IMPORTANT 68000 INSTRUCTIONS

AND - Logical AND

This instruction logically ANDs bits in the source operand with the same number of bits in the destination operand were the result is left. The number of bits can be 8, 16, or 32 as per the .B, .W, or .L opcode suffix. One or both operands must be a data register. Syntax: AND Dn, Dn or AND Dn, memory or AND memory, Dn.

BRA - Branch Always

This instruction changes the program counter register so execution continues at a different point in the program code. The destination of the jump is specified as a signed displacement to the program counter. This signed displacement can be an 8- or 16- bit quantity. With a bit 8-bit quantities, this allows branches of +126 to -128 bytes; 16-bit quantities can specify branches of +32766 to -32768 bytes. The value of the program counter when the displacement is added is taken to be the first word after the BRA opcode. This is the actual opcode address plus two. Normally an assembler will assume a 16-bit quantity as the displacement, but if an opcode suffix of .S is appended th the BRA, a short 8-bit displacement will be used instead. Syntax: BRA label (16-bit displacement) or BRA.S label (8-bit displacement). Flags affected: None.

BSR - Branch to Subroutine

This instruction causes control to be passed unconditionally to the specified program counter displacement as in the BRA opcode. However, before the branch is made, the address of the opcode following the BSR is saved on the stack so return can later be made to that address to continue processing at that point. This is achived as follows: 1. The 24-bit address following the opcode is pushed on the stack as two words. 2. The program counter is loaded with its new value and processing continues at the new address. Syntax: BSR label (16-bit displacement) or BSR.S label (8-bit displacement). Flags affected: None.

Bcc - Branch Conditionally

Other variants of the BRA instruction allow a branch to be made only if a certain condition is met in the condition code register. These Bcc instructions can be divided into three different categories. Whether or not this instruction is actually executed depends on the required condition, which is verified by means of the flags. A minus sign before a flag indicates that it must be cleared to satisfy the condition. Logical operations are indicated with "*" for AND and "/" for OR. Branches depending on flag status:

BCC	- Branch if carry clear -C
BCS	- Branch if carry set C
BNE	- Branch if zero clear -Z
BEQ	- Branch if zero set Z
BVC	- Branch if overflow clear -V
BVS	- Branch if overflow set V
BPL	- Branch if negative clear -N
BMI	- Branch if negative set N Branches after unsigned comparison:
BHI	- Branch if higher than -C * -Z
BHS	- Branch if higher than or same as
BLO	- Branch if lower than
BLS	- Branch if lower than or same as C / Z
BEQ	- Branch if equal to Z
BNE	- Branch if not equal to -Z Branches after signed comparison:
BGT	- Branch if greater than N * V * -Z / -N * -V * -Z
BGE	- Branch if greater than or equal to N * V / -N * -V
BLT	- Branch if less than N * -V / -N * V
BLE	- Branch if less than or equal to Z / N * -V / -N * V
BEQ	- Branch if equal to Z
BNE	- Branch if not equal to -Z Syntax: Bcc label (16-bit displacement) or Bcc.S label (8-bit displacement). Flags affected: None.

CMP - Compare

This instruction compares two operands and sets flags in the condition code register according to the result. Except for the Extend flag, the flags are set as if the source operand were subtracted from the destination. However, the result of this subtraction is not actually retained so the destination remains unchanged. The information about the comparison that is stored in the condition flags can then be acted upon by a Bcc-instruction. CMP may be used with byte, word, or long word source operands. Note that although any addressing mode can be used to specify the source operand, an address register can only be used if a word or long word comparison is performed. Syntax: CMP address, Dn.

JMP - Jump

This instruction allows execution of the program to be transferred anywhere within the entire addressing space of the 68000. The jump address can be specified using any memory mode except register indirect with postincrement or predecrement. It should be borne in mind that an absolute address specified in a jump instruction will load the program counter immediately with that value. Because absolute addresses are not position- independent. If the program is moved in memory it has to be reassembled if the label is contained within the program. The JMP instruction with an absolute address is more properly used for jumps to static locations such as ROM routines. To keep the jump position-independent, a program-counter-relative address should be specified. Syntax: JMP address where address is any addressing mode except (An)+ and -(An). Flags affected: None.

JSR - Jump to Subroutine

This instruction allows control to be redirected in a similar manner to the JMP instruction; however, before the jump is made, the address of the following opcode is pushed onto the stack. (See BSR for a description of the stack save process.) Thus a subroutine can perform a task, and when it finishes, it can execute a Return instruction to return to the address saved on the stack. As far as the destination address of the JSR instruction is concerned, the same caveats apply as for the JMP instruction. Absolute addresses, even as labels inside your program, should be avoided where possible to avoid a program which is not position-independent. Unless using such things as ROM routines or memory-mapped hardware locations, which have absolute addresses, use program counter relative or address register indirect addressing. Syntax: JSR address where address is any addressing mode except (An)+ and -(An). Flags affected: None.

LEA - Load Effective Address

This instruction provides a simple way of loading any address register with the address resulting from nearly any addressing mode. Only two such modes are excluded from the list of possibilities. Due to the fact that address register indirect with postincrement or predecrement represent a dynamically increasing or decreasing addresses, these two modes cannot be used with LEA. But any other address, no matter how complicated, (including address register indirect with displacement and index) can be loaded into the specified address register. This saves performing the address arithmetic within the program. The processor will automatically take the same value as the calculated address - or in other words "the effective address". Only address registers can be used with this instruction, and the destination address register is loaded with a 32-bit long value even though the address will only be 24 bits long. Syntax: LEA address, An where address is any addressing mode except postincrement and predecrement. Flags affected: None.

MOVEA - Move Address

This specialized version of the MOVE command is used when the destination is an address register. The instruction only allows transfers of 16 or 32 bits in length. Byte transfers are not allowed with an address register as the destination. Also note that unlike the normal MOVE command, no flag bits are affected. Syntax: MOVEA source, An where source is any addressing mode. Flags affected: None.

MOVEM - Move Multiple

This variation of the MOVE instruction allows multiple registers to be saved and restored using a single operation. Any of the 16 data or address registers can be moved this way. At the source code level, the registers chosen to be saved or restored are specified to the assembler in a list separated by slashes. Thus, to save D0, D3 and A1, the register list would be specified as D0/D3/A1. If a consecutive number of registers are included in the list, they can be idetified as such by a hyphen. So to save D0, D1, D2, D5 and A1, the register list can be specified as D5/D0-D2/A1. Notice that the order of register between slashes is unimportant; however, when the 68000 saves these registers, it does so in a definite order. It also retrives them in a definite (but opposite) order, so that if the registers are saved on the stack, they can be pulled off in a typical stack-like fashion (that is, last in first out). The order in which the 68000 saves registers is first A7 through A0, and then D7 through D0. Then in reverse order, D0 is restored first, and restoration continues all the way through to A7. As the registers are most often saved in a stack formation, normally an address register is chosen to point to that stack. Then a predecrement addressing mode is used to push the registers down onto the stack. Conversely, when registers are being restored, a postincrement addressing mode is used. As an example, to save two registers at a memory location pointed to by A3, the instruction MOVEM D1/A1, -(A3) might be used. To restore them at another point in program, MOVEM (A3)+, D1/A1 would be correct. Note that registers can only be saved as words or long words. If they are saved as 16-bit words, then when they are restored, the upper half of the register is automtically sign-extended so that bit 15 fills the upper half of the register. Although less memory is used to save registers this way, such a loss of control of the upper 16 bits of every restored register may present problems unless you remain acutely aware of the possible corruption of an upper register half. the MOVEM instruction may be used with addressing modes other than predecrement and postincrement. By specifying other addressing modes as the source or destination of the multiple transfer, registers can be saved and restored in ascending locations in memory. The same register order is used, but they will not be stacked in at last in, first out order. Note that no flags are affected by this operation. Thus a subroutine can affect the condition code register, restore multiple registers with MOVEM, and return with the condition code register still intact. Syntax: MOVEM register list, destination address or MOVEM source address, register list or MOVEM register list, -(An) or MOVEM (An)+, register list. Flags affected: None.

MOVEP - Move Peripheral Data

This variation of the MOVE instruction is used to transfer data between the 68000 and certain peripherals. As input and output on the 68000 is memory-mapped, certain addresses will not actually be memory at all but will instead be external devices. The 68000 has a special design to allow it to use the many hardware interfaces that exist for 8-bit microprocessors, in particular the 6800. What this means to the programmer is that if a peripheral is interfaced to the 68000 and is normally addressed at consecutive address on an 8-bit microprocessor, it will be addressed at every other address on the 68000 due to the design of its peripheral hardware bus. Thus the MOVEP instruction was included to address such peripherals. A long word of data from a data register can be transferred high byte first to every alternate memory (pheripheral) address with a single MOVEP to the first address. This also works the other way round in that every other word will be addressed starting with the source address specified in the MOVEP instruction. Only word or long word transfers are allowed. (A normal MOVE would be used for a single byte.) The only addressing mode allowed to specify the memory location is address register indirect with displacement, and only a data register can be used as the other operand. Syntax: MOVEP Dn, disp(An) or MOVEP disp(An), Dn where disp is a 16-bit displacement.

MOVEQ - Move Quick

This variation on the MOVE instruction allows the quick loading of a data register with an immediate value. The MOVEQ variant works like a MOVE immediate value to the data register except that MOVEQ is much faster and only takes up two bytes in memory. The immediate value that is moved into a data register can only be in the range -128 to +127. This value is sign- extended into the entire 32 bits of the data register, so it is always of type .L despite the small immediate value. As this instruction works so fast, it is quicker to clear a data register with a MOVE #0, Dn than to use CLR Dn. MOVEQ cannot, however, be used with address registers or numbers larger than eight bits. Syntax: MOVEQ #imm 8-bit signed value, Dn. Flags affected: The Negative and Zero flags are set as per the immediate value; the Overflow and Carry flags are reset to zero, and the Extent flag is unaffected.

NOP - No Operation

This instruction is a do-nothing opcode. It is used during program developement to leave room in a section of code. This space can be patched with machine-code instruction as necessary during debugging to test new routines within a previously written machine code level by substituting NOP instruction for the instructions and operands. Syntax: NOP. Flags affected: None.

NOT - Logical NOT

This instruction takes its operand and simply inverts all of its bits. (Each one-bit becomes zero and each zero-bit becomes one.) The operand can either be in a data register or memory and can be 8, 16, or 32 bits in length as per the .B, .W, or .L operand suffix. Syntax: NOT Dn or NOT address where address is any memory addressing mode except program counter relative.

OR - Logical OR

The OR opcode performs a logical OR operation. A number of bits in the source operand are ORed with the same number of bits in the destination operand where the result is left. The number of bits can be 8, 16, or 32 as the .B, .W, or .L opcode suffix. One or both operands must be a data register. Syntax: OR Dn, Dn or OR Dn, address or OR address, Dn where address is any addressing mode with the proviso that program counter relative may not be used as destination.

SUB - Subtract Binary

The SUB instruction subtracts the source operand from the destination operand with the result appearing in the destination. It is possible to subtract bytes, words, or long words with this opcode by appending .B, .W, or .L to the mnemonic. Either the source or destination (or both) must be a data register. The source operand can be any memory location or data register, and the destination operand can also be any memory location or data register. Syntax: SUB Dn, Dn or SUB address, Dn or SUB Dn, address Flags affected: The Extend, Negative, Zero, Overflow, and Carry flags are all affected as per the result of the subtraction.

TRAP - Software Trap

This instruction causes a trap to occur in the same manner as if it had been caused by a hardware-detected condition. The processor will jump to one of the 16 special addresses set up in the first 1024 bytes of memory. The actual address that will be jumped to is determined by the operand supplied with the opcode. This will be a number from 0 to 15. The software trap vectors are 32-bit addresses stored in memory starting at location #128. Before the specified vector is taken, the status register and program counter are pushed onto the stack to facilitate a return via an RTE instruction. Syntax: TRAP #imm where #imm is an immediate value from 0 to 15. Flags affected: None

TST - Test Operand

This instruction causes the processor to scan the operand and set the condition code flags according to the contents. The operand can be 8, 16, or 32 bytes as specified in the .B, .W, or .L opcode modifier. No registers other than the condition code register are changed. The operand can be either a data register or a memory location. Syntax: TST Dn or TST address where address is any addressing mode except program counter relative.

Greetings to Italico for the instructions