

CSCE 313

Embedded Systems Programming

2003/2/20

Spring 2003 – Lecture 15

M68000: Instruction Codes

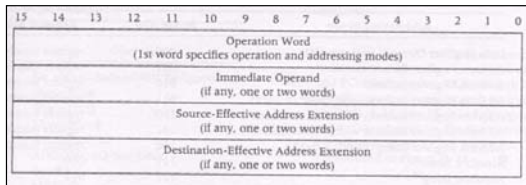
Figures: MacKenzie © 1995 Prentice Hall Publishers, Inc.

Lectures 15 & 16 - Outline

- Objectives.
 - ✓ Understanding these additional elements of the 68000 architecture:
 - ✦ *Instruction formats* – going from the textual instruction specification to the actual hexadecimal or binary sequence of bits read by the processor (hand assembly) and reading machine codes and determining the instruction (disassembly). Appendices B & C.
 - ✦ *Instruction timing* – knowing how many clock cycles a given instruction will take to execute, with its selected addressing modes for its operands. This includes the number of cycles to fetch it from memory, decode it, fetch the operands from memory, decode them, and actually execute what is dictated in the instruction. Appendix D.
- Importance.
 - ✓ Debugging programs
 - ✦ Often, you may be required to read the actual contents of memory, or watch the binary bits scroll across the screen of a logic analyzer. You need to have an understanding of the instruction format and bit encoding, to be able to recognize the instructions.
 - ✓ Understanding the behavior of the processor
 - ✦ Often, you can determine a lot about the internal workings of the CPU by looking at how the instruction set is designed. Some processors have well-design, “clean” instruction sets that are intuitive; others have instructions that have different ways to interpret the bits, depending on the type of instruction. Usually, instruction set designers and processor architects will attempt to save “resources” (cycles, register bits, etc.) by “overloading” the meaning of bits, making it important for the programmer to understand the usage to know the essence of how the processor will behave given a specific instruction bit set.
 - ✓ Programming the processor most efficiently
 - ✦ Often, there are many ways to write a “correct” program, but the issue in embedded systems is often how “tight” you can make the code; it should be correct *and* consume the least amount of resources as possible (memory space, register bits, clock cycles, battery power, etc.).

M68K– General Layout of an Instruction

Source: Figure 3-19. MacKenzie © 1995 Prentice Hall Publishers



Bits 15 through 12	Operation
0000	Bit Manipulation/MOVEP/Immediate
0001	Move Byte
0010	Move Long
0011	Move Word
0100	Miscellaneous
0101	ADDQ/SUBQ/ScC/DBcc
0110	Bcc/BSR
0111	MOVEQ
1000	OR/DIV/SBCD
1001	SUB/SUBX
1010	(Unassigned)
1011	CMP/EOR
1100	AND/MUL/ABCD/EXG
1101	ADD/ADDX
1110	Shift/Rotate
1111	(Unassigned)

Source: Table 3-13. MacKenzie © 1995 Prentice Hall Publishers



Memory Organization

- ✓ A program "segment" has a sequence of instructions located sequentially in memory.
- ✓ The PC Register points to the first word of the instruction to be read from memory to be decoded and executed.
- ✓ A given instruction can be from 1 to 5 16-bit words in length, depending on the number of operands, source & destination of these operands, and how the effective address is specified for the operands.

Instruction Layout

- ✓ A given instruction is organized as consecutive 16-bit words in memory (in ascending order, starting with the Operation Word).
- ✓ The figure 3-19 in MacKenzie (at left) illustrates the different components of a possible instruction; **note** that it doesn't really represent any specific instruction in memory. So, a given instruction could include any of the 3 fields after the Opcode word.
- ✓ For example, a **MOVE.W #FF44, D0** would have the following: Opcode Word + Immediate operand. So, it would be a two-word instruction.

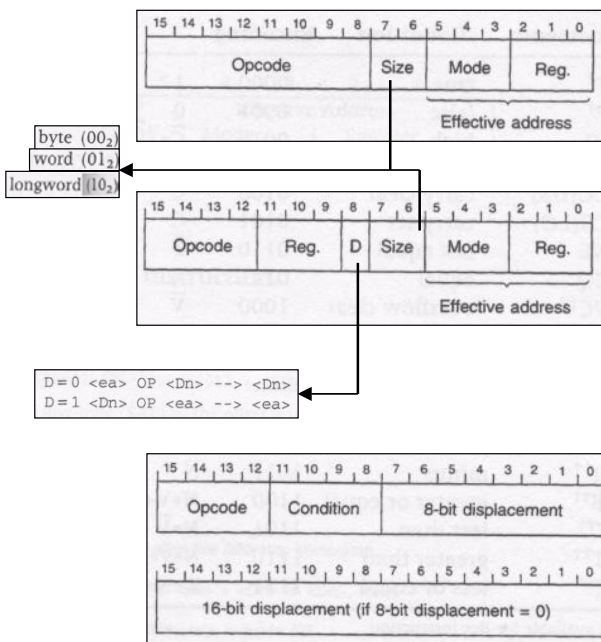
Instruction Category

- ✓ We use the most significant 4-bits (15 down to 12) to determine the basic "category" of the instruction.
- ✓ From this, we can break apart the remainder of the instruction to determine its meaning.

© 2002 Dr. James P. Davis Page 3

M68K– Instruction Categories & Formats

Source: MacKenzie, Fig 3-20 © 1995 Prentice Hall Publishers



Single Operand Instructions

- ✓ Examples: CLR.L D0; TST.B D1...etc.
- ✓ Information in the general format: (1) **Opcode** (8-bits), (2) **Size** (Byte, Word Longword, in 2-bits, but value '11' is reserved), (3) **Operand Addressing Mode** (3-bits, encoding 8 possible modes, except when it is '111', where it uses the Register Mode bits to encode an additional 5 mode choices), and (4) **Register Bits** (3-bits, to encode 1 of the 8 address, A0-A7, or 8 data, D0-D7, registers, except when Mode = '111', when the bits in this field are used to encode additional Effective Address or Mode information).

Double Operand Instructions

- ✓ Examples: MOVE.W \$9000, D0; ADD.L D0, D1...etc.
- ✓ Most 2-operand instructions require one operand to be in a register. The "D" bit (bit 8) indicates whether the register is the source or the destination operand in the instruction. Note: Both operands could be stored in registers, but the D bit is used for most general case.

Conditional Branch Instructions

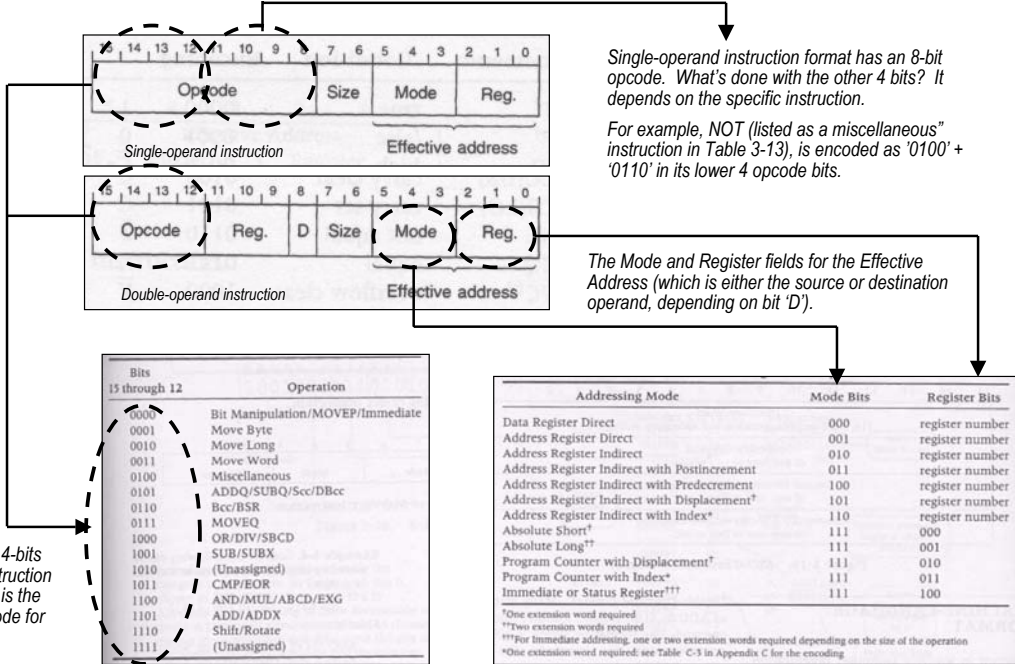
- ✓ Three instruction types: Branch on condition (Bcc), Set by Condition Code (ScC), and Decrement and Branch on condition (DBcc).
- ✓ Examples: BLT \$8040, ST \$9004, DBLE \$8020 (**Note**: most of these would follow one of the Compare CMP instructions, or some other instruction, e.g. SUB, that sets the bits in the CCR).
- ✓ The 8-bit displacement allows "short" branches to be made, if the effective address can be represented in 8-bits; otherwise, this set of 8 bits is zeroed, and the full 16-bit displacement is stored in the second instruction word.



© 2002 Dr. James P. Davis Page 4

M68K Instructions – Format Decoding-1

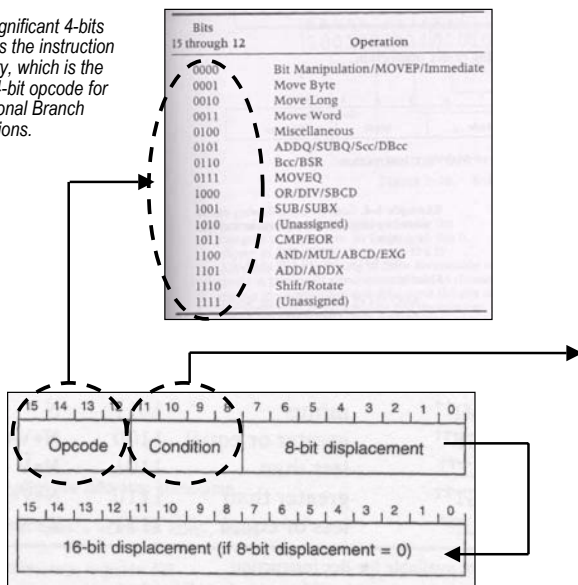
Source: MacKenzie © 1995 Prentice Hall Publishers



M68K Instructions – Format Decoding-2

Figures: MacKenzie © 1995 Prentice Hall Publishers

Most-significant 4-bits encodes the instruction category, which is the whole 4-bit opcode for Conditional Branch instructions.



Next 4-bits (11 down to 8) encodes the Condition category, which is indicated in Table 3-12 as the Encoding of the bits. The instruction mnemonic is added to B<>, S<> or DB<> to form the particular instruction which, when assembled, is encoded as shown below, having the indicated meaning. For example, B+NE for BNE for "branch if not equal to zero".

You'll need to memorize these mnemonics, so you'll recognize these instructions when you see them, and know their meaning.

Mnemonic	Condition	Encoding	Test
T [†]	true	0000	1
F [†]	false	0001	0
HI	high	0010	$\overline{C} \cdot \overline{Z}$
LS	low or same	0011	$C \cdot Z$
CC(HS)	carry clear	0100	\overline{C}
CS(LO)	carry set	0101	C
NE	not equal	0110	\overline{Z}
EQ	equal	0111	Z
VC ^{††}	overflow clear	1000	\overline{V}
VS ^{††}	overflow set	1001	V
PL ^{††}	plus	1010	\overline{N}
MI ^{††}	minus	1011	N
GE ^{††}	greater or equal	1100	$N \cdot V + \overline{N} \cdot \overline{V}$
LT ^{††}	less than	1101	$N \cdot \overline{V} + \overline{N} \cdot V$
GT ^{††}	greater than	1110	$N \cdot V \cdot \overline{Z} + \overline{N} \cdot \overline{V} \cdot \overline{Z}$
LE ^{††}	less or equal	1111	$Z \cdot N \cdot \overline{V} + \overline{N} \cdot V$

[†] Not available for Bcc instruction
^{††} Two's complement arithmetic, cleared numbers

The Displacement is used to calculate either the "jump" address or the location that gets cleared or set (with S<> instruction). If the Displacement field is #0, then CPU will fetch the next word and use it to calculate a 16-bit offset address.

