# PUSH—Push Word or Doubleword Onto the Stack

| Opcode | Instruction | Description |
|---|---|---|
| FF /6 | PUSH *r/m16* | Push *r/m16* |
| FF /6 | PUSH *r/m32* | Push *r/m32* |
| 50+*rw* | PUSH *r16* | Push *r16* |
| 50+*rd* | PUSH *r32* | Push *r32* |
| 6A | PUSH *imm8* | Push *imm8* |
| 68 | PUSH *imm16* | Push *imm16* |
| 68 | PUSH *imm32* | Push *imm32* |
| 0E | PUSH CS | Push CS |
| 16 | PUSH SS | Push SS |
| 1E | PUSH DS | Push DS |
| 06 | PUSH ES | Push ES |
| 0F A0 | PUSH FS | Push FS |
| 0F A8 | PUSH GS | Push GS |

## Description

This instruction decrements the stack pointer and then stores the source operand on the top of the stack. The address-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits), and the operand-size attribute of the current code segment determines the amount the stack pointer is decremented (two bytes or four bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is decremented by four and, if they are 16, the 16-bit SP register is decremented by 2.(The B flag in the stack segment's segment descriptor determines the stack's address-size attribute, and the D flag in the current code segment's segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the source operand.) Pushing a 16-bit operand when the stack address-size attribute is 32 can result in a misaligned the stack pointer (that is, the stack pointer is not aligned on a doubleword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. Thus, if a PUSH instruction uses a memory operand in which the ESP register is used as a base register for computing the operand address, the effective address of the operand is computed before the ESP register is decremented.

In the real-address mode, if the ESP or SP register is 1 when the PUSH instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

## Intel Architecture Compatibility

For Intel Architecture processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true in the real-address and virtual-8086 modes.) For the Intel 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

# PUSH—Push Word or Doubleword Onto the Stack (Continued)

## Operation

```
IF StackAddrSize = 32
THEN
    IF OperandSize = 32
        THEN
            ESP ← ESP – 4;
            SS:ESP ← SRC; (* push doubleword *)
        ELSE (* OperandSize = 16*)
            ESP ← ESP – 2;
            SS:ESP ← SRC; (* push word *)
    FI;
ELSE (* StackAddrSize = 16*)
    IF OperandSize = 16
        THEN
            SP ← SP – 2;
             SS:SP ← SRC; (* push word *)
        ELSE (* OperandSize = 32*)
            SP ← SP – 4;
            SS:SP ← SRC; (* push doubleword *)
    FI;
FI;
```

## Flags Affected

None.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# XOR—Logical Exclusive OR

| Opcode | Instruction | Description |
|---|---|---|
| 34 *ib* | XOR AL,*imm8* | AL XOR *imm8* |
| 35 *iw* | XOR AX,*imm16* | AX XOR *imm16* |
| 35 *id* | XOR EAX,*imm32* | EAX XOR *imm32* |
| 80 /6 *ib* | XOR *r/m8,imm8* | *r/m8* XOR *imm8* |
| 81 /6 *iw* | XOR *r/m16,imm16* | *r/m16* XOR *imm16* |
| 81 /6 *id* | XOR *r/m32,imm32* | *r/m32* XOR *imm32* |
| 83 /6 *ib* | XOR *r/m16,imm8* | *r/m16* XOR *imm8 (sign-extended)* |
| 83 /6 *ib* | XOR *r/m32,imm8* | *r/m32* XOR *imm8 (sign-extended)* |
| 30 /*r* | XOR *r/m8,r8* | *r/m8* XOR *r8* |
| 31 /*r* | XOR *r/m16,r16* | *r/m16* XOR *r16* |
| 31 /*r* | XOR *r/m32,r32* | *r/m32* XOR *r32* |
| 32 /*r* | XOR *r8,r/m8* | *r8* XOR *r/m8* |
| 33 /*r* | XOR *r16,r/m16* | *r8* XOR *r/m8* |
| 33 /*r* | XOR *r32,r/m32* | *r8* XOR *r/m8* |

## Description

This instruction performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

## Operation

DEST ← DEST XOR SRC;

## Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

# MOV—Move

| Opcode | Instruction | Description |
|---|---|---|
| 88 /*r* | MOV *r/m8,r8* | Move *r8* to *r/m8* |
| 89 /*r* | MOV *r/m16,r16* | Move *r16* to *r/m16* |
| 89 /*r* | MOV *r/m32,r32* | Move *r32* to *r/m32* |
| 8A /*r* | MOV *r8,r/m8* | Move *r/m8* to *r8* |
| 8B /*r* | MOV *r16,r/m16* | Move *r/m16* to *r16* |
| 8B /*r* | MOV *r32,r/m32* | Move *r/m32* to *r32* |
| 8C /*r* | MOV *r/m16,Sreg*** | Move segment register to *r/m16* |
| 8E /*r* | MOV *Sreg,r/m16*** | Move *r/m16* to segment register |
| A0 | MOV AL,*moffs8** | Move byte at (*seg:offset*) to AL |
| A1 | MOV AX,*moffs16** | Move word at (*seg:offset*) to AX |
| A1 | MOV EAX,*moffs32** | Move doubleword at (*seg:offset*) to EAX |
| A2 | MOV *moffs8**,AL | Move AL to (*seg:offset*) |
| A3 | MOV *moffs16**,AX | Move AX to (*seg:offset*) |
| A3 | MOV *moffs32**,EAX | Move EAX to (*seg:offset*) |
| B0+ *rb* | MOV *r8,imm8* | Move *imm8* to *r8* |
| B8+ *rw* | MOV *r16,imm16* | Move *imm16* to *r16* |
| B8+ *rd* | MOV *r32,imm32* | Move *imm32* to *r32* |
| C6 /*0* | MOV *r/m8,imm8* | Move *imm8* to *r/m8* |
| C7 /*0* | MOV *r/m16,imm16* | Move *imm16* to *r/m16* |
| C7 /*0* | MOV *r/m32,imm32* | Move *imm32* to *r/m32* |

**NOTES:**

* The *moffs8*, *moffs16*, and *moffs32* operands specify a simple offset relative to the segment base, where 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

** In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (refer to the following “Description” section for further information).

## Description

This instruction copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, or a doubleword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

# ADD—Add

| Opcode | Instruction | Description |
|---|---|---|
| 04 *ib* | ADD AL,*imm8* | Add *imm8* to AL |
| 05 *iw* | ADD AX,*imm16* | Add *imm16* to AX |
| 05 *id* | ADD EAX,*imm32* | Add *imm32* to EAX |
| 80 /0 *ib* | ADD *r/m8,imm8* | Add *imm8* to *r/m8* |
| 81 /0 *iw* | ADD *r/m16,imm16* | Add *imm16* to *r/m16* |
| 81 /0 *id* | ADD *r/m32,imm32* | Add *imm32* to *r/m32* |
| 83 /0 *ib* | ADD *r/m16,imm8* | Add sign-extended *imm8* to *r/m16* |
| 83 /0 *ib* | ADD *r/m32,imm8* | Add sign-extended *imm8* to *r/m32* |
| 00 /*r* | ADD *r/m8,r8* | Add *r8* to *r/m8* |
| 01 /*r* | ADD *r/m16,r16* | Add *r16* to *r/m16* |
| 01 /*r* | ADD *r/m32,r32* | Add r32 to *r/m32* |
| 02 /*r* | ADD *r8,r/m8* | Add *r/m8* to *r8* |
| 03 /*r* | ADD *r16,r/m16* | Add *r/m16* to *r16* |
| 03 /*r* | ADD *r32,r/m32* | Add *r/m32* to *r32* |

## Description

This instruction adds the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

## Operation

DEST ← DEST + SRC;

## Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

# ADC—Add with Carry

| Opcode | Instruction | Description |
|---|---|---|
| 14 *ib* | ADC AL,*imm8* | Add with carry *imm8* to AL |
| 15 *iw* | ADC AX,*imm16* | Add with carry *imm16* to AX |
| 15 *id* | ADC EAX,*imm32* | Add with carry *imm32* to EAX |
| 80 /2 *ib* | ADC *r/m8*,*imm8* | Add with carry *imm8* to *r/m8* |
| 81 /2 *iw* | ADC *r/m16,imm16* | Add with carry *imm16* to *r/m16* |
| 81 /2 *id* | ADC *r/m32,imm32* | Add with CF *imm32* to *r/m32* |
| 83 /2 *ib* | ADC *r/m16,imm8* | Add with CF sign-extended *imm8* to *r/m16* |
| 83 /2 *ib* | ADC *r/m32,imm8* | Add with CF sign-extended *imm8* into *r/m32* |
| 10 /*r* | ADC *r/m8,r8* | Add with carry byte register to *r/m8* |
| 11 /*r* | ADC *r/m16,r16* | Add with carry *r16* to *r/m16* |
| 11 /*r* | ADC *r/m32,r32* | Add with CF *r32* to *r/m32* |
| 12 /*r* | ADC *r8,r/m8* | Add with carry *r/m8* to byte register |
| 13 /*r* | ADC *r16,r/m16* | Add with carry *r/m16* to *r16* |
| 13 /*r* | ADC *r32,r/m32* | Add with CF *r/m32* to *r32* |

## Description

This instruction adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

## Operation

```
DEST ← DEST + SRC + CF;
```

## Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

# LOOP/LOOP*cc*—Loop According to ECX Counter

| Opcode | Instruction | Description |
|---|---|---|
| E2 *cb* | LOOP *rel8* | Decrement count; jump short if count ≠ 0 |
| E1 *cb* | LOOPE *rel8* | Decrement count; jump short if count ≠ 0 and ZF=1 |
| E1 *cb* | LOOPZ *rel8* | Decrement count; jump short if count ≠ 0 and ZF=1 |
| E0 *cb* | LOOPNE *rel8* | Decrement count; jump short if count ≠ 0 and ZF=0 |
| E0 *cb* | LOOPNZ *rel8* | Decrement count; jump short if count ≠ 0 and ZF=0 |

## Description

These instructions perform a loop operation using the ECX or CX register as a counter. Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop. If the address-size attribute is 32 bits, the ECX register is used as the count register; otherwise the CX register is used.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of –128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP*cc*) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the LOOP*cc* instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

# LOOP/LOOP*cc*—Loop According to ECX Counter (Continued)

## Operation

```
IF AddressSize = 32
    THEN
        Count is ECX;
    ELSE (* AddressSize = 16 *)
        Count is CX;
FI;
Count ← Count – 1;

IF instruction is not LOOP
    THEN
        IF (instruction = LOOPE) OR (instruction = LOOPZ)
            THEN
                IF (ZF =1) AND (Count ≠ 0)
                    THEN BranchCond ← 1;
                    ELSE BranchCond ← 0;
                FI;
        FI;
        IF (instruction = LOOPNE) OR (instruction = LOOPNZ)
            THEN
                IF (ZF =0 ) AND (Count ≠ 0)
                    THEN BranchCond ← 1;
                    ELSE BranchCond ← 0;
                FI;
        FI;
    ELSE (* instruction = LOOP *)
        IF (Count ≠ 0)
            THEN BranchCond ← 1;
            ELSE BranchCond ← 0;
        FI;
FI;
IF BranchCond = 1
    THEN
         EIP ← EIP + SignExtend(DEST);
        IF OperandSize = 16
            THEN
                EIP ← EIP AND 0000FFFFH;
        FI;
    ELSE
        Terminate loop and continue program execution at EIP;
FI;
```

# SHRD—Double Precision Shift Right

| Opcode | Instruction | Description |
|---|---|---|
| 0F AC | SHRD *r/m16,r16,imm8* | Shift *r/m16* to right *imm8* places while shifting bits from *r16* in from the left |
| 0F AD | SHRD *r/m16,r16*,CL | Shift *r/m16* to right CL places while shifting bits from *r16* in from the left |
| 0F AC | SHRD *r/m32,r32,imm8* | Shift *r/m32* to right *imm8* places while shifting bits from *r32* in from the left |
| 0F AD | SHRD *r/m32,r32*,CL | Shift *r/m32* to right CL places while shifting bits from *r32* in from the left |

## Description

This instruction shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand). The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be an immediate byte or the contents of the CL register. Only bits 0 through 4 of the count are used, which masks the count to a value between 0 and 31. If the count is greater than the operand size, the result in the destination operand is undefined.

If the count is one or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, the flags are not affected.

The SHRD instruction is useful for multiprecision shifts of 64 bits or more.

# SHRD—Double Precision Shift Right (Continued)

## Operation

```
COUNT ← COUNT MOD 32;
SIZE ← OperandSize
IF COUNT = 0
    THEN
        no operation
    ELSE
        IF COUNT ≥ SIZE
            THEN (* Bad parameters *)
                DEST is undefined;
                CF, OF, SF, ZF, AF, PF are undefined;
            ELSE (* Perform the shift *)
                CF ← BIT[DEST, COUNT – 1]; (* last bit shifted out on exit *)
                FOR i ← 0 TO SIZE – 1 – COUNT
                    DO
                        BIT[DEST, i] ← BIT[DEST, i – COUNT];
                    OD;
                FOR i ← SIZE – COUNT TO SIZE – 1
                    DO
                        BIT[DEST,i] ← BIT[inBits,i+COUNT – SIZE];
                    OD;
        FI;
FI;
```

## Flags Affected

If the count is one or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than one bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

## Protected Mode Exceptions

| | |
|---|---|
| #GP(0) | If the destination is located in a nonwritable segment. |
| | If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. |
| | If the DS, ES, FS, or GS register contains a null segment selector. |
| #SS(0) | If a memory operand effective address is outside the SS segment limit. |
| #PF(fault-code) | If a page fault occurs. |
| #AC(0) | If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. |

# XCHG—Exchange Register/Memory with Register

| Opcode | Instruction | Description |
|---|---|---|
| 90+*rw* | XCHG AX,*r16* | Exchange *r16* with AX |
| 90+*rw* | XCHG *r16*,AX | Exchange AX with *r16* |
| 90+*rd* | XCHG EAX,*r32* | Exchange *r32* with EAX |
| 90+*rd* | XCHG *r32*,EAX | Exchange EAX with *r32* |
| 86 /*r* | XCHG *r/m8,r8* | Exchange *r8* (byte register) with byte from *r/m8* |
| 86 /*r* | XCHG *r8,r/m8* | Exchange byte from *r/m8* with *r8* (byte register) |
| 87 /*r* | XCHG *r/m16,r16* | Exchange *r16* with word from *r/m16* |
| 87 /*r* | XCHG *r16,r/m16* | Exchange word from *r/m16* with *r16* |
| 87 /*r* | XCHG *r/m32,r32* | Exchange *r32* with doubleword from *r/m32* |
| 87 /*r* | XCHG *r32,r/m32* | Exchange doubleword from *r/m32* with *r32* |

## Description

This instruction exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. Refer to the LOCK prefix description in this chapter for more information on the locking protocol.

This instruction is useful for implementing semaphores or similar data structures for process synchronization. Refer to Section 7.1.2., *Bus Locking* in Chapter 7, *Multiple-Processor Management* of the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on bus locking.

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

## Operation

```
TEMP ← DEST
DEST ← SRC
SRC ← TEMP
```

## Flags Affected

None.

# POP—Pop a Value from the Stack

| Opcode | Instruction | Description |
|---|---|---|
| 8F /0 | POP *m16* | Pop top of stack into *m16*; increment stack pointer |
| 8F /0 | POP *m32* | Pop top of stack into *m32*; increment stack pointer |
| 58+ *rw* | POP *r16* | Pop top of stack into *r16*; increment stack pointer |
| 58+ *rd* | POP *r32* | Pop top of stack into *r32*; increment stack pointer |
| 1F | POP DS | Pop top of stack into DS; increment stack pointer |
| 07 | POP ES | Pop top of stack into ES; increment stack pointer |
| 17 | POP SS | Pop top of stack into SS; increment stack pointer |
| 0F A1 | POP FS | Pop top of stack into FS; increment stack pointer |
| 0F A9 | POP GS | Pop top of stack into GS; increment stack pointer |

## Description

This instruction loads the value from the top of the stack to the location specified with the destination operand and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

The current operand-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits—the source address size), and the operand-size attribute of the current code segment determines the amount the stack pointer is incremented (two bytes or four bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is incremented by four and, if they are 16, the 16-bit SP register is incremented by two. (The B flag in the stack segment's segment descriptor determines the stack's address-size attribute, and the D flag in the current code segment's segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the destination operand.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (refer to the "Operation" section below).

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a null value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is null.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0h as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

## POP—Pop a Value from the Stack (Continued)

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

A POP SS instruction inhibits all interrupts, including the NMI interrupt, until after execution of the next instruction. This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt[1]. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

### Operation

```
IF StackAddrSize = 32
    THEN
        IF OperandSize = 32
            THEN
                DEST ← SS:ESP; (* copy a doubleword *)
                ESP ← ESP + 4;
            ELSE (* OperandSize = 16*)
                DEST ← SS:ESP; (* copy a word *)
            ESP ← ESP + 2;
        FI;
    ELSE (* StackAddrSize = 16* )
        IF OperandSize = 16
            THEN
                DEST ← SS:SP; (* copy a word *)
                SP ← SP + 2;
            ELSE (* OperandSize = 32 *)
                DEST ← SS:SP; (* copy a doubleword *)
                SP ← SP + 4;
        FI;
FI;
```

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```
IF SS is loaded;
    THEN
        IF segment selector is null
            THEN #GP(0);
```

1. Note that in a sequence of instructions that individually delay interrupts past the following instruction, only the first instruction in the sequence is guaranteed to delay the interrupt, but subsequent interrupt-delaying instructions may not delay the interrupt. Thus, in the following instruction sequence:
   ```
   STI
   POP SS
   POP ESP
   ```
   interrupts may be recognized before the POP ESP executes, because STI also delays interrupts for one instruction.

# RET—Return from Procedure

| Opcode | Instruction | Description |
|---|---|---|
| C3 | RET | Near return to calling procedure |
| CB | RET | Far return to calling procedure |
| C2 *iw* | RET *imm16* | Near return to calling procedure and pop *imm16* bytes from stack |
| CA *iw* | RET *imm16* | Far return to calling procedure and pop *imm16* bytes from stack |

## Description

This instruction transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- Near return—A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- Far return—A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- Inter-privilege-level far return—A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. Refer to Section 4.3., *Calling Procedures Using CALL and RET* in Chapter 4, *Procedure Calls, Interrupts, and Exceptions* of the *Intel Architecture Software Developer's Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

## RET—Return from Procedure (Continued)

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure's stack and the calling procedure's stack (that is, the stack being returned to).

### Operation

```
(* Near return *)
IF instruction = near return
    THEN;
        IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
                EIP ← Pop();
            ELSE (* OperandSize = 16 *)
                IF top 6 bytes of stack not within stack limits
                    THEN #SS(0)
                FI;
                tempEIP ← Pop();
                tempEIP ← tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits THEN #GP(0); FI;
                EIP ← tempEIP;
        FI;
    IF instruction has immediate operand
        THEN IF StackAddressSize=32
            THEN
                ESP ← ESP + SRC; (* release parameters from stack *)
            ELSE (* StackAddressSize=16 *)
                SP ← SP + SRC; (* release parameters from stack *)
        FI;
    FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) OR (PE = 1 AND VM = 1)) AND instruction = far return
    THEN;
```

**MOV{<cond>}{S}  Rd, <shifter_operand>**

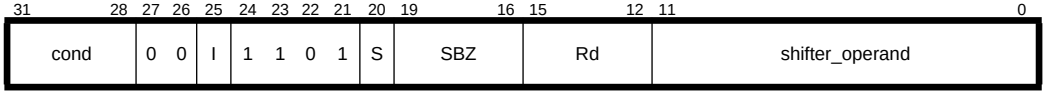
**Description** The MOV (Move) instruction is used to:

- move a value from one register to another
- put a constant value into a register
- perform a shift without any other arithmetic or logical operation

When the PC is the destination of the instruction, a branch occurs, and `MOV PC, LR` can be used to return from a subroutine call (see the B and BL instructions) and to return from some types of exception (See *2.5 Exceptions* on page 2-6).

MOV moves the value of `<shifter_operand>` to the destination register Rd, and optionally updates the condition code flags (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in *3.3 The Condition Field* on page 3-4.




**Operation**

```
if ConditionPassed(<cond>) then
       Rd = <shifter_operand>
       if S == 1 and Rd == R15 then
          CPSR = SPSR
       else if S == 1 then
          N Flag = Rd[31]
          Z Flag = if Rd == 0 then 1 else 0
          C Flag = <shifter_carry_out>
          V Flag = unaffected
```

**Exceptions** None

**Qualifiers** Condition Code
S updates condition code flags N,Z and C

**Notes**

**Shifter operand:** The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 3-84.

**Writing to R15:** When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

**The I bit:** Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

## LDR

`LDR{<cond>}  Rd, <addressing_mode>`

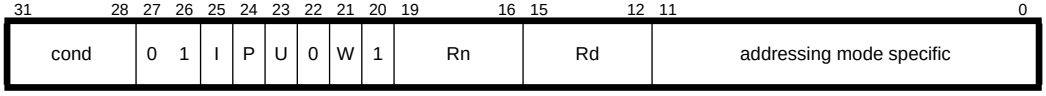
**Description**

Combined with a suitable addressing mode, the LDR (Load register) instruction allows 32-bit memory data to be loaded into a general-purpose register where its value can be manipulated. If the destination register is the PC, this instruction loads a 32-bit address from memory and branches to that address (precede the LDR instruction with `MOV LR, PC` to synthesize a branch and link).

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

LDR loads a word from the memory address calculated by `<addressing_mode>` and writes it to register Rd. If the address is not word-aligned, the loaded data is rotated so that the addressed byte occupies the least-significant byte of the register. If the PC is specified as register Rd, the instruction loads a branch to the address (data) into the PC.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in *3.3 The Condition Field* on page 3-4.




**Operation**

```
if ConditionPassed(<cond>) then
       if <address>[1:0] == 0b00
          Rd = Memory[<address>,4]
       else if <address>[1:0] == 0b01
          Rd = Memory[<address>,4] Rotate_Right 8
       else if <address>[1:0] == 0b10
          Rd = Memory[<address>,4] Rotate_Right 16
       else /* <address>[1:0] == 0b11 */
          Rd = Memory[<address>,4] Rotate_Right 24
```

**Exceptions**

Data Abort

**Qualifiers**

Condition Code

**Notes**

**Addressing modes:** The I, P, U and W bits specify the type of `<addressing_mode>` (see *Addressing Mode 2* starting on page 3-98).

**Register Rn:** Specifies the base register used by `<addressing_mode>`.

**Data Abort:** If a data abort is signalled and `<addressing_mode>` uses pre-indexed or post-indexed addressing, the value left in Rn is IMPLEMENTATION DEFINED, but is either the original base register value or the updated base register value (even if the same register is specified for Rd and Rn).

**Operand restrictions:** If `<addressing_mode>` uses pre-indexed or post-indexed addressing, and the same register is specified for Rd and Rn, the results are UNPREDICTABLE.

**Alignment:** If an implementation includes a System Control Coprocessor (See *Chapter 7*), and alignment checking is enabled, an address with bits[1:0] != 0b00 will cause an alignment exception.

**ADD{<cond>}{S}  Rd, Rn, <shifter_operand>**

**Description**

The ADD instruction adds the value of <shifter_operand> to the value of register Rn, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

ADD is used to add two values together to produce a third.

To increment a register value (in Rx), use:

```
ADD Rx, Rx, #1
```

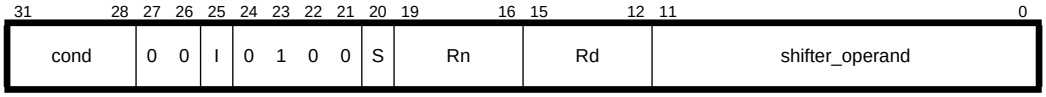
Constant multiplication (of Rx) by $2^n$+1 (into Rd) can be performed with:

```
ADD Rd, Rx, Rx LSL #n
```

To form a PC-relative address, use:

```
ADD Rs, PC, #offset
```

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in *3.3 The Condition Field* on page 3-4.




**Operation**

```
if ConditionPassed(<cond>) then
       Rd = Rn + <shifter_operand>
       if S == 1 and Rd == R15 then
          CPSR = SPSR
       else if S == 1 then
          N Flag = Rd[31]
          Z Flag = if Rd == 0 then 1 else 0
          C Flag = CarryFrom(Rn + <shifter_operand>)
          V Flag = OverflowFrom (Rn + <shifter_operand>)
```

**Exceptions**

None

**Qualifiers**

Condition Code

S updates condition code flags N,Z,C,V

**Notes**

**Shifter operand:** The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 3-84.

**The I bit:** Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.

**Writing to R15:** When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

**ADC{<cond>}{S}  Rd, Rn, <shifter_operand>**

**Description**

The ADC (Add with Carry) instruction adds the value of <shifter_operand> and the value of the Carry flag to the value of register Rn, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

ADC is used to synthesize multi-word addition. If register pairs R0,R1 and R2,R3 hold 64-bit values (where 0 and R2 hold the least-significant words), the following instructions leave the 64-bit sum in R4,R5:

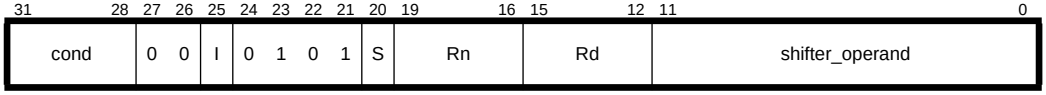
```
ADDS    R4,R0,R2
ADC     R5,R1,R3
```

The instruction:

```
ADCS R0,R0,R0
```

produces a single-bit Rotate Left with Extend operation (33-bit rotate though the carry flag) on R0. See 3-97 for more information.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in *3.3 The Condition Field* on page 3-4.




**Operation**

```
if ConditionPassed(<cond>) then
   Rd = Rn + <shifter_operand> + C Flag
   if S == 1 and Rd == R15 then
        CPSR = SPSR
   else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + <shifter_operand> + C Flag)
        V Flag = OverflowFrom (Rn + <shifter_operand> + C Flag)
```

**Exceptions**

None

**Qualifiers**

Condition Code

S updates condition code flags N,Z,C,V

**Notes**

**Shifter operand:** The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 3-84.

**The I bit:** Bit 25 is used to distinguish between the immediate and register forms of <shifter_operand>.

**Writing to R15:** When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

**SUB{<cond>}{S}  Rd, Rn, <shifter_operand>**

**Description** The SUB (Subtract) instruction is used to subtract one value from another to produce a third. To decrement a register value (in Rx) use:

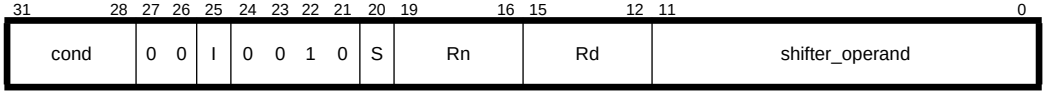
```
SUB Rx, Rx, #1
```

SUB subtracts the value of `<shifter_operand>` from the value of register Rn, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

SUBS is useful as a loop counter decrement, as the loop branch can test the flags for the appropriate termination condition, without the need for a `CMP Rx, #0`.

Use `SUBS PC, LR, #4` to return from an interrupt.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in *3.3 The Condition Field* on page 3-4.




**Operation**

```
if ConditionPassed(<cond>) then
       Rd = Rn - <shifter_operand>
       if S == 1 and Rd == R15 then
          CPSR = SPSR
       else if S == 1 then
          N Flag = Rd[31]
          Z Flag = if Rd == 0 then 1 else 0
          C Flag = NOT BorrowFrom(Rn - <shifter_operand>)
          V Flag = OverflowFrom (Rn - <shifter_operand>)
```

**Exceptions** None

**Qualifiers** Condition Code
S updates condition code flags N,Z,C and V

**Notes**

**Shifter operand:** The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 3-84.

**Writing to R15:** When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

**The I bit:** Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

**B{L}{<cond>}  <target address>**

**Description**

The B (Branch) and BL (Branch and Link) instructions provide both conditional and unconditional changes to program flow. The Branch with Link instruction is used to perform a subroutine call; the return from subroutine is achieved by copying the LR to the PC.

B and BL cause a branch to a target address. The branch target address is calculated by:

1 shifting the 24-bit signed (two's complement) offset left two bits

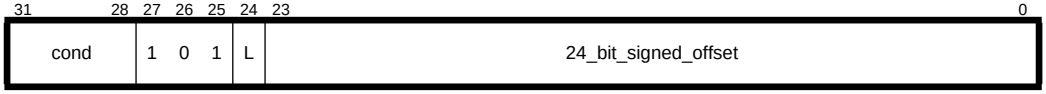
2 sign-extending the result to 32 bits

3 adding this to the contents of the PC (which contains the address of the branch instruction plus 8)

The instruction can therefore specify a branch of +/- 32Mbytes.

In the BL variant of the instruction, the L (link) bit is set, and the address of the instruction following the branch is copied into the link register (R14).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in *3.3 The Condition Field* on page 3-4.




**Operation**

```
if ConditionPassed(<cond>) then
   if L == 1 then
        LR = address of the instruction after the branch instruction
   PC = PC + (SignExtend(<24_bit_signed_offset>) << 2)
```

**Exceptions** None

**Qualifiers** Condition Code

L (Link) stores a return address in the LR (R14) register

**Notes**

**Offset calculation:** An assembler will calculate the branch offset address from the difference between the address of the current instruction and the address of the target (given as a program label) minus eight (because the PC holds the address of the current instruction plus eight).

**Memory bounds:** Branching backwards past location zero and forwards over the end of the 32-bit address space is UNPREDICTABLE.

## 3.3 The Condition Field

All ARM instructions can be conditionally executed, which means that their execution may or may not take place depending on the values of values of the N, Z, C and V flags in the CPSR. Every instruction contains a 4-bit condition code field in bits 31 to 28, as shown in *Figure 3-1: Condition code fields*.

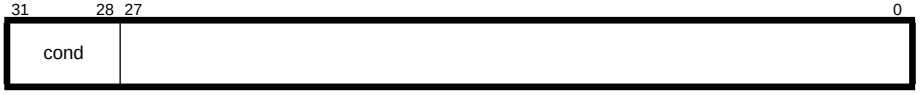



***Figure 3-1: Condition code fields***

### 3.3.1 Condition codes

This field specifies one of 16 conditions as described in *Table 3-2: Condition codes* on page 3-5. Every instruction mnemonic may be extended with the letters defined in the mnemonic extension field.

If the always (AL) condition is specified, the instruction will be executed irrespective of the value of the condition code flags. Any instruction that uses the never (NV) condition is UNPREDICTABLE. The absence of a condition code on an instruction mnemonic implies the always (AL) condition code.

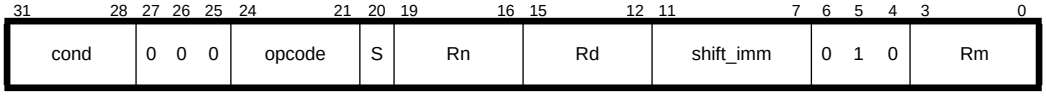
| Opcode [31:28] | Mnemonic Extension | Meaning | Status flag state |
|---|---|---|---|
| 0000 | EQ | Equal | Z set |
| 0001 | NE | Not Equal | Z clear |
| 0010 | CS/HS | Carry Set /Unsigned Higher or Same | C set |
| 0011 | CC/LO | Carry Clear /Unsigned Lower | C clear |
| 0100 | MI | Minus / Negative | N set |
| 0101 | PL | Plus /Positive or Zero | N clear |
| 0110 | VS | Overflow | V set |
| 0111 | VC | No Overflow | V clear |
| 1000 | HI | Unsigned Higher | C set and Z clear |
| 1001 | LS | Unsigned Lower or Same | C clear or Z set |
| 1010 | GE | Signed Greater Than or Equal | N set and V set, or<br>N clear and V clear (N = V) |
| 1011 | LT | Signed Less Than | N set and V clear, or<br>N clear and V set (N != V) |
| 1100 | GT | Signed Greater Than | Z clear, and either N set and V set, or<br>N clear and V clear (Z = 0,N = V) |
| 1101 | LE | Signed Less Than or Equal | Z set, or N set and V clear, or<br>N clear and V set (Z = 1, N != V) |
| 1110 | AL | Always (unconditional) | - |
| 1111 | NV | Never | - |

***Table 3-2: Condition codes***

**Rm, LSR #<shift_imm>**

**Description** This data-processing operand is used to provide the unsigned value of a register shifted right (divided by a constant power of two).

It is produced by the value of register Rm logically shifted right by an immediate value in the range 1 to 32. Zeros are inserted into the vacated bit positions. A shift by 32 is encoded by `<shift_imm> = 0`.




**Operation**

```
if <shift_imm> == 0 then
       <shifter_operand> = 0
       <shifter_carry_out> = Rm[31]
else /* <shift_imm> > 0 */
       <shifter_operand> = Rm Logical_Shift_Right <shift_imm>
       <shifter_carry_out> = Rm[<shift_imm> - 1]
```

**Notes** **Use of R15:** If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

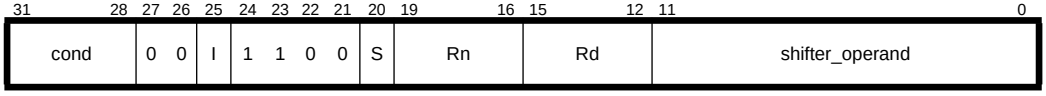
**ORR{<cond>}{S}  Rd, Rn, <shifter_operand>**

**Description**

The ORR (Logical OR) instruction can be used to set selected bits in a register; for each bit OR with 1 will set the bit, OR with 0 will leave it unchanged.

ORR performs a bitwise (inclusive) OR of the value of register Rn with the value of `<shifter_operand>`, and stores the result in the destination register Rd. The condition code flags are optionally updated (based on the result).

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in *3.3 The Condition Field* on page 3-4.




**Operation**

```
if ConditionPassed(<cond>) then
       Rd = Rn OR <shifter_operand>
       if S == 1 and Rd == R15 then
          CPSR = SPSR
       else if S == 1 then
          N Flag = Rd[31]
          Z Flag = if Rd == 0 then 1 else 0
          C Flag = <shifter_carry_out>
          V Flag = unaffected
```

**Exceptions**

None

**Qualifiers**

Condition Code
S updates condition code flags N, Z and C

**Notes**

**Shifter operand:** The shifter operands for this instruction are given in *Addressing Mode 1* starting on page 3-84.

**Writing to R15:** When Rd is R15 and the S flag in the instruction is not set, the result of the operation is placed in the PC. When Rd is R15 and the S flag is set, the result of the operation is placed in the PC and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of the instruction is UNPREDICTABLE in User mode and System mode.

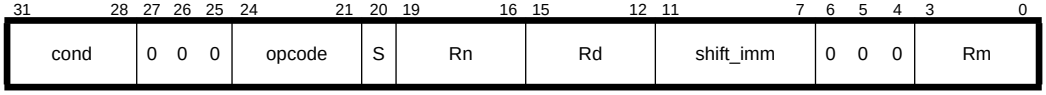
**The I bit:** Bit 25 is used to distinguish between the immediate and register forms of `<shifter_operand>`.

**Rm, LSL #<shift_imm>**

**Description**

This data-processing operand is used to provide either the value of a register directly (lone register operand (see page 3-88), or the value of a register shifted left (multiplied by a constant power of two).

This instruction operand is produced by the value of register Rm, logically shifted left by an immediate value in the range 0 to 31. Zeros are inserted into the vacated bit positions. The carry-out from the shifter is the last bit shifted out, or the C flag if no shift is specified (lone register operand, see page 3-88).




**Operation**

```
if <shift_imm> == 0 then /* Register Operand */
       <shifter_operand> = Rm
       <shifter_carry_out> = C Flag
else /* <shift_imm> > 0 */
       <shifter_operand> = Rm Logical_Shift_Left <shift_imm>
       <shifter_carry_out> = Rm[32 - <shift_imm>]
```

**Notes**

**Default shift:** If the value of `<shift_imm> == 0`, the operand may be written as just Rm, (see page 3-88).

**Use of R15:** If R15 is specified as register Rm or Rn, the value used is the address of the current instruction plus 8.

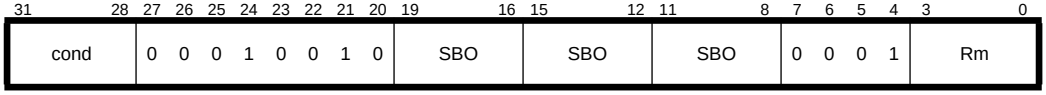
**BX{<cond>} Rm**

**Description**

The BX (Branch and Exchange instructions set) is UNDEFINED on ARM Architecture Version 4. On ARM Architecture Version 4T, this instruction branches and selects the instruction set decoder to use to decode the instructions at the branch destination. The branch target address is the value of register Rm. The T flag is updated with bit 0 of the value of register Rm.

BX is used to branch between ARM code and THUMB code. On ARM Architecture 4, it causes an UNDEFINED instruction exception to allow the THUMB instruction set to be emulated.

The instruction is only executed if the condition specified in the instruction matches the condition code status. The conditions are defined in *3.3 The Condition Field* on page 3-4.




**Operation**

```
if ConditionPassed(<cond>) then
       T Flag = Rm[0]
       PC = Rm[31:1] << 1
```

**Exceptions**

None

**Operation**

Condition Code

**Notes**

**Transferring to THUMB:** When transferring to the THUMB instruction set, bit[0] of PC will be cleared (set to zero), and bits[31:1] will be copied from Rm to the PC.

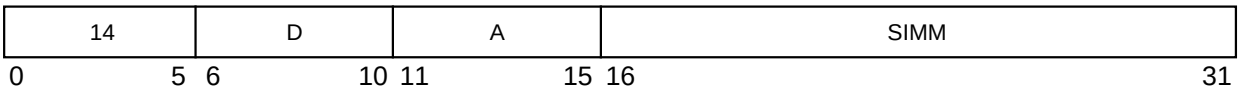
**Transferring to ARM:** When transferring to the ARM instruction set, bit[0] of PC will be cleared (set to zero), and bits[31:1] will be copied from Rm to the PC. If bit[1] of Rm is set, the result is UNPREDICTABLE.

# addi

addi

Add Immediate (x'3800 0000')

**addi** **r**D**,r**A**,**SIMM




```
if rA = 0
        then rD ←EXTS(SIMM)
        else rD←(rA) + EXTS(SIMM)
```

The sum (**r**A|0) + sign extended SIMM is placed into **r**D.

The **addi** instruction is preferred for addition because it sets few status bits.

**NOTE:** **addi** uses the value 0, not the contents of GPR0, if **r**A = 0.

Other registers altered:

- None

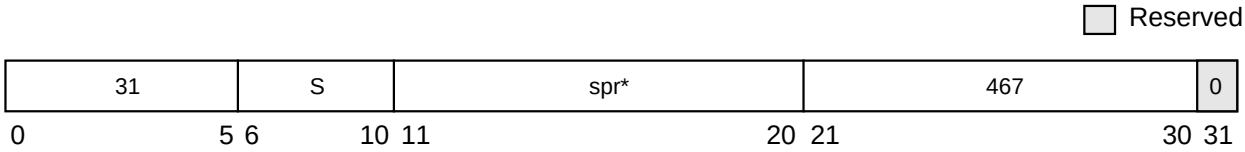
Simplified mnemonics:

| | | | | |
|---|---|---|---|---|
| **li** | **r**D**,**value | equivalent to | **addi** | **r**D**,0,**value |
| **la** | **r**D**,**disp**(r**A**)** | equivalent to | **addi** | **r**D**,r**A**,**disp |
| **subi** | **r**D**,r**A**,**value | equivalent to | **addi** | **r**D**,r**A**,**–value |

| PowerPC Architecture Level | Supervisor Level | PowerPC Optional | Form |
|---|---|---|---|
| UISA | | | D |

Move to Special-Purpose Register (x'7C00 03A6')

**mtspr** SPR,**rS**




**NOTE:** This is a split field.

```
n ← spr[5–9] || spr[0–4]
SPR(n) ← (rS)
```

In the PowerPC UISA, the SPR field denotes a special-purpose register, encoded as shown in Table 8-12. The contents of **r**S are placed into the designated special-purpose register.

**Table 8-12. PowerPC UISA SPR Encodings for mtspr**

| SPR** | | | Register Name |
|---|---|---|---|
| Decimal | spr[5–9] | spr[0–4] | |
| 1 | 00000 | 00001 | XER |
| 8 | 00000 | 01000 | LR |
| 9 | 00000 | 01001 | CTR |

** **Note**: The order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.

If the SPR field contains any value other than one of the values shown in Table 8-12, and the processor is operating in user mode, one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor instruction error handler is invoked.
- The results are boundedly undefined.

Other registers altered:

- See Table 8-12.

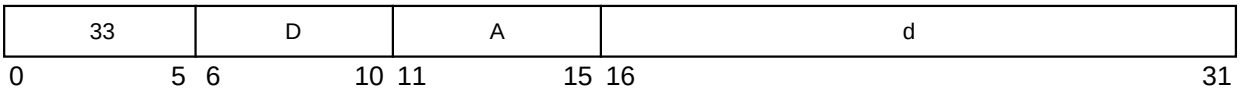
Simplified mnemonics:

| | | | | | |
|---|---|---|---|---|---|
| **mtxer** | **r**D | equivalent to | **mtspr** | **1,r**D |
| **mtlr** | **r**D | equivalent to | **mtspr** | **8,r**D |
| **mtctr** | **r**D | equivalent to | **mtspr** | **9,r**D |

# lwzu lwzu

Load Word and Zero with Update (x'8400 0000')

**lwzu** **r**D**,**d**(r**A**)**




```
EA ← (rA) + EXTS(d)
rD ← MEM(EA, 4)
rA ← EA
```

EA is the sum (**r**A) + d. The word in memory addressed by EA is loaded into **r**D.

EA is placed into **r**A.

If **r**A = 0, or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

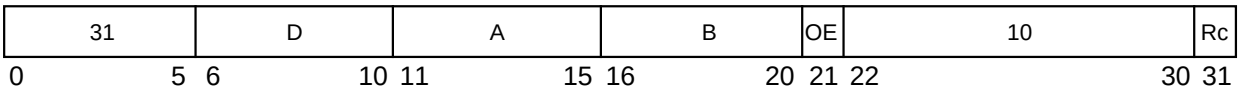
- None

8

| PowerPC Architecture Level | Supervisor Level | PowerPC Optional | Form |
|---|---|---|---|
| UISA | | | D |

Add Carrying (x'7C00 0014')

| | | |
|---|---|---|
| **addc** | **r**D,**r**A,**r**B | (OE = 0 Rc = 0) |
| **addc.** | **r**D,**r**A,**r**B | (OE = 0 Rc = 1) |
| **addco** | **r**D,**r**A,**r**B | (OE = 1 Rc = 0) |
| **addco.** | **r**D,**r**A,**r**B | (OE = 1 Rc = 1) |




**r**D← (**r**A) + (**r**B)

The sum (**r**A) + (**r**B) is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (If Rc = 1)

  **NOTE:** CR0 field may not reflect the infinitely precise result if overflow occurs (see next bullet item).

- XER:

  Affected: CA

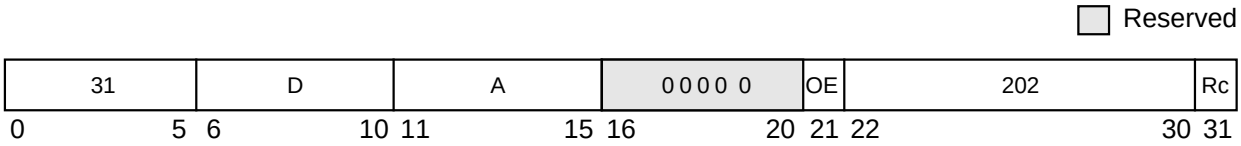
  Affected: SO, OV (If OE = 1)

  **NOTE:** For more information on condition codes see Section 2.1.3, "Condition Register," and Section 2.1.5, "XER Register."

| PowerPC Architecture Level | Supervisor Level | PowerPC Optional | Form |
|---|---|---|---|
| UISA | | | XO |

Add to Zero Extended (x'7C00 0194')

| | | |
|---|---|---|
| **addze** | **r**D,**r**A | (OE = 0 Rc = 0) |
| **addze.** | **r**D,**r**A | (OE = 0 Rc = 1) |
| **addzeo** | **r**D,**r**A | (OE = 1 Rc = 0) |
| **addzeo.** | **r**D,**r**A | (OE = 1 Rc = 1) |




```
rD← (rA) + XER[CA]
```

The sum (**r**A) + XER[CA] is placed into **r**D.

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (If Rc = 1)

  **NOTE:** CR0 field may not reflect the infinitely precise result if overflow occurs (see next).

- XER:

  Affected: CA

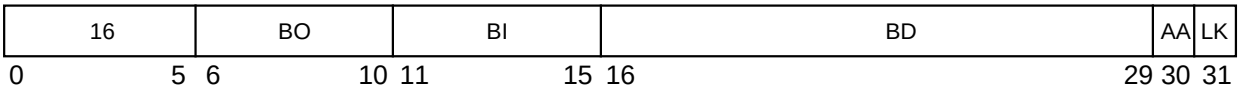
  Affected: SO, OV (If OE = 1)

  **NOTE:** For more information on condition codes see Section 2.1.3, "Condition Register," and Section 2.1.5, "XER Register".

| PowerPC Architecture Level | Supervisor Level | PowerPC Optional | Form |
|---|---|---|---|
| UISA | | | XO |

Branch Conditional (x'4000 0000')

| | | |
|---|---|---|
| **bc** | BO**,**BI**,**target_addr | (AA = 0 LK = 0) |
| **bca** | BO**,**BI**,**target_addr | (AA = 1 LK = 0) |
| **bcl** | BO**,**BI**,**target_addr | (AA = 0 LK = 1) |
| **bcla** | BO**,**BI**,**target_addr | (AA = 1 LK = 0) |




```
if ¬ BO[2]
    then CTR ← CTR – 1
ctr_ok ← BO[2] | ((CTR ≠ 0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok
    then
         if AA = 1
              then NIA ←iea EXTS(BD || 0b00)
              else NIA ←iea CIA + EXTS(BD || 0b00)
if LK = 1
    then LR ←iea CIA + 4
```



The BI field specifies the bit in the condition register (CR) to be used as the condition of the branch. The BO field is encoded as described in Table 8-6. Additional information about BO field encoding is provided in Section 4.2.4.2, "Conditional Branch Control".

**Table 8-6. BO Operand Encodings**

| BO | Description |
|---|---|
| 0000*y* | Decrement the CTR, then branch if the decremented CTR 0 and the condition is FALSE. |
| 0001*y* | Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE. |
| 001*zy* | Branch if the condition is FALSE. |
| 0100*y* | Decrement the CTR, then branch if the decremented CTR 0 and the condition is TRUE. |
| 0101*y* | Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE. |
| 011*zy* | Branch if the condition is TRUE. |
| 1*z*00*y* | Decrement the CTR, then branch if the decremented CTR 0. |
| 1*z*01*y* | Decrement the CTR, then branch if the decremented CTR = 0. |
| 1*z*1*zz* | Branch always. |

In this table, *z* indicates a bit that is ignored.
**Note:** The *z* bits should be cleared, as they may be assigned a meaning in some future version of the PowerPC architecture.

The *y* bit provides a hint about whether a conditional branch is likely to be taken, and may be used by some PowerPC implementations to improve performance.

target_addr specifies the branch target address.

If AA = 0, the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction.

If AA = 1, the branch target address is the value BD || 0b00 sign-extended.

If LK = 1, the effective address of the instruction following the branch instruction is placed into the link register.

Other registers altered:

Affected: Count Register (CTR) (If BO[2] = 0)

Affected: Link Register (LR) (If LK = 1)

Simplified mnemonics:

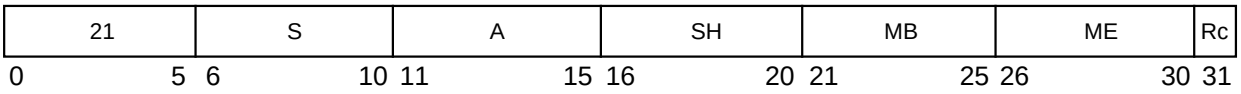
| | | | | |
|---|---|---|---|---|
| **blt** | target | equivalent to | **bc** | **12,0,**target |
| **bne** | **cr2**,target | equivalent to | **bc** | **4,10,**target |
| **bdnz** | target | equivalent to | **bc** | **16,0,**target |



| PowerPC Architecture Level | Supervisor Level | PowerPC Optional | Form |
|---|---|---|---|
| UISA | | | D |

Rotate Left Word Immediate then AND with Mask (x'5400 0000')

**rlwinm** **r**A**,r**S**,**SH**,**MB**,**ME (Rc = 0)
**rlwinm.** **r**A**,r**S**,**SH**,**MB**,**ME (Rc = 1)




```
n ← SH
r ← ROTL(rS, n)
m ← MASK(MB , ME)
rA ← r & m
```

The contents of **r**S are rotated left the number of bits specified by operand SH. A mask is generated having 1 bits from bit MB through bit ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

**NOTE:** **rlwinm** can be used to extract, rotate, shift, and clear bit fields using the methods shown below:

- To extract an *n*-bit field, that starts at bit position *b* in **r**S, right-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), set SH = *b* + *n*, MB = 32 – *n*, and ME = 31.
- To extract an *n*-bit field, that starts at bit position *b* in **r**S, left-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), set SH = *b*, MB = 0, and ME = *n* – 1.
- To rotate the contents of a register left (or right) by *n* bits, set SH = *n* (32 – *n*), MB = 0, and ME = 31.
- To shift the contents of a register right by *n* bits, by setting SH = 32 – *n*, MB = *n*, and ME = 31. It can be used to clear the high-order *b* bits of a register and then shift the result left by *n* bits by setting SH = *n*, MB = *b* – *n* and ME = 31 – *n*.
- To clear the low-order *n* bits of a register, by setting SH = 0, MB = 0, and ME = 31 – *n*..

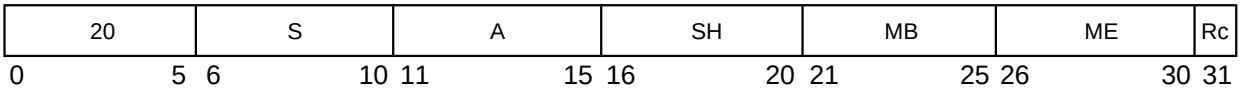
Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

Rotate Left Word Immediate then Mask Insert (x'5000 0000')

**rlwimi** **r**A,**r**S,SH,MB,ME (Rc = 0)
**rlwimi.** **r**A,**r**S,SH,MB,ME (Rc = 1)




```
n ← SH
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← (r & m) | (rA & ¬ m)
```

The contents of **r**S are rotated left the number of bits specified by operand SH. A mask is generated having 1 bits from bit MB through bit ME and 0 bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask.

8

**NOTE:** **rlwimi** can be used to copy a bit field of any length from register **r**S into the contents of **r**A. This field can start from any bit position in **r**S and be placed into any position in **r**A. The length of the field can range from 0 to 32 bits. The remaining bits in register **r**A remain unchanged:

- To copy byte_0 (bits 0-7) from rS into byte_3 (bits 24-31) of rA, set SH = 8 , MB = *24*, and ME = 31.
- In general, to copy an *n*-bit field that starts in bit position b in register rS into register rA starting a bit position c: set SH = 32 - c + b Mod(32), set MB = *c*, and set ME = (*c* + *n*) – 1 Mod(32).

Other registers altered:

- Condition Register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

Simplified mnemonics:
inslwi rA,rS,n,b equivalent to rlwimi rA,rS,32 – b,b,b + n – 1
insrwi rA,rS,n,b (n > 0) equivalent to rlwimi rA,rS,32 – (b + n),b, (b + n) – 1

| PowerPC Architecture Level | Supervisor Level | PowerPC Optional | Form |
|---|---|---|---|
| UISA | | | M |

Table F-6 provides the simplified mnemonics for the **bclr** and **bcclr** instructions without link register updating, and the syntax associated with these instructions.

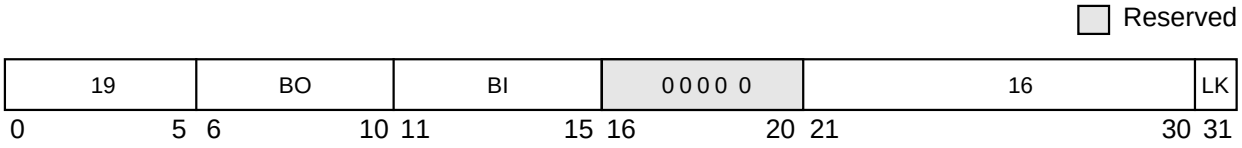
**NOTE:** The default condition register specified by the simplified mnemonics in the table is CR0.

**Table F-6. Simplified Branch Mnemonics for bclr and bcclr Instructions without Link Register Update**

| Branch Semantics | LR Update Not Enabled | | | |
|---|---|---|---|---|
| | **bclr** to LR | Simplified Mnemonic | **bcctr** to CTR | Simplified Mnemonic |
| Branch unconditionally | **bclr** 20,0 | **blr** | **bcctr** 20,0 | **bctr** |
| Branch if condition true | **bclr** 12,0 | **btlr** 0 | **bcctr** 12,0 | **btctr** 0 |
| Branch if condition false | **bclr** 4,0 | **bflr** 0 | **bcctr** 4,0 | **bfctr** 0 |
| Decrement CTR, branch if CTR nonzero | **bclr** 16,0 | **bdnzlr** | — | — |
| Decrement CTR, branch if CTR nonzero AND condition true | **bclr** 10,0 | **bdztlr** 0 | — | — |
| Decrement CTR, branch if CTR nonzero AND condition false | **bclr** 0,0 | **bdnzflr** 0 | — | — |
| Decrement CTR, branch if CTR zero | **bclr** 18,0 | **bdzlr** | — | — |
| Decrement CTR, branch if CTR zero AND condition true | **bclr** 10,0 | **bdztlr** 0 | — | — |
| Decrement CTR, branch if CTR zero AND condition false | **bcctr** 0,0 | **bdzflr** 0 | — | — |

Branch Conditional to Link Register (x'4C00 0020')

| | | |
|---|---|---|
| **bclr** | BO,BI | (LK = 0) |
| **bclrl** | BO,BI | (LK = 1) |




```
if ¬ BO[2]
     then CTR ← CTR – 1
ctr_ok ← BO[2] | ((CTR ≠   0)⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok
     then   NIA ←iea LR[0–29] || 0b00
if LK
     then LR ←iea CIA + 4
```

The BI field specifies the bit in the condition register to be used as the condition of the branch. The BO field is encoded as described in Table 8-8. Additional information about BO field encoding is provided in Section 4.2.4.2, "Conditional Branch Control".



**Table 8-8. BO Operand Encodings**

| BO | Description |
|---|---|
| 0000*y* | Decrement the CTR, then branch if the decremented CTR  0 and the condition is FALSE. |
| 0001*y* | Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE. |
| 001*zy* | Branch if the condition is FALSE. |
| 0100*y* | Decrement the CTR, then branch if the decremented CTR  0 and the condition is TRUE. |
| 0101*y* | Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE. |
| 011*zy* | Branch if the condition is TRUE. |
| 1*z*00*y* | Decrement the CTR, then branch if the decremented CTR  0. |
| 1*z*01*y* | Decrement the CTR, then branch if the decremented CTR = 0. |
| 1*z*1*zz* | Branch always. |

If the BO field specifies that the CTR is to be decremented, the entire 32-bit CTR is decremented.

In this table, *z* indicates a bit that is ignored.
**Note**: The *z* bits should be cleared, as they may be assigned a meaning in some future version of the PowerPC architecture.

The *y* bit provides a hint about whether a conditional branch is likely to be taken, and may be used by some PowerPC implementations to improve performance.

The branch target address is LR[0–29] || 0b00.

Table 2-5 *Synthetic Instruction to Hardware Instruction Mapping— Continued*

| *Synthetic Instruction* | | *Hardware Equivalent(s)* | | *Comment* |
|---|---|---|---|---|
| `set` | *value*, *reg$_{rd}$* | `sethi` | `%hi(`*value*`)`, *reg$_{rd}$* | *(if ((value&0x1ff) == 0))* |
| `set` | *value*, *reg$_{rd}$* | `sethi`<br>`or` | `%hi(`*value*`)`, *reg$_{rd}$*;<br>*reg$_{rd}$*, `%lo(`*value*`)`, *reg$_{rd}$* | *(otherwise)*<br><br>*Warning: do not use* `set` *in an instruction's delay slot.* |
| `not` | *reg$_{rs1}$*, *reg$_{rd}$* | `xnor` | *reg$_{rs1}$*, `%g0`, *reg$_{rd}$* | *(one's complement)* |
| `not` | *reg$_{rd}$* | `xnor` | *reg$_{rd}$*, `%g0`, *reg$_{rd}$* | *(one's complement)* |
| `neg` | *reg$_{rs2}$*, *reg$_{rd}$* | `sub` | `%g0`, *reg$_{rs2}$*, *reg$_{rd}$* | *(two's complement)* |
| `neg` | *reg$_{rd}$* | `sub` | `%g0`, *reg$_{rd}$*, *reg$_{rd}$* | *(two's complement)* |
| `inc` | *reg$_{rd}$* | `add` | *reg$_{rd}$*, `1`, *reg$_{rd}$* | *(increment by 1)* |
| `inc` | *const13*, *reg$_{rd}$* | `add` | *reg$_{rd}$*, *const13*,*reg$_{rd}$* | *(increment by const13)* |
| `inccc` | *reg$_{rd}$* | `addcc` | *reg$_{rd}$*, `1`, *reg$_{rd}$* | *(increment by 1 and set icc)* |
| `inccc` | *const13*,*reg$_{rd}$* | `addcc` | *reg$_{rd}$*, *const13*,*reg$_{rd}$* | *(increment by const13 and set icc)* |
| `dec` | *reg$_{rd}$* | `sub` | *reg$_{rd}$*, `1`, *reg$_{rd}$* | *(decrement by 1)* |
| `dec` | *const13*, *reg$_{rd}$* | `sub` | *reg$_{rd}$*, *const13*,*reg$_{rd}$* | *(decrement by const13)* |
| `deccc` | *reg$_{rd}$* | `subcc` | *reg$_{rd}$*, `1`, *reg$_{rd}$* | *(decrement by 1 and set icc)* |
| `deccc` | *const13*, *reg$_{rd}$* | `subcc` | *reg$_{rd}$*, *const13*,*reg$_{rd}$* | *(decrement by const13 and set icc)* |
| `btst` | *reg_or_imm*, *reg$_{rs1}$* | `andcc` | *reg$_{rs1}$*, *reg_or_imm*, `%g0` | *(bit test)* |
| `bset` | *reg_or_imm*, *reg$_{rd}$* | `or` | *reg$_{rd}$*, *reg_or_imm*, *reg$_{rd}$* | *(bit set)* |
| `bclr` | *reg_or_imm*, *reg$_{rd}$* | `andn` | *reg$_{rd}$*, *reg_or_imm*, *reg$_{rd}$* | *(bit clear)* |
| `btog` | *reg_or_imm*, *reg$_{rd}$* | `xor` | *reg$_{rd}$*, *reg_or_imm*, *reg$_{rd}$* | *(bit toggle)* |
| `clr` | *reg$_{rd}$* | `or` | `%g0`, `%g0`, *reg$_{rd}$* | *(clear(zero) register)* |
| `clrb` | [*address*] | `stb` | `%g0`, [*address*] | *(clear byte)* |
| `clrh` | [*address*] | `sth` | `%g0`, [*address*] | *(clear halfword)* |
| `clr` | [*address*] | `st` | `%g0`, [*address*] | *(clear word)* |
| `mov` | *reg_or_imm*, *reg$_{rd}$* | `or` | `%g0`, *reg_or_imm*, *reg$_{rd}$* | |
| `mov` | `%y`, *reg$_{rs1}$* | `rd` | `%y`, *reg$_{rs1}$* | |
| `mov` | `%psr`, *reg$_{rs1}$* | `rd` | `%psr`, *reg$_{rs1}$* | |
| `mov` | `%wim`, *reg$_{rs1}$* | `rd` | `%wim`, *reg$_{rs1}$* | |
| `mov` | `%tbr`, *reg$_{rs1}$* | `rd` | `%tbr`, *reg$_{rs1}$* | |
| `mov` | *reg_or_imm*, `%y` | `wr` | `%g0`, *reg_or_imm*, `%y` | |
| `mov` | *reg_or_imm*, `%psr` | `wr` | `%g0`, *reg_or_imm*, `%psr` | |
| `mov` | *reg_or_imm*, `%wim` | `wr` | `%g0`, *reg_or_imm*, `%wim` | |
| `mov` | *reg_or_imm*, `%tbr` | `wr` | `%g0`, *reg_or_imm*, `%tbr` | |

# OR — Inclusive-Or — OR

**Operation:** r[rd] ← r[rs1] OR (r[rs2] or sign extnd(simm13))

**Assembler Syntax:** or *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

**Description:** This instruction does a bitwise logical OR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

**Traps:** none

**Format:**

# LD — Load Word — LD

**Operation:** r[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

**Assembler Syntax:** ld [*address*], $reg_{rd}$

**Description:** The LD instruction moves a word from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

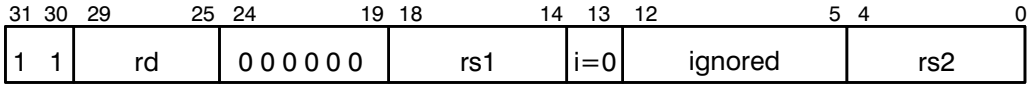
If LD takes a trap, the contents of the destination register remain unchanged.

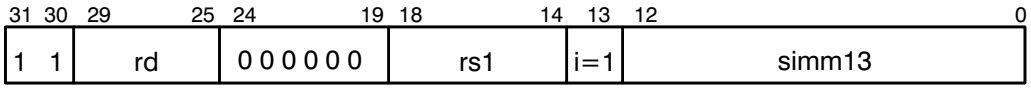
If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

*Programming note:* If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

**Traps:** memory_address_not_aligned
data_access_exception

**Format:**



# ADDcc — Add and modify icc — ADDcc

**Operation:** r[rd] ← r[rs1] + operand2, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
c ← (r[rs1]<31> AND operand2<31>)
OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))

**Assembler Syntax:** addcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

**Description:** ADDcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. The result is placed in the register specified in the *rd* field. In addition, ADDcc modifies all the integer condition codes in the manner described above.

**Traps:** none

**Format:**

# ADDX — Add with Carry — ADDX

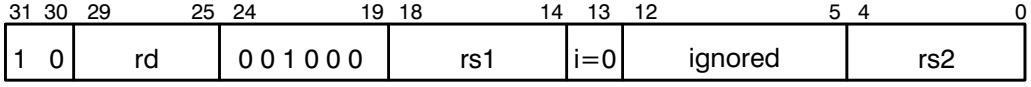
**Operation:** r[rd] ← r[rs1] + (r[rs2] or sign extnd(simm13)) + c

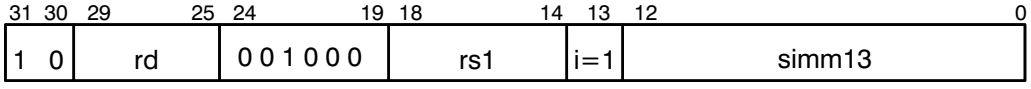
**Assembler Syntax:** addx *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

**Description:** ADDX adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. It then adds the PSR's carry bit (*c*) to that result. The final result is placed in the register specified in the *rd* field.

**Traps:** none

**Format:**



# SUBcc — Subtract and modify icc — SUBcc

**Operation:** r[rd] ← r[rs1] - operand2, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>)
OR (not r[rs1]<31> AND operand2<31> AND r[rd]<31>)
c ← (not r[rs1]<31> AND operand2<31>)
OR (r[rd]<31> AND (not r[rs1]<31> OR operand2<31>))

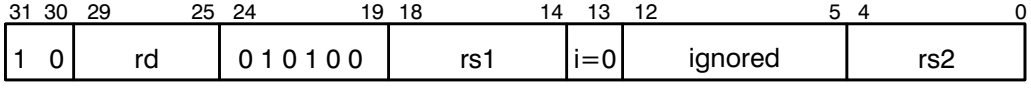
**Assembler Syntax:** subcc *regrs1, reg_or_imm, regrd*

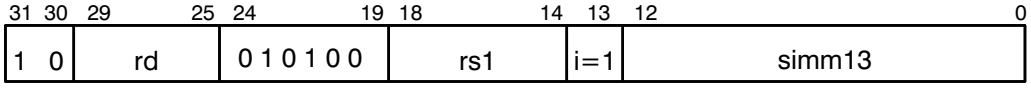
**Description:** The SUBcc instruction subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. The result is placed in register r[rd]. In addition, SUBcc modifies all the integer condition codes in the manner described above.

*Programming note:* A SUBcc instruction with *rd* = 0 can be used for signed and unsigned integer comparison.

**Traps:** none

**Format:**



# Bicc — Integer Conditional Branch — Bicc

**Operation:** PC ← nPC

If condition true then nPC ← PC + (sign extnd(disp22) x 4)

else nPC ← nPC + 4

**Assembler Syntax:**

| | | |
|---|---|---|
| ba{,a} | *label* | |
| bn{,a} | *label* | |
| bne{,a} | *label* | synonym: bnz |
| be{,a} | *label* | synonym: bz |
| bg{,a} | *label* | |
| ble{,a} | *label* | |
| bge{,a} | *label* | |
| bl{,a} | *label* | |
| bgu{,a} | *label* | |
| bleu{,a} | *label* | |
| bcc{,a} | *label* | synonym: bgeu |
| bcs{,a} | *label* | synonym: blu |
| bpos{,a} | *label* | |
| bneg{,a} | *label* | |
| bvc{,a} | *label* | |
| bvs{,a} | *label* | |

*Note:* The instruction's annul bit field, *a*, is set by appending ",a" after the branch name. If it is not appended, the *a* field is automatically reset. ",a" is shown in braces because it is optional.

**Description:** The Bicc instructions (except for BA and BN) evaluate specific integer condition code combinations (from the PSR's *icc* field) based on the branch type as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken. Refer to Section NO TAG for additional information on control transfer instructions.

If the branch is not taken, the annul bit field (*a*) is checked. If *a* is set, the instruction immediately following the branch instruction (the delay instruction) *is not* executed (i.e., it is annulled). If the annul field is zero, the delay instruction *is* executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed. See Section NO TAG regarding delay-branch instructions.

Branch Never (BN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

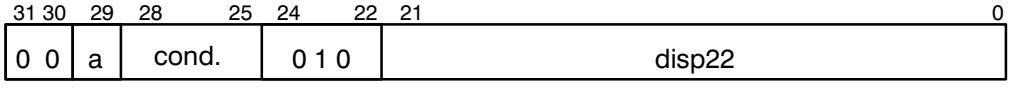
Branch Always (BA), because it always branches regardless of the condition codes, would normally ignore the annul field. Instead, it follows the same annul field rules: if $a=1$, the delay instruction is annulled; if $a=0$, the delay instruction is executed.

The delay instruction following a Bicc (other than BA) should not be a delayed-control-transfer instruction. The results of following a Bicc with another delayed control transfer instruction are implementation-dependent and therefore unpredictable.

**Traps:** none

| Mnemonic | Cond. | Operation | icc Test |
|---|---|---|---|
| BN | 0000 | Branch Never | No test |
| BE | 0001 | Branch on Equal | z |
| BLE | 0010 | Branch on Less or Equal | z OR (n XOR v) |
| BL | 0011 | Branch on Less | n XOR v |
| BLEU | 0100 | Branch on Less or Equal, Unsigned | c OR z |
| BCS | 0101 | Branch on Carry Set<br>(Less than, Unsigned) | c |
| BNEG | 0110 | Branch on Negative | n |
| BVS | 0111 | Branch on oVerflow Set | v |
| BA | 1000 | Branch Always | No test |
| BNE | 1001 | Branch on Not Equal | not z |
| BG | 1010 | Branch on Greater | not(z OR (n XOR v)) |
| BGE | 1011 | Branch on Greater or Equal | not(n XOR v) |
| BGU | 1100 | Branch on Greater, Unsigned | not(c OR z) |
| BCC | 1101 | Branch on Carry Clear<br>(Greater than or Equal, Unsigned) | not c |
| BPOS | 1110 | Branch on Positive | not n |
| BVC | 1111 | Branch on oVerflow Clear | not v |

**Format:**



# ADD — Add — ADD

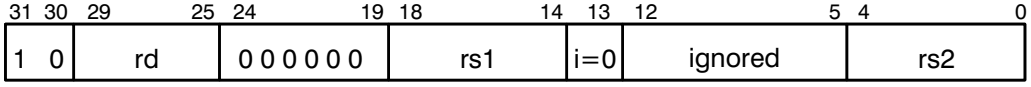
**Operation:** r[rd] ← r[rs1] + (r[rs2] or sign extnd(simm13))

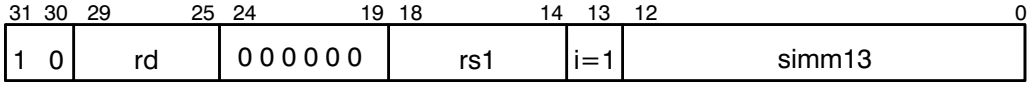
**Assembler Syntax:** add $reg_{rs1}$, *reg_or_imm*, $reg_{rd}$

**Description:** The ADD instruction adds the contents of the register named in the *rs1* field ,r[rs1], to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The result is placed in the register specified in the *rd* field.

**Traps:** none

**Format:**



# SRL — Shift Right Logical — SRL

**Operation:** r[rd] ← r[rs1] SRL by (r[rs2] or shcnt)

**Assembler Syntax:** srl *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

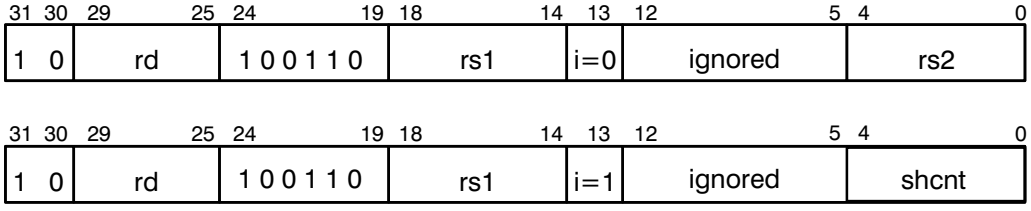
**Description:** SRL shifts the contents of r[rs1] right by the number of bits specified by the shift count, filling the vacated positions with zeros. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

If the i bit field equals zero, the shift count for SRL is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SRL is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt*.

This instruction does *not* modify the condition codes.

**Traps:** none

**Format:**



# SLL — Shift Left Logical — SLL

**Operation:** r[rd] ← r[rs1] SLL by (r[rs2] or shcnt)

**Assembler Syntax:** sll *$reg_{rs1}$, reg_or_imm, $reg_{rd}$*

**Description:** SLL shifts the contents of r[rs1] left by the number of bits specified by the shift count, filling the vacated positions with zeros. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

If the i bit field equals zero, the shift count for SLL is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SLL is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt*.

This instruction does *not* modify the condition codes.

**Traps:** none

**Format:**

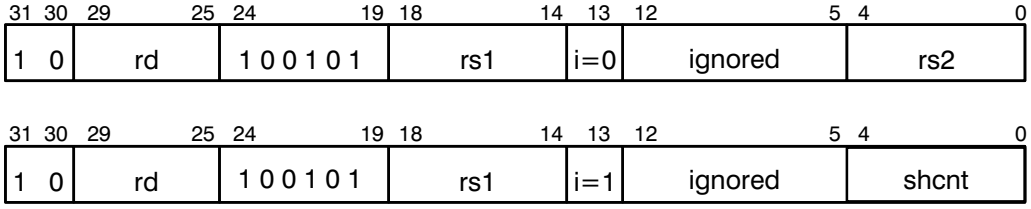


Table 2-3 *Floating-point Instructions— Continued*

| *SPARC* | *Mnemonic* | *Argument List* | *Description* |
|---|---|---|---|
| FCMPEs<br>FCMPEd<br>FCMPEq | fcmpes<br>fcmped<br>fcmpeq | $freg_{rs1}$, $freg_{rs2}$<br>$freg_{rs1}$, $freg_{rs2}$<br>$freg_{rs1}$, $freg_{rs2}$ | *Compare, Generate exception if unordered.* |

## 2.4. Coprocessor Instructions

All `cpop`*n* instructions take all operands from and return all results to coprocessor registers. The data types supported by the coprocessor are coprocessor-dependent. Operand alignment is coprocessor-dependent.

If the EC field of the PSR is 0, or if no coprocessor is present, a `cpop`*n* instruction causes a `cp_disabled` trap.

The conditions causing a `cp_exception` trap are coprocessor-dependent.

*NOTE* *A non-* `cpop`*n (non-coprocessor-operate) instruction must be executed between a* `cpop2` *instruction and a subsequent* `cbccc` *instruction.*

Table 2-4 *Coprocessor Instructions*

| *SPARC* | *Mnemonic* | *Argument List* | *Name* | *Comments* |
|---|---|---|---|---|
| CPop1 | cpop1 | *opd*, $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | *Coprocessor operation* | |
| CPop2 | cpop2 | *opd*, $reg_{rs1}$, $reg_{rs2}$, $reg_{rd}$ | *Coprocessor operation* | *(may modify ccc's)* |

## 2.5. Synthetic Instructions

This section describes the mapping of synthetic instructions to hardware instructions.

Table 2-5 *Synthetic Instruction to Hardware Instruction Mapping*

| *Synthetic Instruction* | | *Hardware Equivalent(s)* | | *Comment* |
|---|---|---|---|---|
| cmp | $reg_{rs1}$, *reg_or_imm* | subcc | $reg_{rs1}$, *reg_or_imm*, %g0 | *(compare)* |
| jmp | *address* | jmpl | *address*, %g0 | |
| call | *reg_or_imm* | jmpl | *reg_or_imm*, %o7 | |
| tst | $reg_{rs1}$ | orcc | $reg_{rs1}$, %g0, %g0 | *(test)* |
| ret<br>retl | | jmpl<br>jmpl | %i7+8, %g0<br>%o7+8, %g0 | *(return from subroutine)*<br>*(return from leaf subroutine)* |
| restore<br>save | | restore<br>save | %g0, %g0, %g0<br>%g0, %g0, %g0 | *(trivial* restore*)*<br>*(trivial* save*)*<br>*Warning: trivial* save *should only be used in kernel code!* |
| set | *value*, $reg_{rd}$ | or | %g0, *value*, $reg_{rd}$ | *(if -4096 ≤ value ≤ 4095)* |

# JMPL — Jump and Link — JMPL

**Operation:** r[rd] ← PC
PC ← nPC
nPC ← r[rs1] + (r[rs2] or sign extnd(simm13))

**Assembler Syntax:** jmpl *address*, $reg_{rd}$

**Description:** JMPL first provides linkage by saving its return address into the register specified in the *rd* field. It then causes a register-indirect, delayed control transfer to an address specified by the sum of the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If either of the low-order two bits of the jump address is nonzero, a memory_address_not_aligned trap is generated.

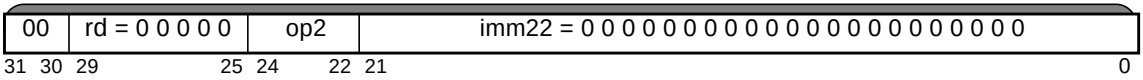
*Programming note:* A register-indirect CALL can be constructed using a JMPL instruction with *rd* set to 15. JMPL can also be used to return from a CALL. In this case, *rd* is set to 0 and the return (jump) address would be equal to r[31] + 8.

**Traps:** memory_address_not_aligned

**Format:**

# 7.67 No Operation

| Instruction | op2 | Operation | Assembly Language Syntax | Class |
|---|---|---|---|---|
| NOP | 100 | No Operation | nop | **A1** |




*Description* The NOP instruction changes no program-visible state (except that of the PC register).

NOP is a special case of the SETHI instruction, with imm22 = 0 and rd = 0.

**Programming Note** There are many other opcodes that may execute as NOPs; however, this dedicated NOP instruction is the only one guaranteed to be implemented efficiently across all implementations.

*Exceptions* None

# SETHI — Set High 22 bits of *r register* — SETHI

**Operation:** r[rd]<31:10> ← imm22
r[rd]<9:0> ← 0

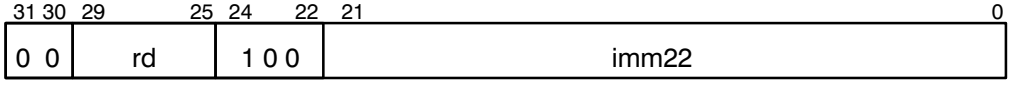
**Assembler Syntax:** sethi *const22, reg$_{rd}$*
sethi %hi *value*, *reg$_{rd}$*

**Description:** SETHI zeros the ten least significant bits of the contents of r[rd] and replaces its high-order 22 bits with *imm22*. The condition codes are not affected.

*Programming note:* SETHI 0, %0 is the preferred instruction to use as a NOP, because it will not increase execution time if it follows a load instruction.

**Traps:** none

**Format:**



automatically if you omit ***s***: It will treat **addu *d*,*s*** in exactly the same way as **addu *d*,*d*,*s***.

Unary operations like **neg, not** are always synthesized from one or more of the three-register instructions. The assembler expects a maximum of two operands for these instructions, so **negu *d*,*s*** is the same as **subu *d*,zero,*s*** and **not *d*** will be assembled as **nor *d*,zero,*d***.

Probably the most common register-to-register operation is **move *d*,*s***. The assembler implements this ubiquitous instruction as **or *d*,zero,*s***.

### 9.3.2 *Immediates: Computational Instructions with Constants*

In assembly or machine language, a constant value encoded within an instruction is called an *immediate* value. Many of the MIPS arithmetical and logical operations have an alternative form that uses a 16-bit immediate in place of **t**. The immediate value is first sign-extended or zero-extended to 32 bits; the choice of how it's extended depends on the operation, but in general arithmetical operations sign-extend and logical operations zero-extend.

Although an immediate operand produces a different machine instruction from its three-register version (e.g., **addiu** instead of **addu**), there is no need for the programmer to write this explicitly. The assembler checks whether the final operand is a register or an immediate and chooses the correct machine instruction accordingly:

```
addu    $2, $4, 64      ⇒    addiu   $2, $4, 64
```

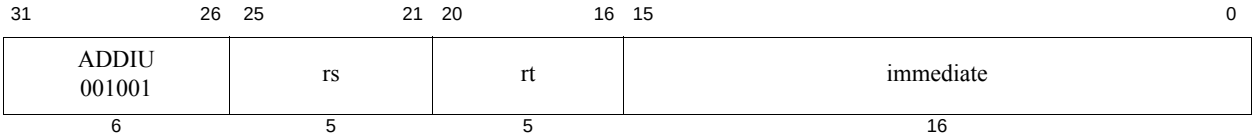
If an immediate value is too large to fit into the 16-bit field in the machine instruction, then the assembler helps out again. It automatically loads the constant into the *assembler temporary register* **at**/**$1** and then uses it to perform the operation:

```
addu     $4, 0x12345    ⇒    li      at, 0x12345
                             addu    $4, $4, at
```

Note the **li** (load immediate) instruction, which you won't find in the machine's instruction set; **li** is a heavily used macro instruction that loads an arbitrary 32-bit integer value into a register without the programmer having to worry about how it gets there—the assembler automatically chooses the best way to code the operation, according to the properties of the integer value.

When the 32-bit value lies between ±32 K, the assembler can use a single **addiu** with **$0**; when bits 16–31 are all zero, it can use **ori**; when bits 0–15 are all zero, it will use **lui**; and when none of these is possible, it will choose a **lui**/**ori** pair:

```
li      $3, -5          ⇒    addiu   $3, $0, -5
li      $4, 0x8000      ⇒    ori     $4, $0, 0x8000
```




**Format:** `ADDIU rt, rs, immediate` **MIPS32**

**Purpose:** Add Immediate Unsigned Word

To add a constant to a 32-bit integer.

**Description:** `GPR[rt] ← GPR[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:**

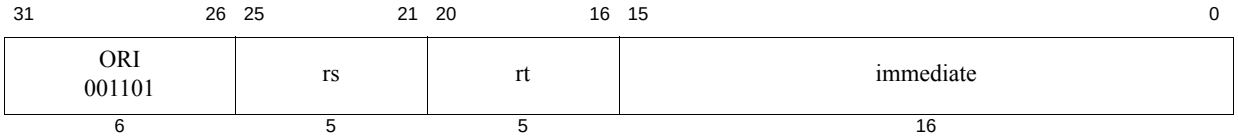
```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt] ← temp
```

**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.




**Format:** `ORI rt, rs, immediate` **MIPS32**

**Purpose:** Or Immediate

To do a bitwise logical OR with a constant.

**Description:** `GPR[rt] ← GPR[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:**

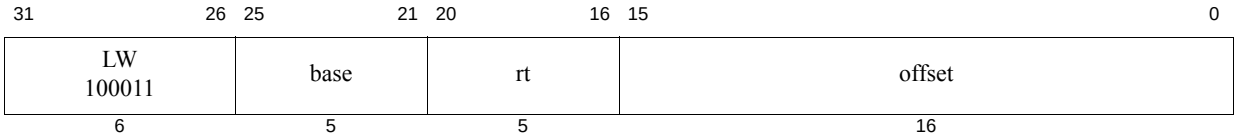
None

**Operations:**

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

**Exceptions:**

None




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

**Purpose:** Load Word

To load a word from memory as a signed value

**Description:** `GPR[rt] ← memory[GPR[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:**

Pre-Release 6: 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.

Release 6 allows hardware to provide address misalignment support in lieu of requiring natural alignment.

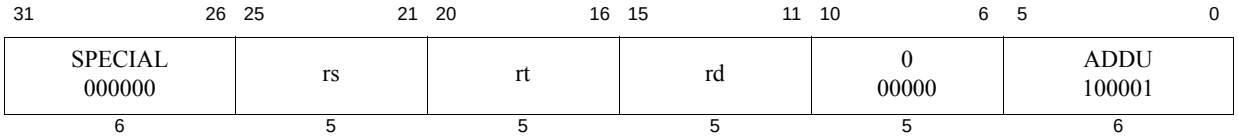
Note: The pseudocode is not completely adapted for Release 6 misalignment support as the handling is implementation dependent.

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
(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, Watch




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

**Purpose:** Add Unsigned Word

To add 32-bit integers.

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

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

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**

None

**Operation:**

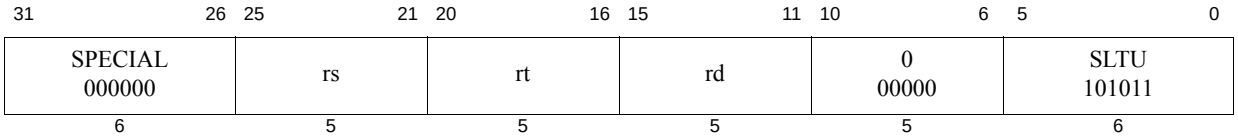
```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

**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.




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

**Purpose:** Set on Less Than Unsigned

To record the result of an unsigned less-than comparison.

**Description:** `GPR[rd] ← (GPR[rs] < GPR[rt])`

Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; 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:**

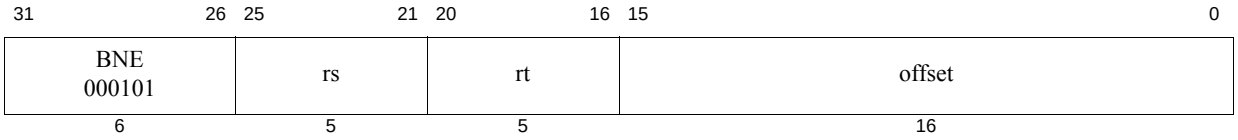
None

**Operation:**

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

**Exceptions:**

None




**Format:** `BNE rs, rt, offset` **MIPS32**

**Purpose:** Branch on Not Equal

To compare GPRs then do a PC-relative conditional branch

**Description:** `if GPR[rs] ≠ GPR[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:**

*Control Transfer Instructions (CTIs) should not be placed in branch delay slots or Release 6 forbidden slots.* CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE.

Pre-Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

Release 6: If a control transfer instruction (CTI) is executed in the delay slot of a branch or jump, Release 6 implementations are required to signal a Reserved Instruction exception.

**Operation:**

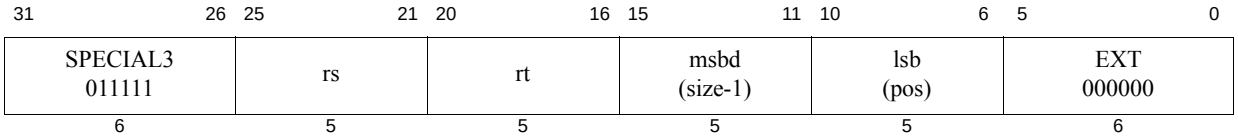
```
I:      target_offset ← sign_extend(offset || 0^2)
        condition ← (GPR[rs] ≠ GPR[rt])
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

**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) to branch to addresses outside this range.




**Format:** `EXT rt, rs, pos, size` **MIPS32 Release 2**

**Purpose:** Extract Bit Field

To extract a bit field from GPR *rs* and store it right-justified into GPR *rt*.

**Description:** `GPR[rt] ← ExtractField(GPR[rs], msbd, lsb)`

The bit field starting at bit *pos* and extending for *size* bits is extracted from GPR *rs* and stored zero-extended and right-justified in GPR *rt*. The assembly language arguments *pos* and *size* are converted by the assembler to the instruction fields *msbd* (the most significant bit of the destination field in GPR *rt*), in instruction bits **15..11**, and *lsb* (least significant bit of the source field in GPR *rs*), in instruction bits **10..6**, as follows:

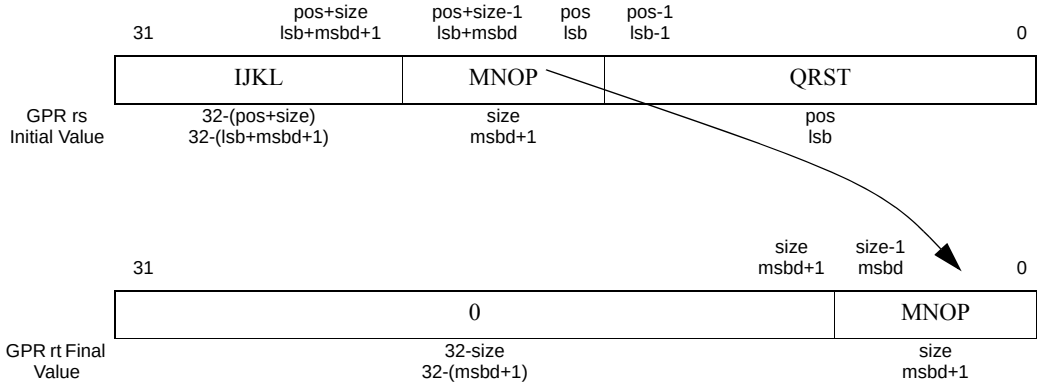
```
msbd ← size-1
lsb ← pos
```

The values of *pos* and *size* must satisfy all of the following relations:

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

Figure 3-9 shows the symbolic operation of the instruction.

**Figure 3.5 Operation of the EXT Instruction**



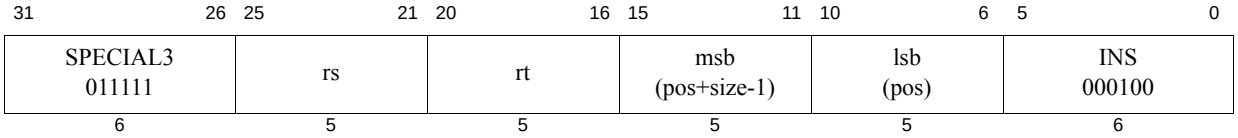

**Restrictions:**

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction exception.

The operation is **UNPREDICTABLE** if *lsb*+*msbd* > 31.

**Operation:**

```
if (lsb + msbd) > 31) then
   UNPREDICTABLE
endif
temp ← 0^(32-(msbd+1)) || GPR[rs]_(msbd+lsb..lsb)
GPR[rt] ← temp
```




**Format:** `INS rt, rs, pos, size` **MIPS32 Release 2**

**Purpose:** Insert Bit Field

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

**Description:** `GPR[rt] ← InsertField(GPR[rt], GPR[rs], msb, lsb)`

The right-most *size* bits from GPR *rs* are merged into the value from GPR *rt* starting at bit position *pos*. The result is placed back in GPR *rt*. The assembly language arguments *pos* and *size* are converted by the assembler to the instruction fields *msb* (the most significant bit of the field), in instruction bits 15..11, and *lsb* (least significant bit of the field), in instruction bits 10..6, as follows:

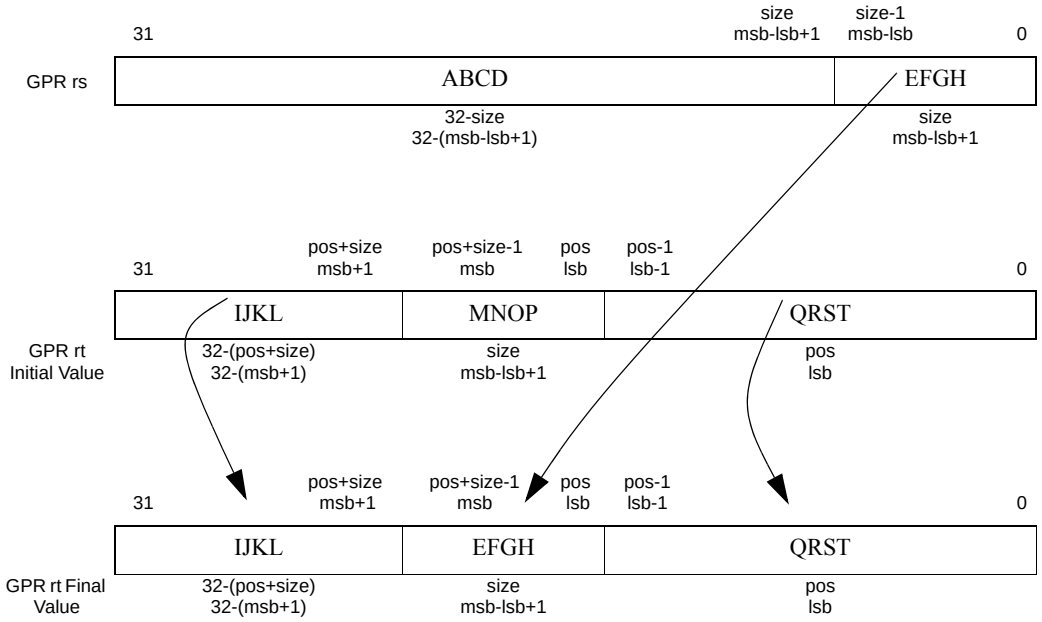
```
msb ← pos+size-1
lsb ← pos
```

The values of *pos* and *size* must satisfy all of the following relations:

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

Figure 3-10 shows the symbolic operation of the instruction.

**Figure 3.6 Operation of the INS Instruction**




**Restrictions:**

In implementations prior to Release 2 of the architecture, this instruction resulted in a Reserved Instruction exception.

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

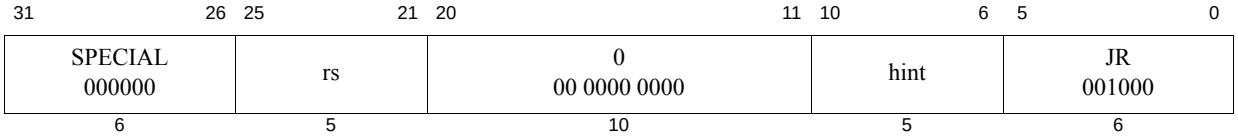
**Operation:**

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

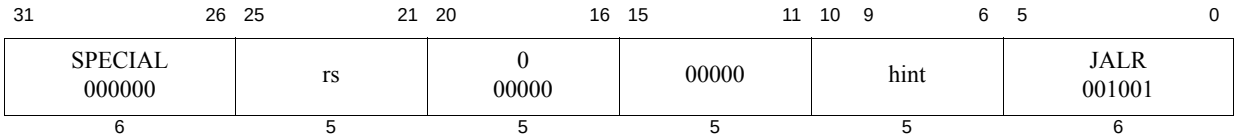
**Exceptions:**

Reserved Instruction

pre-Release 6:




Release 6:




**Format:** `JR rs`

**MIPS32**
**Assembly idiom MIPS32 Release 6**

**Purpose:** Jump Register

To execute a branch to an instruction address in a register

**Description:** `PC ← GPR[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 do not implement the MIPS16e or microMIPS ISA:*

- Jump to the effective target address in GPR *rs*. If the target address is not 4-byte aligned, an Address Error exception will occur when the target address is fetched.

*For processors that do implement the MIPS16e or microMIPS ISA:*

- Jump to the effective target address in GPR *rs*. Set the ISA Mode bit to the value in GPR *rs* bit 0. Set bit 0 of the target address to zero. If the target ISA Mode bit is 0 and the target address is not 4-byte aligned, an Address Error exception will occur when the target instruction is fetched.

**Restrictions:**

*Control Transfer Instructions (CTIs) should not be placed in branch delay slots or Release 6 forbidden slots.* CTIs include all branches and jumps, NAL, ERET, ERETNC, DERET, WAIT, and PAUSE.

Pre-Release 6: Processor operation is **UNPREDICTABLE** if a control transfer instruction (CTI) is placed in the delay slot of a branch or jump.

Release 6: If a control transfer instruction (CTI) is executed in the delay slot of a branch or jump, Release 6 implementations are required to signal a Reserved Instruction exception.

*Restrictions Related to Multiple Instruction Sets:* This instruction can change the active instruction set, if more than one instruction set is implemented.

If only one instruction set is implemented, then the effective target address must obey the alignment rules of the instruction set. If multiple instruction sets are implemented, the effective target address must obey the alignment rules of the intended instruction set of the target address as specified by the bit 0 or GPR *rs*.

For processors that do not implement the microMIPS ISA, the effective target address in GPR *rs* must be naturally-aligned. For processors that do not implement the MIPS16e ASE or microMIPS ISA, 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 MIPS16e ASE or microMIPS ISA, 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.

In release 1 of the architecture, the only defined hint field value is 0, which sets default handling of JR. In Release 2 of the architecture, bit 10 of the hint field is used to encode an instruction hazard barrier. See the JR.HB instruction description for additional information.

**Availability and Compatibility:**

*Release 6 maps JR and JR.HB to JALR and JALR.HB with rd = 0:*

Pre-Release 6, JR and JALR were distinct instructions, both with primary opcode SPECIAL, but with distinct function codes.

Release 6: JR is defined to be JALR with the destination register specifier *rd* set to 0. The primary opcode and function field are the same for JR and JALR. The pre-Release 6 instruction encoding for JR is removed in Release 6.

Release 6 assemblers should accept the JR and JR.HB mnemonics, mapping them to the Release 6 instruction encodings.

**Operation:**

```
I:  temp ← GPR[rs]
I+1:if (Config3ISA = 0) and (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.

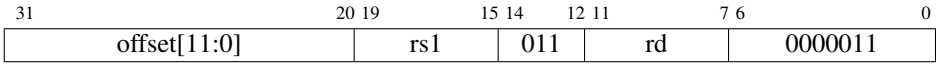
## ld rd, offset(rs1)

`x[rd] = M[x[rs1] + sext(offset)][63:0]`

*Load Doubleword*. I-type, RV64I only.

Loads eight bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*].

*Compressed forms*: **c.ldsp** rd, offset; **c.ld** rd, offset(rs1)



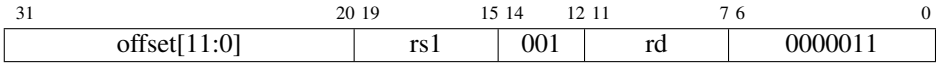

---

## lh rd, offset(rs1)

`x[rd] = sext(M[x[rs1] + sext(offset)][15:0])`

*Load Halfword*. I-type, RV32I and RV64I.

Loads two bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*], sign-extending the result.



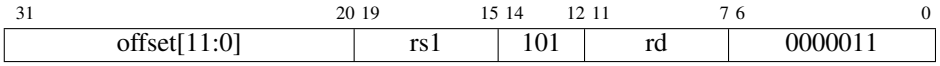

---

## lhu rd, offset(rs1)

`x[rd] = M[x[rs1] + sext(offset)][15:0]`

*Load Halfword, Unsigned*. I-type, RV32I and RV64I.

Loads two bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*], zero-extending the result.




---

## li rd, immediate

`x[rd] = immediate`

*Load Immediate*. Pseudoinstruction, RV32I and RV64I.

Loads a constant into x[*rd*], using as few instructions as possible. For RV32I, it expands to **lui** and/or **addi**; for RV64I, it's as long as **lui**, **addi**, **slli**, **addi**, **slli**, **addi**, **slli**, **addi**.

---

## lla rd, symbol

`x[rd] = &symbol`

*Load Local Address*. Pseudoinstruction, RV32I and RV64I.

Loads the address of *symbol* into x[*rd*]. Expands into **auipc** rd, offsetHi then **addi** rd, rd, offsetLo.

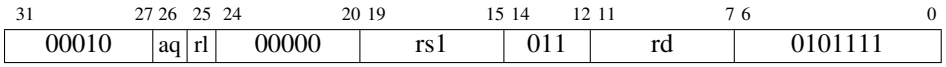
---

## lr.d rd, (rs1)

`x[rd] = LoadReserved64(M[x[rs1]])`

*Load-Reserved Doubleword*. R-type, RV64A only.

Loads the eight bytes from memory at address x[*rs1*], writes them to x[*rd*], and registers a reservation on that memory doubleword.



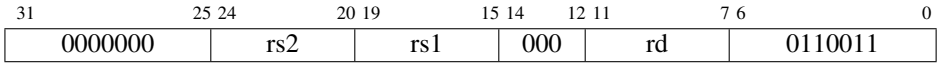

---

## add rd, rs1, rs2 `x[rd] = x[rs1] + x[rs2]`

*Add*. R-type, RV32I and RV64I.

Adds register x[*rs2*] to register x[*rs1*] and writes the result to x[*rd*]. Arithmetic overflow is ignored.

*Compressed forms*: **c.add** rd, rs2; **c.mv** rd, rs2



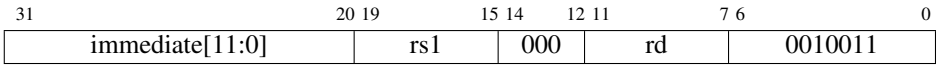

---

## addi rd, rs1, immediate `x[rd] = x[rs1] + sext(immediate)`

*Add Immediate*. I-type, RV32I and RV64I.

Adds the sign-extended *immediate* to register x[*rs1*] and writes the result to x[*rd*]. Arithmetic overflow is ignored.

*Compressed forms*: **c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm



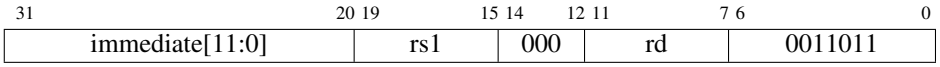

---

## addiw rd, rs1, immediate `x[rd] = sext((x[rs1] + sext(immediate))[31:0])`

*Add Word Immediate*. I-type, RV64I only.

Adds the sign-extended *immediate* to x[*rs1*], truncates the result to 32 bits, and writes the sign-extended result to x[*rd*]. Arithmetic overflow is ignored.

*Compressed form*: **c.addiw** rd, imm



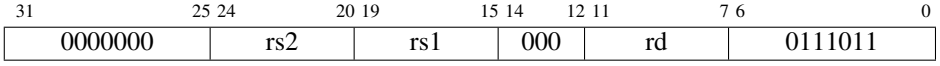

---

## addw rd, rs1, rs2 `x[rd] = sext((x[rs1] + x[rs2])[31:0])`

*Add Word*. R-type, RV64I only.

Adds register x[*rs2*] to register x[*rs1*], truncates the result to 32 bits, and writes the sign-extended result to x[*rd*]. Arithmetic overflow is ignored.

*Compressed form*: **c.addw** rd, rs2

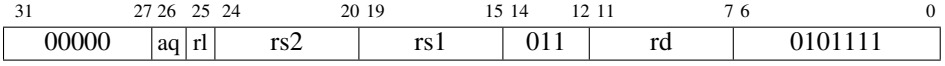



---

## amoadd.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] + x[rs2])`

*Atomic Memory Operation: Add Doubleword*. R-type, RV64A only.

Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to *t* + x[*rs2*]. Set x[*rd*] to *t*.

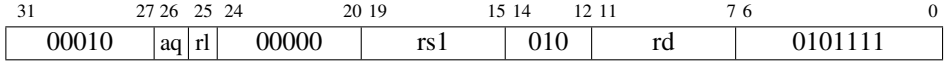



---

## lr.w rd, (rs1)

`x[rd] = LoadReserved32(M[x[rs1]])`

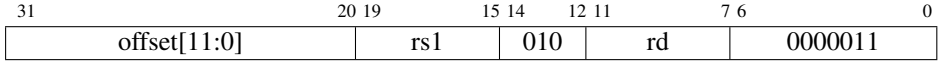
*Load-Reserved Word*. R-type, RV32A and RV64A.
Loads the four bytes from memory at address x[*rs1*], writes them to x[*rd*], sign-extending the result, and registers a reservation on that memory word.




## lw rd, offset(rs1)

`x[rd] = sext(M[x[rs1] + sext(offset)][31:0])`

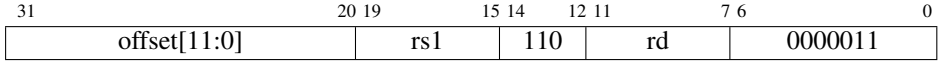
*Load Word*. I-type, RV32I and RV64I.
Loads four bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*]. For RV64I, the result is sign-extended.
*Compressed forms*: **c.lwsp** rd, offset; **c.lw** rd, offset(rs1)




## lwu rd, offset(rs1)

`x[rd] = M[x[rs1] + sext(offset)][31:0]`

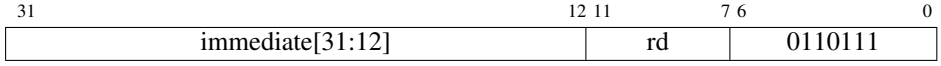
*Load Word, Unsigned*. I-type, RV64I only.
Loads four bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*], zero-extending the result.




## lui rd, immediate

`x[rd] = sext(immediate[31:12] << 12)`

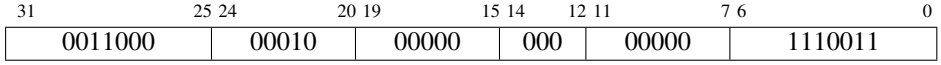
*Load Upper Immediate*. U-type, RV32I and RV64I.
Writes the sign-extended 20-bit *immediate*, left-shifted by 12 bits, to x[*rd*], zeroing the lower 12 bits.
*Compressed form*: **c.lui** rd, imm




## mret

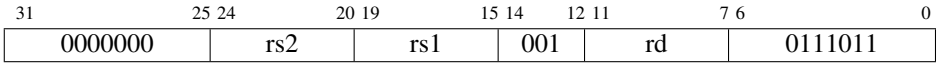
`ExceptionReturn(Machine)`

*Machine-mode Exception Return*. R-type, RV32I and RV64I privileged architectures.
Returns from a machine-mode exception handler. Sets the *pc* to CSRs[mepc], the privilege mode to CSRs[mstatus].MPP, CSRs[mstatus].MIE to CSRs[mstatus].MPIE, and CSRs[mstatus].MPIE to 1; and, if user mode is supported, sets CSRs[mstatus].MPP to 0.



## sllw rd, rs1, rs2 `x[rd] = sext((x[rs1] << x[rs2][4:0])[31:0])`

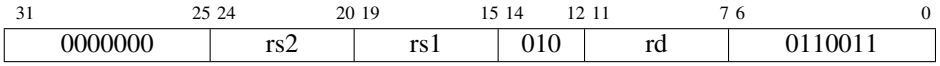
*Shift Left Logical Word*. R-type, RV64I only.
Shifts the lower 32 bits of x[*rs1*] left by x[*rs2*] bit positions. The vacated bits are filled with zeros, and the sign-extended 32-bit result is written to x[*rd*]. The least-significant five bits of x[*rs2*] form the shift amount; the upper bits are ignored.




---

## slt rd, rs1, rs2 `x[rd] = x[rs1]` $<_s$ `x[rs2]`

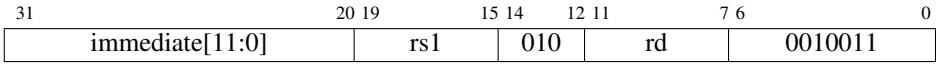
*Set if Less Than*. R-type, RV32I and RV64I.
Compares x[*rs1*] and x[*rs2*] as two's complement numbers, and writes 1 to x[*rd*] if x[*rs1*] is smaller, or 0 if not.




---

## slti rd, rs1, immediate `x[rd] = x[rs1]` $<_s$ `sext(immediate)`

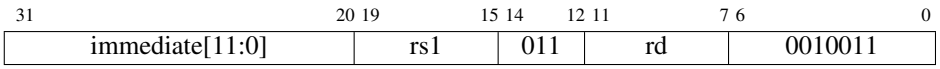
*Set if Less Than Immediate*. I-type, RV32I and RV64I.
Compares x[*rs1*] and the sign-extended *immediate* as two's complement numbers, and writes 1 to x[*rd*] if x[*rs1*] is smaller, or 0 if not.




---

## sltiu rd, rs1, immediate `x[rd] = x[rs1]` $<_u$ `sext(immediate)`

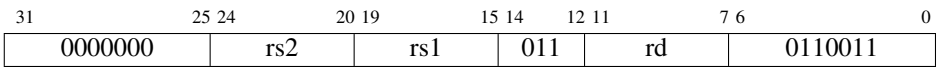
*Set if Less Than Immediate, Unsigned*. I-type, RV32I and RV64I.
Compares x[*rs1*] and the sign-extended *immediate* as unsigned numbers, and writes 1 to x[*rd*] if x[*rs1*] is smaller, or 0 if not.




---

## sltu rd, rs1, rs2 `x[rd] = x[rs1]` $<_u$ `x[rs2]`

*Set if Less Than, Unsigned*. R-type, RV32I and RV64I.
Compares x[*rs1*] and x[*rs2*] as unsigned numbers, and writes 1 to x[*rd*] if x[*rs1*] is smaller, or 0 if not.




---

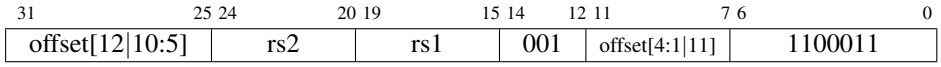
## bne rs1, rs2, offset

`if (rs1 ≠ rs2) pc += sext(offset)`

*Branch if Not Equal*. B-type, RV32I and RV64I.

If register x[*rs1*] does not equal register x[*rs2*], set the *pc* to the current *pc* plus the sign-extended *offset*.

*Compressed form*: **c.bnez** rs1, offset




---

## bnez rs1, offset

`if (rs1 ≠ 0) pc += sext(offset)`

*Branch if Not Equal to Zero*. Pseudoinstruction, RV32I and RV64I.

Expands to **bne** rs1, x0, offset.

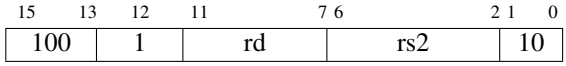
---

## c.add rd, rs2

`x[rd] = x[rd] + x[rs2]`

*Add*. RV32IC and RV64IC.

Expands to **add** rd, rd, rs2. Invalid when rd=x0 or rs2=x0.



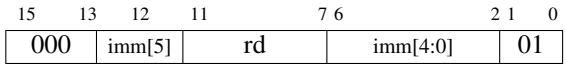

---

## c.addi rd, imm

`x[rd] = x[rd] + sext(imm)`

*Add Immediate*. RV32IC and RV64IC.

Expands to **addi** rd, rd, imm.



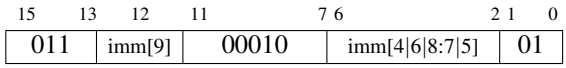

---

## c.addi16sp imm

`x[2] = x[2] + sext(imm)`

*Add Immediate, Scaled by 16, to Stack Pointer*. RV32IC and RV64IC.

Expands to **addi** x2, x2, imm. Invalid when imm=0.



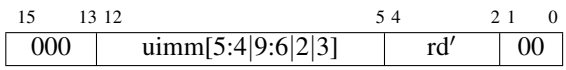

---

## c.addi4spn rd′, uimm

`x[8+rd′] = x[2] + uimm`

*Add Immediate, Scaled by 4, to Stack Pointer, Nondestructive*. RV32IC and RV64IC.

Expands to **addi** rd, x2, uimm, where rd=8+rd′. Invalid when uimm=0.

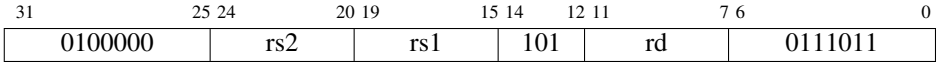



---

## sraw rd, rs1, rs2

`x[rd] = sext(x[rs1][31:0] >>$_s$ x[rs2][4:0])`

*Shift Right Arithmetic Word*. R-type, RV64I only.
Shifts the lower 32 bits of x[*rs1*] right by x[*rs2*] bit positions. The vacated bits are filled with x[*rs1*][31], and the sign-extended 32-bit result is written to x[*rd*]. The least-significant five bits of x[*rs2*] form the shift amount; the upper bits are ignored.



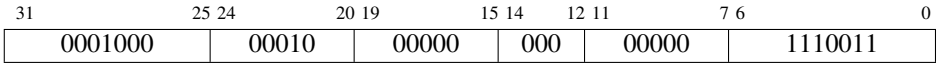
| 31–25 | 24–20 | 19–15 | 14–12 | 11–7 | 6–0 |
|---|---|---|---|---|---|
| 0100000 | rs2 | rs1 | 101 | rd | 0111011 |

---

## sret

`ExceptionReturn(Supervisor)`

*Supervisor-mode Exception Return*. R-type, RV32I and RV64I privileged architectures.
Returns from a supervisor-mode exception handler. Sets the *pc* to CSRs[sepc], the privilege mode to CSRs[sstatus].SPP, CSRs[sstatus].SIE to CSRs[sstatus].SPIE, CSRs[sstatus].SPIE to 1, and CSRs[sstatus].SPP to 0.



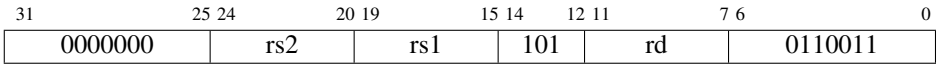
| 31–25 | 24–20 | 19–15 | 14–12 | 11–7 | 6–0 |
|---|---|---|---|---|---|
| 0001000 | 00010 | 00000 | 000 | 00000 | 1110011 |

---

## srl rd, rs1, rs2

`x[rd] = x[rs1] >>$_u$ x[rs2]`

*Shift Right Logical*. R-type, RV32I and RV64I.
Shifts register x[*rs1*] right by x[*rs2*] bit positions. The vacated bits are filled with zeros, and the result is written to x[*rd*]. The least-significant five bits of x[*rs2*] (or six bits for RV64I) form the shift amount; the upper bits are ignored.



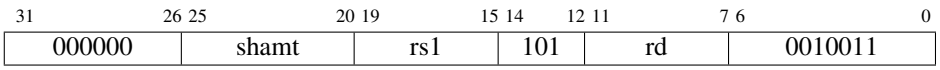
| 31–25 | 24–20 | 19–15 | 14–12 | 11–7 | 6–0 |
|---|---|---|---|---|---|
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 |

---

## srli rd, rs1, shamt

`x[rd] = x[rs1] >>$_u$ shamt`

*Shift Right Logical Immediate*. I-type, RV32I and RV64I.
Shifts register x[*rs1*] right by *shamt* bit positions. The vacated bits are filled with zeros, and the result is written to x[*rd*]. For RV32I, the instruction is only legal when *shamt*[5]=0.
*Compressed form*: **c.srli** rd, shamt



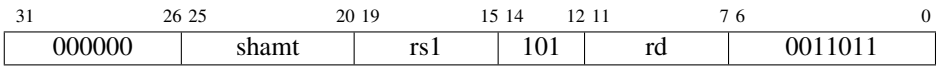
| 31–26 | 25–20 | 19–15 | 14–12 | 11–7 | 6–0 |
|---|---|---|---|---|---|
| 000000 | shamt | rs1 | 101 | rd | 0010011 |

---

## srliw rd, rs1, shamt

`x[rd] = sext(x[rs1][31:0] >>$_u$ shamt)`

*Shift Right Logical Word Immediate*. I-type, RV64I only.
Shifts the lower 32 bits of x[*rs1*] right by *shamt* bit positions. The vacated bits are filled with zeros, and the sign-extended 32-bit result is written to x[*rd*]. The instruction is only legal when *shamt*[5]=0.



| 31–26 | 25–20 | 19–15 | 14–12 | 11–7 | 6–0 |
|---|---|---|---|---|---|
| 000000 | shamt | rs1 | 101 | rd | 0011011 |

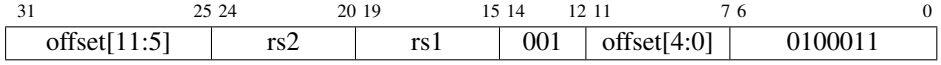
---

## sh rs2, offset(rs1)

`M[x[rs1] + sext(offset)] = x[rs2][15:0]`

*Store Halfword*. S-type, RV32I and RV64I.

Stores the two least-significant bytes in register x[*rs2*] to memory at address x[*rs1*] + *sign-extend*(*offset*).



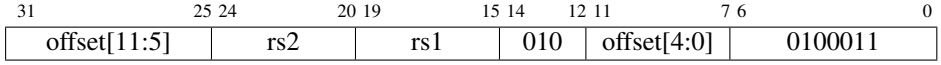

## sw rs2, offset(rs1)

`M[x[rs1] + sext(offset)] = x[rs2][31:0]`

*Store Word*. S-type, RV32I and RV64I.

Stores the four least-significant bytes in register x[*rs2*] to memory at address x[*rs1*] + *sign-extend*(*offset*).

*Compressed forms*: **c.swsp** rs2, offset; **c.sw** rs2, offset(rs1)

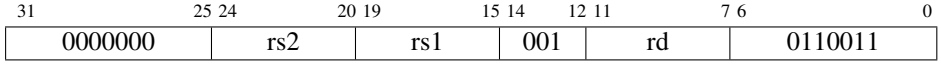



## sll rd, rs1, rs2

`x[rd] = x[rs1] << x[rs2]`

*Shift Left Logical*. R-type, RV32I and RV64I.

Shifts register x[*rs1*] left by x[*rs2*] bit positions. The vacated bits are filled with zeros, and the result is written to x[*rd*]. The least-significant five bits of x[*rs2*] (or six bits for RV64I) form the shift amount; the upper bits are ignored.



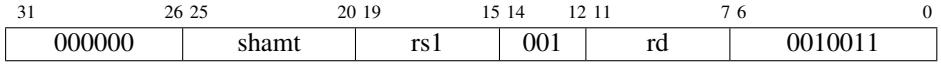

## slli rd, rs1, shamt

`x[rd] = x[rs1] << shamt`

*Shift Left Logical Immediate*. I-type, RV32I and RV64I.

Shifts register x[*rs1*] left by *shamt* bit positions. The vacated bits are filled with zeros, and the result is written to x[*rd*]. For RV32I, the instruction is only legal when *shamt*[5]=0.

*Compressed form*: **c.slli** rd, shamt

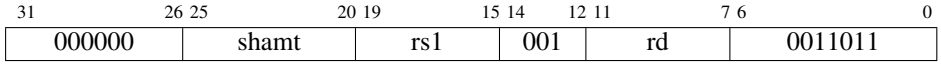



## slliw rd, rs1, shamt

`x[rd] = sext((x[rs1] << shamt)[31:0])`

*Shift Left Logical Word Immediate*. I-type, RV64I only.

Shifts x[*rs1*] left by *shamt* bit positions. The vacated bits are filled with zeros, the result is truncated to 32 bits, and the sign-extended 32-bit result is written to x[*rd*]. The instruction is only legal when *shamt*[5]=0.



## neg rd, rs2

`x[rd] = -x[rs2]`

*Negate*. Pseudoinstruction, RV32I and RV64I.
Writes the two's complement of x[*rs2*] to x[*rd*]. Expands to **sub** rd, x0, rs2.

---

## negw rd, rs2

`x[rd] = sext((-x[rs2])[31:0])`

*Negate Word*. Pseudoinstruction, RV64I only.
Computes the two's complement of x[*rs2*], truncates the result to 32 bits, and writes the sign-extended result to x[*rd*]. Expands to **subw** rd, x0, rs2.

---

## nop

*Nothing*

*No operation*. Pseudoinstruction, RV32I and RV64I.
Merely advances the *pc* to the next instruction. Expands to **addi** x0, x0, 0.

---

## not rd, rs1

`x[rd] = ∼x[rs1]`

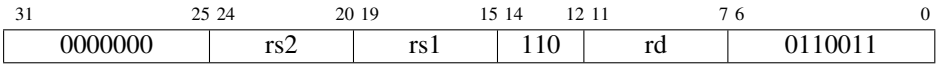
*NOT*. Pseudoinstruction, RV32I and RV64I.
Writes the ones' complement of x[*rs1*] to x[*rd*]. Expands to **xori** rd, rs1, -1.

---

## or rd, rs1, rs2

`x[rd] = x[rs1] | x[rs2]`

*OR*. R-type, RV32I and RV64I.
Computes the bitwise inclusive-OR of registers x[*rs1*] and x[*rs2*] and writes the result to x[*rd*].
*Compressed form*: **c.or** rd, rs2

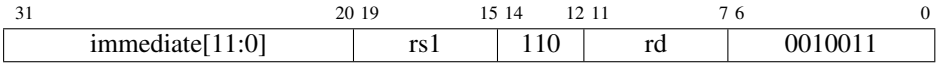



---

## ori rd, rs1, immediate

`x[rd] = x[rs1] | sext(immediate)`

*OR Immediate*. I-type, RV32I and RV64I.
Computes the bitwise inclusive-OR of the sign-extended *immediate* and register x[*rs1*] and writes the result to x[*rd*].




---

## rdcycle rd

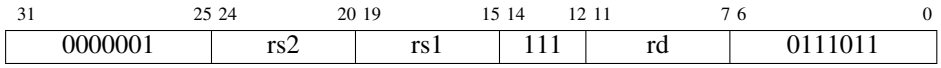
`x[rd] = CSRs[cycle]`

*Read Cycle Counter*. Pseudoinstruction, RV32I and RV64I.
Writes the number of cycles that have elapsed to x[*rd*]. Expands to **csrrs** rd, cycle, x0.

---

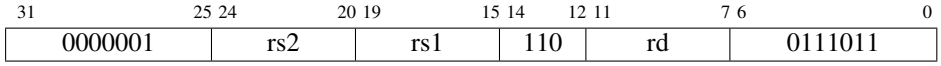
## remuw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] %u x[rs2][31:0])`

*Remainder Word, Unsigned*. R-type, RV64M only.
Divides the lower 32 bits of x[*rs1*] by the lower 32 bits of x[*rs2*], rounding towards zero, treating the values as unsigned numbers, and writes the sign-extended 32-bit remainder to x[*rd*].




## remw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] %s x[rs2][31:0])`

*Remainder Word*. R-type, RV64M only.
Divides the lower 32 bits of x[*rs1*] by the lower 32 bits of x[*rs2*], rounding towards zero, treating the values as two's complement numbers, and writes the sign-extended 32-bit remainder to x[*rd*].

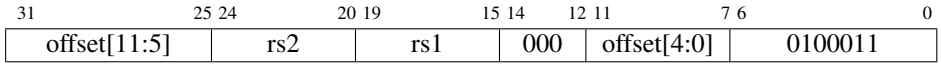



## ret `pc = x[1]`

*Return*. Pseudoinstruction, RV32I and RV64I.
Returns from a subroutine. Expands to **jalr** x0, 0(x1).

## sb rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][7:0]`

*Store Byte*. S-type, RV32I and RV64I.
Stores the least-significant byte in register x[*rs2*] to memory at address x[*rs1*] + *sign-extend*(*offset*).




## sc.d rd, rs2, (rs1) `x[rd] = StoreConditional64(M[x[rs1]], x[rs2])`

*Store-Conditional Doubleword*. R-type, RV64A only.
Stores the eight bytes in register x[*rs2*] to memory at address x[*rs1*], provided there exists a load reservation on that memory address. Writes 0 to x[*rd*] if the store succeeded, or a nonzero error code otherwise.