UNIVERSITY of WASHINGTON

CSS 422 Hardware and Computer Organization

68000 instruction decomposition

Ver. 1

Professor: Wooyoung Kim

The slides are re-produced by the courtesy of Dr. Arnie Berger

Before we start

- Turn off your phone or at least sound off
- Do not open your laptop, or turn on your computer unless you are asked to do
  - Computers for class notes ONLY. Otherwise, I will consider you are distracting the class, and ask you to leave
  - I tend to ask the students with the laptop open, just to see if you are following my lecture
- No talk with your friends unless I ask to discuss
- If I think you are distracting this class, I will ask you to leave the class

Warning: You will see this slide in EVERY class, because you keep forgetting to do so

Topic

- 68000 Instruction Set Architecture
  - Fundamental Effective Addressing Mode
  - Instruction Set Decomposition

  - 68K manual
  - Chapter 8 (Berger)
  - Chapter 2, 3 (Clements)

68000 instruction format

- Instruction set in a memory
  - Assembly codes are assembled into binary numbers and store into a memory
  - One instruction code can be up to 5 words: 80 bits

  For example,
  - MOVE.B #14, D0 (human code) → 103C 000E (machine code in hex)
  - MOVE.W $0010AA00,$00103000 → 33F9 0010AA00 00103000
- The first 16-bit word ($103C, $33F9) of an instruction is called the ***Opcode Word***
  - Opcode word contains all of the information needed to decode the *rest of the instruction*
  - Contains the opcode (what to do) and *effective address fields (EA)*
- The complete instruction in memory must contain the op code word and *may contain additional words to complete the instruction*
  - One instruction may require up to 5 words of memory

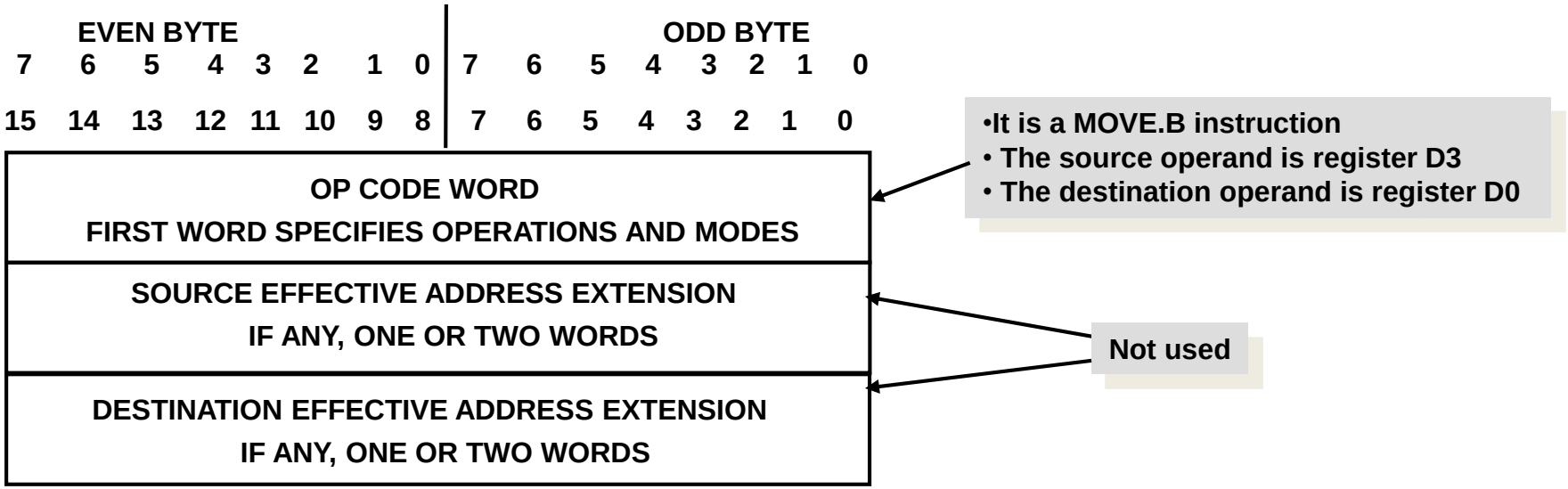
Effective Address

- The effective address, EA, determines how the operands of an instruction are to be accessed by the processor
- Different types of EA's determine the *addressing modes* of the architecture
- Consider the form of the ***opcode word*** shown below:

- **MOVE.W** instruction: Move the ***word*** contents at the memory location specified by the **src EA** to the memory location specified by the **dst EA**
  - This information is encoded in the 16 bits of the op code word
    - OPCODE/SIZE = DB15-DB12 (MOVE.W)
    - Destination Effective Address = DB11-DB6
    - Source Effective Address = DB5-DB0
- May have to retrieve **additional words** from memory to complete the instruction
  - Note: *Not all instructions have the same form as the MOVE instruction*
  - *Refer the 68K manual*

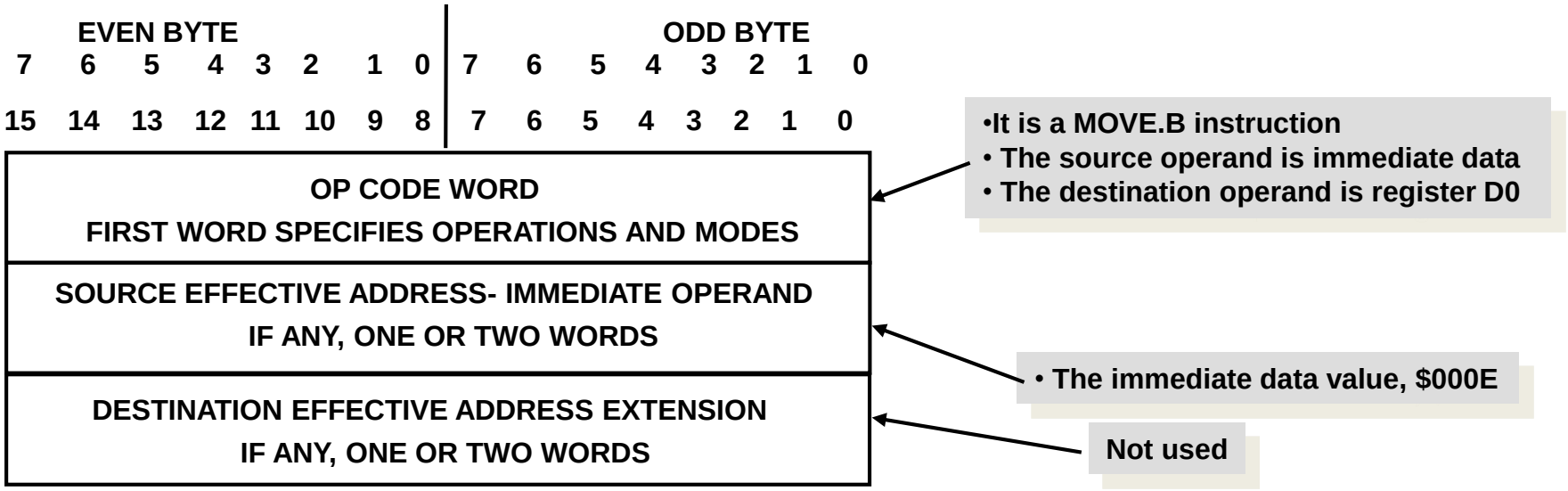
Instruction format in memory

- Example: Only op-code word
  - Generally represented as **OP CODE**
  - Example: **MOVE.B D3, D0 → 1003**



Instruction format in memory (2)

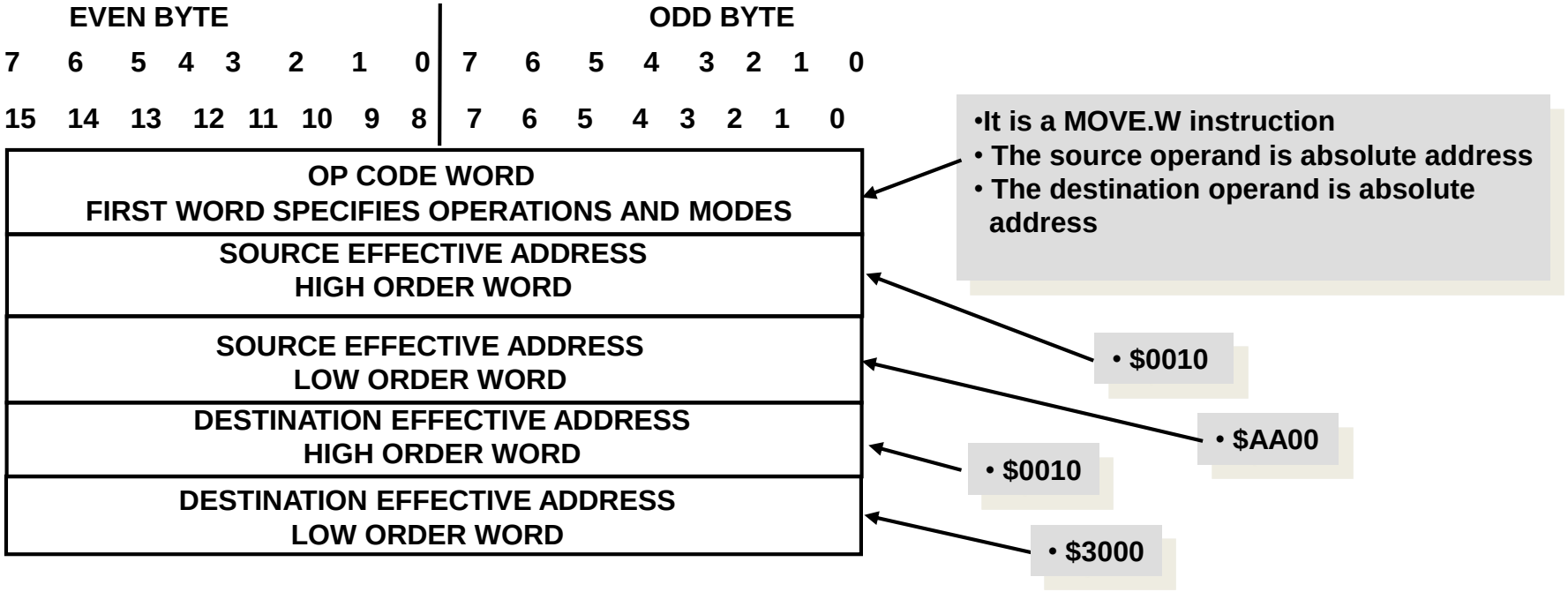
- Example: an ***immediate operand*** is the actual data value
  - Generally represented as **OP CODE** ***#DATA*** **(min. unit is word)**
  - Example: **MOVE.B #14, D0 → 103C *000E***



Instruction format in memory(3)

- Example: an ***absolute operand*** is the actual data value
  - Generally represented as **OP CODE** ***source EA, dest EA***
  - Example:

MOVE.W $0010AA00,$00103000 → **33F9** ***0010AA00 00103000***



Instruction decomposition

- Assemble/decompose: Translate assembly code to machine code
  - Why need assembling?
    - Your memory system can store binary numbers only
    - Basically, all assembly languages should be translated into a set of binary numbers to be stored or understood by the computer system
    - Why you only see hex numbers in the assembled code?

- Disassemble: Translate machine code to assembly code (human readable)
  - Why need disassembling?
    - Because it is a final project

68K manual

MOVE **Move Data from Source to Destination** **MOVE**
(M68000 Family)

Operation: Source → Destination

Assembler Syntax: MOVE < ea > , < ea >

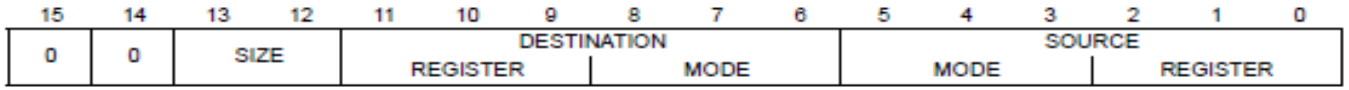
Attributes: Size = (Byte, Word, Long)

Description: Moves the data at the source to the destination location and sets the condition codes according to the data. The size of the operation may be specified as byte, word, or long. Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X — Not affected.
N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Always cleared.
C — Always cleared.

Instruction Format:



Instruction Fields:

Size field—Specifies the size of the operand to be moved.
01 — Byte operation
11 — Word operation
10 — Long operation

68K manual

Destination Effective Address field—Specifies the destination location. Only data alterable addressing modes can be used as listed in the following tables:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

MC68020, MC68030, and MC68040 only

(bd,An,Xn)*	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

(bd,PC,Xn)*	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Can be used with CPU32.

68K manual

Source Effective Address field—Specifies the source operand. All addressing modes can be used as listed in the following tables:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

MC68020, MC68030, and MC68040 only

(bd,An,Xn)**	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

(bd,PC,Xn)**	111	011
([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	111	011

*For byte size operation, address register direct is not allowed.
**Can be used with CPU32.

NOTE

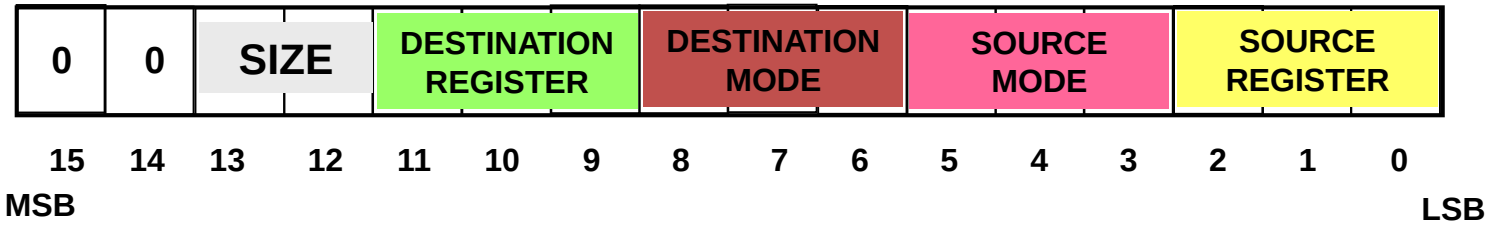
Most assemblers use MOVEA when the destination is an address register.

MOVEQ can be used to move an immediate 8-bit value to a data register.

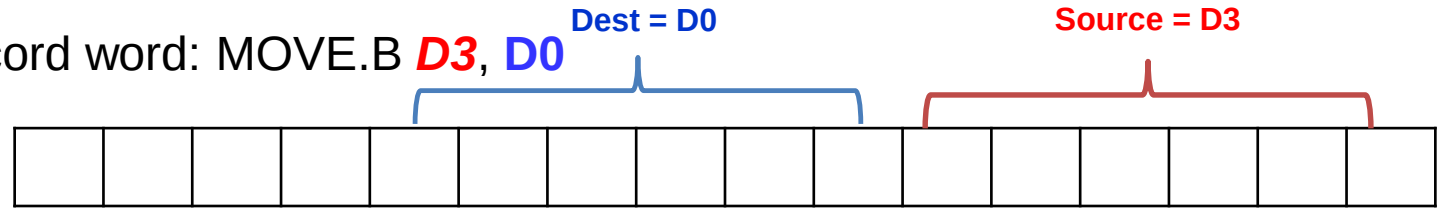
Decomposing the MOVE instruction

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



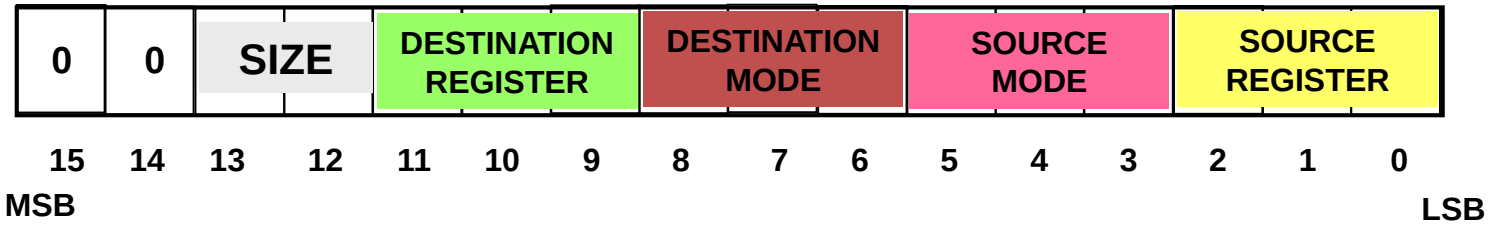
- Opcord word: MOVE.B ***D3***, **D0**



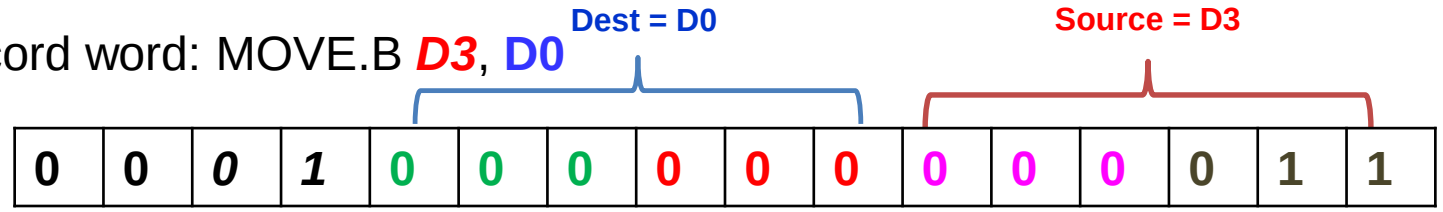
Decomposing the MOVE instruction

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



- Opcord word: MOVE.B ***D3***, **D0**

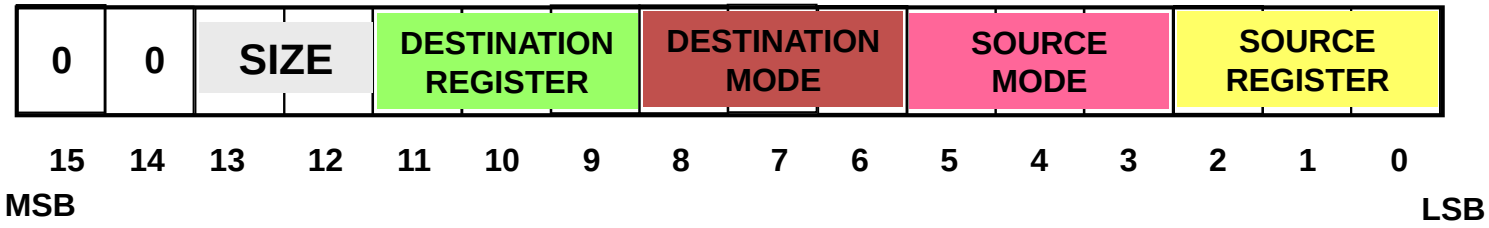


- **1003**

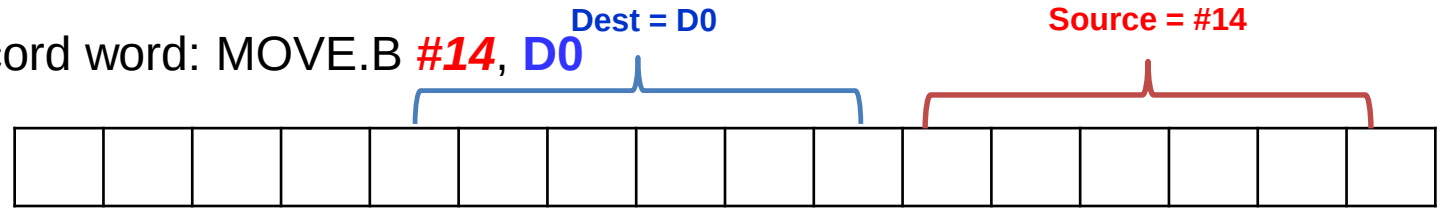
Decomposing the MOVE instruction (2)

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



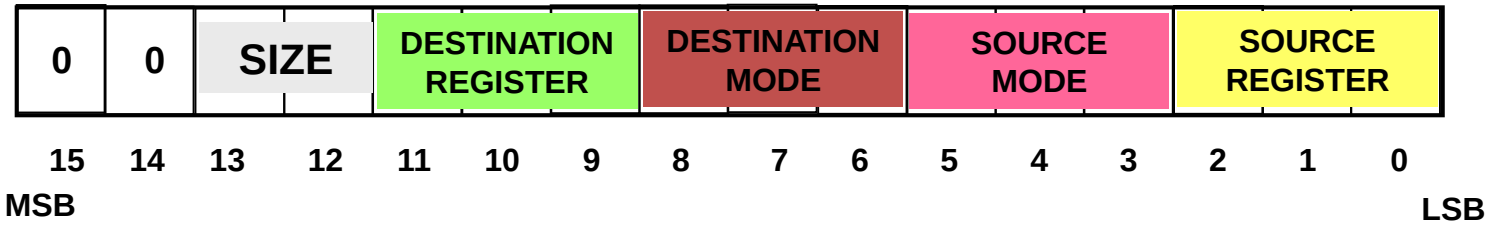
- Opcord word: MOVE.B ***#14***, **D0**



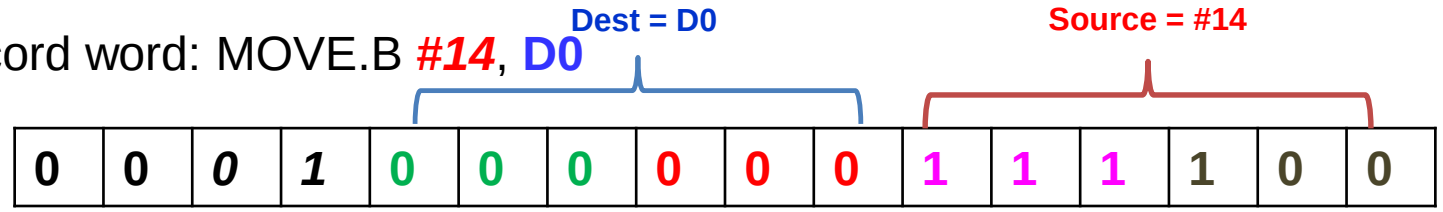
Decomposing the MOVE instruction (2)

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



- Opcord word: MOVE.B ***#14***, **D0**

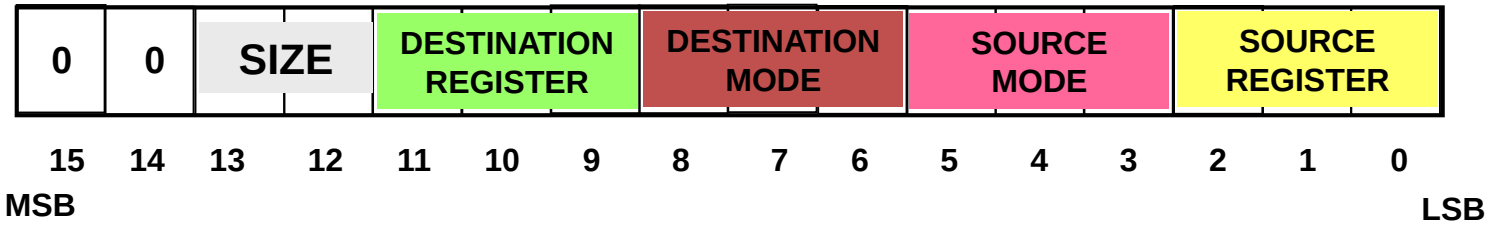


- **Finished?**

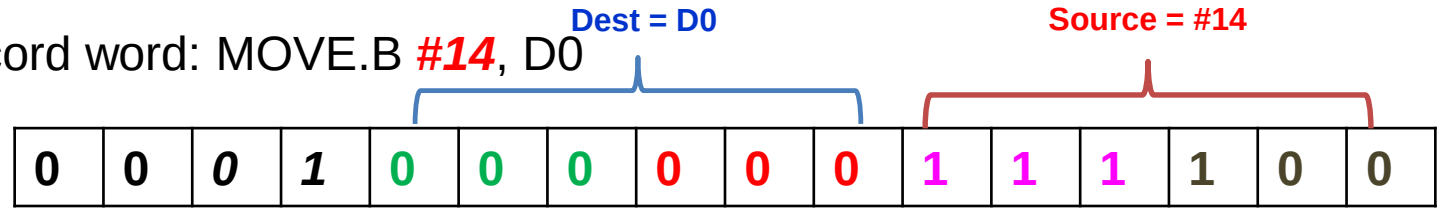
Decomposing the MOVE instruction(2)

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



- Opcord word: MOVE.B ***#14***, D0

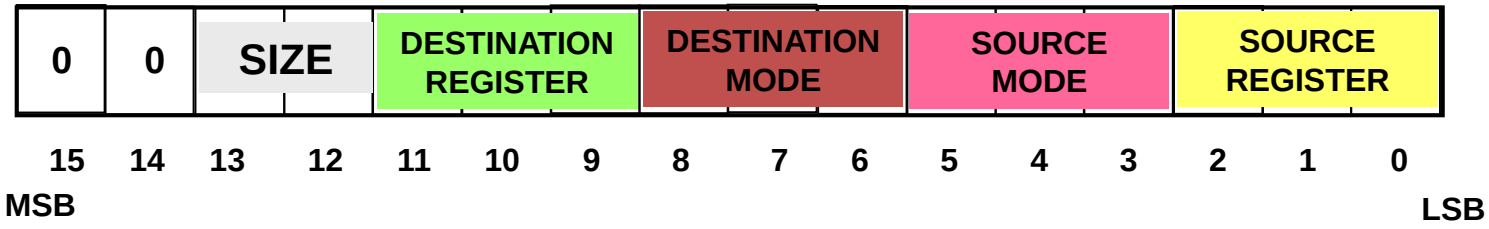


- **Finished? No, you need one more word for the number #14**

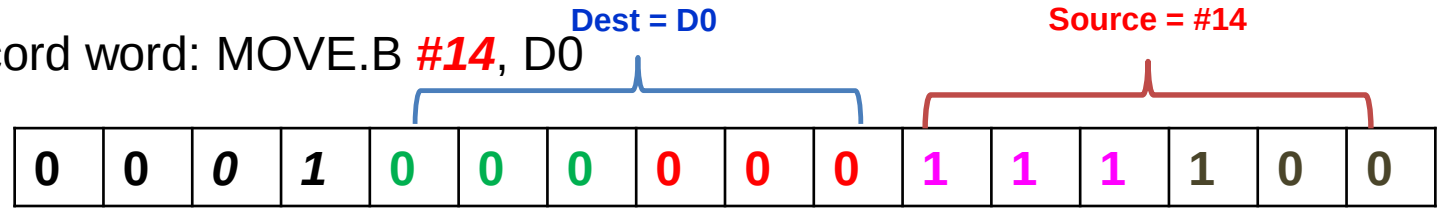
Decomposing the MOVE instruction(2)

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



- Opcord word: MOVE.B ***#14***, D0

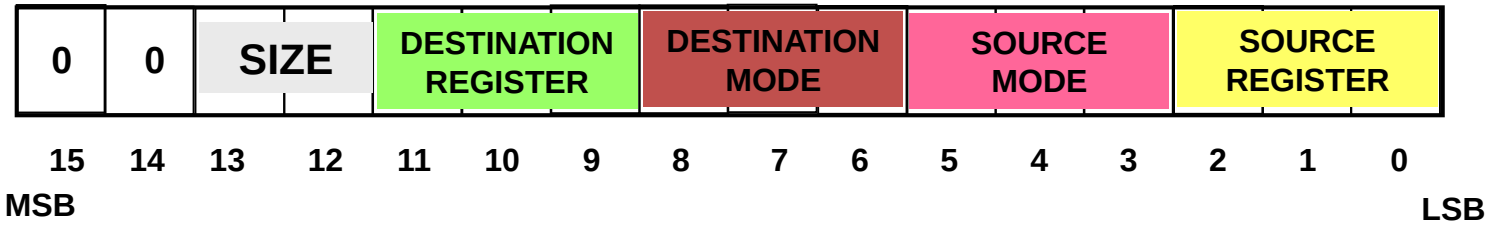


- **103C 000E**

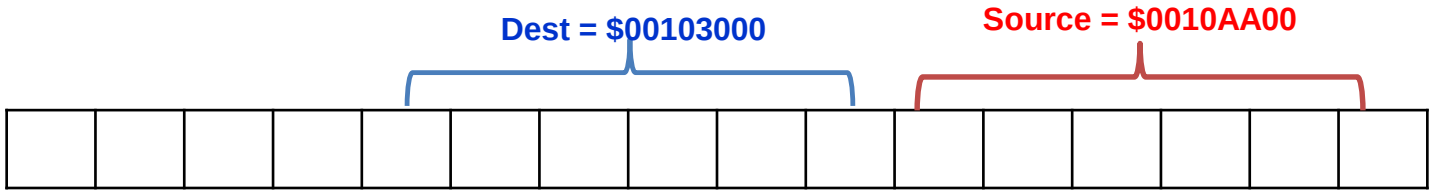
Decomposing the MOVE instruction (3)

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



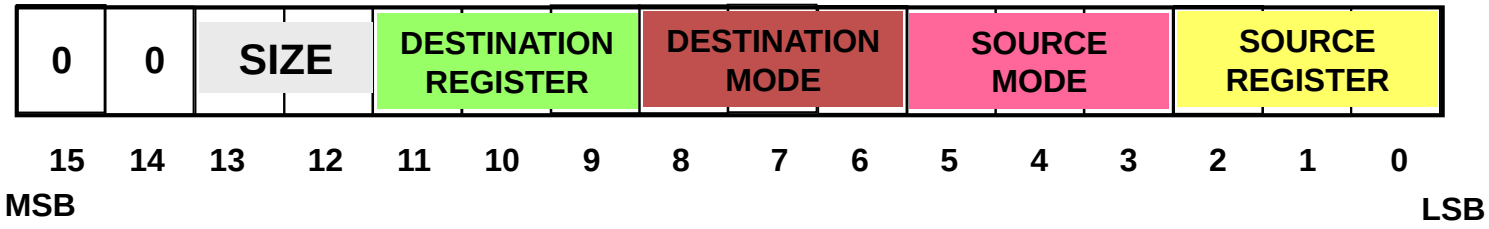
- Opcord word: MOVE.W $0010AA00, $00103000



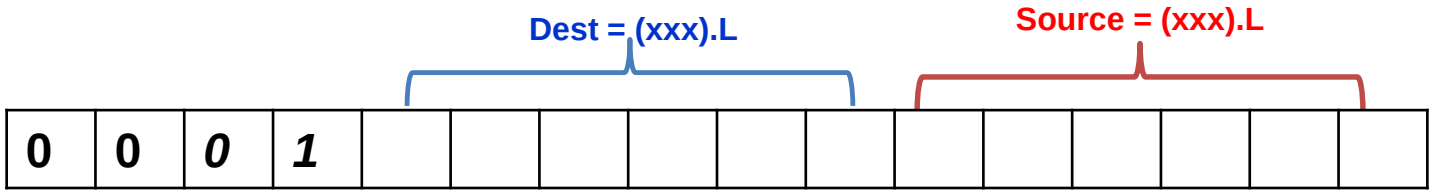
Decomposing the MOVE instruction (3)

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



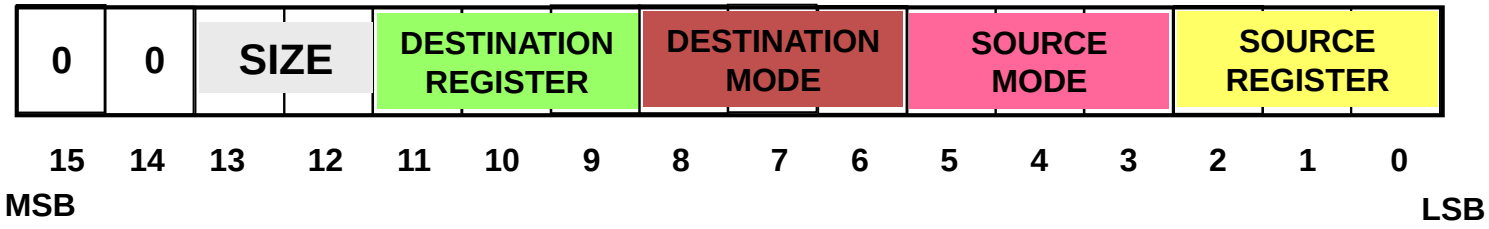
- Opcord word: MOVE.W $0010AA00, $00103000



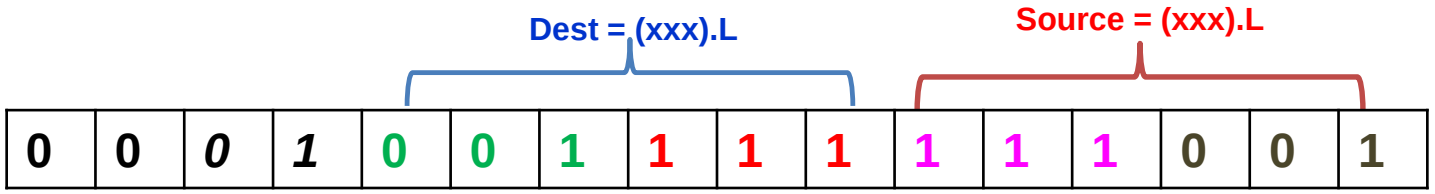
Decomposing the MOVE instruction (3)

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



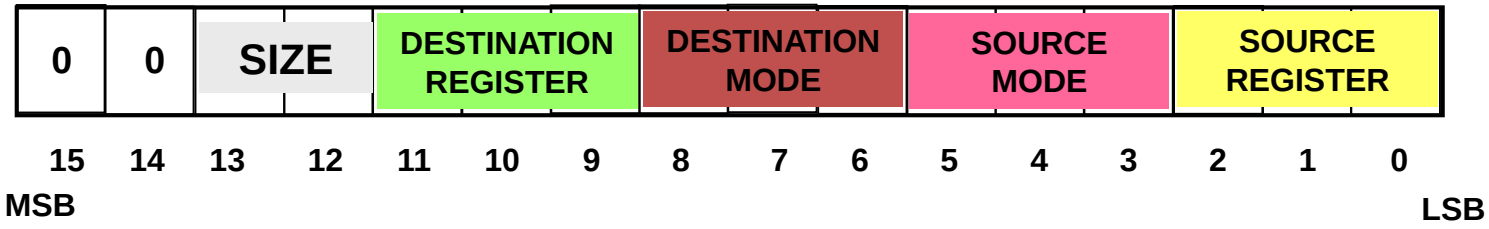
- Opcord word: MOVE.W $0010AA00, $00103000



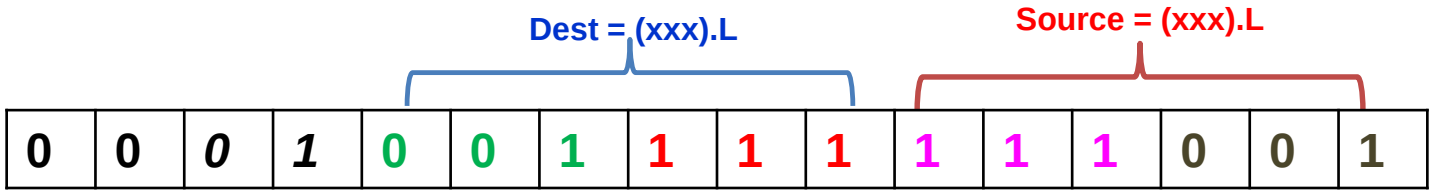
Decomposing the MOVE instruction (3)

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



- Opcord word: MOVE.W $0010AA00, $00103000

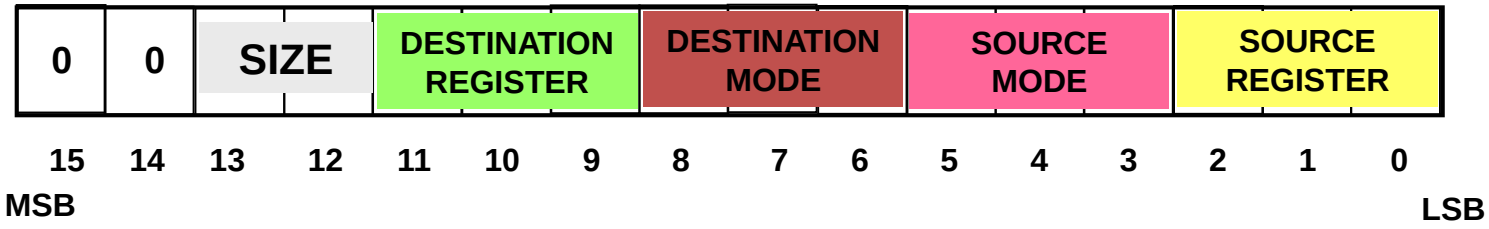


- **Finished?**

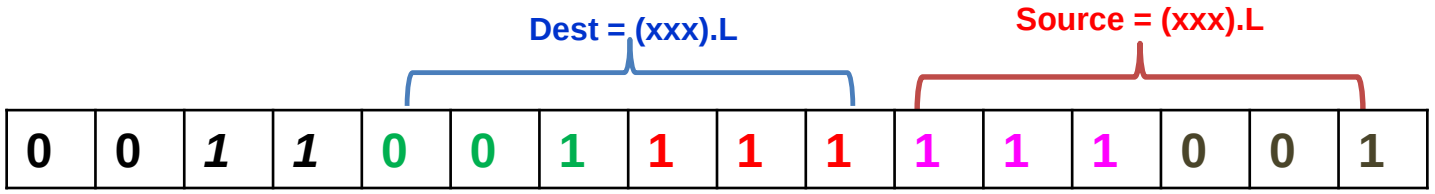
Decomposing the MOVE instruction (3)

- Refer the 68K manual
- Recall that the MOVE instruction format was shown as:

- We can decompose this further to:



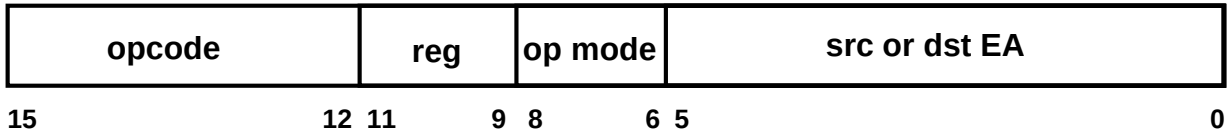
- Opcord word: MOVE.W $0010AA00, $00103000



- **13F9 0010AA00 00103000**

Format of the 68000 instructions set (2)

- The MOVEA instruction is a special form of the MOVE instruction and is used if the destination (dst) is an ***Address Register***
- We can also use an internal register as the source or destination of the operation (ie ADD, AND, CMP)



Some notes

- MOVE instruction
  - MOVE.W src, dst
  - MOVEA.W src, An
- Question: MOVEA.W and MOVE.W have identical opcodes, what distinguishes the two instructions from each other?
  - Hint: Look closely at the mode/register fields of the destination effective address for both instructions
- Why did Motorola do this?
  - Answer: Recall that word accesses must be aligned on word boundaries. If the same instruction was used for storing data in an address register, then it would be possible to store an odd address value and cause a non-aligned access to occur.

Illegal instruction

- You should refer the 68K manual to find the illegal syntax for each instruction

  For example,

  – MOVE.W D0, A3

  is illegal : explain why?

  - MOVE.B #$2400, D3 is illegal, but, MOVE.B $2400, D3 is legal. Why?

  - ADD.B D3, D4 is legal, but ADD.B $2500, $5000 is illegal, why?

Now you try it...

- In class exercise:

1. Assemble by hand the following assembly language MOVE and ADD instructions

- MOVE.B (A0) D7
- MOVE.L $1234 D7
- MOVEA.W D7 A0
- ADD.W D0 D7

illegal instructions

- In exercise:

2. Explain why the following codes are illegal

- MOVE.W $2233 A5

- MOVE.B #$2233 D6

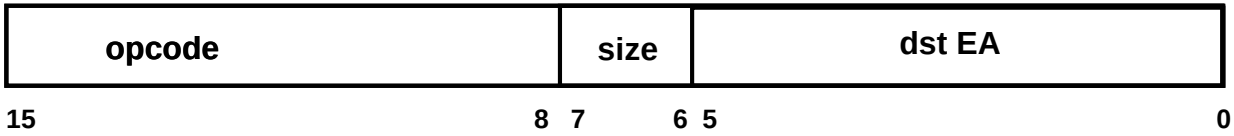
- ADD.W D0 #$1000

- MOVEA.B D7 A0

- MOVEA.W A0 $1234

Format of the 68000 instructions set(3)

- Single operand instructions, i.e CLR (clear the contents of the dst EA)
  - An opcode word + rest of instruction (sometimes do not have)

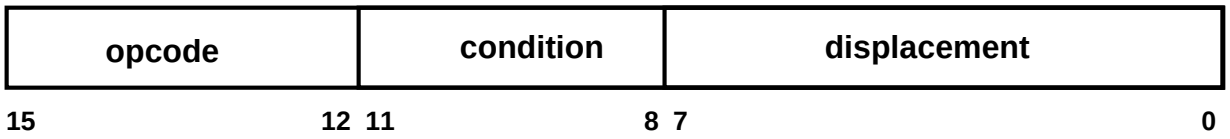


- JMP (Jump) and JSR (Jump to Sub-Routine) are single-operand instructions
  - JMP: Change the value of the Program Counter (PC)
  - Next instruction is fetched from <PC>
- JSR is a special type of jump instruction
  - Replaces <PC> with operand but also saves the current <PC> on the stack
  - Can return to starting point with RTS
  - Used for interrupt subroutines or ISR's and function calls

29

Format of the 68000 instructions set(4)

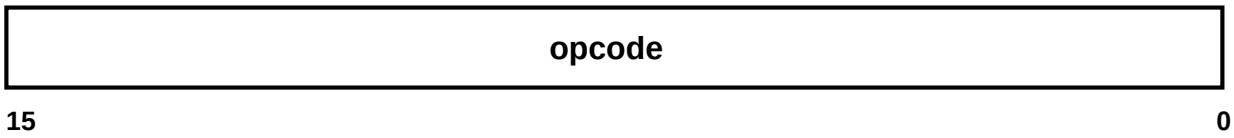
- Branch instructions
  - Change the program counter value to a new value if a test condition is true
  - Test conditions are represented by the state of the *flags* in the *Condition Code Register CCR*
    - Tests can be, zero, overflow, carry or borrow, negative



- Destination of the branch is calculated by adding the current value of the program counter to the displacement value in the instruction
  - Uses 2's complement, signed addition

Format of the 68000 instructions set(6)

- Inherent addressing: The effect (dst or src) of the opcode is inherently contained in the function of the opcode
  - RTS: ReTurn from Subroutine:
    - JSR instruction PUSHed the return location on the stack
    - RTS only needs to POP the <PC> in order to get back from the subroutine
    - Major gotcha: If the stack PUSHes and POPs after entering the JSR and before the RTS don't cancel out, the RTS is guaranteed to create an interesting result



Some representative instructions

- CLR.B ddst 0100001000dddddd
  - Clear (set to zero) the byte of the data destination operand, ddst
- BNE displacement 01100110dddddddd
  - Branch if the result is Not Equal to zero (Zero flag = 0)
- BEQ displacement 01100111dddddddd
  - Branch if the result is EQual to zero (Zero flag = 1)
- JMP cdst 0100111011dddddd
  - Jump to the address defined by control destination operand, cdst
- RTS 0x4E75
  - Notice that the RTS instruction does not require an operand

Immediate instructions

- The previous example represents the “normal” instruction
- Other classes of instructions require special considerations
  - MOVE instruction: Two effective address fields
  - All immediate instructions: Immediate addressing mode is hard coded into the instruction and is not a source operand
    - Example: AND and ANDI
  - Immediate source operands ALWAYS follow the op-code word
  - For example, the instruction, **`ADDI.L #$5A5A5A00,$3456FFEE`** is 5 words long
  - The “instruction” is 3 word long and the effective address is two words long
- Of course, some instructions violate this rule as well
  - Example: MULU, MULS, DIVU, DIVS