

# EE 357 Unit 4b

## CF Assembly Basics

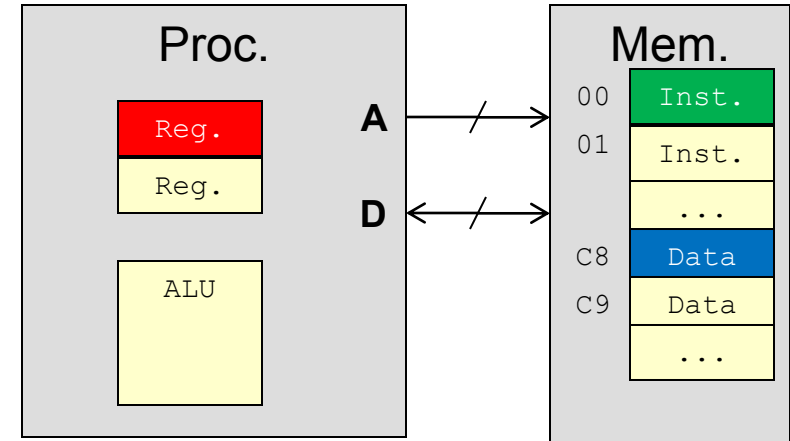
# OPERATIONS AND INSTRUCTIONS

# Coldfire Instruction Classes

- Data Transfer
  - Move data between processor & memory
  - Move data between registers w/in processor
  - Can specify .B, .W, .L size
- ALU
  - Performs arithmetic and logic operations
  - Only .L size => Ops. must be on a full 32-bit longword contents
- Control / Program Flow
  - Unconditional/Conditional Branch
  - Subroutine Calls
- Privileged / System Instructions
  - Instructions that can only be used by OS or other “supervisor” software (e.g. STOP, certain HW access instructions, etc.)

# Operand Locations

- In almost all instruction sets, operands can be...
  - A register value (e.g. D0)
  - A value in a memory location (e.g. value at address 0xC8)
  - A constant stored in the instruction itself (known as an 'immediate' value)  
[e.g. ADDI #1,D0]
- Thus, our instructions must be able to specify which of these three locations is where the operand is located

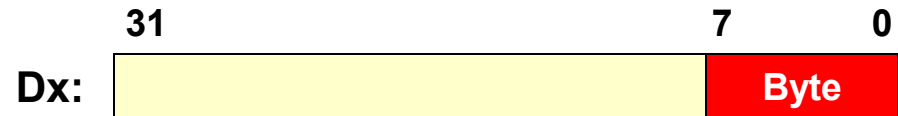


# Data Transfer Instruction

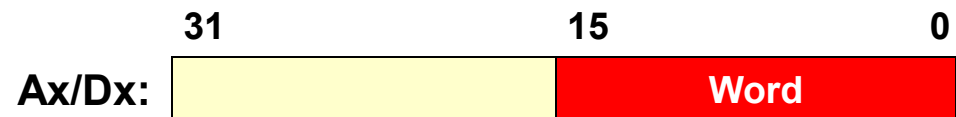
- MOVE.s src,dst
  - .s = .B, .W, .L
  - Copies src operand to dst operand
  - SRC operand specifies data in a:
    - Reg. (e.g. D0 or A4)
    - Mem. = Specified with the address of desired source location
    - Immediate = Constant (preceded w/ '#' sign in assembly instruc.)
  - DST operand specifies a location to put the source data & can be a:
    - Reg.
    - Mem. = Specified with the address of desired destination location
- Examples
  - MOVE.B D0,0x1C ; 0x = hex modifier
    - Moves the byte from reg. D0 to memory byte @ address 0x1c
  - MOVE.W #0x1C,D0 ; # = Immediate (no # = address/mem. oprnd.)
    - Moves the constant 001C hex to D0

# Registers & Data Size

- Most CF instructions specify what size of data to operate on
  - MOVE.B
  - MOVE.W
  - MOVE.L
- Register sizes are right-justified (start at LSB and work left)



Byte operations only access bits 7-0 of a register (upper bits are left alone)



Word operations only access bits 15-0 of a register (upper bits are left alone)



Longword operations use the entire 32-bit value

# Examples

- Initial Conditions:

D0: 

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

D1: 

A	A	A	A	B	B	B	B
---	---	---	---	---	---	---	---

- MOVE.B D0,D1

D1: 

A	A	A	A	B	B	7	8
---	---	---	---	---	---	---	---

- MOVE.W #0xFEDC,D1

D1: 

A	A	A	A	F	E	D	C
---	---	---	---	---	---	---	---

immediate (constant)            base hex

- MOVE.L D0,D1

D1: 

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

# Memory & Data Size

Recall...

- Memory operands are addressed by their starting address and size
  - Whereas registers operands always start with LSB and get appropriate size data, memory operands get the appropriate sized data from starting address and continuing towards larger addresses
  - This makes it look left justified due to the big-endian ordering
- Valid words start @ addresses that is multiples of 2
- Valid longwords start @ addresses that are multiples of 4

Address

0x000

5A	13	F8	7C
----	----	----	----

Word @ Addr. 0

Dx:

31	15	0
		Word

Word in D0

Longword @ Addr. 0

0x000

Word @ Addr. 0		Word @ Addr. 2	
5A	13	F8	7C
Byte @ 0	Byte @ 1	Byte @ 2	Byte @ 3



# Examples

- Initial Conditions:

000000	12	34	56	78
000004	9A	BC	DE	F0

- MOVE.B 1,2

↑  
no '#' so treat as address

000000	12	34	34	78
000004	9A	BC	DE	F0

- MOVE.W 0x2,0x4

000000	12	34	56	78
000004	56	78	DE	F0

- MOVE.L #\$FEDCBA98,4

↑ ↑    another hex indicator    ↑  
immediate                                  address

000000	12	34	56	78
000004	FE	DC	BA	98

# Examples

- Initial Conditions:

000000	12 34 56 78							
000004	9A BC DE F0							
D0:	1 2 3 4 5 6 7 8							

- MOVE.B 0x5,D0

D0:	1	2	3	4	5	6	B	C
-----	---	---	---	---	---	---	---	---

- MOVE.W 4,D0

D0:	1	2	3	4	9	A	B	C
-----	---	---	---	---	---	---	---	---

- MOVE.L #3,4

000000	12	34	56	78
000004	00	00	00	03

# ALU Instruction(s)

- ADD.L src1,src2/dst
  - Size is always .L (cannot choose .B or .W)
  - Adds src1 and src2/dst and places result in src2/dst overwriting the original value
  - One operand MUST BE a data register though it can be either source or dest. (e.g. ADD.L Dn,dst or ADD.L src,Dn)
- SRC can be {reg, mem., immediate}
- DST can be {reg, mem.}

# Examples

- Initial Conditions:

000000	5	6	7	8	1	2	3	4
D0:	1	2	3	4	5	4	3	2
D1:	C	8	2	D	F	E	9	8

- ADD.L D0,D1

	1	2	3	4	5	4	3	2
	+ A	B	C	D	F	E	9	8
D1:	D	A	6	2	5	2	C	A

- ADD.L D0, 0

	1	2	3	4	5	4	3	2
	+ 5	6	7	8	1	2	3	4
000000	6	8	A	C	6	6	6	6

- ADDI.L #0x341E, D0

	0	0	0	0	3	4	1	E
	+ 1	2	3	4	5	4	3	2
D0:	1	2	3	4	8	8	5	0

# ADD and MOVE Examples

Instruction	M[0x7000]	M[0x7004]	D0	Operation
	5A13 F87C	2933 ABC0	0000 0000	
MOVE.L #\$26CE071B, \$7000	26CE 071B	2933 ABC0	0000 0000	
MOVE.B \$7003,D0	26CE 071B	2933 ABC0	0000 001B	
MOVE.W \$7004,D0	26CE 071B	2933 ABC0	0000 2933	
MOVE.W #\$4431,\$7004	26CE 071B	4431 ABC0	0000 2933	
ADD.L \$7000,D0	26CE 071B	4431 ABC0	26CE 304E	
MOVE.B D0,\$7004	26CE 071B	4E31 ABC0	26CE 304E	
MOVE.W D0,\$7006	26B7 071B	F931 304E	26CE 304E	

# Data Transfer Instructions

C operator	Assembly	Notes
=	MOVE.s src,dst	
= 0	CLR.s dst	CLR is faster to execute than MOVE.s #0,dst

# Arithmetic and Logic Instructions

C operator	Assembly	Notes
+	ADD.s src1,src2/dst	
-	SUB.s src1,src2/dst	Order: src2 – src1
&	AND.s src1,src2/dst	
	OR.s src1,src2/dst	
^	EOR.s src1,src2/dst	
~	NOT.s src/dst	
-	NEG.s src/dst	Performs 2's complementation
* (signed)	MULS src1,src2/dst	Implied size .W
* (unsigned)	MULU src1, src2/dst	Implied size .W
/ (signed)	DIVS src1,src2/dst	Implied size .W
/ (unsigned)	DIVU src1, src2/dst	Implied size .W
<< (signed)	ASL.s cnt, src/dst	
<< (unsigned)	LSL.s cnt, src/dst	
>> (signed)	ASR.s cnt, src/dst	
>> (unsigned)	LSR.s cnt, src/dst	
==, <, >, <=, >=, != (src2 ? src1)	CMP.s src1, src2	Order: src2 – src1

- Logic operations on numbers means performing the operation on each pair of bits

*Initial Conditions:* D0 = 0x000000F0, D1 = 0x0000003C

① AND.L D0,D1      →      0xF0      →      1111 0000  
                               AND 0x3C      AND 0011 1100  
    (D1) = 0x00000030   ←      0x30      ←      0011 0000

② OR.L D0,D1      →      0xF0      →      1111 0000  
                                       OR 0x3C      OR 0011 1100  
                                       ———  
    (D1) = 0x000000FC   ←      0xFC      ←      1111 1100

③ EOR.L D0,D1      →      0xF0      →      1111 0000  
                               EOR 0x3C      EOR 0011 1100  
 (D1) = 0x000000CC ←      0xCC      ←      1100 1100



# Logical Operations

- Logic operations on numbers means performing the operation on each pair of bits

*Initial Conditions:* D0 = 0xFFFFFFFF0, D1 = 0x0000003C

$$\begin{array}{ccccc}
 \textcircled{1} \text{ NOT.L D0} & \longrightarrow & \text{NOT 0xF0} & \longrightarrow & \text{NOT 1111 0000} \\
 (\text{D0}) = 0x0000000F & \longleftarrow & 0x0F & \longleftarrow & 0000 1111
 \end{array}$$

$$\begin{array}{ccccc}
 \textcircled{2} \text{ NEG.L D1} & \longrightarrow & \text{2's Comp. 0x3C} & \longrightarrow & 0xffffffffc3 \\
 & & & & + \quad \quad \quad 1 \\
 (\text{D1}) = 0xFFFFFFFFC4 & \longleftarrow & 0xFFFFFFFFC4 & \longleftarrow & 0xffffffffc4
 \end{array}$$

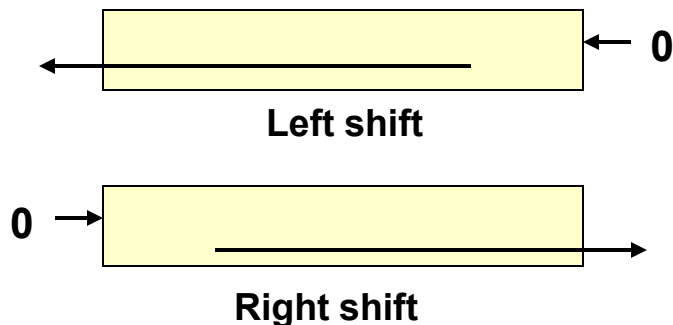
# Logical Operations

- Logic operations are often used for “bit” fiddling
  - Change the value of 1-bit in a number w/o affecting other bits
  - C operators:  $\&$  = AND,  $|$  = OR,  $\wedge$  = XOR,  $\sim$  = NOT
- Examples (Assume an 8-bit variable,  $v$ )
  - Set the LSB to ‘0’ w/o affecting other bits
    - $v = v \& 0xfe;$
  - Check if the MSB = ‘1’ regardless of other bit values
    - $\text{if}(v \& 0x80) \{ \text{code} \}$
  - Set the MSB to ‘1’ w/o affecting other bits
    - $v = v | 0x80;$
  - Flip the LS 4-bits w/o affecting other bits
    - $v = v \wedge 0x0f;$

# Logical Shift vs. Arithmetic Shift

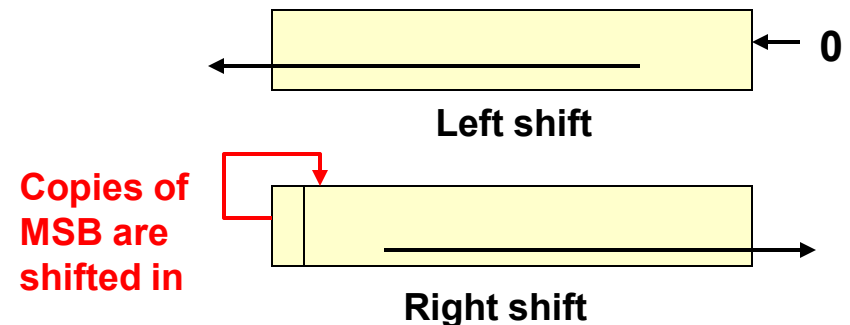
- Logical Shift

- Use for unsigned or non-numeric data
- Will always shift in 0's whether it be a left or right shift



- Arithmetic Shift

- Use for signed data
- Left shift will shift in 0's
- Right shift will sign extend (replicate the sign bit) rather than shift in 0's
  - If negative number...stays negative by shifting in 1's
  - If positive...stays positive by shifting in 0's



# Logical Shift

- 0's shifted in
- Only use for operations on *unsigned* data
  - Right shift by n-bits = Dividing by  $2^n$
  - Left shift by n-bits = Multiplying by  $2^n$

0 0 0 0 1 1 0 0 = +12

Logical Right Shift by 2 bits:

0's shifted in...

0 0 0 0 0 0 1 1 = +3

Logical Left Shift by 2 bits:

0's shifted in...

0 0 1 1 0 0 0 0 = +48

# Arithmetic Shift

- Use for operations on *signed* data
- Arithmetic Right Shift – replicate MSB
  - Right shift by  $n$ -bits = Dividing by  $2^n$
- Arithmetic Left Shift – shifts in 0's
  - Left shift by  $n$ -bits = Multiplying by  $2^n$

**1 1 1 1 1 1 0 0** = -4

Arithmetic Right Shift by 2 bits:

*MSB replicated and shifted in...*

**1 1 1 1 1 1 1 1** = -1

Arithmetic Left Shift by 2 bits:

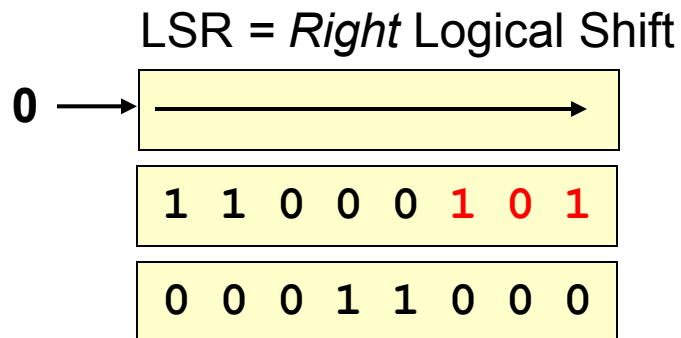
*0's shifted in...*

**1 1 1 1 0 0 0 0** = -16

*Notice if we shifted in 0's (like a logical right shift) our result would be a positive number and the division wouldn't work*

# Logical Shift Instructions

- LSR (Logical Shift Right) & LSL (Logical Left Shift)
- Specify a shift count and the data to be shifted
- 2 forms of each instruction
  - LSx.L Dx, Dy
    - $Dx \bmod 64$  = shift (rotate) count
    - Dy = data to be shifted
  - LSx.L #imm,Dy
    - imm = shift (rotate) count [1-8]
    - Dy = data to be shifted

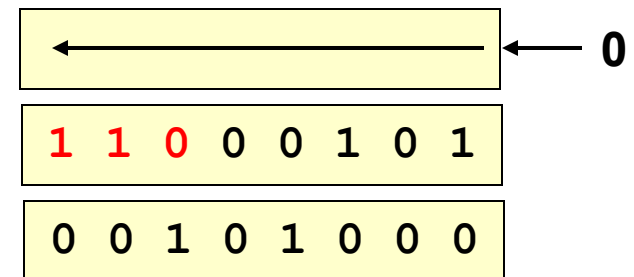


Operation

Initial Value

Shifted by 3-bits

LSL = *Left* Logical Shift



# LSL and LSR Instructions

**D2**

**MOVE.L #0xF0000003,D2**

**F000 0003**

1111 0000 0000 0000 0000 0000 0000 0011

**MOVE.L #4,D3**

*Right Shift by 1-bit*

**LSR.L #1,D2**

0111 1000 0000 0000 0000 0000 0000 0001

**D2 = 0x78000001**

*Left Shift by 4-bits*

**LSL.L D3,D2**

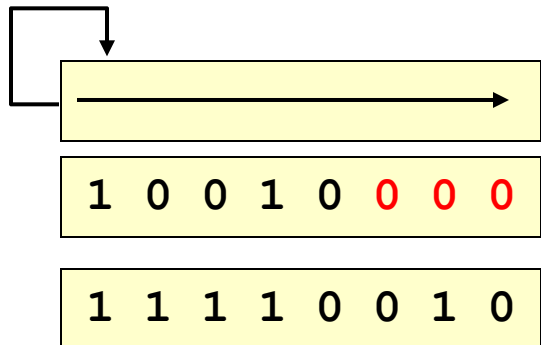
1000 0000 0000 0000 0000 0000 0000 1000

**D2 = 0x80000010**

# Arithmetic Shift Instructions

- ASR (Arithmetic Shift Right) & ASL (Arithmetic Shift Left)
- Same format as LSR and LSL
- 2 forms of each instruction
  - ASx.s Dx, Dy
    - Dx mod 64 = shift (rotate) count
    - Dy = data to be rotated
  - ASx.s #imm,Dy
    - imm = shift (rotate) count [1-8]
    - Dy = data to be rotated

ASR = *Right* Arithmetic Shift

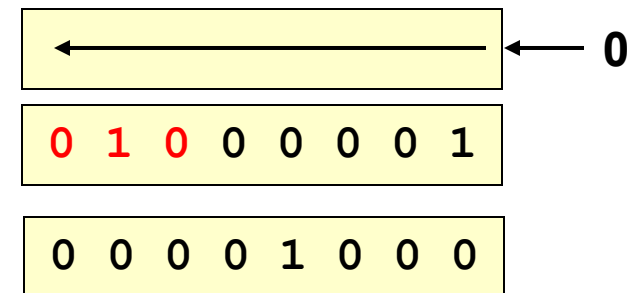


Operation

Initial Value

Shifted by 3-  
bits

ASL = *Left* Arithmetic Shift





# ASL and ASR Instructions

**D2**

**MOVE.L #0x80000003,D2**

8000 0003
1000 0000 0000 0000 0000 0000 0000 0011

**MOVE.L #1,D3**

*Right Shift by 5-bits*

**ASR.L #5,D2**

1111 1100 0000 0000 0000 0000 0000 0000
---

**D2 = 0xFC000000**

*Left Shift by 1-bit*

**ASL.L D3,D2**

1111 1000 0000 0000 0000 0000 0000 0000
---

**D2 = 0xF8000000**

Addressing Modes

# DEALING WITH OPERANDS

# Addressing Modes and Operands

- Operands of an instruction can be a register, memory, or immediate value however how we specify these (especially memory values) can be done with a variety of methods which we call *addressing modes*
- Addressing modes refer to the methods an instruction can use to specify the location of an operand
  - RISC approach: few, simple addressing modes
  - CISC approach: complex, multi-operation modes
- Definition: EA = effective address
  - The final *location* of the operand the instruction will use

# Shorthand Notation

- $M[x]$  = Value in memory @ address  $x$
- $D[n]$  or  $A[n]$  = Register value of  $Dn$  or  $An$
- $R[n]$  = Either data or address register value
- Examples:
  - $M[4] = D0$ 
    - Memory at address 4 gets the value of  $D0$
  - $M[A[1]] = D1$ 
    - Memory at the address specified by  $A1$  gets the value of  $D1$

# Overview

Location of Operand	Addressing Modes	Operand	Assembly Notation
<b>Register Contents</b>	<b>Data Register Direct</b>	<b>D[n]</b>	<b>D0</b>
	<b>Address Register Direct</b>	<b>A[n]</b>	<b>A1</b>
<b>Memory Location</b>	<b>Address Register Indirect (A.R.I.)</b>	<b>M[A[n]]</b>	<b>(A2)</b>
	<b>A.R.I. w/ postincrement</b>	<b>Operand: M[A[n]] Side Effect: A[n] = A[n] + {1,2,4}</b>	<b>(A2)+</b>
	<b>A.R.I. w/ predecrement</b>	<b>Operand: M[A[n] - {1,2,4}] Side Effect: A[n] = A[n] - {1,2,4}</b>	<b>-(A2)</b>
	<b>A.R.I. w/ displacement</b>	<b>M[A[n] + displacement]</b>	<b>(0x24,A2)</b>
	<b>A.R.I. w/ scaled index &amp; displacement</b>	<b>M[A[n] + disp. + R[m]*size]</b>	<b>(0x24,A2,D0.L*4)</b>
	<b>Absolute Addressing Short</b>	<b>M[address] (e.g. M[0x7060])</b>	<b>0x7060</b>
	<b>Absolute Addressing Long</b>	<b>M[address] (e.g. M[0x18000])</b>	<b>0x18000</b>
<b>Immediate</b>	<b>Immediate</b>	<b>Immediate</b>	<b>#0x7800</b>

# Register Operands

- Two different modes for different register types
  - Data Registers
  - Address Registers

# Data Register Direct

- Specifies the contents of a data register

**Example:**

MOVE .W

D3, D2

*Data Register Direct*

# Data Register Direct

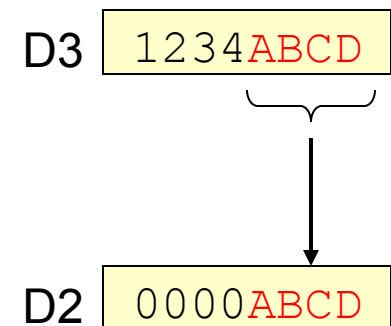
## Example:

① *Initial Conditions:* D[3] = 1234ABCD, D[2] = 00000000

**MOVE.W D3,D2**

② Lower Word of D3 put into D2

③ D3 retains same value



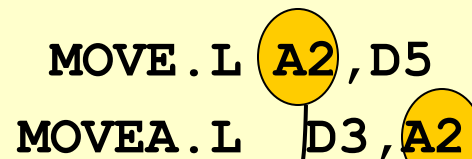


# Address Register Direct

- Specifies the contents of a address register
- Use MOVEA, ADDA, SUBA when address register is destination (though not when source)
- Recommendation: ALWAYS use size .L when destination is address register
  - If you use size .W and destination is an address register, the result will be sign-extended to fill the entire register anyways

## Example:

```
MOVE .L A2, D5
MOVEA .L D3, A2
```



*Address Register Direct*

# Address Register Direct

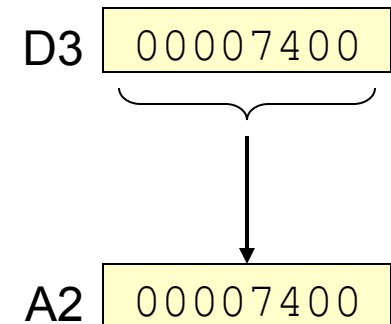
## Example:

① *Initial Conditions:* D[3] = 00007400, A[2] = 000F0420

**MOVEA.L D3,A2**

② Longword of D3 put into A2

③ D3 retains same value



# Address Register Direct

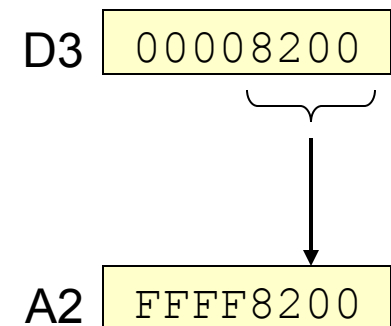
## Example:

① *Initial Conditions:* D[3] = 00008200, A[2] = 00007000

**MOVEA.W D3, A2**

② Word of D3 put into A2 and then sign extended to fill all 32 bits

③ D3 retains same value



Notice the sign extension – MSB of 8200 = '1'...that '1' is extended to fill all the upper bits...so your address is now 0xFFFF8200

# Memory Operands

- 5 Modes for using an address register to specify the address of the memory location
  - Address Register Indirect
  - Address Register Indirect w/ Postincrement
  - Address Register Indirect w/ Predecrement
  - Address Register Indirect w/ Displacement
  - Address Register Indirect w/ Index
- 2 Modes for specifying an address as part of the instruction (called an absolute address)
  - Absolute Short Address Mode
  - Absolute Long Address Mode

# Address Register Indirect

- Specifies a memory location
- Use contents of  $A_n$  as an address (pointer) to the actual data
- Similar idea as pointers in C/C++
  - $(A0)$  in assembly  $\Leftrightarrow$   $*ptr\_A0$  in C/C++

**Example:**

MOVE .L (A0), D2

*Addr. Reg. Indirect*

# Address Register Indirect

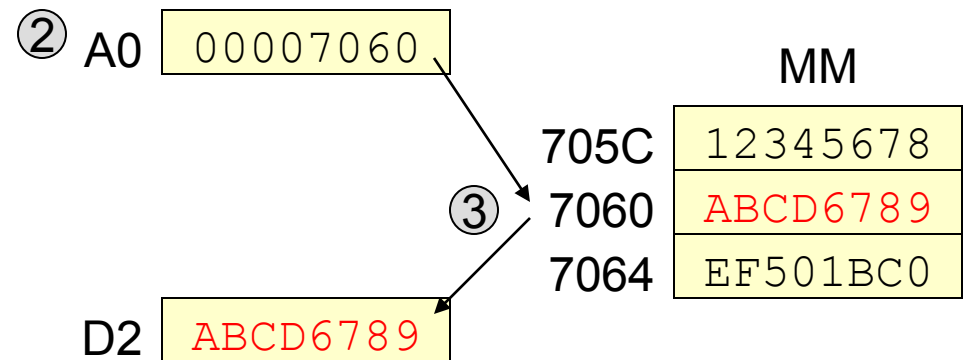
## Example:

① *Initial Conditions:*  $A[0] = 00007060$

**MOVE .L (A0) , D2**

②  $\langle EA \rangle =$  contents of A0

③ Use  $\langle EA \rangle$  to access memory

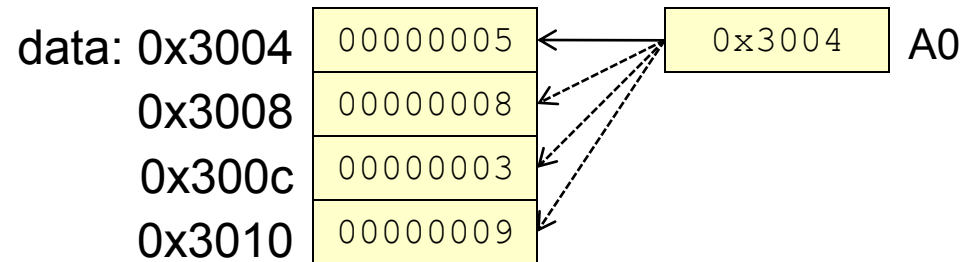


# Address Register Indirect

- Good for use as a pointer

```
int data[4]={5,8,3,9};
for(i=0; i<4; i++)
    data[i] = 1;
```

C Code



MM

```
movea.l #0x3004,a0
loop 4 times {
    move.l #1,(a0)
    adda.l #4,a0
}
```

# Address Register Indirect w/ Postincrement

- Specifies a memory location
- Use contents of  $A_n$  as address to actual data
- After accessing data, contents of  $A_n$  are incremented by 1, 2, or 4 depending on size (.B, .W, .L)
- Similar idea as pointers in C/C++
  - $(A0)+$  in assembly  $\Leftrightarrow$  `*ptr_A0; ptr_A0++;` in C/C++

## Example:

MOVE .L (A0)+, D2

*Postincrement*

*Addr. Reg. Indirect w/  
Postincrement Mode*



# Address Register Indirect w/ Postincrement

## Example:

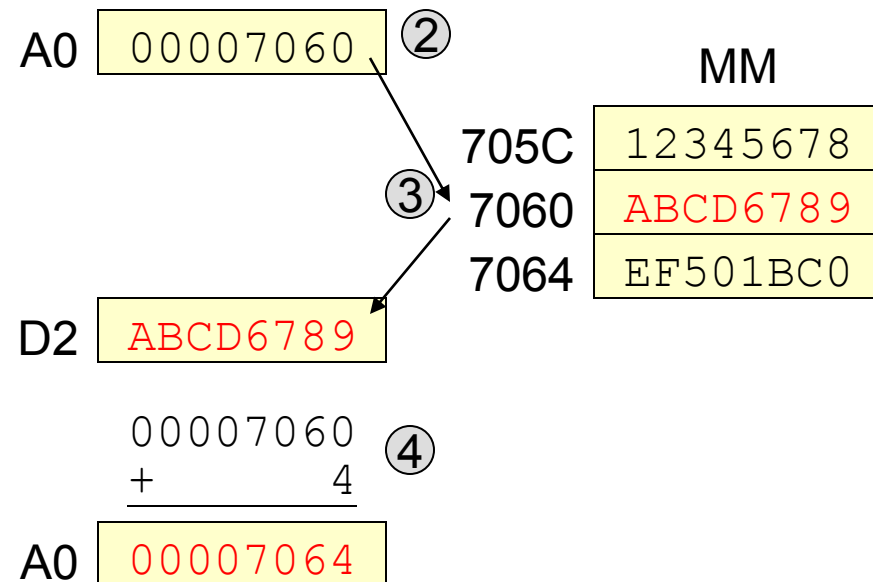
① Initial Conditions:  $A[0] = 00007060$

**MOVE .L** **(A0) +**, D2

② <EA> = contents of A0

③ Increment An by 1, 2, 4 depending on size (.B, .W, .L)

④ Use <EA> to access memory



# Address Register Indirect w/ Postincrement

## Another Example:

① Initial Conditions:  $A[0] = 00007060$

**MOVE .W (A0) +, (A0) +**

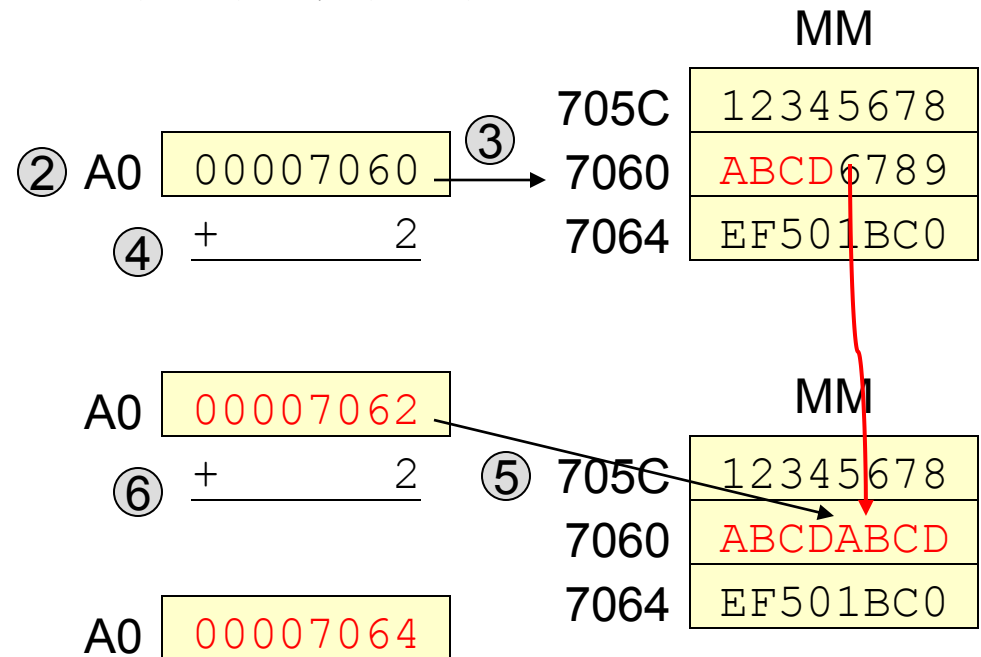
② Source <EA> = contents of A0

③ Use <EA> to access memory

④ Increment A0 by 2 because it is a .W instruction

⑤ Use new contents of A0 as address for destination

⑥ Increment A0 by 2 again



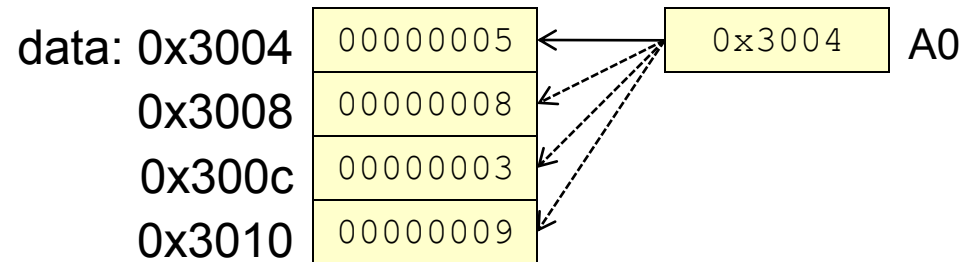
**Summary: Work from left to right**

# Address Register Indirect w/ Postincrement

- Good for use as a pointer moving through an array

```
int data[4]={5,8,3,9};
for(i=0; i<4; i++)
    data[i] = 1;
```

C Code



MM

```
movea.l #0x3004,a0
loop 4 times {
    move.l #1,(a0)+
}
```

# Address Register Indirect w/ Predecrement

- Specifies a memory location
- Use contents of  $A_n$  as address to actual data
- **Before** accessing data, contents of  $A_n$  are decremented by 1, 2, or 4 depending on the size (.B, .W, .L)
- Similar idea as pointers in C/C++
  - $-(A0)$  in assembly  $\Leftrightarrow --ptr\_A0; *ptr\_A0;$  in C/C++

## Example:

MOVE .L **-(A0)**, D2

*Predecrement*

*Addr. Reg. Indirect w/  
Predecrement Mode*

# Address Register Indirect w/ Predecrement

## Example:

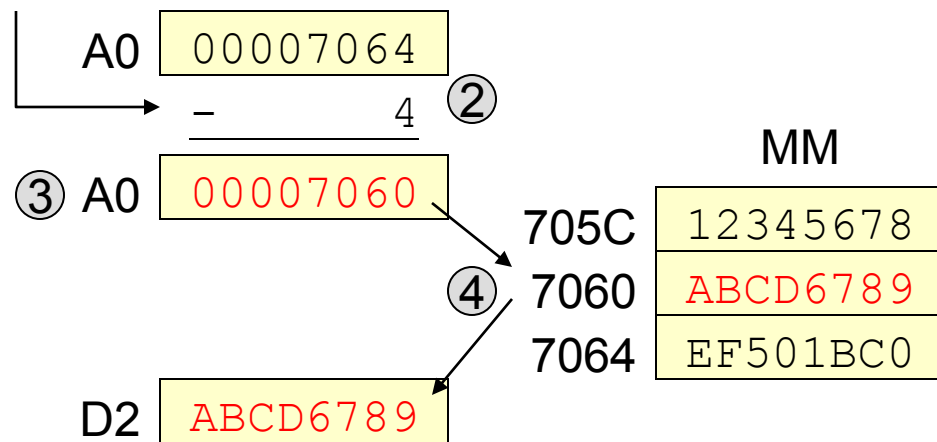
① Initial Conditions:  $A[0] = 00007064$

**MOVE .L - (A0) , D2**

② Decrement A0 by 1, 2, 4 depending on size (.B, .W, .L)

③ Use new A0 to access memory

④  $\langle EA \rangle = \text{contents of A0}$

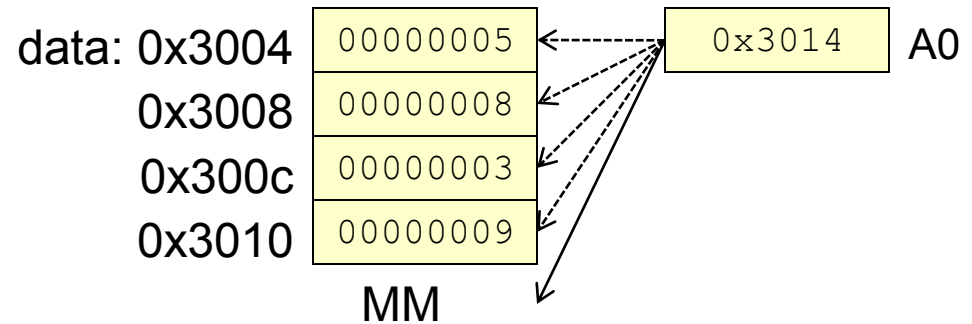


# Address Register Indirect w/ Predecrement

- Good for use as a pointer moving through an array

```
int data[4]={5,8,3,9};
for(i=0; i<4; i++)
    data[i] = 1;
```

C Code



```
movea.l #0x3014,a0
loop 4 times {
    move.l #1,-(a0)
}
```

# Address Register Indirect w/ Displacement

- Specifies a memory location
- Use  $A_n$  as base address and adds a displacement value to come up w/ effective address (<EA>)
- Displacement limited to 16-bit signed number
- $A_n$  not affected (maintains original address)

**Example:**

`MOVE.L (0x24, A0), D2`

*Displacement Value*

*Addr. Reg. Indirect w/  
Displacement Mode*

# Address Register Indirect w/ Displacement

## Example:

① Initial Conditions:  $A[0] = 0000703C$

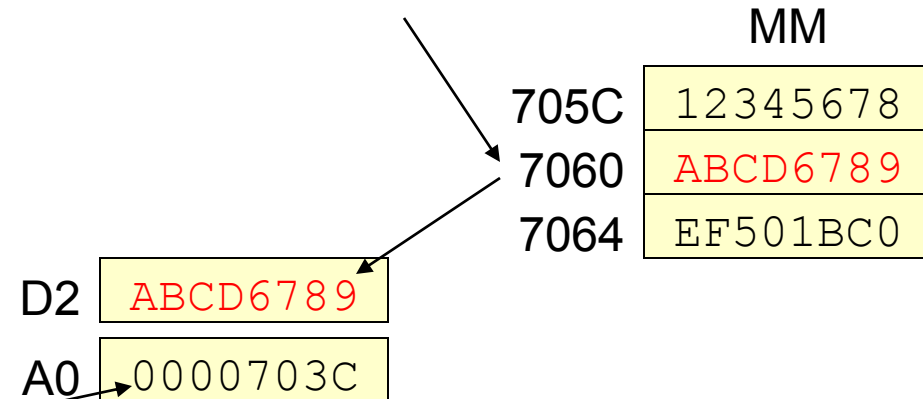
**MOVE .L**     $(0x24, A0), D2$

②  $\langle EA \rangle = A[0] + \text{Displacement}$

③ Use  $\langle EA \rangle$  to access memory

④  $\langle EA \rangle$  discarded ( $A0$  is left w/ original value)

$$\begin{array}{rcl}
 & \underbrace{0x24} & \underbrace{A0} \\
 & \downarrow & \downarrow \\
 & + & 703C = (A0) \\
 & \underline{+ 0024} & = \text{Disp.} \\
 & 7060 & = \langle EA \rangle
 \end{array}$$



*A0 is left unchanged*



# Address Register Indirect w/ Displacement

- Good for use as a pointer to access fields of a record/structure/class

```
struct mystruct {
    int x;
    int y;
} data[2];

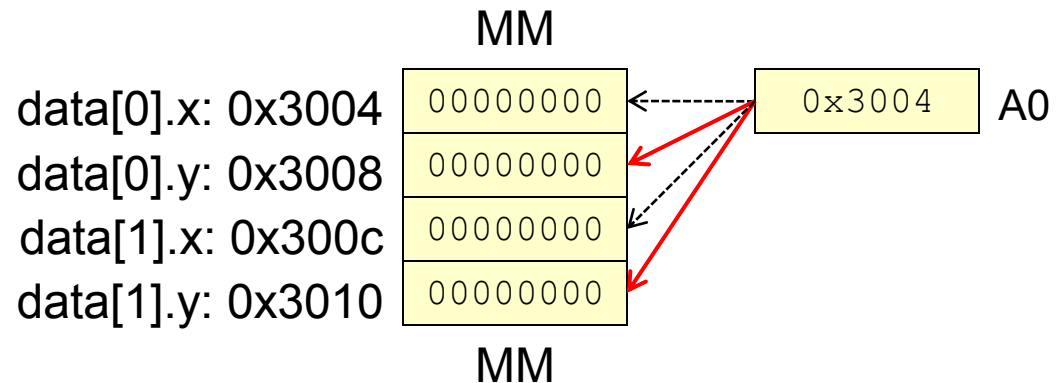
for(i=0; i<2; i++)
    data[i].y = 1;
```

C Code

```
movea.l #0x3004, a0
move.l  #1, d0
loop 2 times {

    move.l d0, (4, a0)
    adda.l #8, a0

}
```



# Address Register Indirect w/ Index

- Specifies a memory location
- Use  $An$  as base address and adds a displacement value and product of the contents of another Data or Address register times a scale factor (1,2,4) to come up w/ effective address (<EA>)
- Displacement limited to 8-bit signed number
- $An$  not affected (maintains original address)

**Example:**

MOVE.L (0x24, A0, D1.L\*4), D2

*Displacement Value*

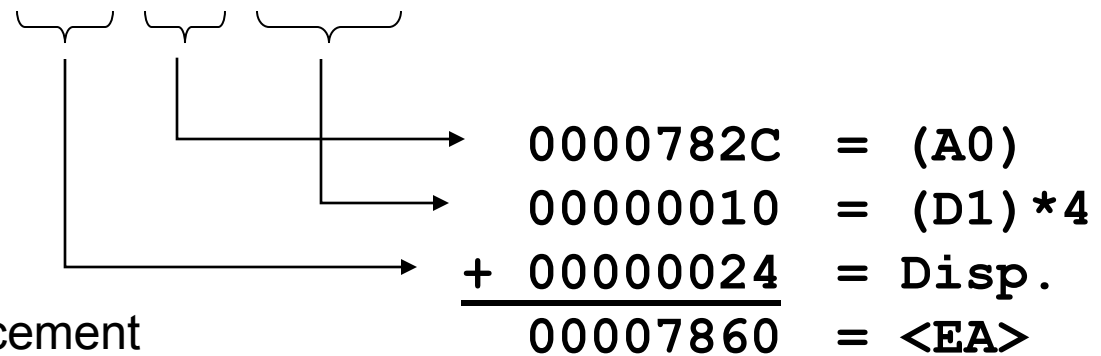
*Addr. Reg. Indirect w/  
Index Mode*

# Address Register Indirect w/ Index

## Example:

① Initial Conditions:  $A[0] = 0000782C$ ,  $D[1] = 00000004$

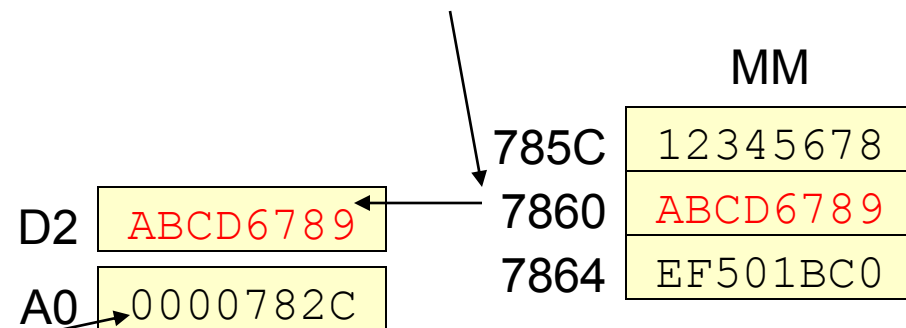
**MOVE.L (0x24, A0, D1.L\*4), D2**



②  $\langle EA \rangle = A[0] + D[1] + \text{Displacement}$

③ Use  $\langle EA \rangle$  to access memory

④  $\langle EA \rangle$  discarded (A0 is left w/ original value)



*A0 is left unchanged*

# Address Register Indirect w/ Index

## Example:

① Initial Conditions:  $A[0] = 0000703C$ ,  $D[1] = FFFFFFF0$

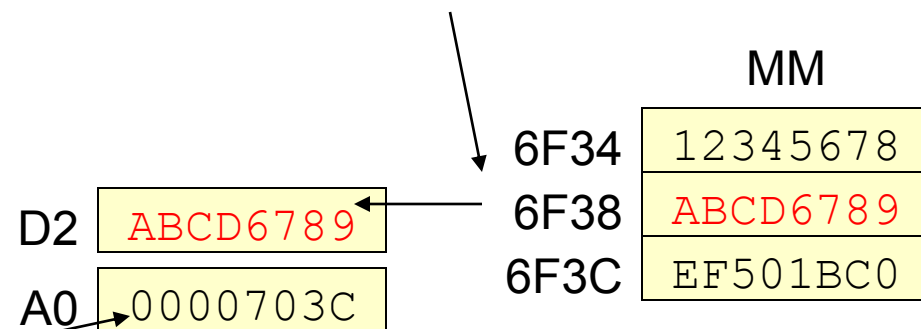
**MOVE.L (0xFC, A0, D1.L\*4), D2**



②  $\langle EA \rangle = A[0] + D[1] + \text{Displacement}$

③ Use  $\langle EA \rangle$  to access memory

④  $\langle EA \rangle$  discarded (A0 is left w/ original value)



*A0 is left unchanged*

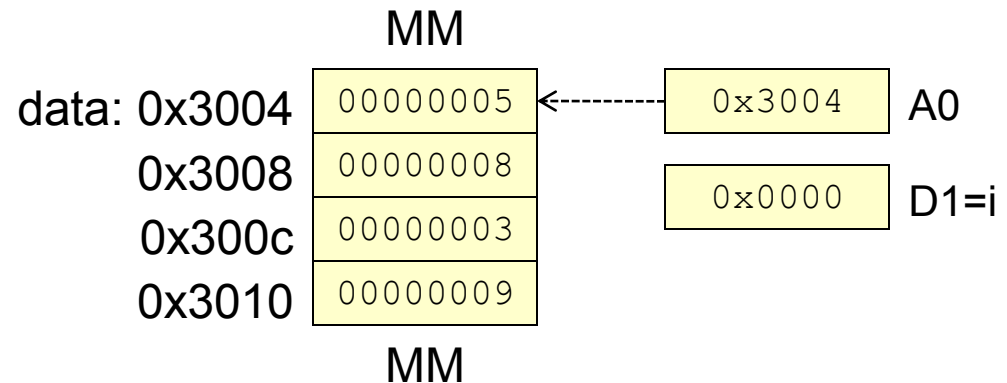
# Address Register Indirect w/ Index

- Good for use as a pointer to access an array

```
int data[4] = {5,8,3,9};
for(i=0; i<4; i++)
    data[i] = 1;
```

C Code

```
movea.l #0x3004,a0 ; base ptr.
move.l #1,d0        ; val to move
move.l #0,d1        ; i cntr.
loop 4 times {
    move.l d0,(0,a0,d1.L*4)
    addi.l #1,d1
}
```



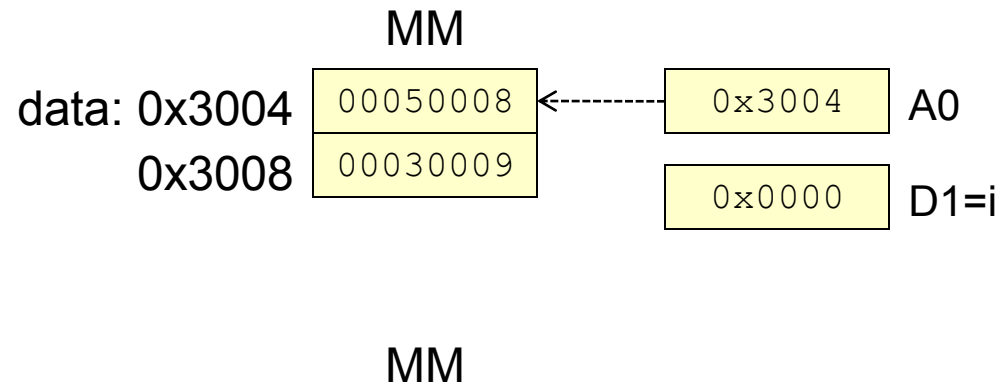
# Address Register Indirect w/ Index

- Good for use as a pointer to access an array

```
short data[4] = {5,8,3,9};
for(i=0; i<4; i++)
    data[i] = 1;
```

C Code

```
movea.l #0x3004,a0 ; base ptr.
move.l #1,d0        ; val to
movemove.l #0,d1      ; i cntr.
loop 4 times {
    move.l d0,(0,a0,d1.L*2)
    addi.l #1,d1
}
```



# Short Absolute Address

- Specifies the exact memory location
- Short address requires only 16-bits but MSB must be 0 ( ∴ 0000-7FFF)

**Example:**

MOVE .L 0x7060 , D2

*Short Absolute Address*

*Absolute Address  
Mode*

# Short Absolute Address

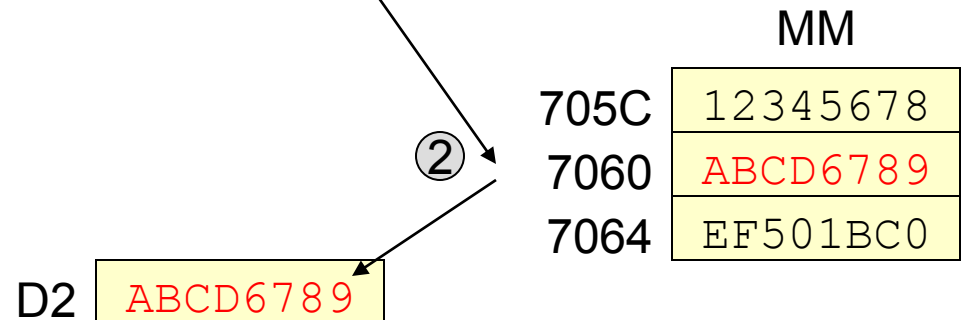
## Example:

*This is considered a short address because it is  $\leq 0x7FFF$*

**MOVE.L    0x7060, D2**

①  $\langle EA \rangle = 0x7060$

② Access Longword at 0x7060





# Long Absolute Address

- Specifies the exact memory location
- Long address requires more than 16-bits (also includes when Short address has MSB=1  $\therefore$  8000-FFFFFF)

**Example:**

MOVE .L 0x18060, D2

*Long Absolute Address*

*Absolute Address  
Mode*

# Long Absolute Address

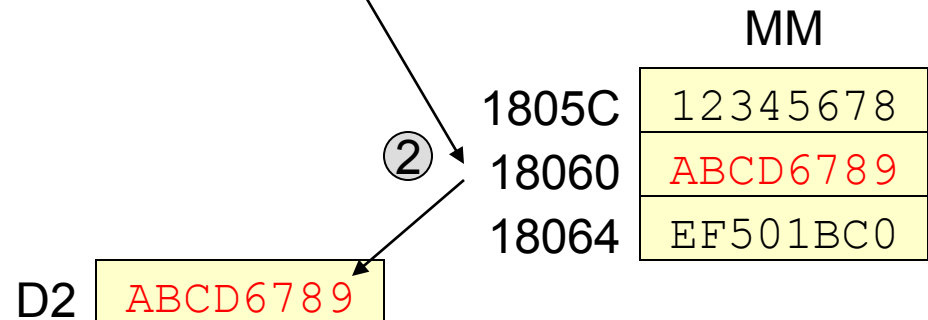
## Example:

*This is considered a long address because it is  $\geq 0x8000$*

**MOVE .L    0x18060 , D2**

① <EA> = 0x18060

② Access Longword at 0x18060



# Immediate Operands

- One mode indicating the operand is stored as part of the instruction
  - Immediate Mode

# Immediate Mode

- Places the exact data into the destination
- '#' indicates Immediate Mode
- The immediate value will be zero-extended to the size indicated by the instruction

**Example:**

MOVE.L #0x7060, D2

*Immediate Value*

*Immediate Mode*

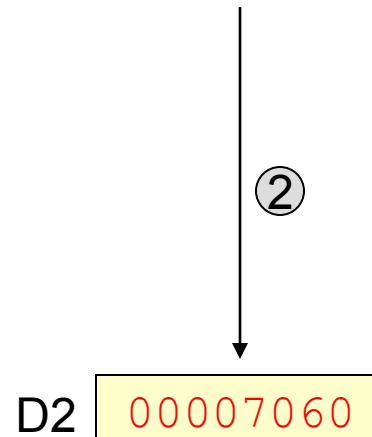
# Immediate Mode

## Example:

① Initial Conditions: (D2) = ABCD1234

**MOVE .L    #0x7060 , D2**

② Move constant 0x00007060 to D2



# Overview

Location of Operand	Addressing Modes	Operand	Assembly Notation
<b>Register Contents</b>	<b>Data Register Direct</b>	<b>D[n]</b>	<b>D0</b>
	<b>Address Register Direct</b>	<b>A[n]</b>	<b>A1</b>
<b>Memory Location</b>	<b>Address Register Indirect (A.R.I.)</b>	<b>M[A[n]]</b>	<b>(A2)</b>
	<b>A.R.I. w/ postincrement</b>	<b>M[A[n]]</b>	<b>(A2)+</b>
	<b>A.R.I. w/ predecrement</b>	<b>M[A[n] - {1,2,4}]</b>	<b>-(A2)</b>
	<b>A.R.I. w/ displacement</b>	<b>M[A[n] + displacement]</b>	<b>(0x24,A2)</b>
	<b>A.R.I. w/ scaled index &amp; displacement</b>	<b>M[A[n] + disp. + R[m]*size]</b>	<b>(0x24,A2,D0.L*4)</b>
	<b>Absolute Addressing Short</b>	<b>M[address] (e.g. M[0x7060])</b>	<b>0x7060</b>
	<b>Absolute Addressing Long</b>	<b>M[address] (e.g. M[0x18000])</b>	<b>0x18000</b>
<b>Immediate</b>	<b>Immediate</b>	<b>Immediate</b>	<b>#0x7800</b>

# Addressing Mode Examples

Initial Contents: A[0] = 0, A[1] = 0, D[0] = 3

		(A0)	(A1)
1	MOVEA.L #0x00007000, A0	0x00007000	
2	MOVEA.L #0x00007008, A1		
3	MOVE.B (A0)+, (A1)+		
4	MOVE.B (A0)+, 0x7(A0)		
5	ADDA.L #1, A0		
6	MOVE.B (A0), 1(A1)		
7	MOVE.B -(A0), -1(A1, D0.L)		

**MOVEA.L #0x00007000, A0**

**Main Memory**

7000	1A	1B	1C	1D
7004	00	00	00	00
7008	00	00	00	00

# Addressing Mode Examples

Initial Contents: A[0] = 0, A[1] = 0, D[0] = 3

		(A0)	(A1)
1	MOVEA.L #0x00007000, A0	0x00007000	
2	MOVEA.L #0x00007008, A1		0x00007008
3	MOVE.B (A0)+, (A1)+		
4	MOVE.B (A0)+, 0x7(A0)		
5	ADDA.L #1, A0		
6	MOVE.B (A0), 1(A1)		
7	MOVE.B -(A0), -1(A1, D0.L)		

**MOVEA.L #0x00007008, A1**

**Main Memory**

7000	1A	1B	1C	1D
7004	00	00	00	00
7008	00	00	00	00



# Addressing Mode Examples

Initial Contents: A[0] = 0, A[1] = 0, D[0] = 3

		(A0)	(A1)
1	MOVEA.L #0x00007000, A0	0x00007000	
2	MOVEA.L #0x00007008, A1		0x00007008
3	MOVE.B (A0)+, (A1)+	0x00007001	0x00007009
4	MOVE.B (A0)+, 0x7(A0)		
5	ADDA.L #1, A0		
6	MOVE.B (A0), 1(A1)		
7	MOVE.B -(A0), -1(A1, D0.L)		

**MOVE.B (A0)+, (A1)+**

1. Get Value pointed to by A0 => 0x1A
2. Increment A0 => 0x7001
3. Place value in location pointed to by A1 => 0x7008
4. Increment A1 => 0x7009

**Main Memory**

7000	1A	1B	1C	1D
7004	00	00	00	00
7008	1A	00	00	00

# Addressing Mode Examples

Initial Contents: A[0] = 0, A[1] = 0, D[0] = 3

		(A0)	(A1)
1	MOVEA.L #0x00007000, A0	0x00007000	
2	MOVEA.L #0x00007008, A1		0x00007008
3	MOVE.B (A0)+, (A1)+	0x00007001	0x00007009
4	MOVE.B (A0)+, 0x7(A0)	0x00007002	
5	ADDA.L #1, A0		
6	MOVE.B (A0), 1(A1)		
7	MOVE.B -(A0), -1(A1, D0.L)		

**MOVE.B (A0)+, 0x7(A0)**

1. Get Value pointed to by A0 => 0x1B
2. Increment A0 => 0x7002
3. Place value in location pointed to by 7+(A0) => 0x7009

**Main Memory**

7000	1A	1B	1C	1D
7004	00	00	00	00
7008	1A	1B	00	00

# Addressing Mode Examples

Initial Contents: A[0] = 0, A[1] = 0, D[0] = 3

		(A0)	(A1)
1	MOVEA.L #0x00007000, A0	0x00007000	
2	MOVEA.L #0x00007008, A1		0x00007008
3	MOVE.B (A0)+, (A1)+	0x00007001	0x00007009
4	MOVE.B (A0)+, 0x7(A0)	0x00007002	
5	ADDA.L #1, A0	0x00007003	
6	MOVE.B (A0), 1(A1)		
7	MOVE.B -(A0), -1(A1, D0.L)		

**ADDA.L #1, A0**

**1. Add 1 to (A0) => 0x7003**

**Main Memory**

7000	1A 1B 1C 1D
7004	00 00 00 00
7008	<b>1A 1B</b> 00 00

# Addressing Mode Examples

Initial Contents: A[0] = 0, A[1] = 0, D[0] = 3

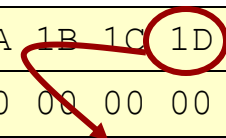
		(A0)	(A1)
1	MOVEA.L #0x00007000, A0	0x00007000	
2	MOVEA.L #0x00007008, A1		0x00007008
3	MOVE.B (A0)+, (A1)+	0x00007001	0x00007009
4	MOVE.B (A0)+, 0x7(A0)	0x00007002	
5	ADDA.L #1, A0	0x00007003	
6	MOVE.B (A0), 1(A1)	0x00007003	0x00007009
7	MOVE.B -(A0), -1(A1, D0.L)		

**MOVE.B (A0), 1(A1)**

1. Get Value pointed to by A0 => 0x1D
2. Place value in location pointed to by 1+(A1) => 0x700A

**Main Memory**

7000	1A	1B	1C	1D
7004	00	00	00	00
7008	1A	1B	1D	00



# Addressing Mode Examples

Initial Contents: A[0] = 0, A[1] = 0, D[0] = 3

		(A0)	(A1)
1	MOVEA.L #0x00007000, A0	0x00007000	
2	MOVEA.L #0x00007008, A1		0x00007008
3	MOVE.B (A0)+, (A1)+	0x00007001	0x00007009
4	MOVE.B (A0)+, 0x7(A0)	0x00007002	
5	ADDA.L #1, A0	0x00007003	
6	MOVE.B (A0), (1, A1)	0x00007003	0x00007009
7	MOVE.B -(A0), (-1, A1, D0.L*1)	0x00007002	0x00007009

**MOVE.B -(A0), (-1, A1, D0.L\*1)**

1. Decrement A0 by 1 => 0x7002
2. Get Value pointed to by A0 => 0x1C
3. Place value in location pointed to by -1+(A1)+(D0) => 0x700B

**Main Memory**

7000	1A	1B	1C	1D
7004	00	00	00	00
7008	1A	1B	1D	1C

Instruction Format and Length

# MACHINE CODE TRANSLATION

# Instruction Format

- Instructions can range from 1 to 3 words
- 1<sup>st</sup> word is always the instruction word
  - Indicates the operation and addressing modes
- 2<sup>nd</sup> – 3<sup>rd</sup> words are extension words
  - Only used with certain addressing modes

1	Instruction Word
2	Ext. Word 1
3	Ext. Word 2 (if needed)

# Extension Words

- Displacement Mode
  - 16-bit disp. value stored as 1 ext. word
- Absolute Addr. Mode
  - Short Mode stored as 1 ext. word
  - Long Mode stored as 2 ext. words

MOVE.W 0x24(A0),D0

MOVE.W
0024

MOVE.B 0x7060,D0

MOVE.B
7060

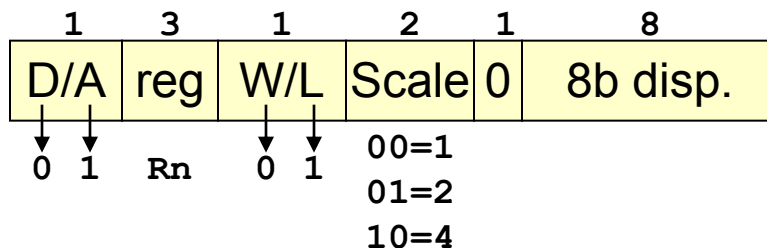
MOVE.B 0x2A7000,(A0)+

MOVE.B
002A
7000



# Extension Words (cont.)

- Immediate Mode
  - .B and .W values stored as 1 ext. word
  - .L stored as 2 ext. words
- Index Mode w/ Disp.
  - 1 ext. word to encode index reg. and disp. value



MOVE.B #-1,D0

MOVE.B
00FF

MOVE.W #-1,D0

MOVE.W
FFFF

MOVE.L #-1,(A0)+

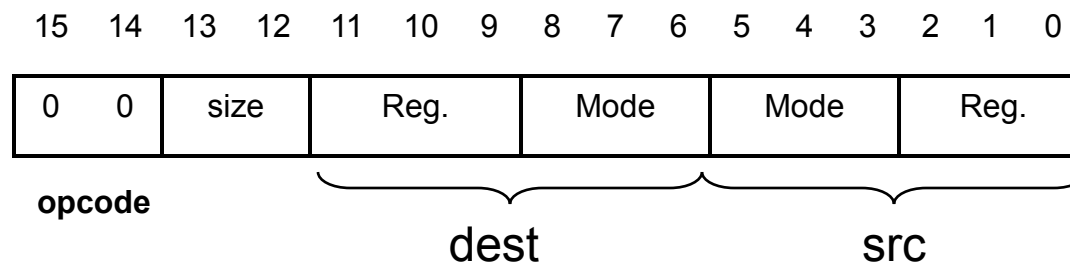
MOVE.B
FFFF
FFFF

MOVE.W 0x24(A0,D1.L\*2),D2

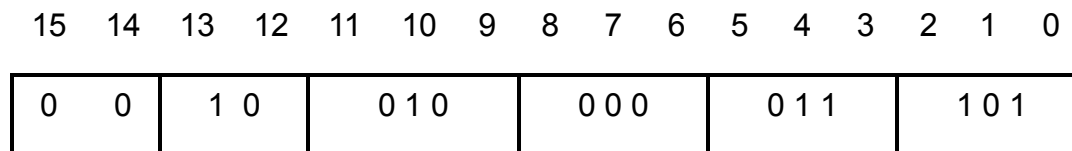
MOVE.W
1A24

# MOVE Translation

- Instruction word format:

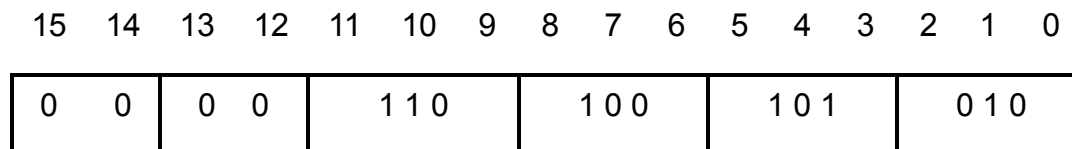


**MOVE.L (A5)+,D2**



**= 241D**

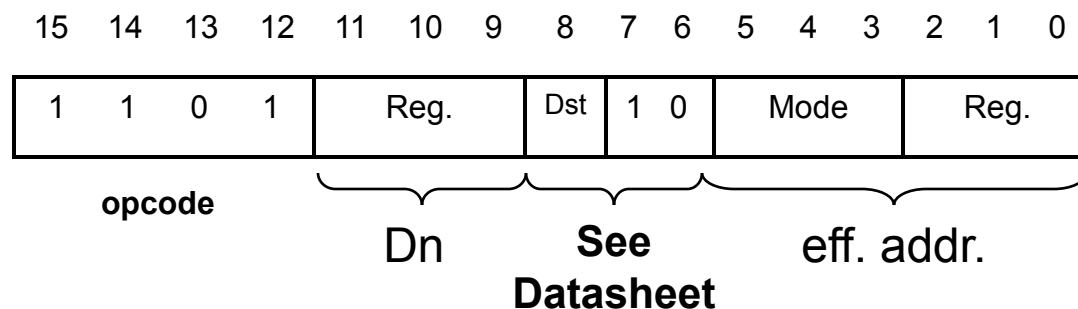
**MOVE.W 0x42 (A2), -(A6)**



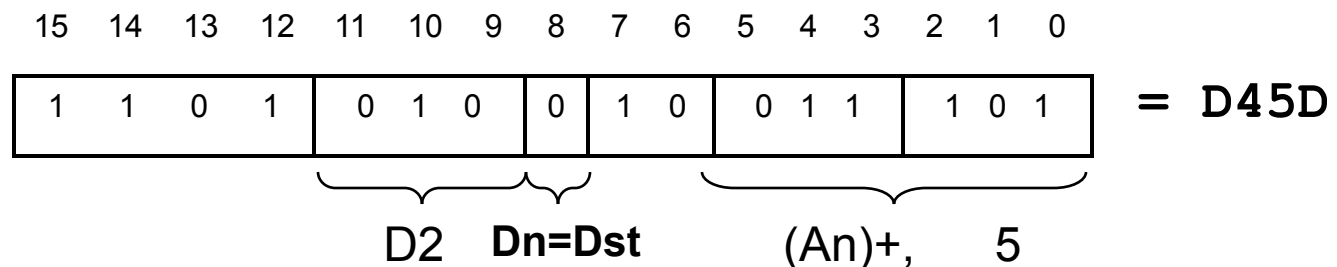
**= 0D2A  
0042**

# ADD Translation

- ALU instructions like ADD must have one operand as the data register
- Instruction word format:



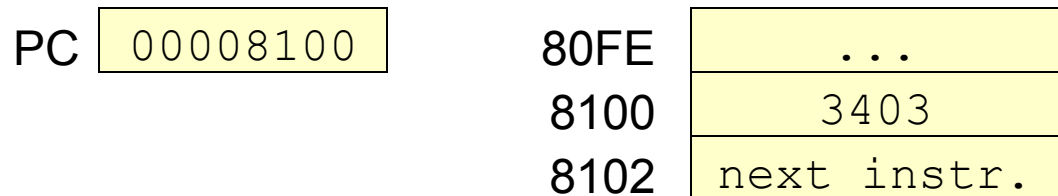
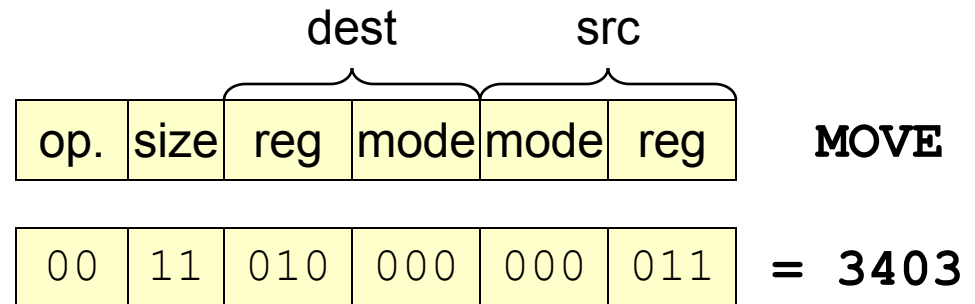
**ADD.L (A5)+, D2**



# Data Register Direct

- Before Instruction  
Fetch: (PC)=8100
- After Instruction  
Fetch: (PC)=8102
- After Execution:  
(PC)=8102

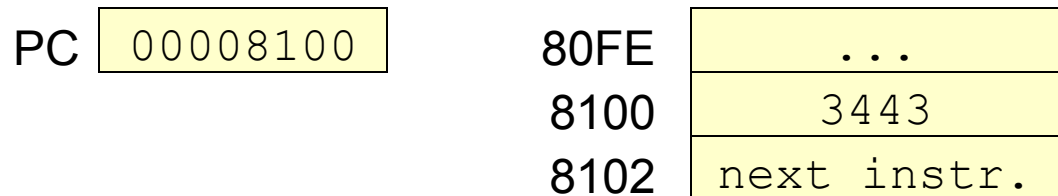
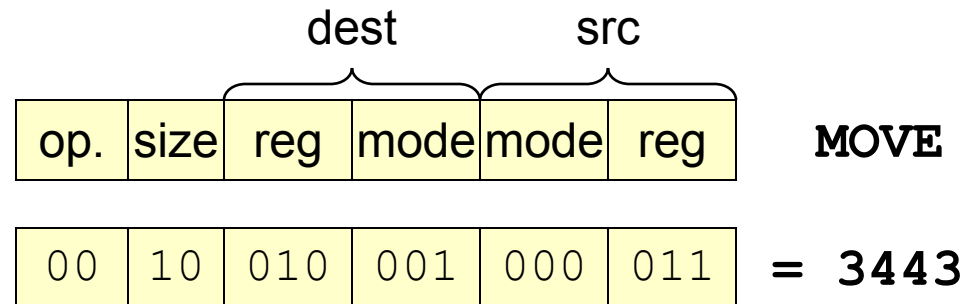
**MOVE.W D3,D2**



# Address Register Direct

- Before Instruction  
Fetch: (PC)=8100
- After Instruction  
Fetch: (PC)=8102
- After Execution:  
(PC)=8102

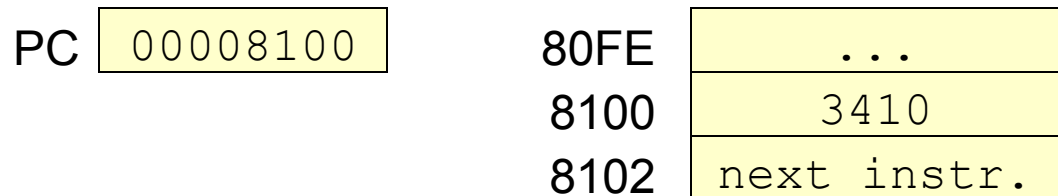
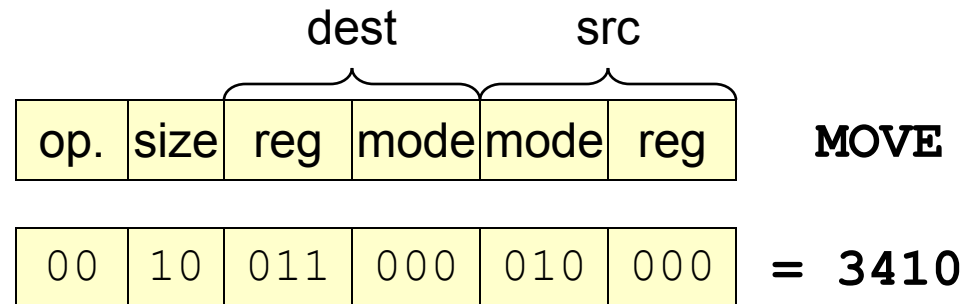
**MOVEA.L D3,A2**



# Address Register Indirect

- Before Instruction  
Fetch: (PC)=8100
- After Instruction  
Fetch: (PC)=8102
- After Execution:  
(PC)=8102

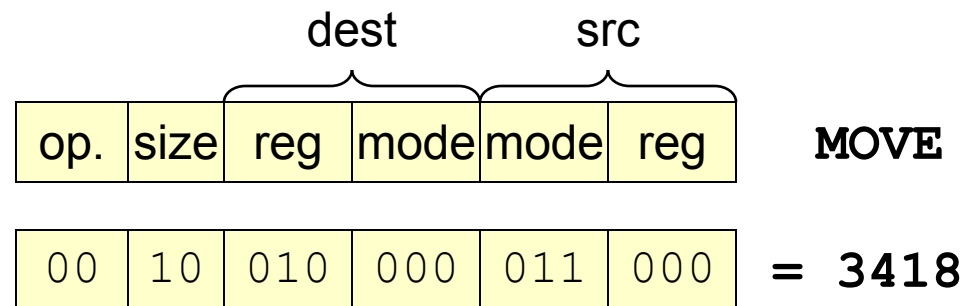
**MOVE.L (A0), D3**



# Address Register Indirect w/ Postincrement

- Before Instruction  
Fetch: (PC)=8100
- After Instruction  
Fetch: (PC)=8102
- After Execution:  
(PC)=8102

**MOVE.L (A0)+,D2**

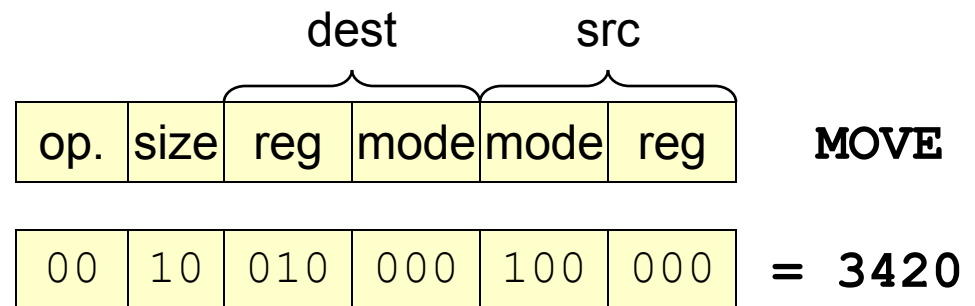


PC	00008100	80FE	...
		8100	3418
		8102	next instr.

# Address Register Indirect w/ Predecrement

- Before Instruction  
Fetch: (PC)=8100
- After Instruction  
Fetch: (PC)=8102
- After Execution:  
(PC)=8102

**MOVE .L - (A0) ,D2**



PC 

00008100
----------

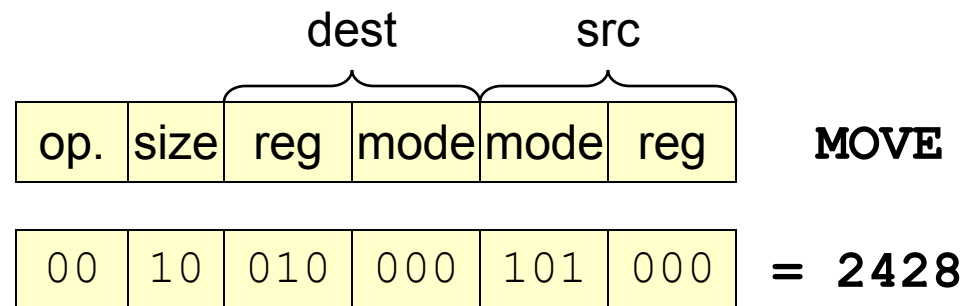
80FE	...
8100	3420
8102	next instr.



# Address Register Indirect w/ Displacement

- Displacement is always stored as a word after the instruction
- Before Instruction Fetch: (PC)=8100
- After Instruction Fetch: (PC)=8102
- After Execution: (PC)=8104

**MOVE.L (0x24, A0), D2**



PC 

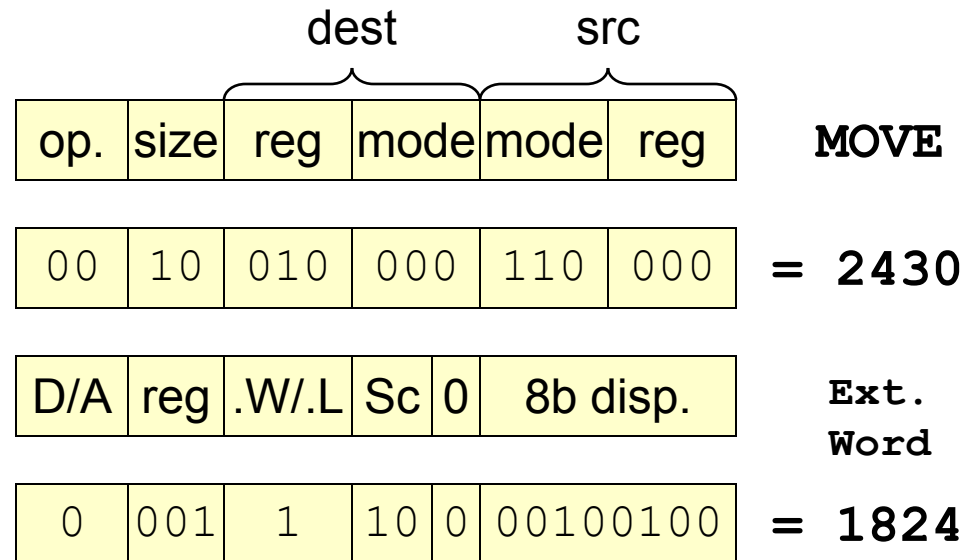
00008100
----------

80FE	...
8100	2428
8102	0024
8104	next instr.

# Address Register Indirect w/ Index

- Index and displacement info is stored as a word after the instruction
- Before Instruction Fetch (PC)=0x8100
- After Instruction Fetch:  
(PC)=0x8102
- After Execution:  
(PC)=0x8104

**MOVE.L (0x24,A0,D1.L\*4),D2**

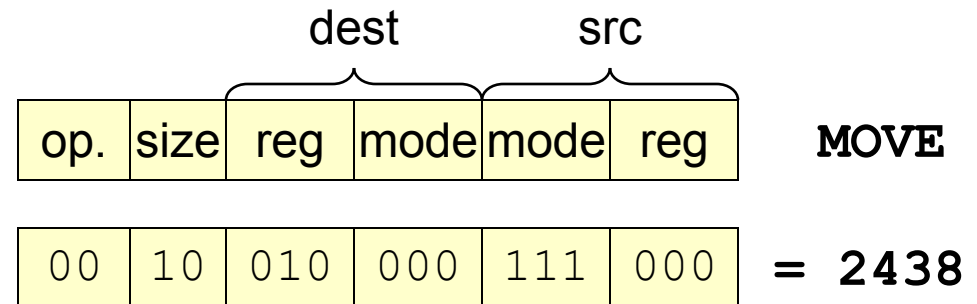


PC	00008100	80FE	...
		8100	2430
		8102	1824
		8104	next instr.

# Short Absolute Address

- Short absolute address is always stored as a word after the instruction
- Before Instruction Fetch: (PC)=8100
- After Instruction Fetch: (PC)=8102
- After Execution: (PC)=8104

**MOVE.L 0x7060,D2**

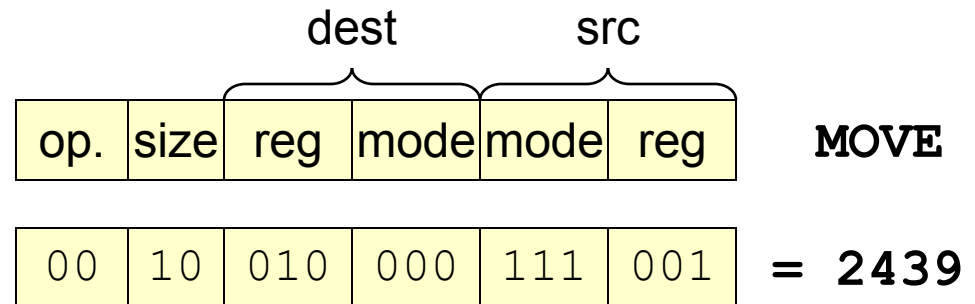


PC	00008100	80FE	...
		8100	2438
		8102	7060
		8104	next instr.

# Long Absolute Address

- Long absolute  
Address is stored  
as a longword after  
instruction
- Before Instruction  
Fetch: (PC)=8100
- After Instruction  
Fetch: (PC)=8102
- After Execution:  
(PC)=8106

**MOVE.L 0x18060,D2**



PC 

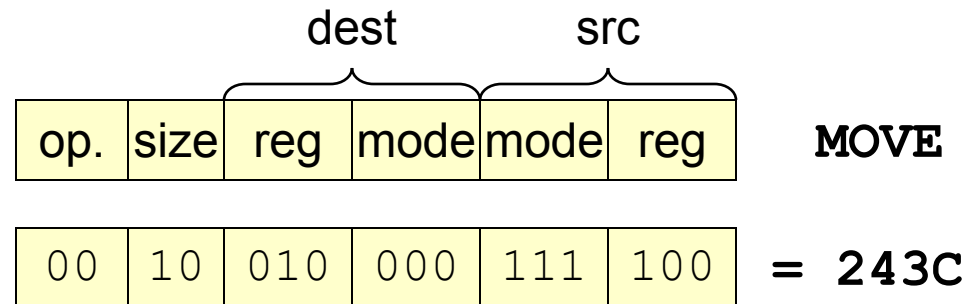
00008100
----------

80FE	...
8100	2439
8102	0001
8104	8060
8106	next instr.

# Immediate Mode

- Immediate data is stored as longword if size .L is used. It is stored as a word if size is .B or .W
- Before Instruction Fetch: (PC)=8100
- After Instruction Fetch: (PC)=8102
- After Execution: (PC)=8106

**MOVE.L #0x7060,D2**



PC	00008100	80FE	...
		8100	243C
		8102	0000
		8104	7060
		8106	next instr.