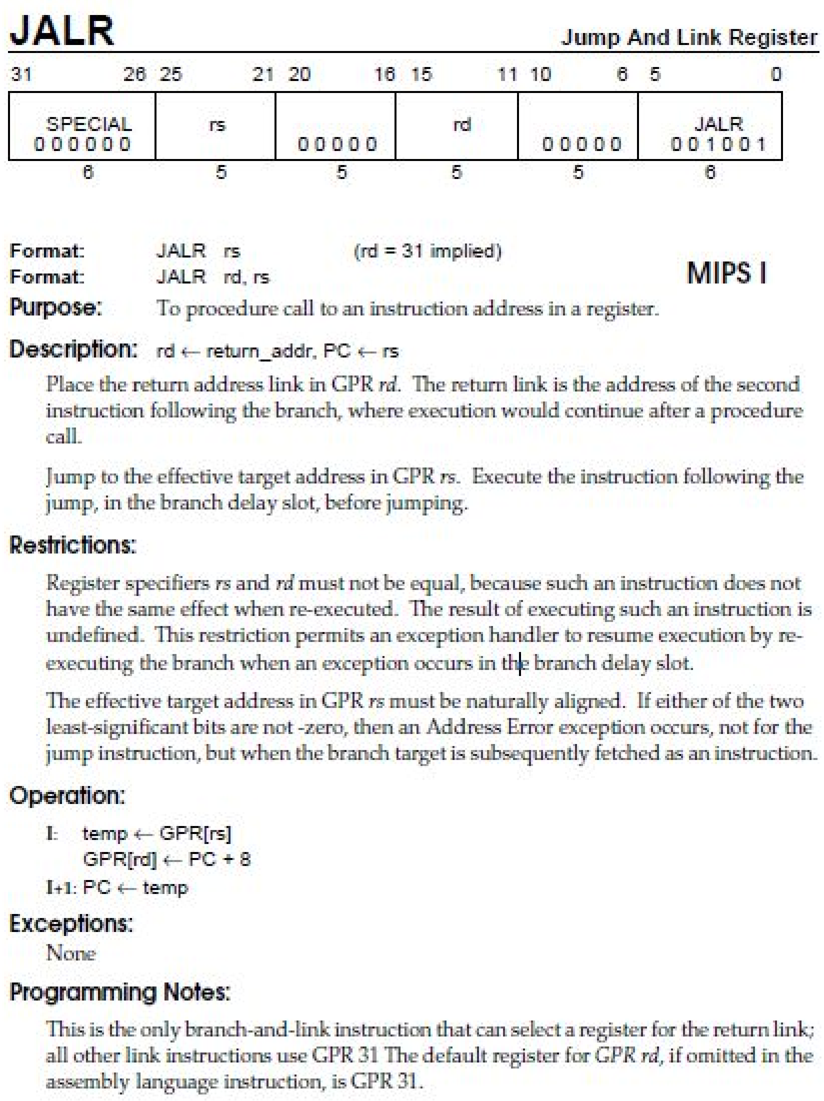
Description: PC ← rs

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

Restrictions:

The effective target address in GPR *rs* must be naturally aligned. If either of the two least-significant bits are not -zero, then an Address Error exception occurs, not for the jump instruction, but when the branch target is subsequently fetched as an instruction.

2. JALR:



JALR **Jump And Link Register**

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	00000	rd	00000	JALR 001001
6	5	5	5	5	6

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

Purpose: To 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 would continue after a procedure call.

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

Restrictions:

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

The effective target address in GPR *rs* must be naturally aligned. If either of the two least-significant bits are not -zero, then an Address Error exception occurs, not for the jump instruction, but when the branch target is subsequently fetched as an instruction.

Operation:

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

Exceptions:

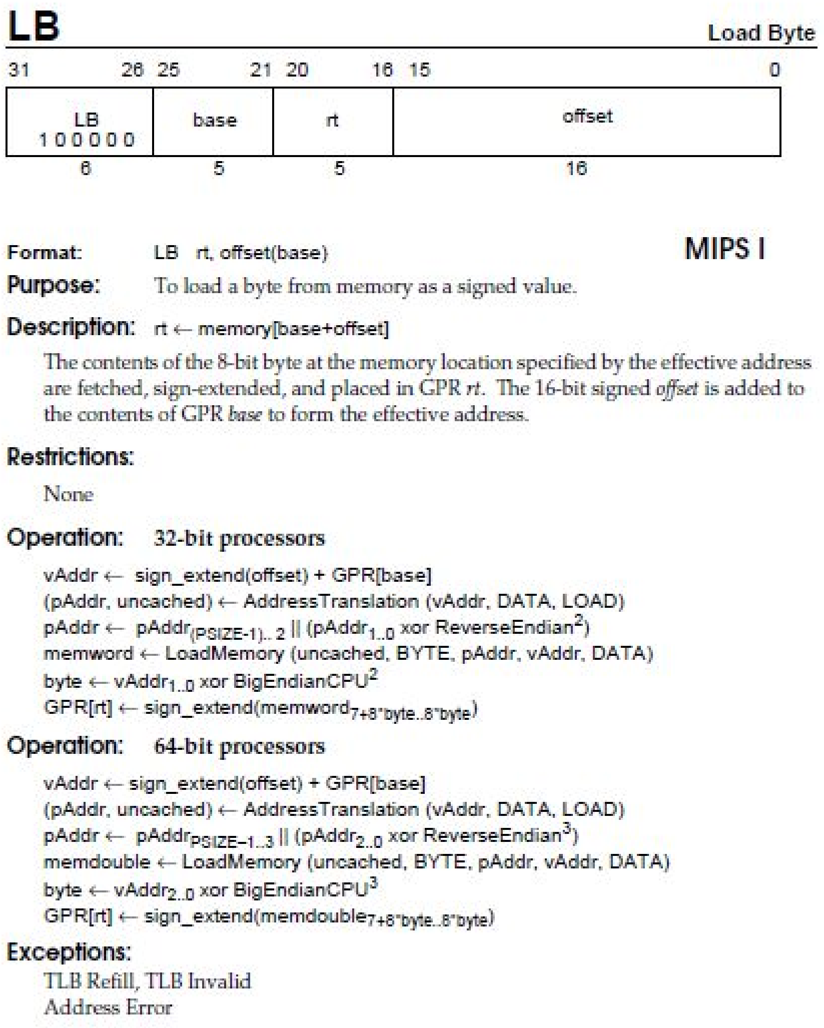
None

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.

Descripción de instrucciones I-Type:

1. LB: Load byte.



LB

Load Byte

31 26	25 21	20 16	15 0
LB 100000	base	rt	offset
6	5	5	16

Format: LB rt, offset(base) **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: **32-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $\text{pAddr}_{(\text{PSIZE-1})..2}$ || ($\text{pAddr}_{1..0}$ xor $\text{ReverseEndian}^2$)
memword ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← $\text{vAddr}_{1..0}$ xor $\text{BigEndianCPU}^2$
GPR[rt] ← sign_extend($\text{memword}_{7+8*\text{byte}..8*\text{byte}}$)
```

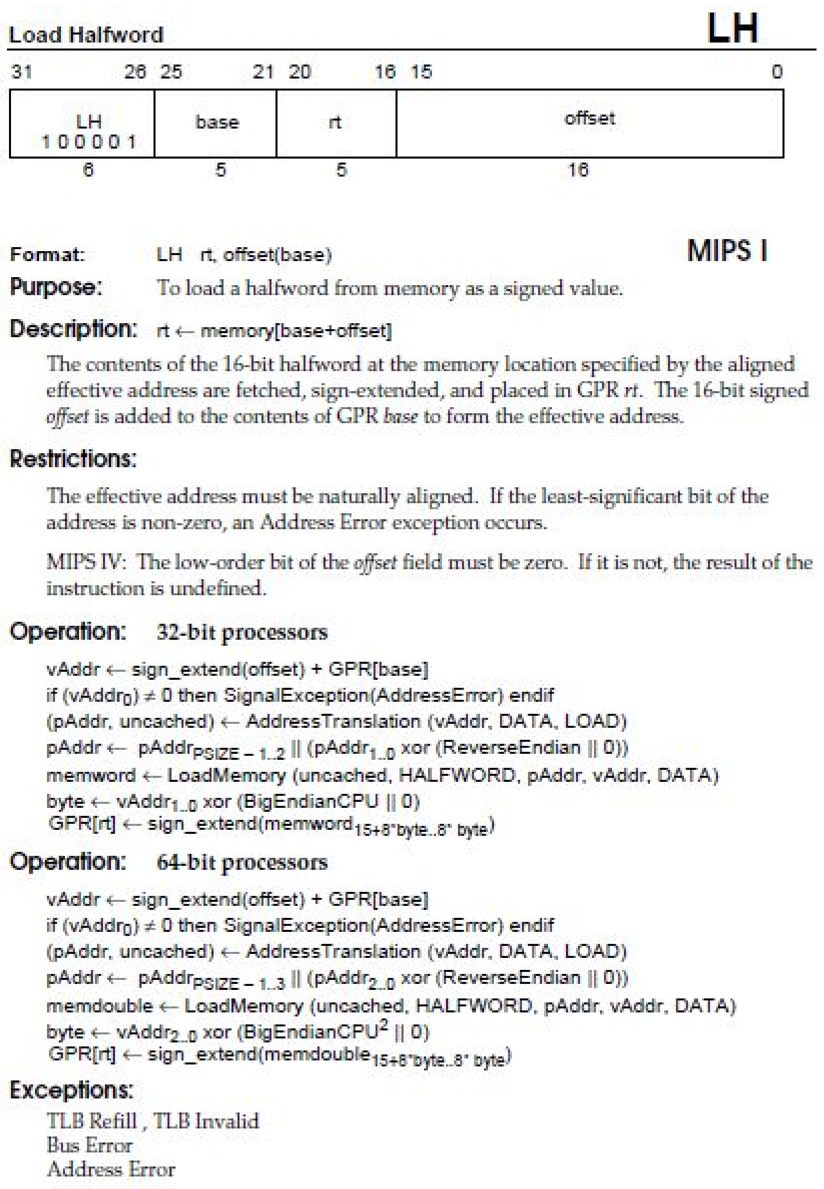
Operation: **64-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3}$ || ($\text{pAddr}_{2..0}$ xor $\text{ReverseEndian}^3$)
memdouble ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← $\text{vAddr}_{2..0}$ xor $\text{BigEndianCPU}^3$
GPR[rt] ← sign_extend($\text{memdouble}_{7+8*\text{byte}..8*\text{byte}}$)
```

Exceptions:

TLB Refill, TLB Invalid
Address Error

2. LH: Load Halfword.



Load Halfword **LH**

31 26	25 21	20 16	15 0
LH 100001	base	rt	offset
6	5	5	16

Format: LH rt, offset(base) **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.

MIPS IV: The low-order bit of the *offset* field must be zero. If it is not, the result of the instruction is undefined.

Operation: 32-bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_0$) ≠ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor (ReverseEndian || 0))
memword ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
byte ← $vAddr_{1..0}$ xor (BigEndianCPU || 0)
GPR[rt] ← sign_extend($memword_{15+8*byte..8*byte}$)
```

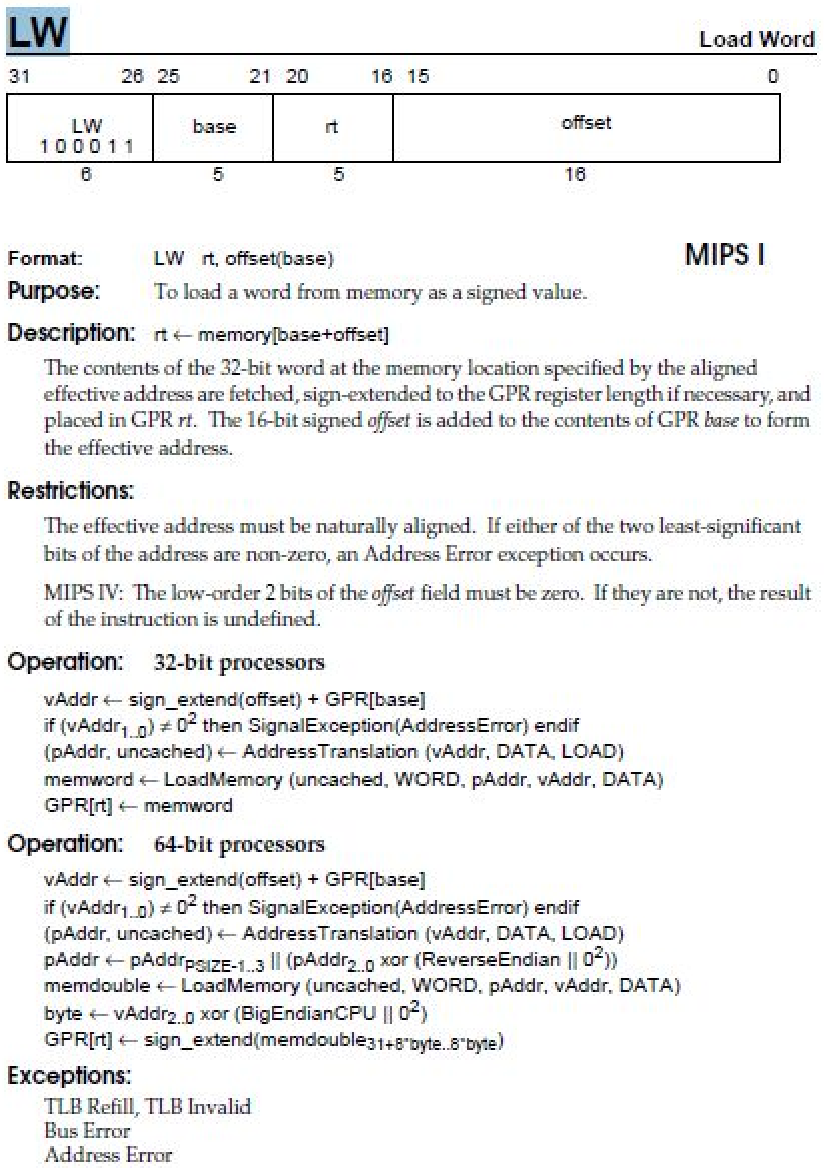
Operation: 64-bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_0$) ≠ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || 0))
memdouble ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
byte ← $vAddr_{2..0}$ xor ($BigEndianCPU^2$ || 0)
GPR[rt] ← sign_extend($memdouble_{15+8*byte..8*byte}$)
```

Exceptions:

TLB Refill , TLB Invalid
Bus Error
Address Error

3. LW: Load Word.



LW

Load Word

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset
6	5	5	16

Format: LW rt, offset(base) **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 two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: **32-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ $0^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
```

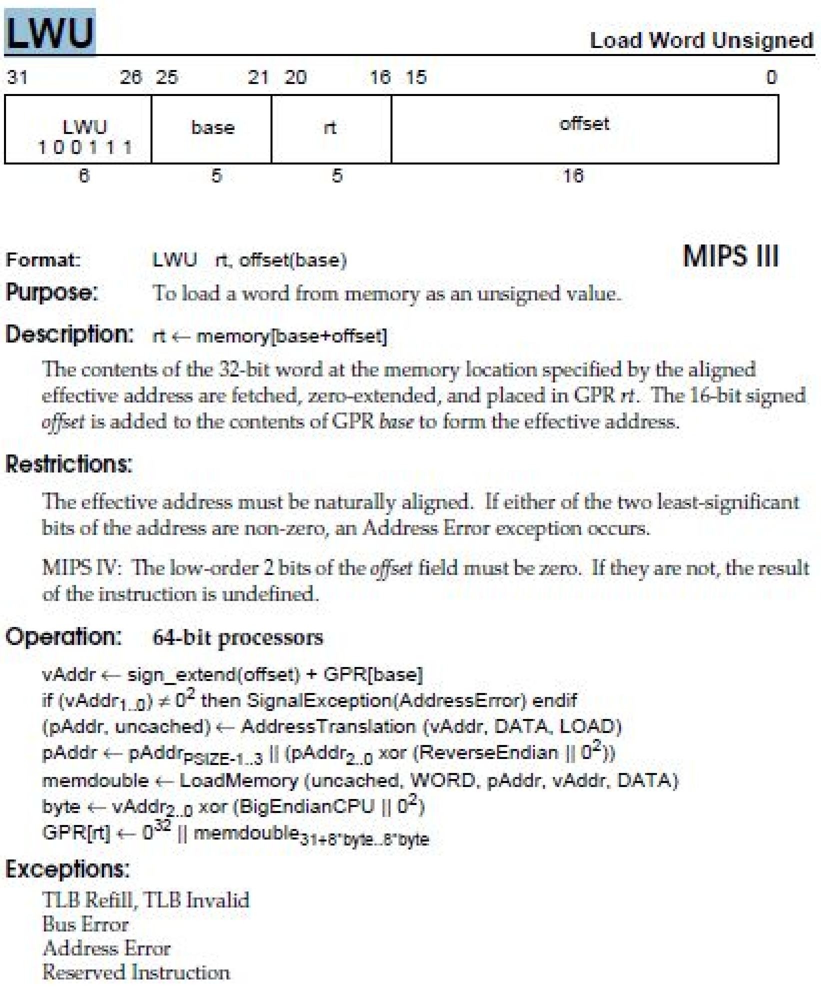
Operation: **64-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ $0^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
memdouble ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
byte ← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
GPR[rt] ← sign_extend($memdouble_{31+8*byte..8*byte}$)
```

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error

4. LWU: Load Word unsigned.



LWU

Load Word Unsigned

31–26	25–21	20–16	15–0
LWU 100111	base	rt	offset
6	5	5	16

Format: LWU rt, offset(base) **MIPS III**

Purpose: To load a word from memory as an unsigned 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, 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 either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

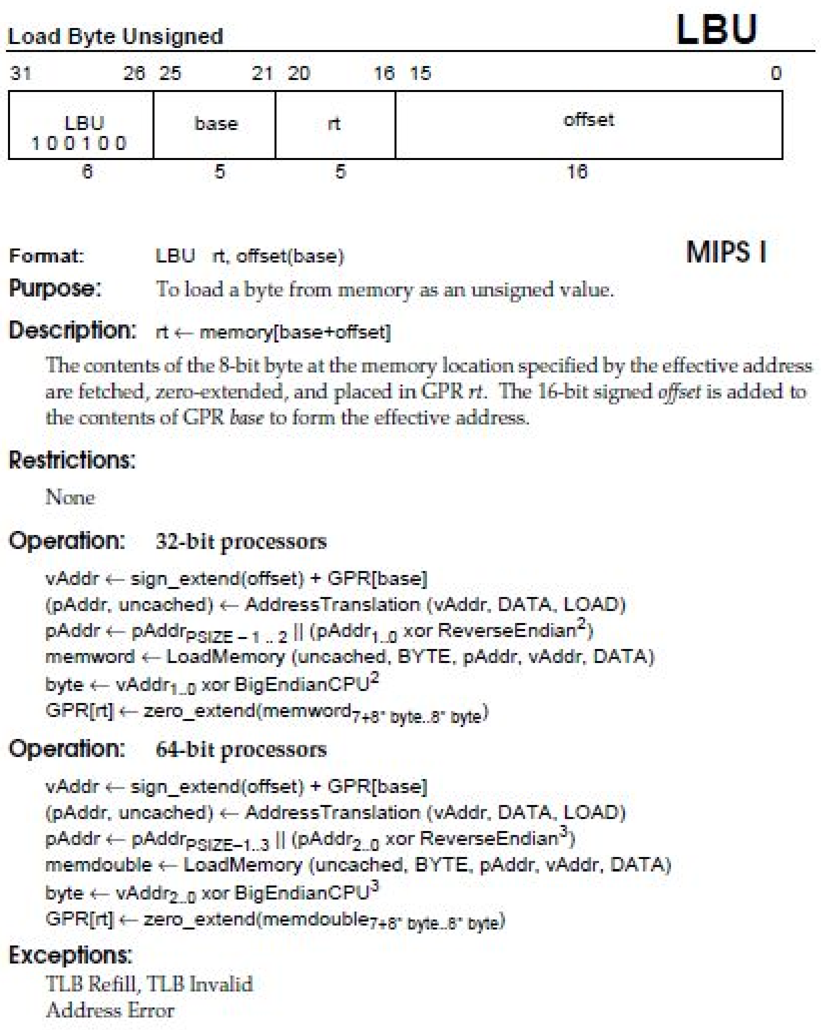
Operation: **64-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ $0^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
memdouble ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
byte ← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
GPR[rt] ← $0^{32}$ || $memdouble_{31+8*byte..8*byte}$
```

Exceptions:

TLB Refill, TLB Invalid
Bus Error
Address Error
Reserved Instruction

5. LBU: Load byte unsigned.



Load Byte Unsigned **LBU**

31 26	25 21	20 16	15 0
LBU 100100	base	rt	offset
6	5	5	16

Format: LBU rt, offset(base) **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: **32-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{PSIZE-1..2}$ || (pAddr$_{1..0}$ xor ReverseEndian$^2$)
memword ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← vAddr$_{1..0}$ xor BigEndianCPU$^2$
GPR[rt] ← zero_extend(memword$_{7+8*byte..8*byte}$)
```

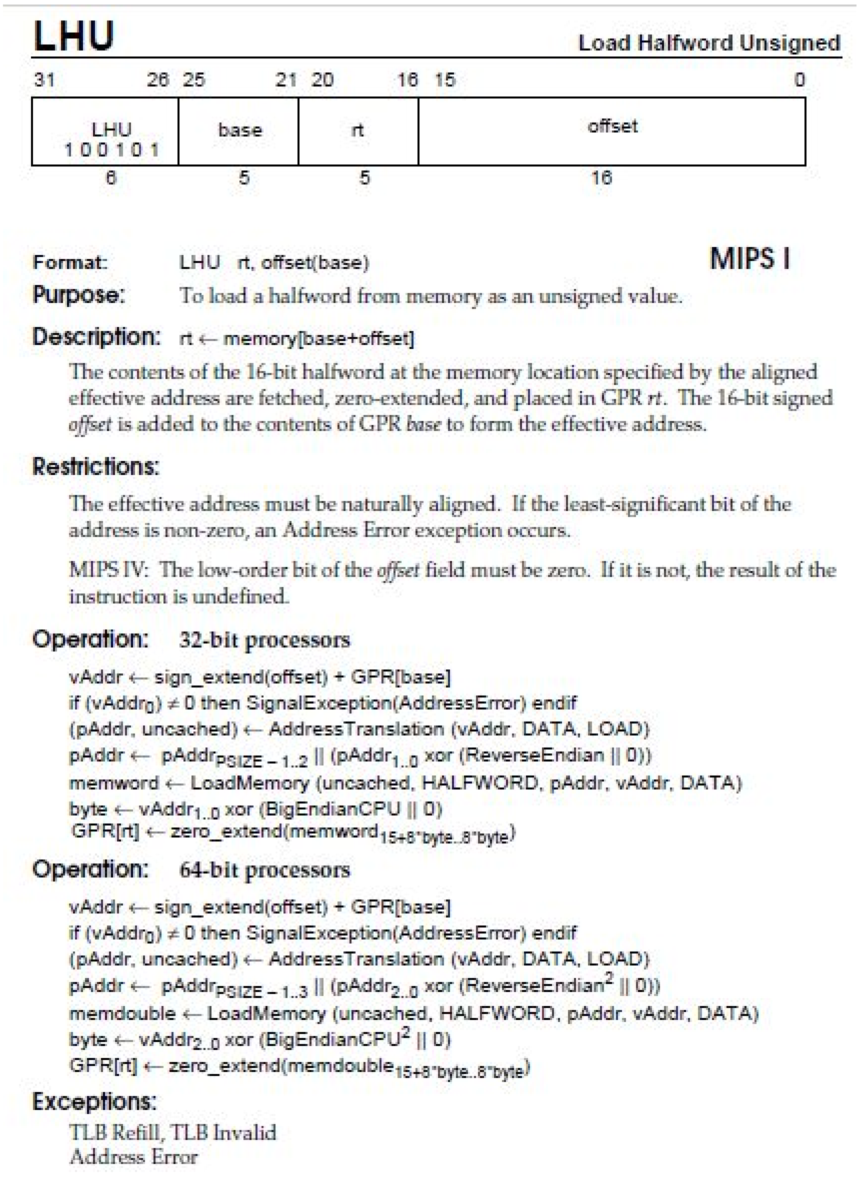
Operation: **64-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^3$)
memdouble ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
byte ← vAddr$_{2..0}$ xor BigEndianCPU$^3$
GPR[rt] ← zero_extend(memdouble$_{7+8*byte..8*byte}$)
```

Exceptions:

TLB Refill, TLB Invalid
Address Error

6. LHU: Load Halfword Unsigned.



LHU — Load Halfword Unsigned

31–26	25–21	20–16	15–0
LHU 100101	base	rt	offset
6	5	5	16

Format: LHU rt, offset(base) **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.

MIPS IV: The low-order bit of the *offset* field must be zero. If it is not, the result of the instruction is undefined.

Operation: **32-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_0$) ≠ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor (ReverseEndian || 0))
memword ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
byte ← $vAddr_{1..0}$ xor (BigEndianCPU || 0)
GPR[rt] ← zero_extend($memword_{15+8*byte..8*byte}$)
```

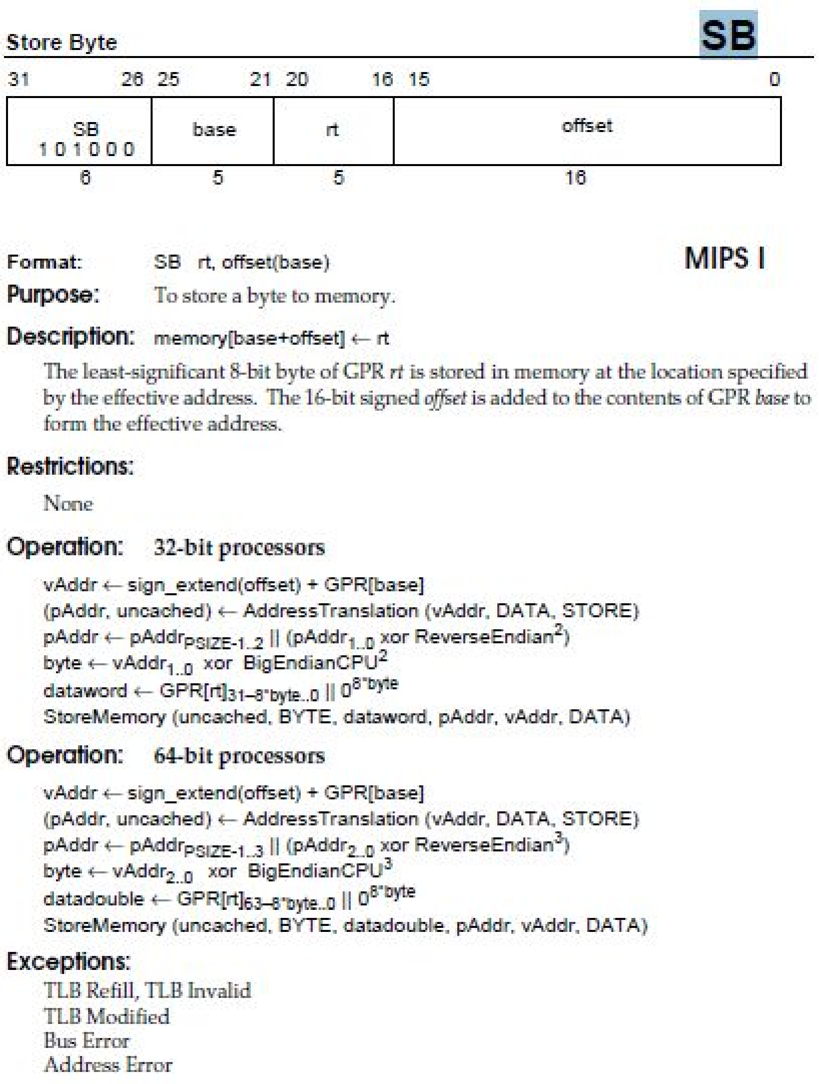
Operation: **64-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_0$) ≠ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor ($ReverseEndian^2$ || 0))
memdouble ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
byte ← $vAddr_{2..0}$ xor ($BigEndianCPU^2$ || 0)
GPR[rt] ← zero_extend($memdouble_{15+8*byte..8*byte}$)
```

Exceptions:

TLB Refill, TLB Invalid
Address Error

7. SB: Store byte.



Store Byte **SB**

31 … 26	25 … 21	20 … 16	15 … 0
SB 101000	base	rt	offset
6	5	5	16

Format: SB rt, offset(base) **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: 32-bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor $ReverseEndian^2$)
byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
dataword ← $GPR[rt]_{31-8*byte..0}$ || $0^{8*byte}$
StoreMemory (uncached, BYTE, dataword, pAddr, vAddr, DATA)
```

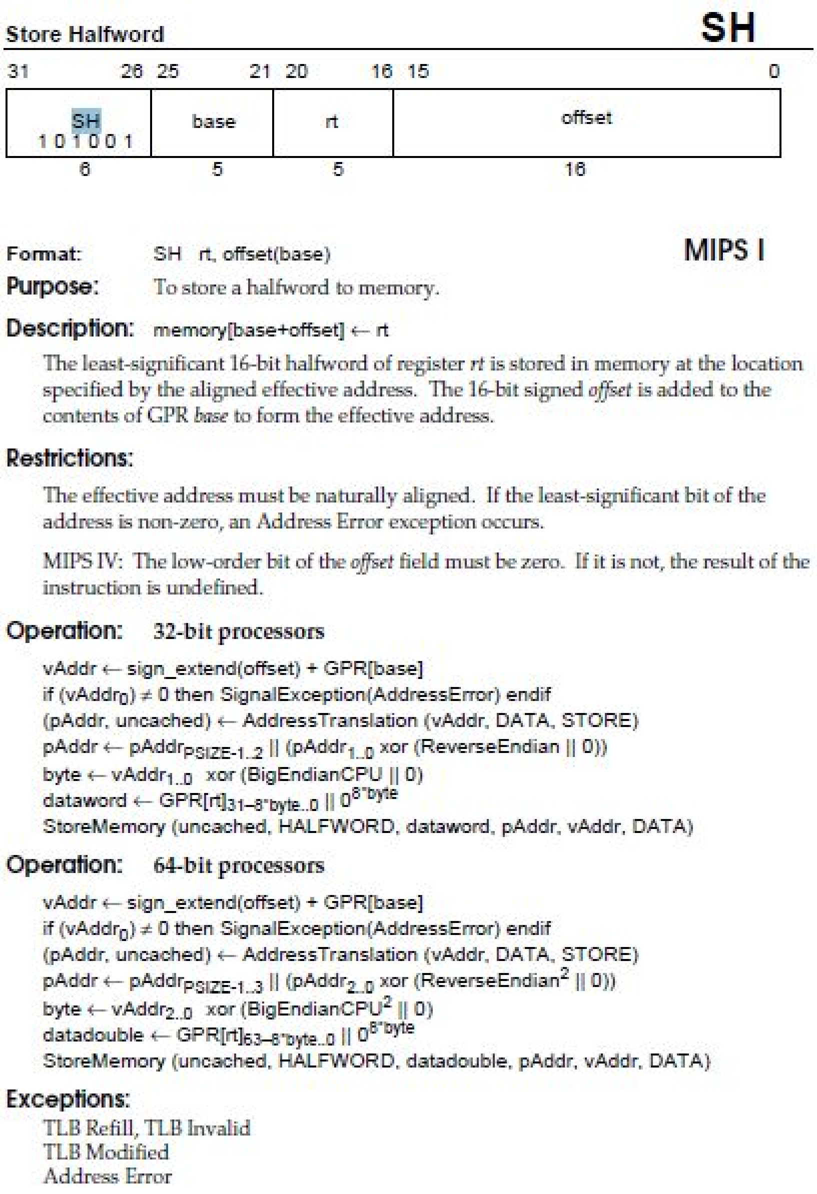
Operation: 64-bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor $ReverseEndian^3$)
byte ← $vAddr_{2..0}$ xor $BigEndianCPU^3$
datadouble ← $GPR[rt]_{63-8*byte..0}$ || $0^{8*byte}$
StoreMemory (uncached, BYTE, datadouble, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Bus Error
Address Error

8. SH: Store HalfWord.



Store Halfword **SH**

31 26	25 21	20 16	15 0
SH 1 0 1 0 0 1	base	rt	offset
6	5	5	16

Format: SH rt, offset(base) **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.

MIPS IV: The low-order bit of the *offset* field must be zero. If it is not, the result of the instruction is undefined.

Operation: **32-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
if (vAddr$_0$) $\neq$ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr$_{PSIZE-1..2}$ || (pAddr$_{1..0}$ xor (ReverseEndian || 0))
byte ← vAddr$_{1..0}$ xor (BigEndianCPU || 0)
dataword ← GPR[rt]$_{31-8*byte..0}$ || $0^{8*byte}$
StoreMemory (uncached, HALFWORD, dataword, pAddr, vAddr, DATA)
```

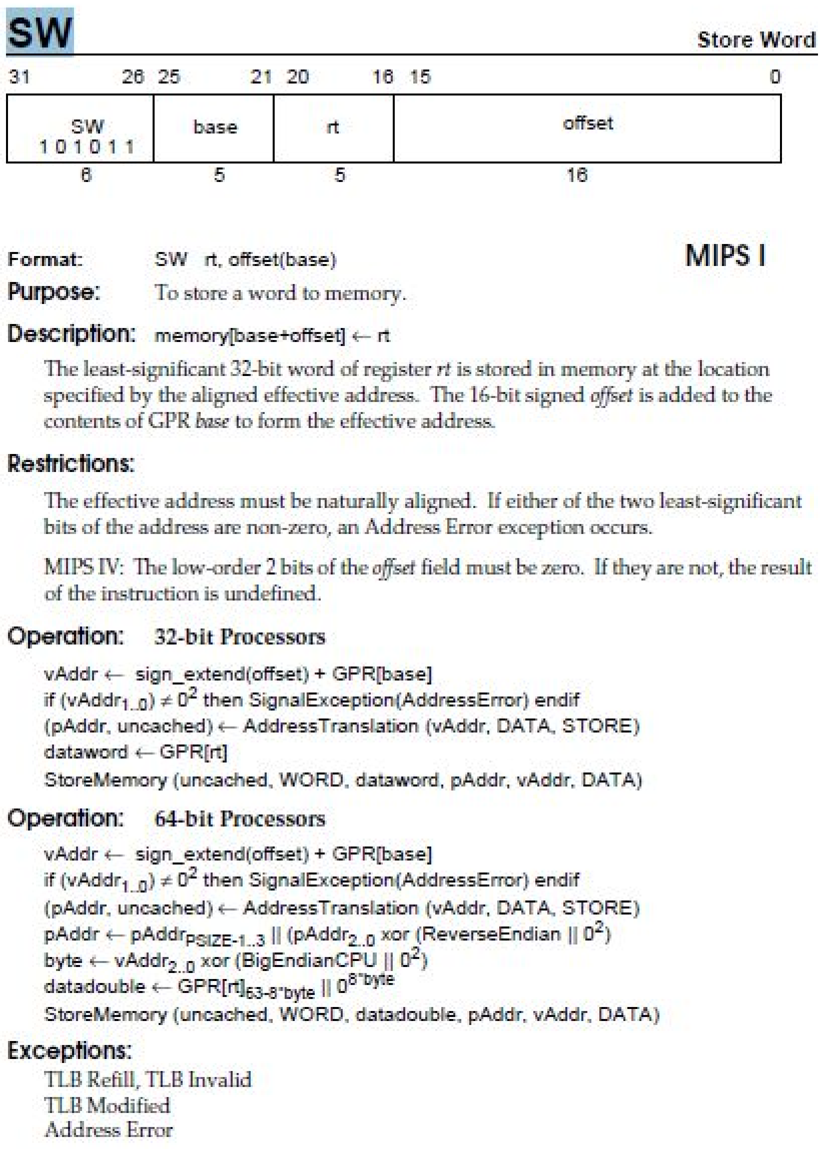
Operation: **64-bit processors**

```
vAddr ← sign_extend(offset) + GPR[base]
if (vAddr$_0$) $\neq$ 0 then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian$^2$ || 0))
byte ← vAddr$_{2..0}$ xor (BigEndianCPU$^2$ || 0)
datadouble ← GPR[rt]$_{63-8*byte..0}$ || $0^{8*byte}$
StoreMemory (uncached, HALFWORD, datadouble, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error

9. SW: Store Word.



SW

Store Word

31 26	25 21	20 16	15 0
SW 101011	base	rt	offset
6	5	5	16

Format: SW rt, offset(base) **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 two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit Processors

```
vAddr ← sign_extend(offset) + GPR[base]
if (vAddr$_{1..0}$) ≠ 0$^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
StoreMemory (uncached, WORD, dataword, pAddr, vAddr, DATA)
```

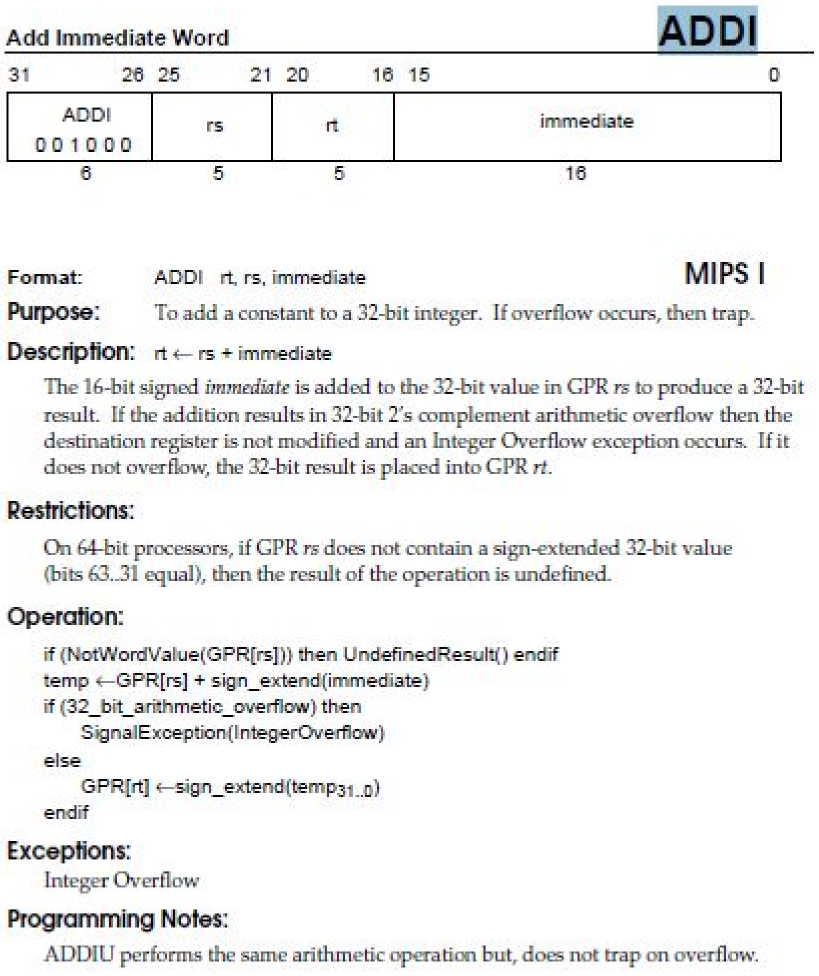
Operation: 64-bit Processors

```
vAddr ← sign_extend(offset) + GPR[base]
if (vAddr$_{1..0}$) ≠ 0$^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian || 0$^2$)
byte ← vAddr$_{2..0}$ xor (BigEndianCPU || 0$^2$)
datadouble ← GPR[rt]$_{63-8*byte}$ || 0$^{8*byte}$
StoreMemory (uncached, WORD, datadouble, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error

10. ADDI: Add Immediate Word.



Add Immediate Word **ADDI**

31–26	25–21	20–16	15–0
ADDI 001000	rs	rt	immediate
6	5	5	16

Format: ADDI rt, rs, immediate **MIPS I**

Purpose: To add a constant to a 32-bit integer. If overflow occurs, then trap.

Description: rt ← rs + immediate

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rt*.

Restrictions:

On 64-bit processors, if GPR *rs* does not contain a sign-extended 32-bit value (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs])) then UndefinedResult() endif
temp ←GPR[rs] + sign_extend(immediate)
if (32_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rt] ←sign_extend(temp_{31..0})
endif
```

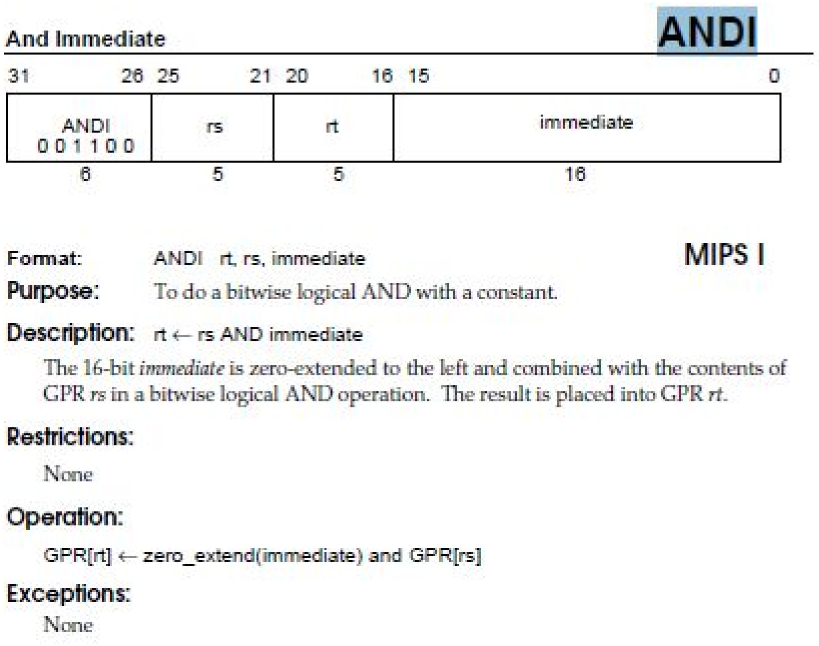
Exceptions:

Integer Overflow

Programming Notes:

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

11. ANDI: And Immediate.



And Immediate **ANDI**

31–26	25–21	20–16	15–0
ANDI 001100	rs	rt	immediate
6	5	5	16

Format: ANDI rt, rs, immediate **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

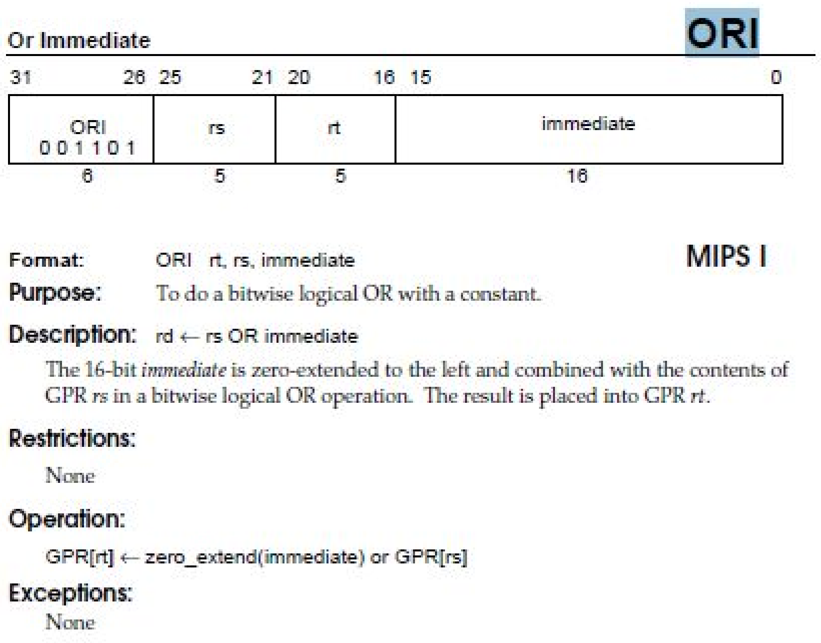
Operation:

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

Exceptions:

None

12. ORI: Or Immediate.



Or Immediate **ORI**

31 … 26	25 … 21	20 … 16	15 … 0
ORI 001101	rs	rt	immediate
6	5	5	16

Format: ORI rt, rs, immediate **MIPS I**

Purpose: To do a bitwise logical OR with a constant.

Description: rd ← 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

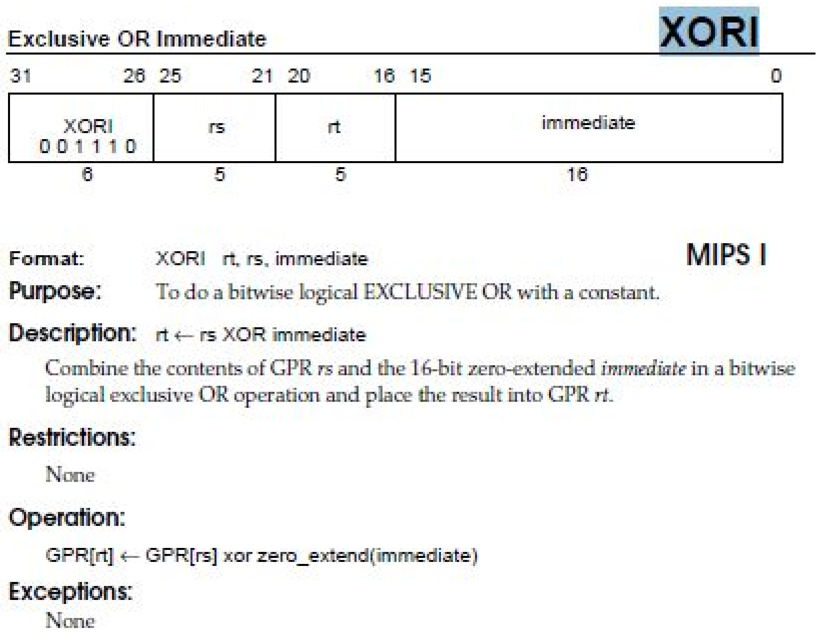
Operation:

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

Exceptions:

None

13. XORI: Exclusive Or Immediate.



Exclusive OR Immediate **XORI**

31 … 26	25 … 21	20 … 16	15 … 0
XORI 001110	rs	rt	immediate
6	5	5	16

Format: XORI rt, rs, immediate **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

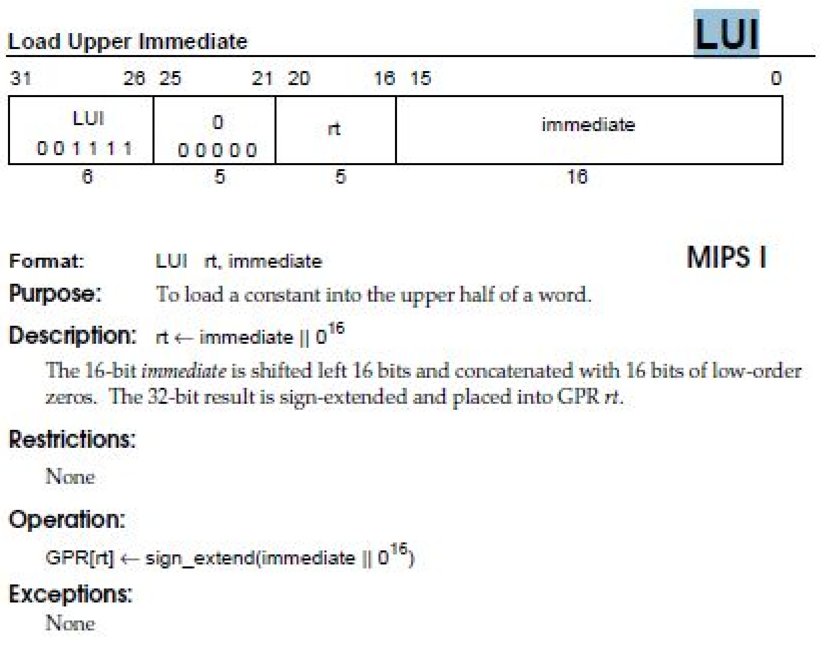
Operation:

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

Exceptions:

None

14. LUI: Load Upper Immediate.



Load Upper Immediate **LUI**

31 26	25 21	20 16	15 0
LUI 001111	0 00000	rt	immediate
6	5	5	16

Format: LUI rt, immediate **MIPS I**

Purpose: To load a constant into the upper half of a word.

Description: $rt \leftarrow immediate \,||\, 0^{16}$

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

Restrictions:

None

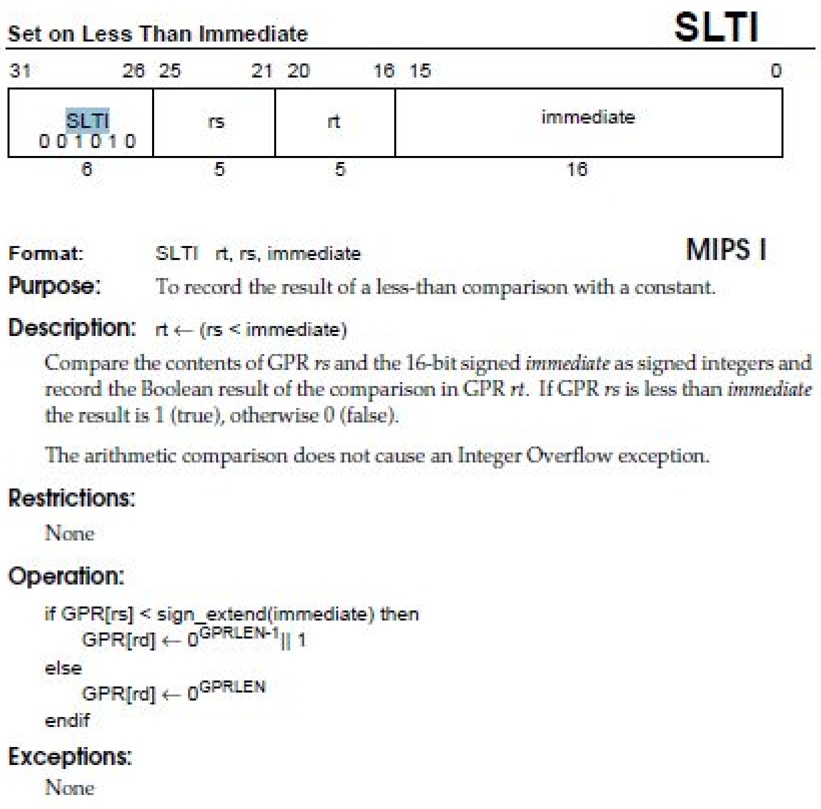
Operation:

```
GPR[rt] ← sign_extend(immediate || 0^16)
```

Exceptions:

None

15. SLTI: Set on Less Than Immediate.



Set on Less Than Immediate **SLTI**

31 26	25 21	20 16	15 0
SLTI 001010	rs	rt	immediate
6	5	5	16

Format: SLTI rt, rs, immediate **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 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

Restrictions:

None

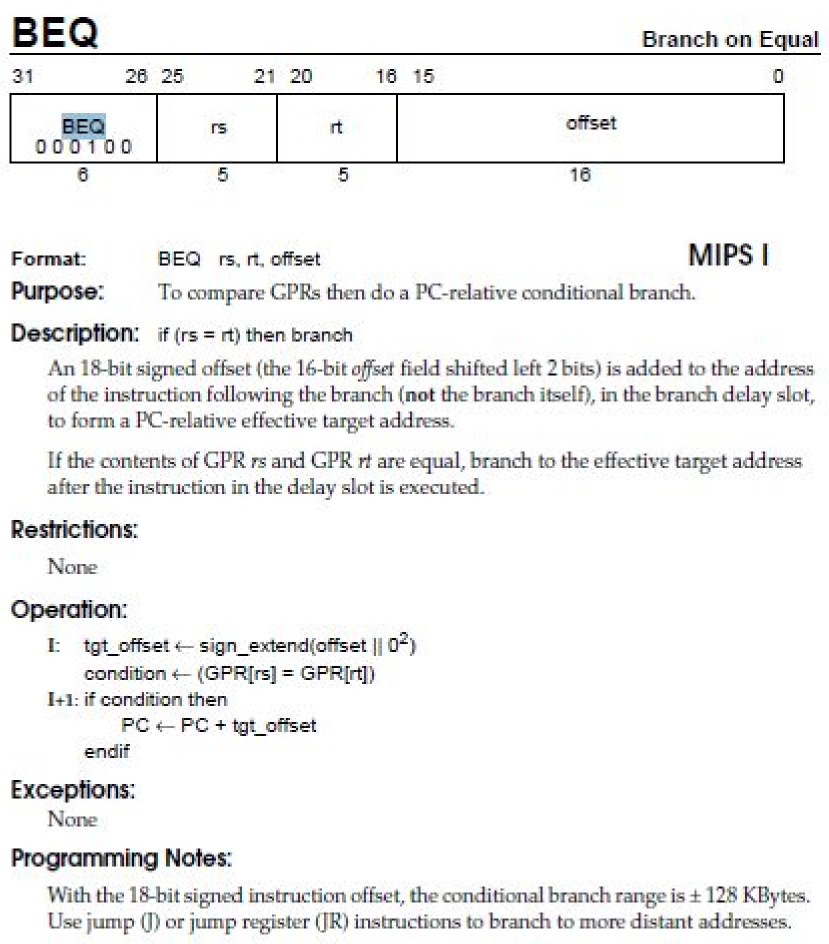
Operation:

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

Exceptions:

None

16. BEQ: Branch on Equal.



BEQ

Branch on Equal

31 26	25 21	20 16	15 0
BEQ 0 0 0 1 0 0	rs	rt	offset
6	5	5	16

Format: BEQ rs, rt, offset **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:

None

Operation:

```
I:   tgt_offset ← sign_extend(offset || $0^2$)
     condition ← (GPR[rs] = GPR[rt])
I+1: if condition then
         PC ← PC + tgt_offset
     endif
```

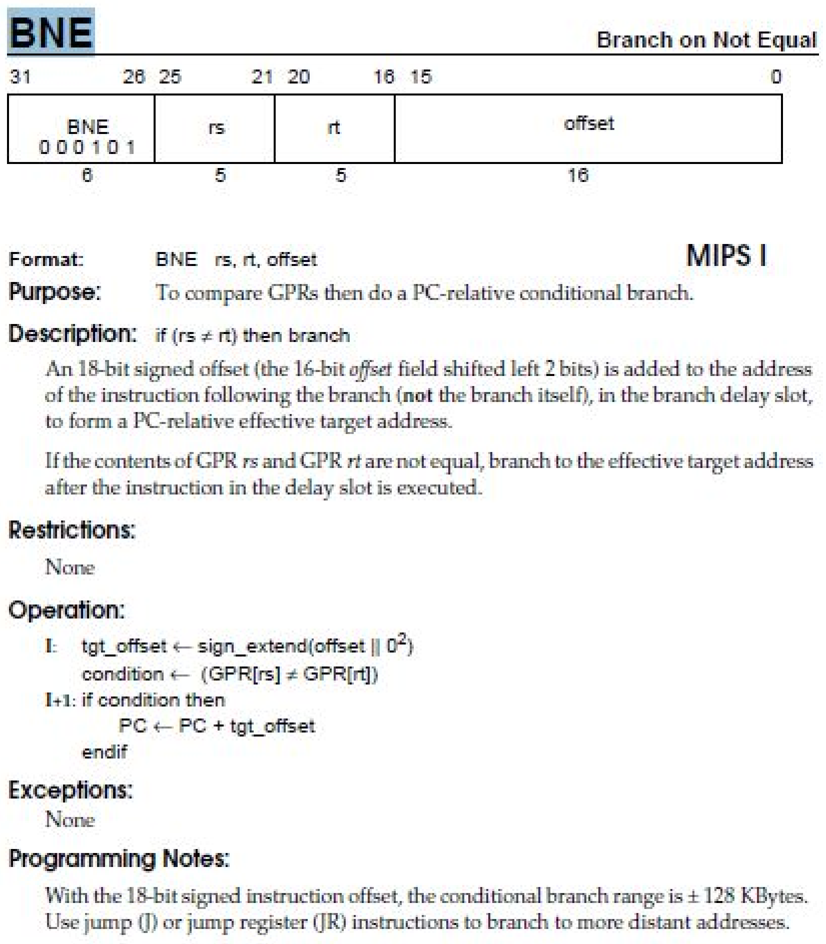
Exceptions:

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 more distant addresses.

17. BNE: Branch on Not Equal.



BNE

Branch on Not Equal

31 26	25 21	20 16	15 0
BNE 000101	rs	rt	offset
6	5	5	16

Format: BNE rs, rt, offset **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:

None

Operation:

```
I:   tgt_offset ← sign_extend(offset || $0^2$)
     condition ←  (GPR[rs] ≠ GPR[rt])
I+1: if condition then
          PC ← PC + tgt_offset
     endif
```

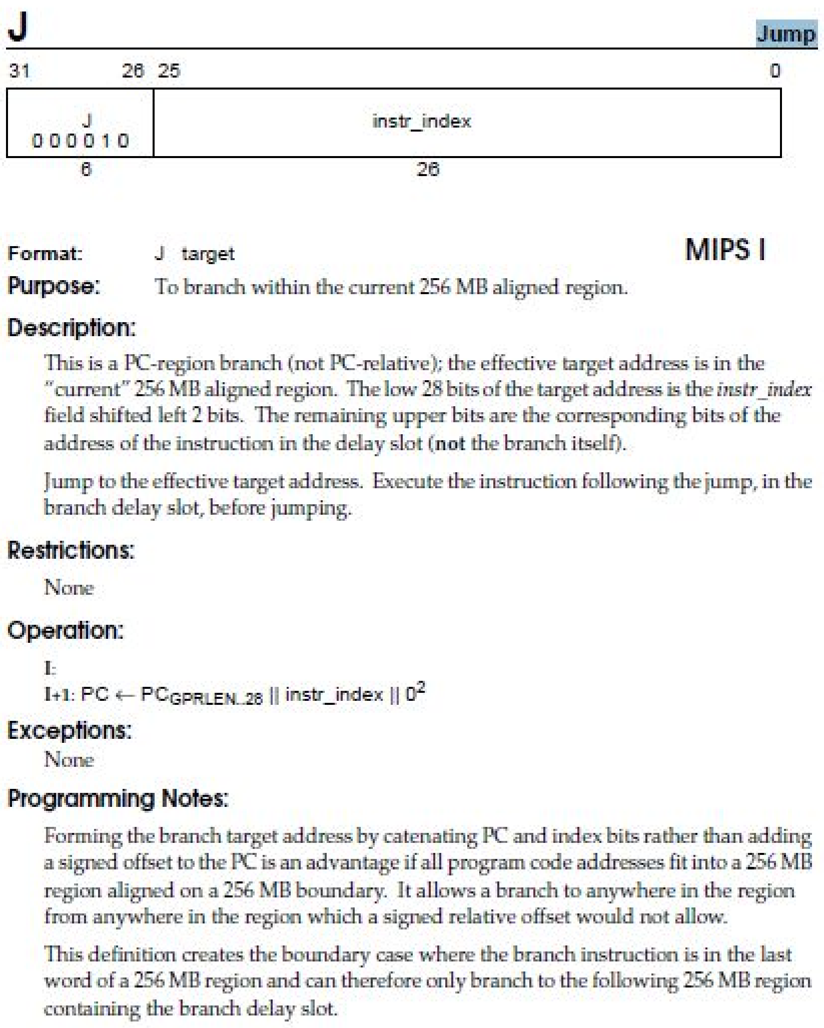
Exceptions:

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 more distant addresses.

18. J: Jump.



J

Jump

31 … 26	25 … 0
J 000010	instr_index
6	26

Format: J target **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 following the jump, in the branch delay slot, before jumping.

Restrictions:

None

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 to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.

19. JAL: Jump and Link.

Jump And Link **JAL**

31 26	25 0
JAL 000011	instr_index
6	26

Format: JAL target **MIPS I**

Purpose: To 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, where execution would continue 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 following the jump, in the branch delay slot, before jumping.

Restrictions:

None

Operation:

```
I:   $GPR[31] \leftarrow PC + 8$
I+1: $PC \leftarrow PC_{GPRLEN..28} \,||\, instr\_index \,||\, 0^2$
```

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 to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.