Address Modes

- An address mode refers to the way in which an operand's location is specified

- Some examples that we have already seen include
  - `ADD.W  D0,D1`
  - `ADD.W  $9000,D1`
  - `ADD.W  #$9000,D1`

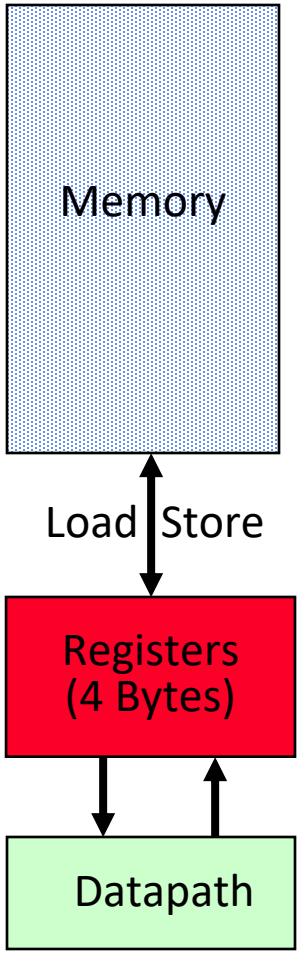
- The 68000 supports 13 unique address modes
  - Not all address modes can be used with all instructions/operands

- Why support more than one address mode?
  - in theory one address mode suffices, but
  - more powerful address modes increases the efficiency of programs

Data Register Direct

- Operand is contained inside a data register
  - fastest address mode
  - shortest instructions
  - typically used for temporary values and frequently accessed variables

Assembler Instruction	Machine Language
`move.b d0,d3`	`1600`



C and Data-Register Direct Addressing

- The C register keyword can be used as a hint to the compiler that the variable should be stored in a register rather than RAM

- Consider the following C code

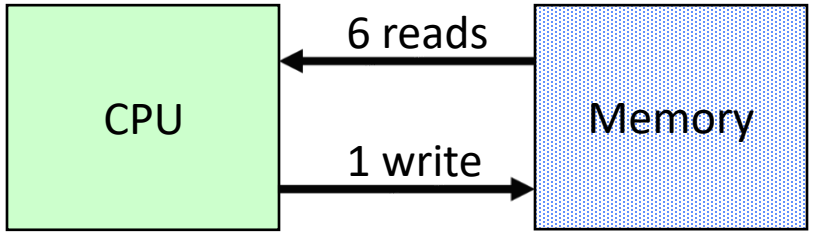
```
for(register int i=0; i < bufsize; i++) {

    // do something
}
```

Absolute Long Addressing

- Operand is in memory and address of operand is contained in two extension words as part of the instruction
  - slowest address mode
  - longest instructions
  - used to access simple global variables

Assembler Instruction	Machine Language
`move.b $9000,$9001`	`13F9 0000 9000 0000 9001`

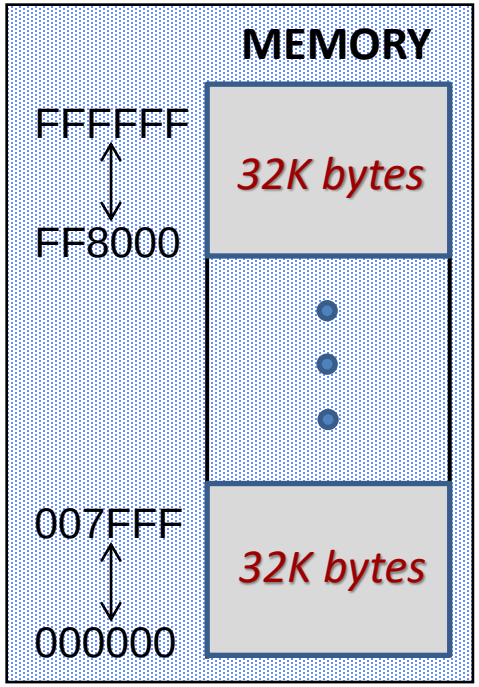


Absolute Short Addressing

- Operand is in memory and address of operand is contained in one extension word as part of the instruction

Assembler Instruction	Machine Language
`move.w $ff8000,d0`	`2038 8000`

Assembler Instruction	Machine Language
`move.w $007fff,d0`	`2038 7fff`



Assembler Treatment of Absolute Addressing

- Absolute addressing is usually determined by the assembler, but can be explicitly stated by the programmer by using
  - Absolute Short: xxxx.W or <xxxx
  - Absolute Long: xxxx.L or >xxxx

```
1    00008000                          ORG       $8000

2    00008000 3038 7000                MOVE.W    $7000,D0
3    00008004 3039 0000 7000           MOVE.W    >$7000,D0

4    0000800A 3039 0000 8000           MOVE.W    $8000,D0
5    00008010 3038 8000                MOVE.B    $FFFF8000,D0

6    00008014 13F8 1234 0000 5678      MOVE.B    $1234.W,$5678.L
```

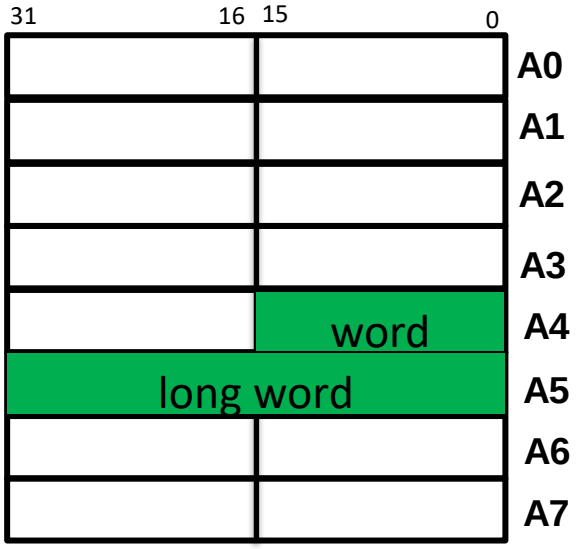
Immediate Addressing

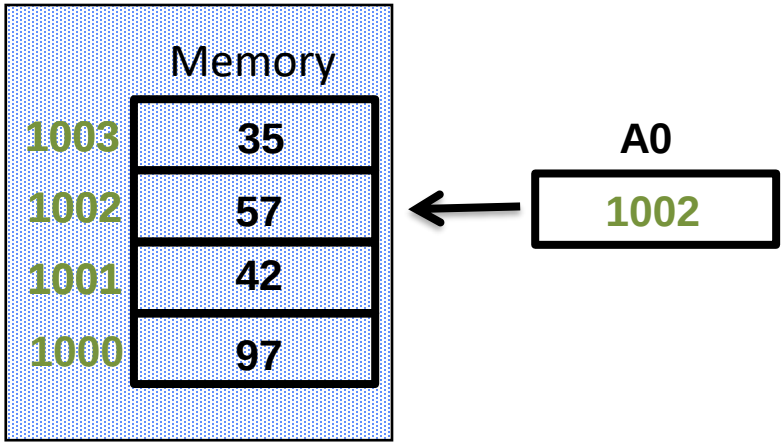
- Operand is contained inside the instruction using one or two extension words
  - specified using # symbol
  - used for all true constants and to initialize memory and registers
  - can only be used to specify a source operand

Assembler Instruction	Machine Language	Result
`move.l #$ff,d0`	`203C 0000 00FF`	`D0 = $000000FF`
`move.w #$ff,d0`	`303C 00FF`	`D0 = $----00FF`
`move.b #$ff,d0`	`103C 00FF`	`D0 = $------FF`
`move.l #$1f,d0`	`203C 0000 001F`	`D0 = $0000001F`
`moveq #$1f,d0`	`701F`	`D0 = $0000001F`

Address Registers

- Address registers are used to hold the address of a memory location
  - Eight address registers, each 32 bits
  - Byte operations not allowed
  - Special one: A7 is used as a stack pointer





Memory[A0] = Memory[1002] = 57

Address Registers != Data Registers

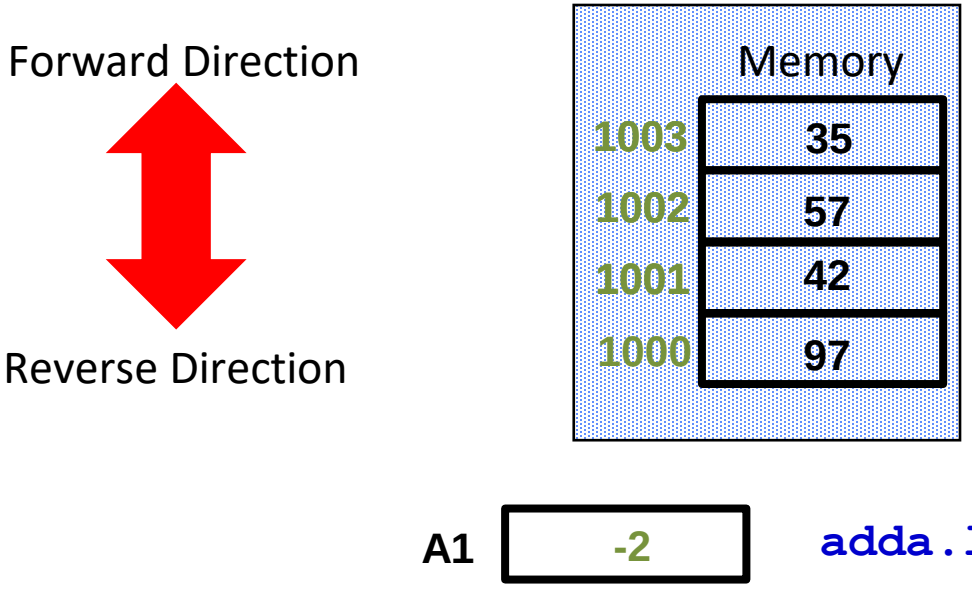
- If an address register is specified as the destination operand, the following instructions must be used
  - MOVEA, ADDA, SUBA, **(CCR flags NOT affected)**

- Data registers use the following instructions
  - MOVE, ADD, SUB **(CCR flags affected)**

Instruction	Correct (Y/N)?
`adda.l d3,a3`	Y
`add.l d3,a3`	N
`add.l a3,d3`	Y

Address Registers != Data Registers

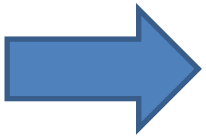
- Address stored in an address register is considered to be a 32-bit signed entity



Sign-Extension of Word Addresses

- All addresses are sign-extended to 32 bits for word operations

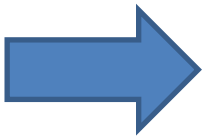
```
adda.l #$fff4,a0
```



```
a0 = a0 + $0000fff4
```

Machine language: D1FC 0000 FFF4

```
adda.w #$fff4,a0
```

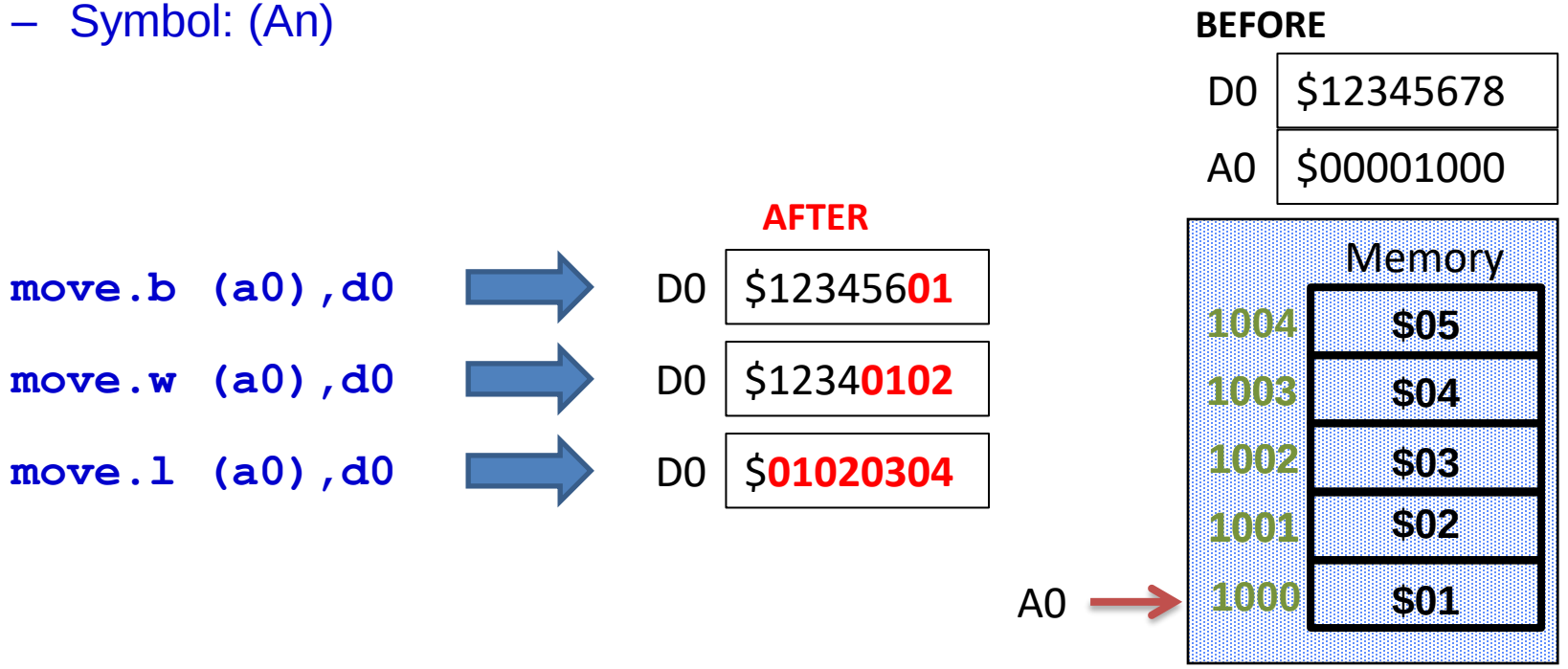


```
a0 = a0 + $ffffffff4
```

Machine language: D0FC FFF4

Indirect Addressing

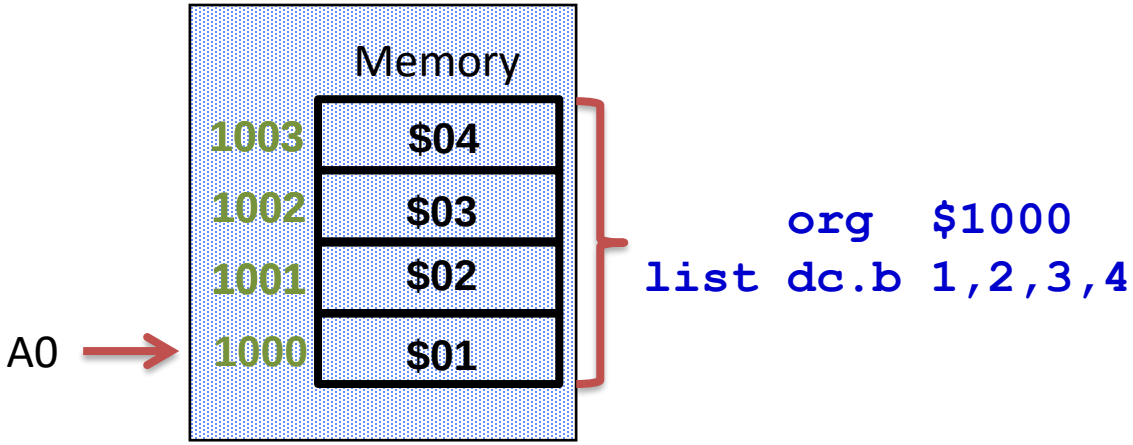
- Operand is in memory and address of operand is contained inside an address register
  - Symbol: (An)



Indirect Addressing and Pointers in C

- Consider the example

```
ptr = &list;
first = *ptr;
```



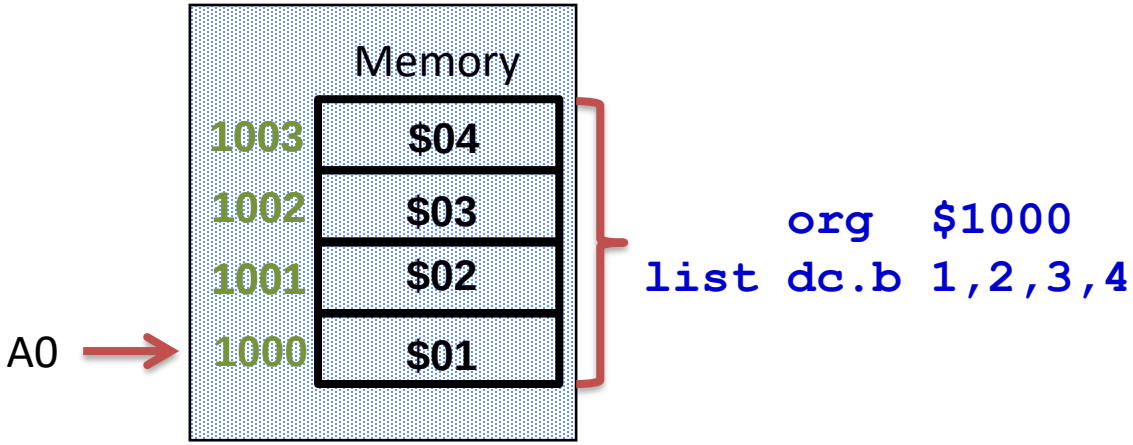
- Assume that the list variables are bytes (i.e., chars) and **`first`** is contained in D0 and **`ptr`** is to be implemented using A0

```
movea.l #list,a0        ;a0 = 1000
move.b  (a0),d0         ;d0 = memory[1000] = 1
```

Indirect Addressing and Pointers in C

- Consider the example

```
ptr = &list;
first = *ptr;
```



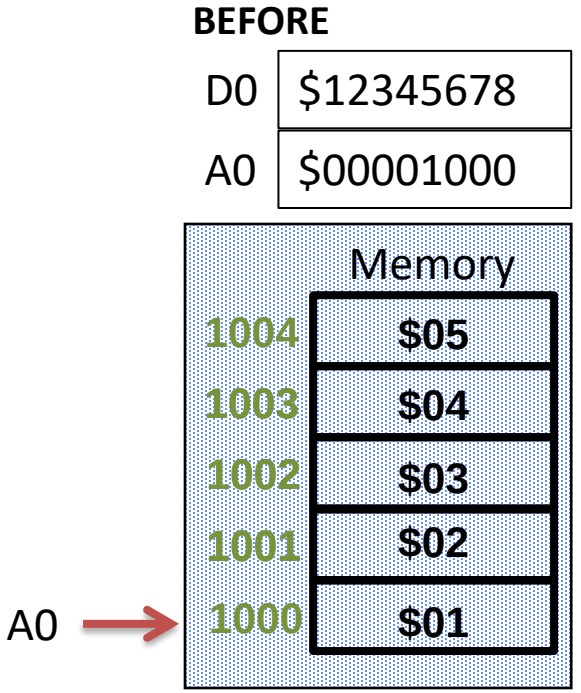
- Assume that the list variables are bytes (i.e., chars) and **`first`** is contained in D0 and **`ptr`** is to be implemented using A0

```
lea     list,a0          ;a0 = 1000
move.b  (a0),d0          ;d0 = memory[1000] = 1
```

Post Increment Indirect Addressing

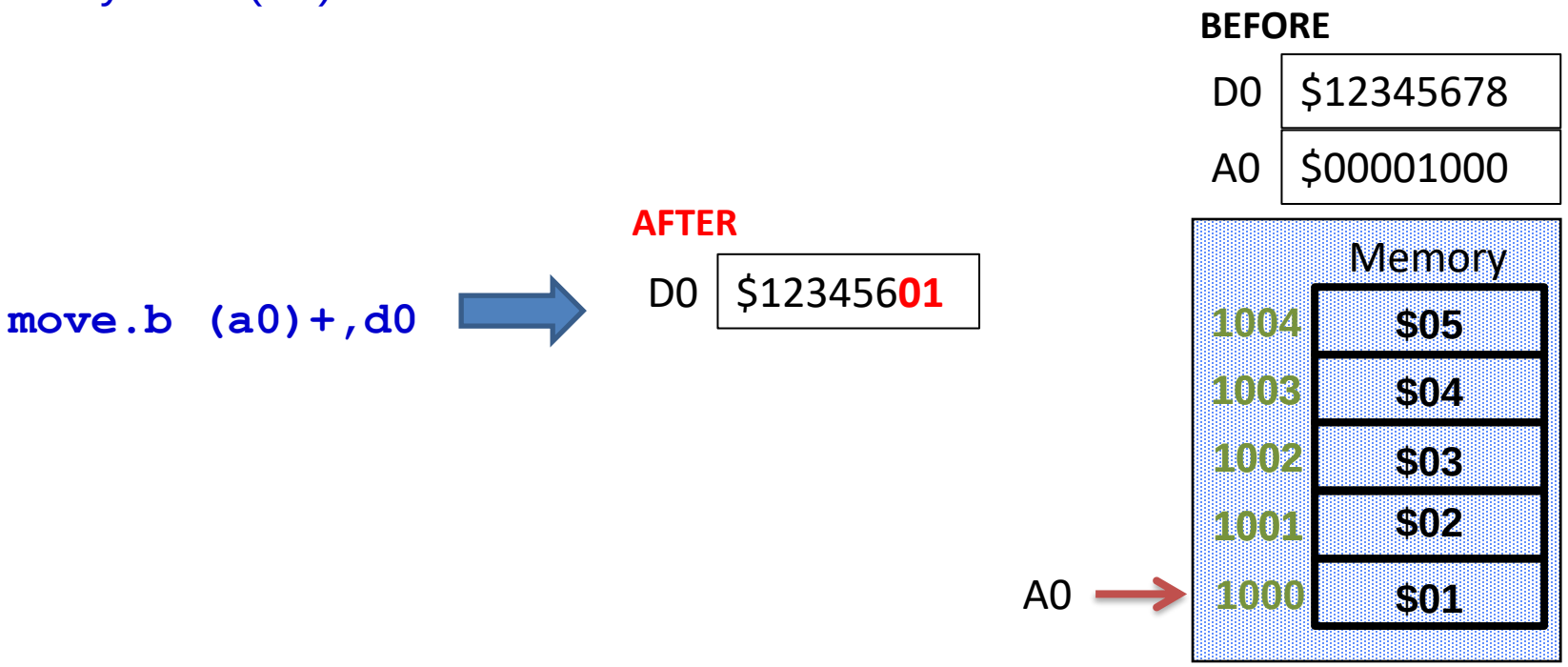
- Same as indirect, but after operand is used, the address in the register is incremented by the size of the operand (in bytes)
  - Symbol: (An)+

```
move.b (a0)+,d0
```



Post Increment Indirect Addressing

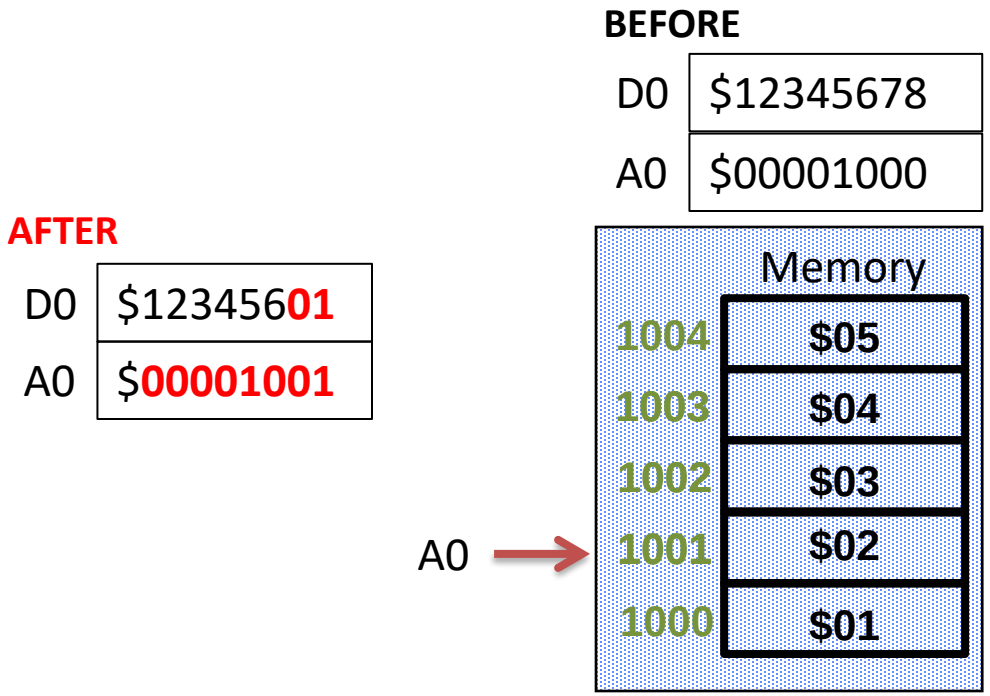
- Same as indirect, but after operand is used, the address in the register is incremented by the size of the operand (in bytes)
  - Symbol: (An)+



Post Increment Indirect Addressing

- Same as indirect, but after operand is used, the address in the register is incremented by the size of the operand (in bytes)
  - Symbol: (An)+

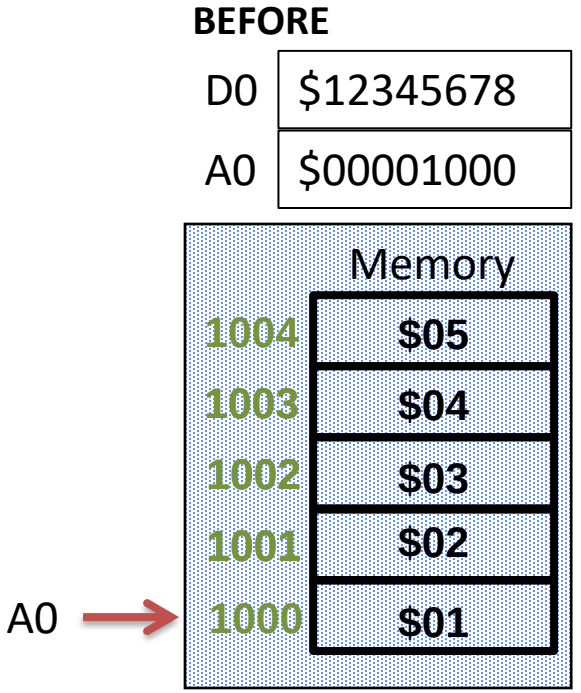
```
move.b (a0)+,d0
```



Post Increment Indirect Addressing

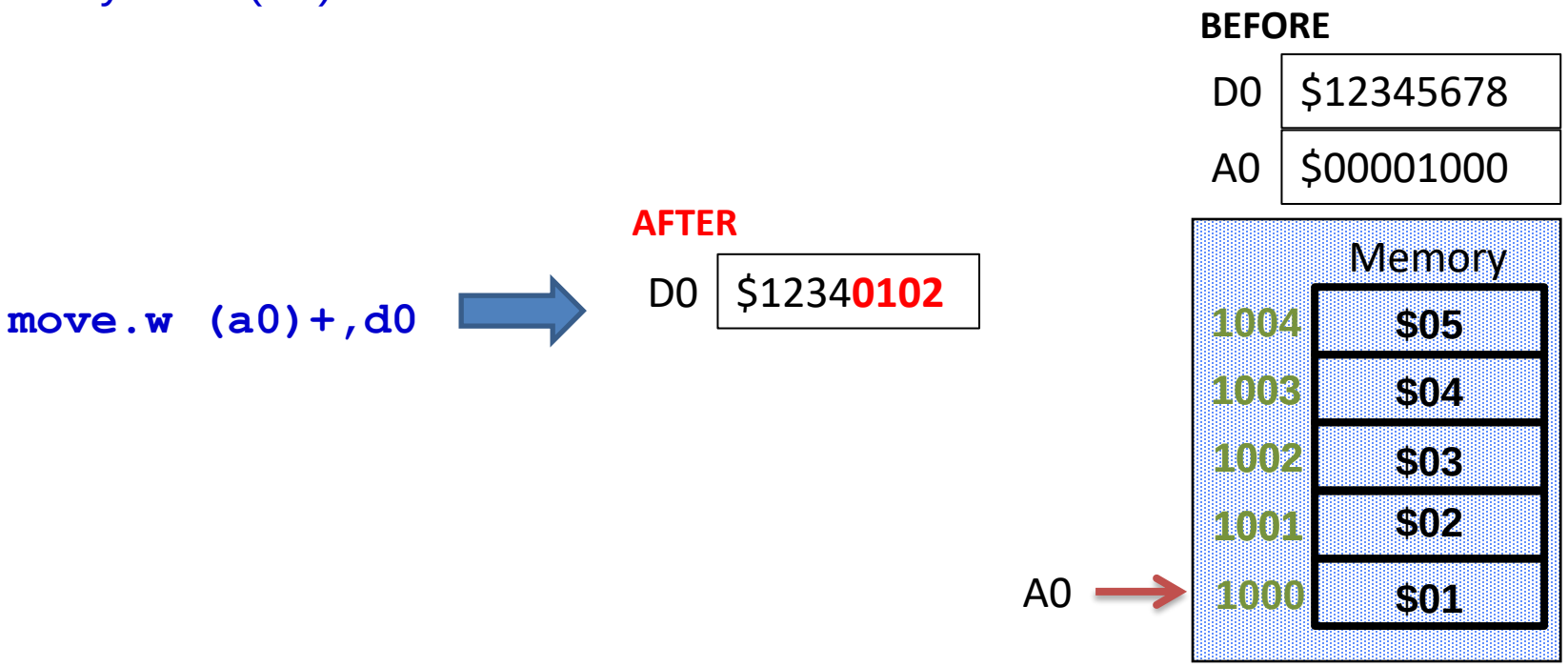
- Same as indirect, but after operand is used, the address in the register is incremented by the size of the operand (in bytes)
  - Symbol: (An)+

```
move.w (a0)+,d0
```



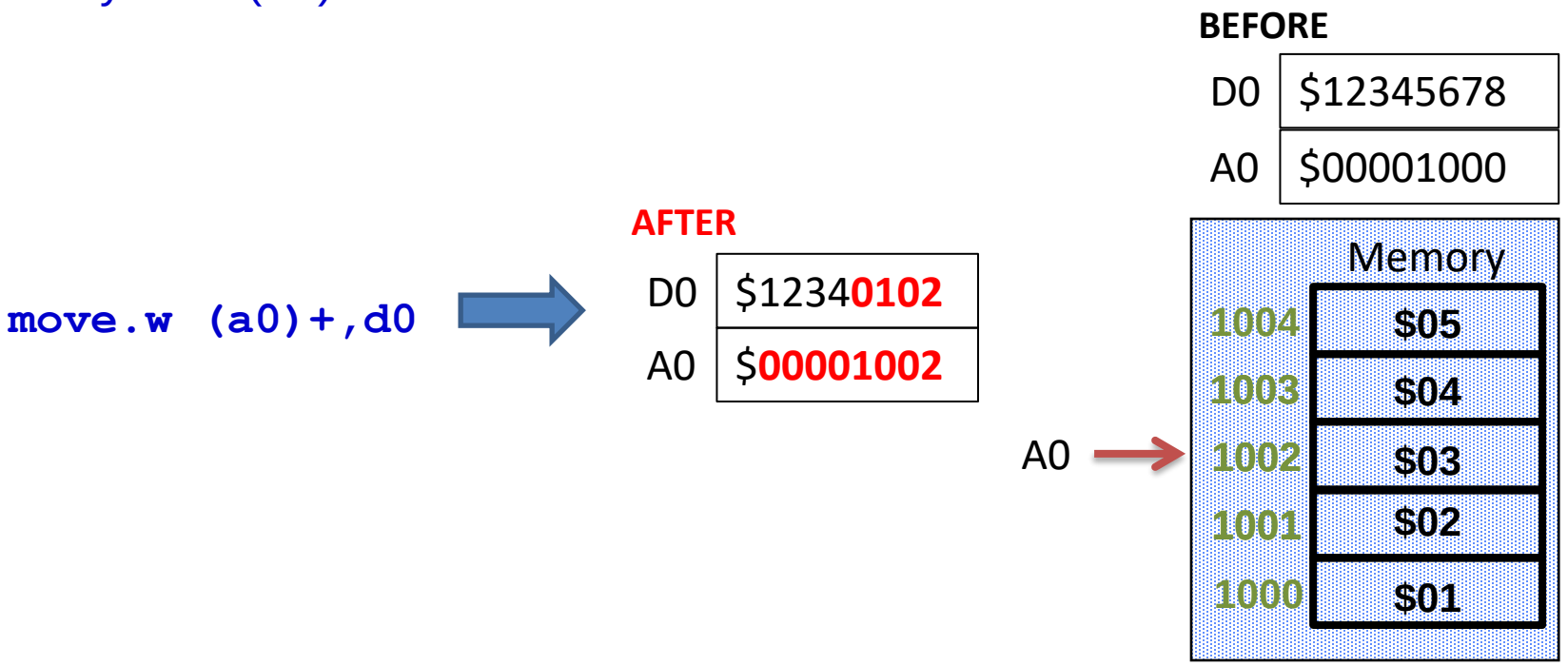
Post Increment Indirect Addressing

- Same as indirect, but after operand is used, the address in the register is incremented by the size of the operand (in bytes)
  - Symbol: (An)+



Post Increment Indirect Addressing

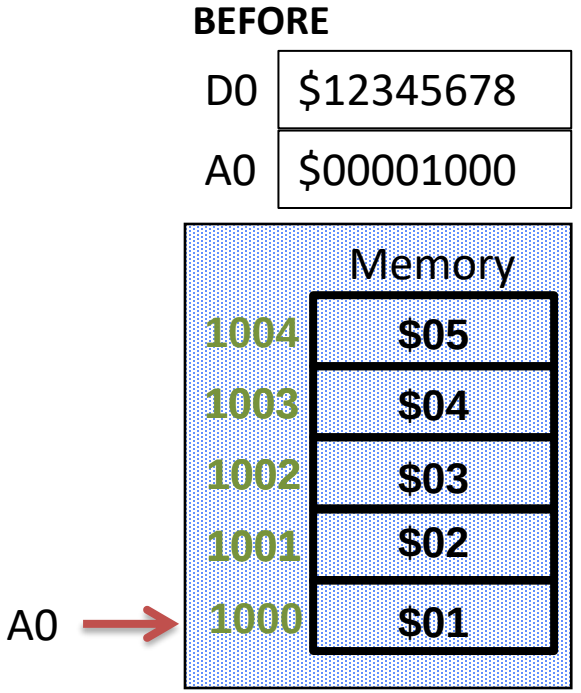
- Same as indirect, but after operand is used, the address in the register is incremented by the size of the operand (in bytes)
  - Symbol: (An)+



Post Increment Indirect Addressing

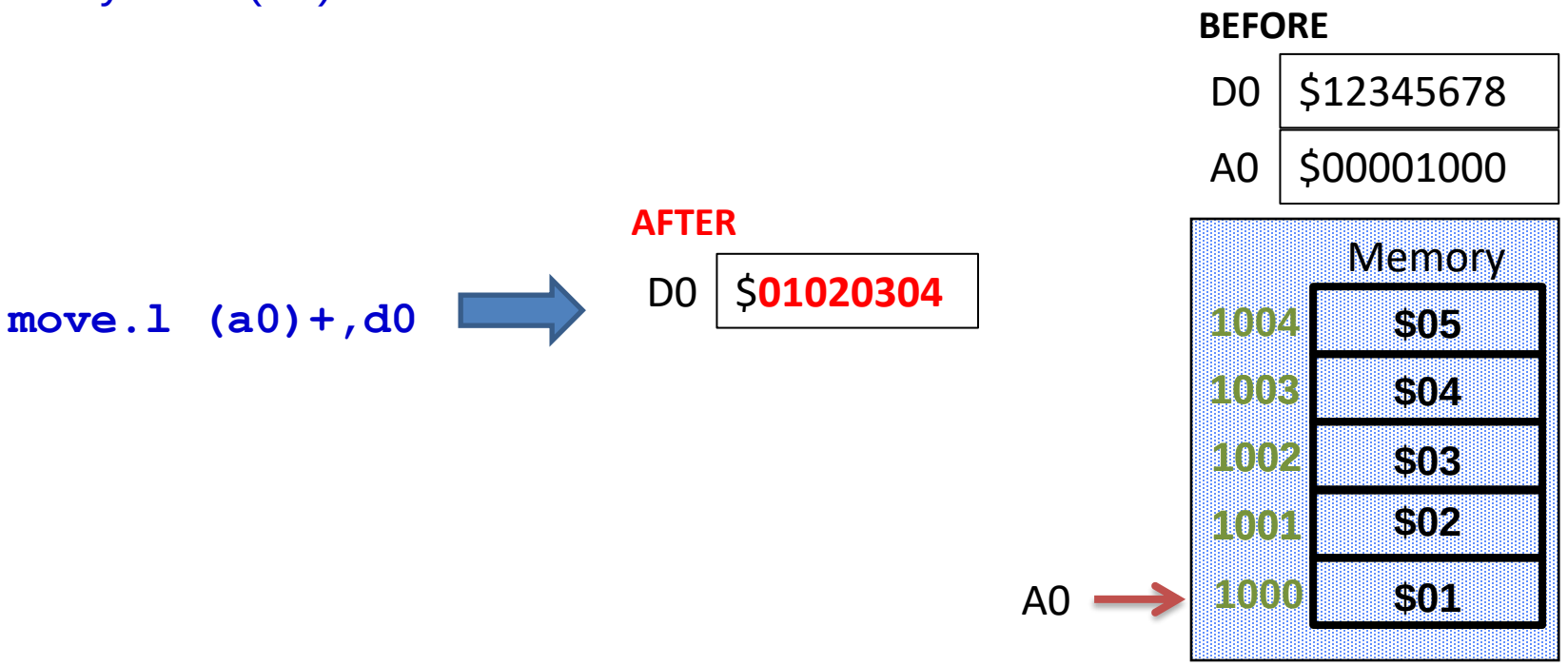
- Same as indirect, but after operand is used, the address in the register is incremented by the size of the operand (in bytes)
  - Symbol: (An)+

```
move.l (a0)+,d0
```



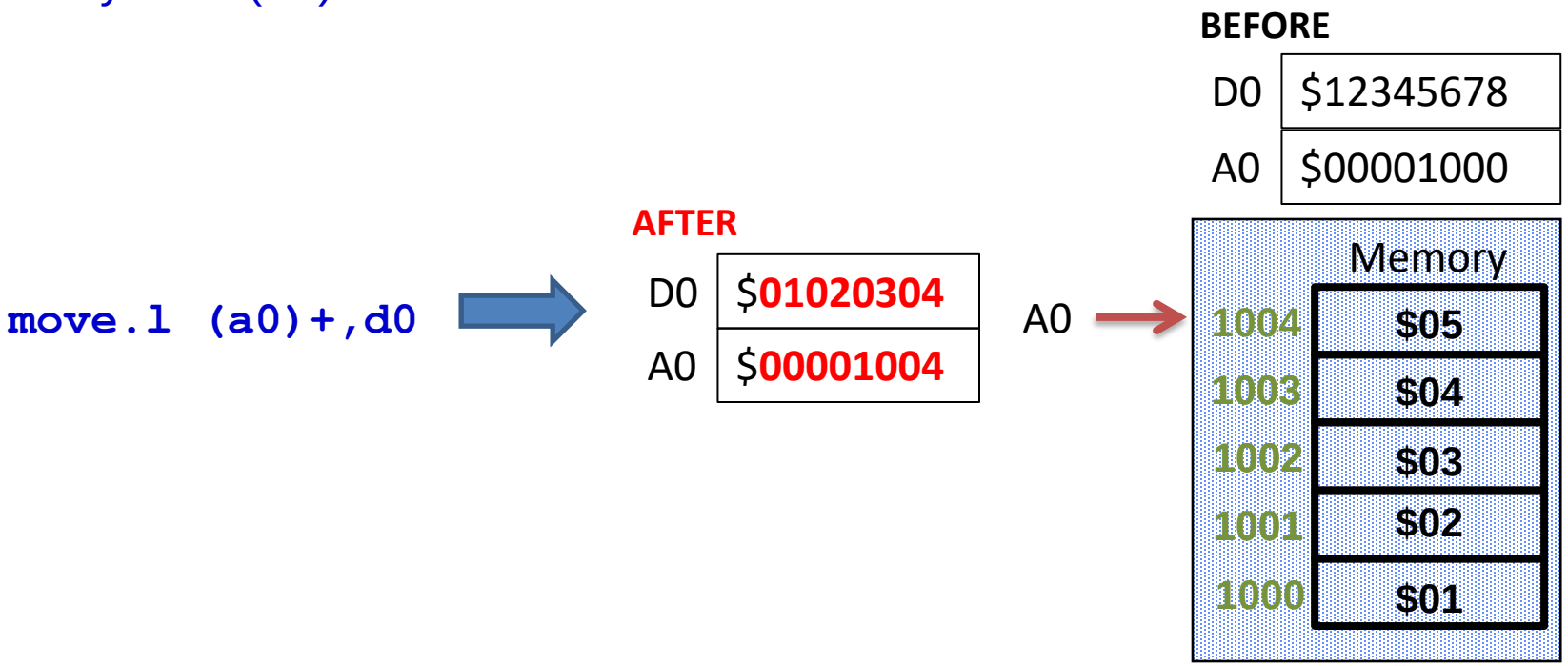
Post Increment Indirect Addressing

- Same as indirect, but after operand is used, the address in the register is incremented by the size of the operand (in bytes)
  - Symbol: (An)+



Post Increment Indirect Addressing

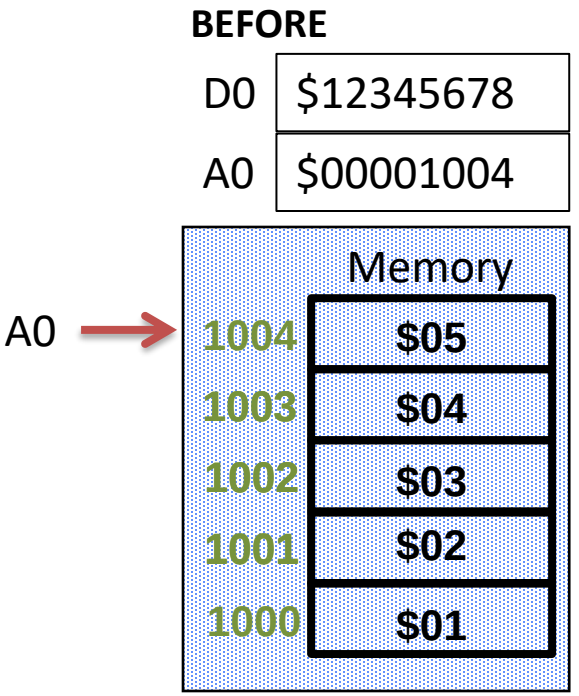
- Same as indirect, but after operand is used, the address in the register is incremented by the size of the operand (in bytes)
  - Symbol: (An)+



Pre Decrement Indirect Addressing

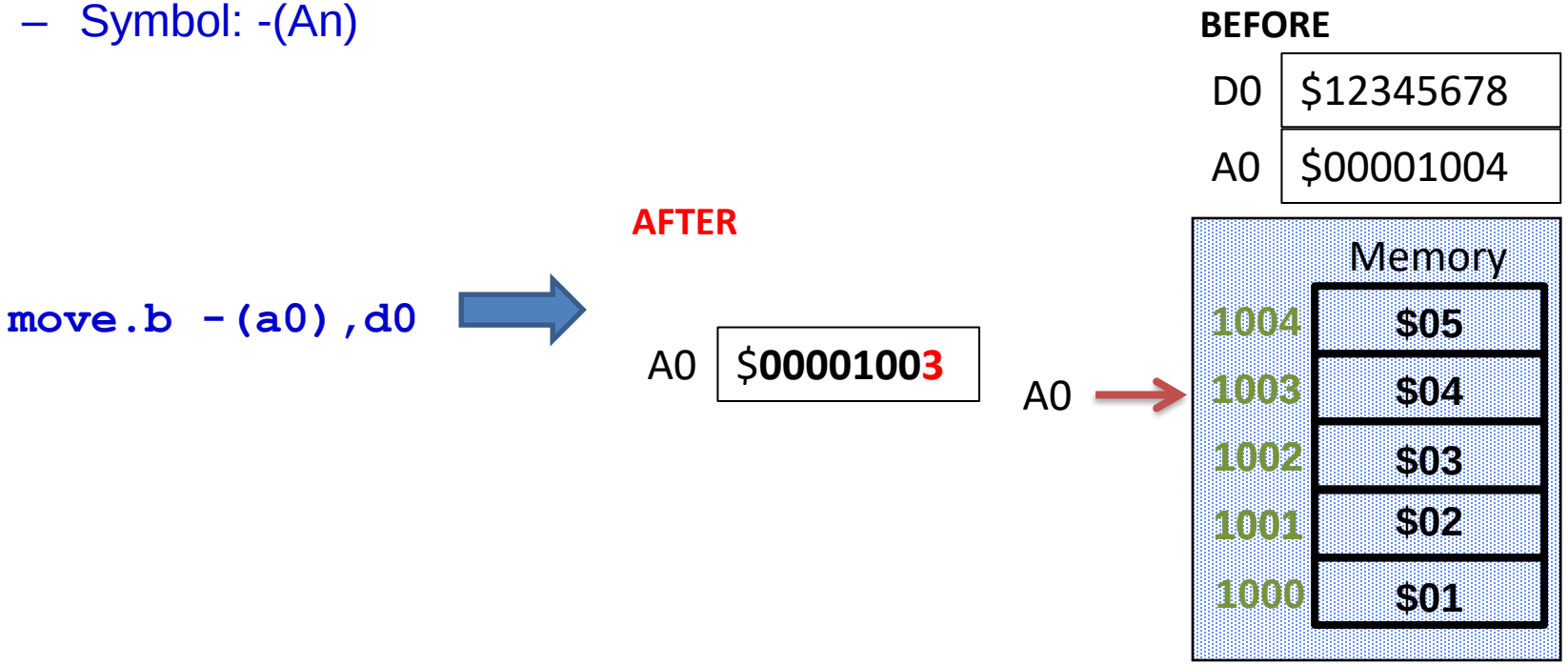
- Same as indirect, but the address in the register is first decremented by the size of the operand (in bytes) before it is used
  - Symbol: -(An)

```
move.b -(a0),d0
```



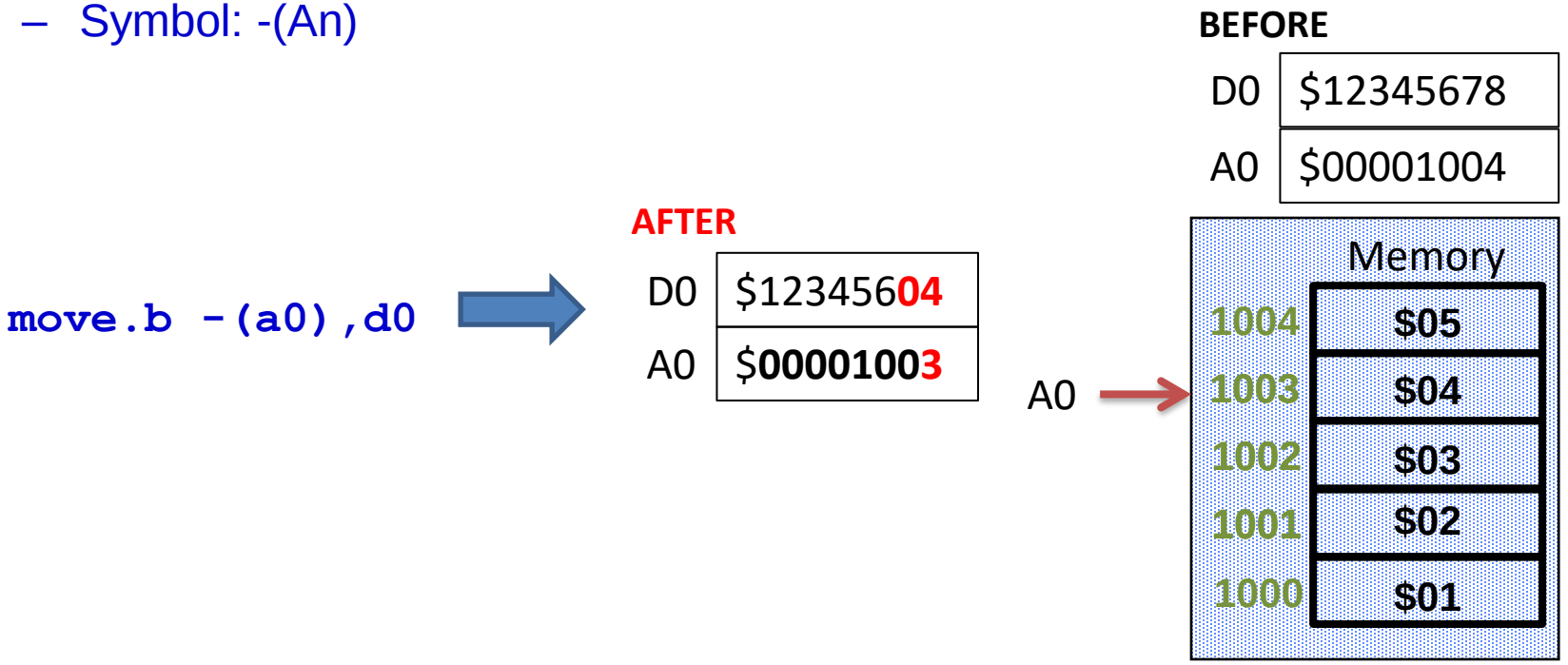
Pre Decrement Indirect Addressing

- Same as indirect, but the address in the register is first decremented by the size of the operand (in bytes) before it is used
  - Symbol: -(An)



Pre Decrement Indirect Addressing

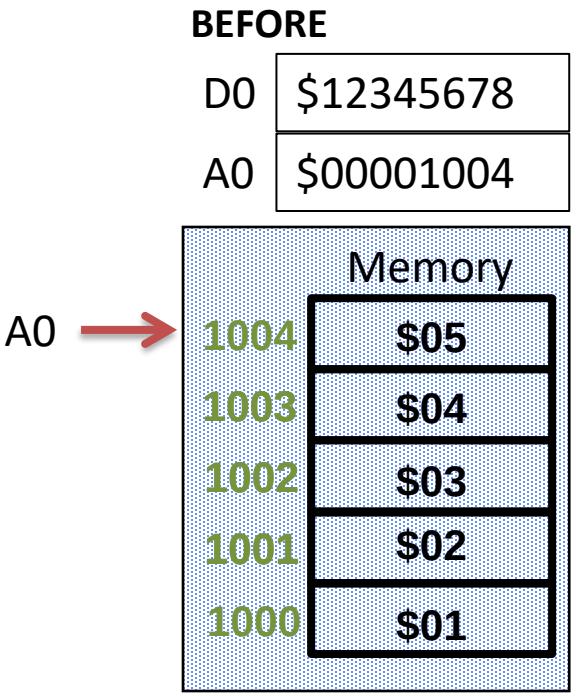
- Same as indirect, but the address in the register is first decremented by the size of the operand (in bytes) before it is used
  - Symbol: -(An)



Pre Decrement Indirect Addressing

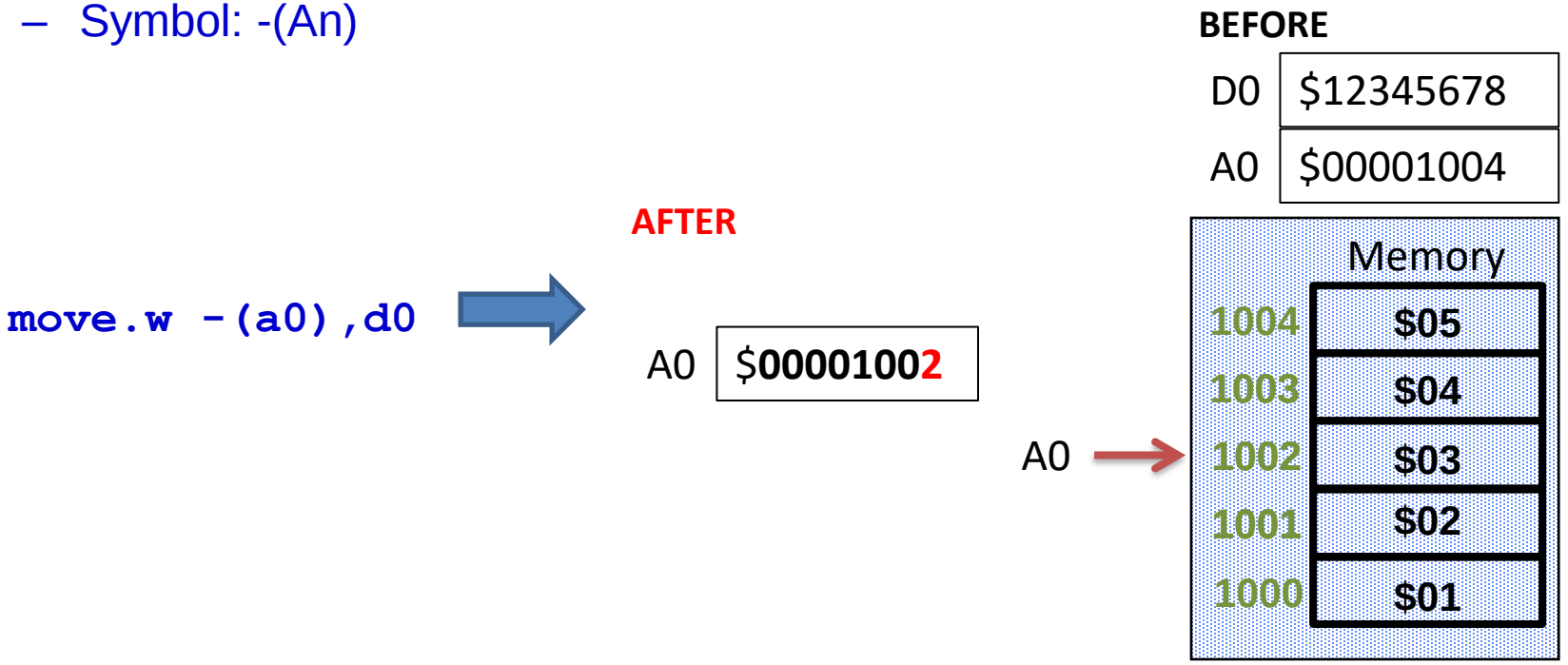
- Same as indirect, but the address in the register is first decremented by the size of the operand (in bytes) before it is used
  - Symbol: -(An)

```
move.w -(a0),d0
```



Pre Decrement Indirect Addressing

- Same as indirect, but the address in the register is first decremented by the size of the operand (in bytes) before it is used
  - Symbol: -(An)



Pre Decrement Indirect Addressing

- Same as indirect, but the address in the register is first decremented by the size of the operand (in bytes) before it is used
  - Symbol: -(An)

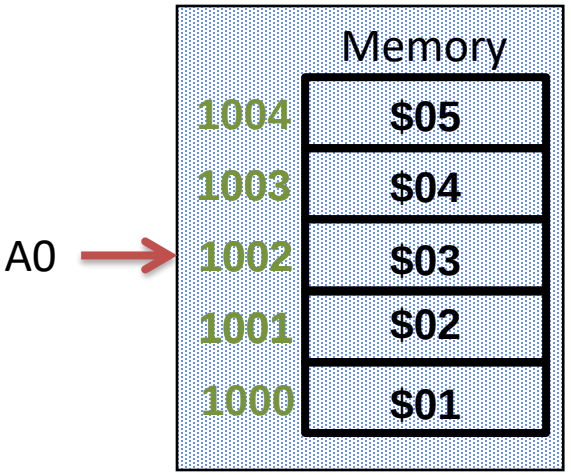
```
move.w -(a0),d0
```

AFTER

D0	$1234**0304**
A0	$**0000100****2**

BEFORE

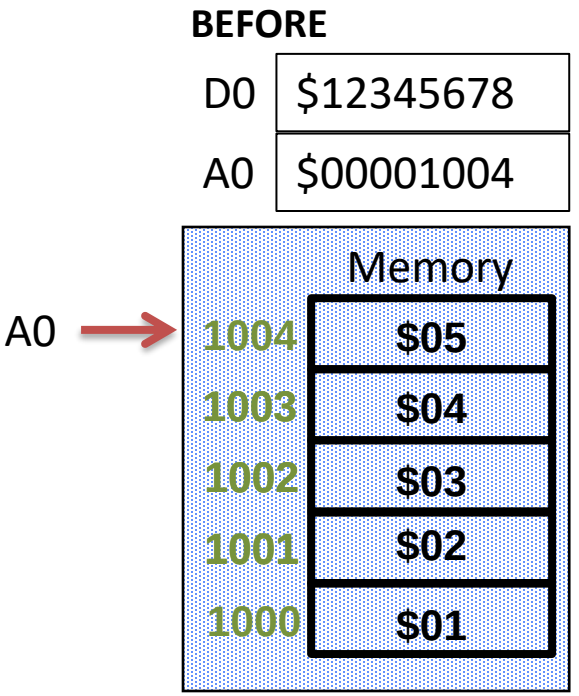
D0	$12345678
A0	$00001004



Pre Decrement Indirect Addressing

- Same as indirect, but the address in the register is first decremented by the size of the operand (in bytes) before it is used
  - Symbol: -(An)

```
move.l -(a0),d0
```



Pre Decrement Indirect Addressing

- Same as indirect, but the address in the register is first decremented by the size of the operand (in bytes) before it is used
  - Symbol: -(An)

```
move.l -(a0),d0
```

AFTER

A0 $00001000

BEFORE

D0	$12345678
A0	$00001004

Memory

1004	$05
1003	$04
1002	$03
1001	$02
1000	$01

A0 →

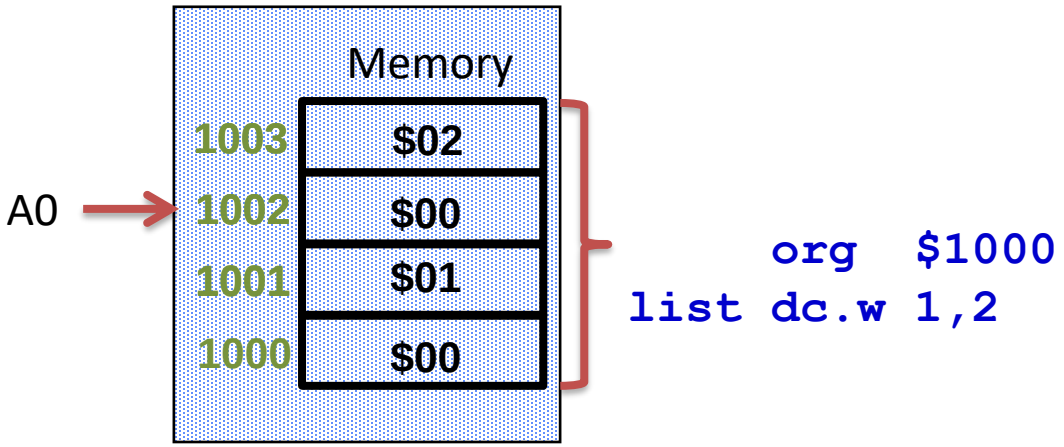
Pre Decrement Indirect Addressing

- Same as indirect, but the address in the register is first decremented by the size of the operand (in bytes) before it is used
  - Symbol: -(An)

Post Increment/Pre Decrement and C

- Consider the example

```
ptr = &list;
sum += *ptr++;
```



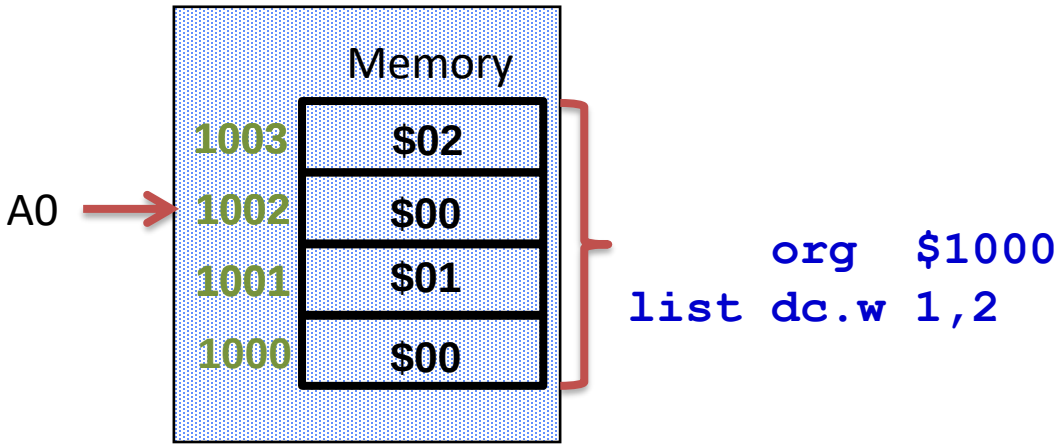
- Assume that the list variables are <u>words</u> (i.e., shorts) and **`sum`** is contained in D0 and the pointer **`ptr`** is to be placed in A0

```
lea     list,a0    ;a0=$00001000
add.w   (a0)+,d0   ;d0=d0+word at 1000 and a0=a0+2
```

Post Increment/Pre Decrement and C

- Consider the example

```
ptr = &list;
sum += *ptr++;
```

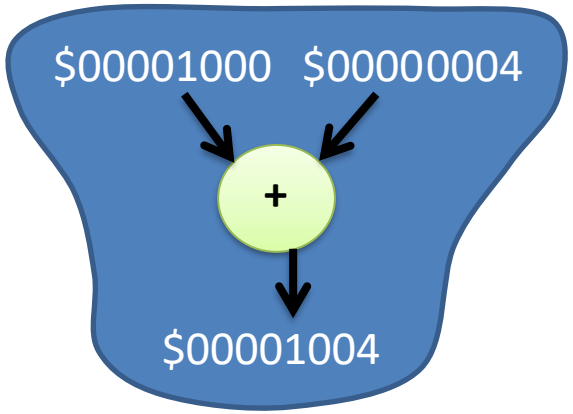


- Assume that the list variables are words (i.e., shorts) and **sum** is contained in D0 and the pointer **ptr** is to be placed in A0

```
lea     #list,a0   ;a0 = $00001000
add.w   (a0)+,d0   ;d0 = d0 + word at 1000 and a0 = a0 + 2
```

Indirect Addressing With Displacement

- Address of operand is contained inside an address register plus a sign-extended 16-bit displacement
  - Symbol: d16(An)



BEFORE

D0	$12345678
A0	$00001000

Memory

1004	$05
1003	$04
1002	$03
1001	$02
1000	$01

```
move.b 4(a0),d0
```

AFTER

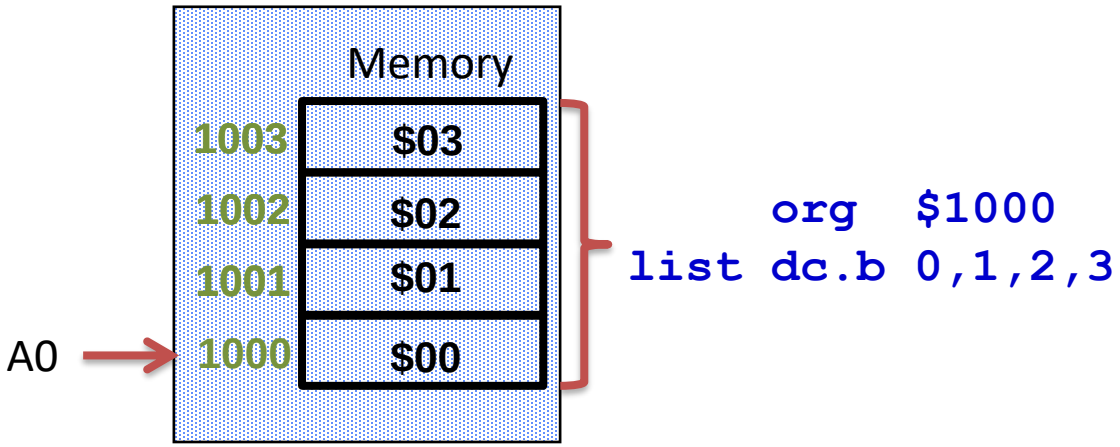
D0	$12345605
A0	$00001000

A0 →

Register Indirect with Displacement and C

- Consider the example

```
a = a + list[2];
```



- Assume that the list variables are bytes (i.e., chars) and **a** is contained in D0 and the array (pointer) **list** is placed in A0

```
add.b   2(a0),d0   ;d0 = d0 + byte at 1002
```

Indirect Addressing With Index and Displacement

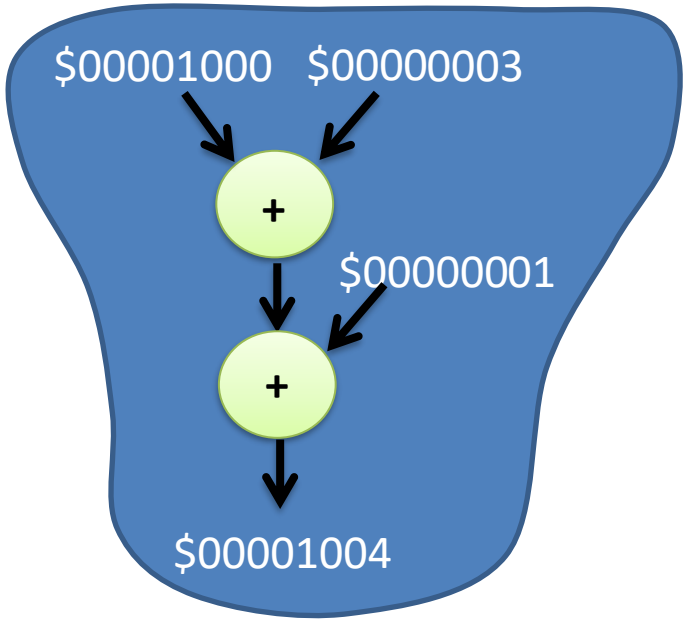
- The address of operand is contained inside an address register plus an index register plus a sign-extended 8-bit displacement
  - Symbol
    - d8(An,Xn)

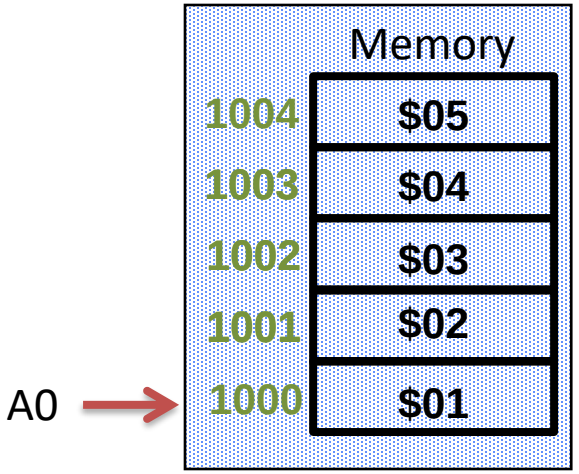
- Index register (Xn) can be either an address register or a data register
  - Size
    - Xn.L
    - Xn.W (which will be sign-extended to 32 bits)

Example

BEFORE

D0	$12345678
D1	$FFFF0003
A0	$00001000






```
move.b 1(a0,d1.w),d0
```

AFTER

D0	$12345605	D1	$FFFF0003
A0	$00001000		

Register Indirect with Index plus Displacement and C

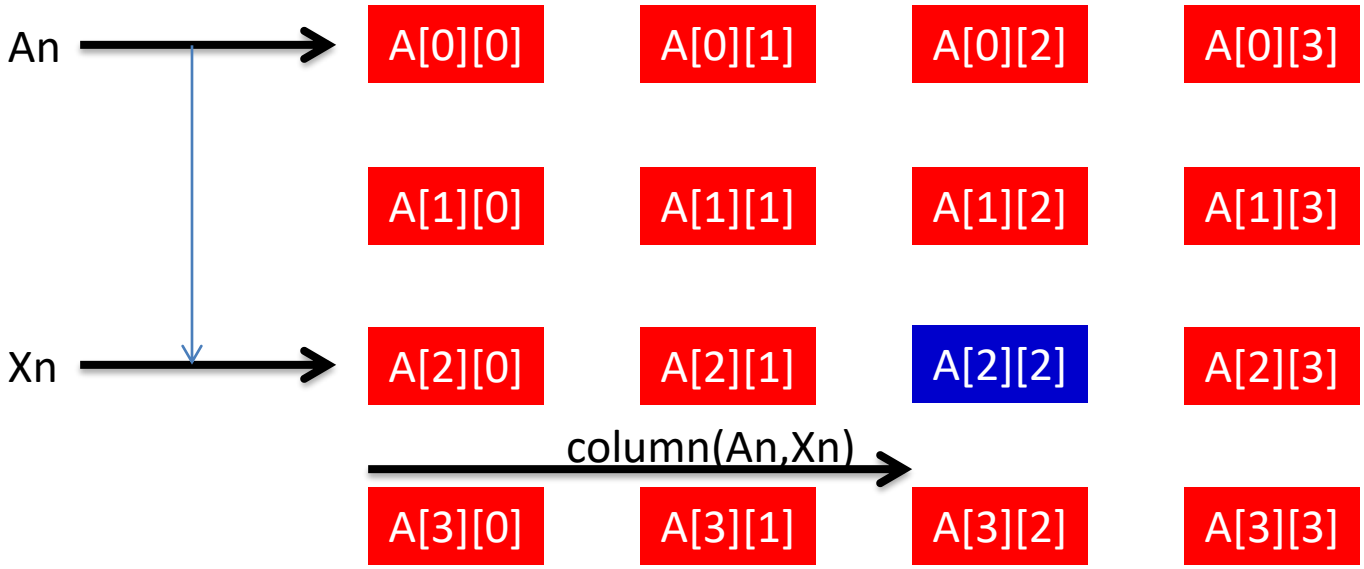
- Consider the example

```
a = a + list[i];
```

- Assume that the list variables are long words (i.e., ints) and **a** is contained in D0, **i** is contained in D1 and the array (pointer) **list** is in A0

```
move.w   #4,d3              ;d3 = 4
mulu     d1,d3              ;d3 = 4 * i
add.l    0(a0,d3.l),d0      ;d0 = d0 + list[i]
```

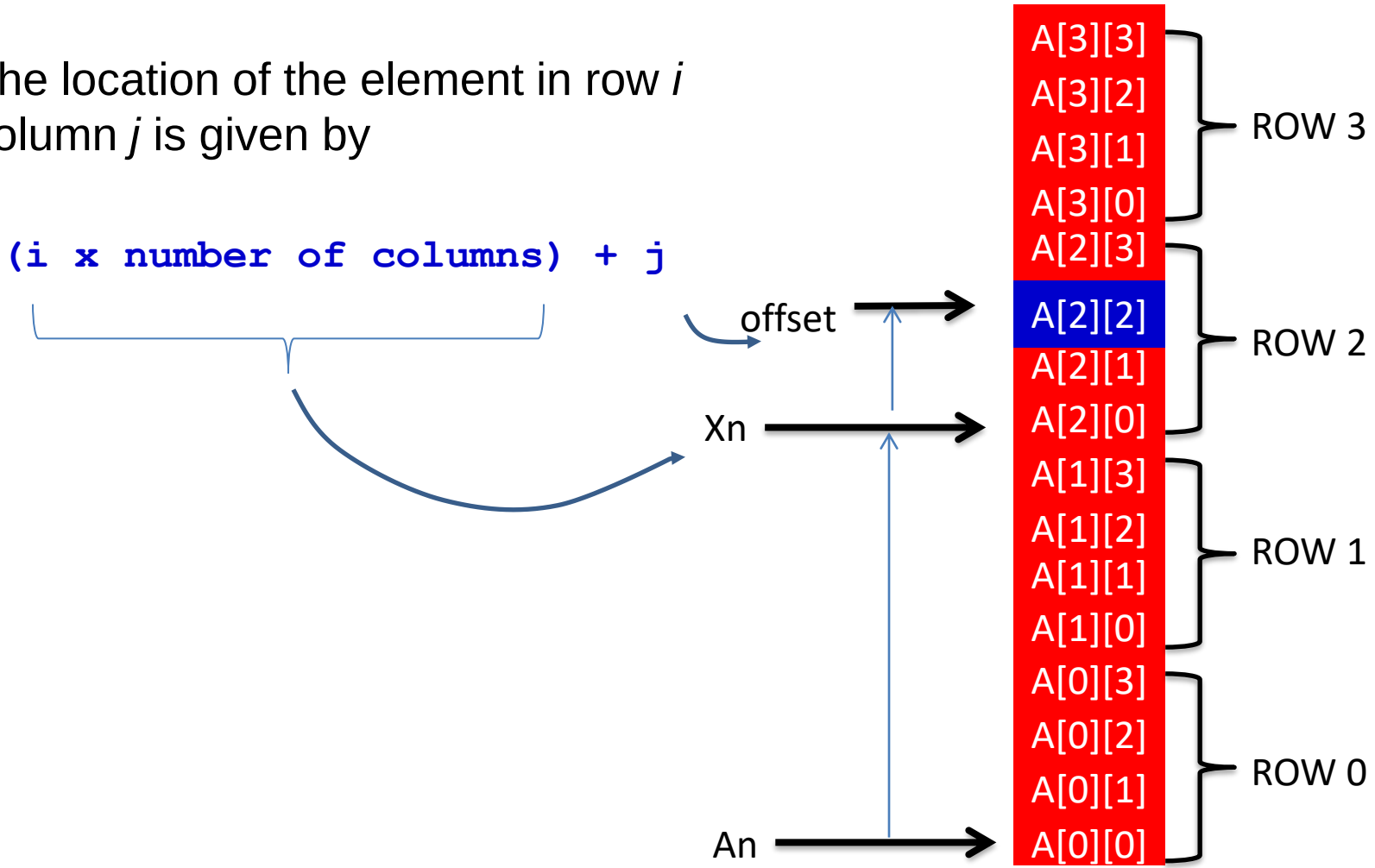
Accessing a 2-dimensional Array



C Stores Array in Row-Major Order

- The location of the element in row *i* column *j* is given by

```
(i x number of columns) + j
```



Assembler Code

- Consider the example

```
char list[4][4];
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

2-d array (list)

- Uninitialized array

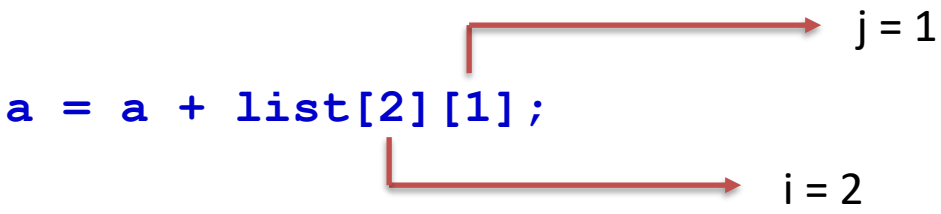
```
ncols    equ      4
nrows    equ      4
list     ds.b     ncols*nrows
```

- Initialized array

```
list     dc.b     1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
```

Assembler Code

- Consider the example

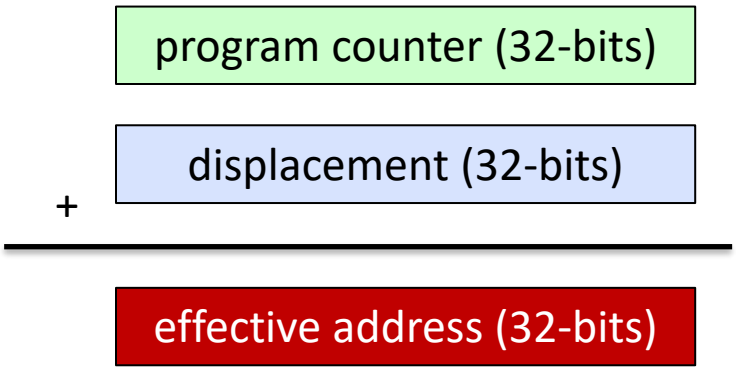


- Assume that the list variables are bytes (i.e., chars) and **a** is contained in D0 and the array (pointer) **list** is in A0

```
move.w   #ncols,d2          ;ncols = 4
mulu     #2,d2              ;i x ncols
add.b    1(a0,d2.l),d0      ;a0 + (i x ncols)+ j
```

Program Counter With Displacement

- Address of operand is computed by adding a 16-bit signed displacement (in extension word) to the PC
  - Symbol: d16(PC)
    - Displacement is usually expressed using a label
    - The value in the PC when the addition takes place is the address of the extension word



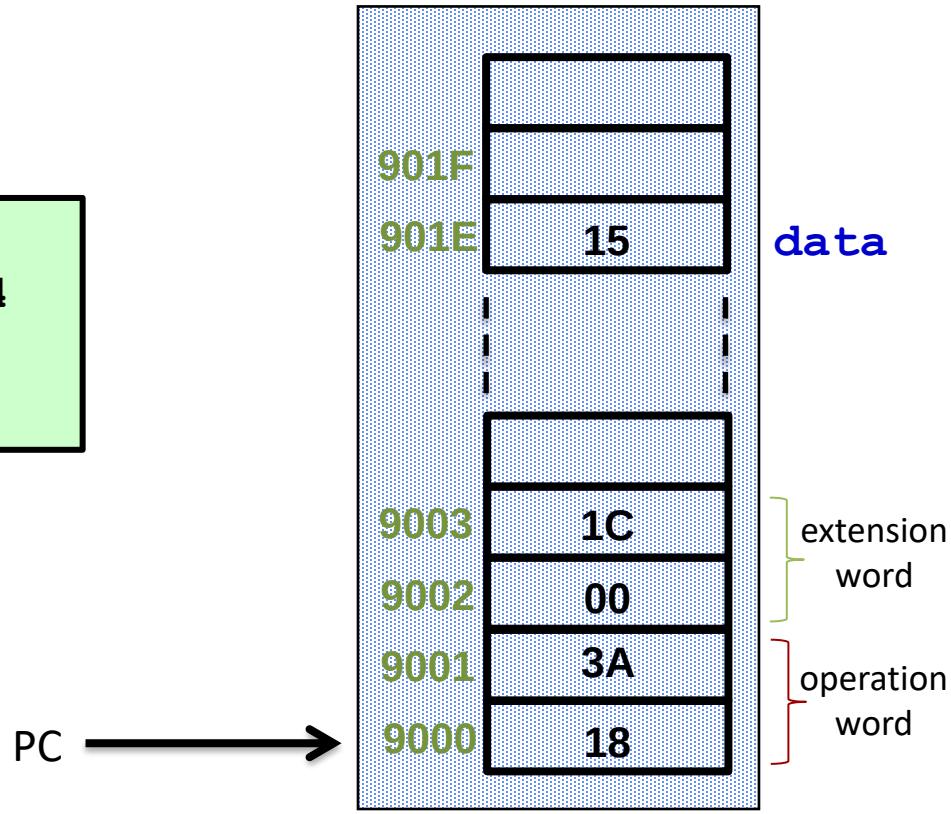
- Allows the program to load and execute at any address, no matter what the ORG value used during assembly

PC with Displacement – Processor Perspective

- Address of operand is computed by adding a 16-bit signed displacement (in extension word) to the PC
  - Symbol: d16(PC)

```
        org     $9000
        move.b  data(pc),d4
        org     $091E
data    dc.b    $15
```

```
PC          009002
DISP    +   00001C
-------------------
EA          00901E
```

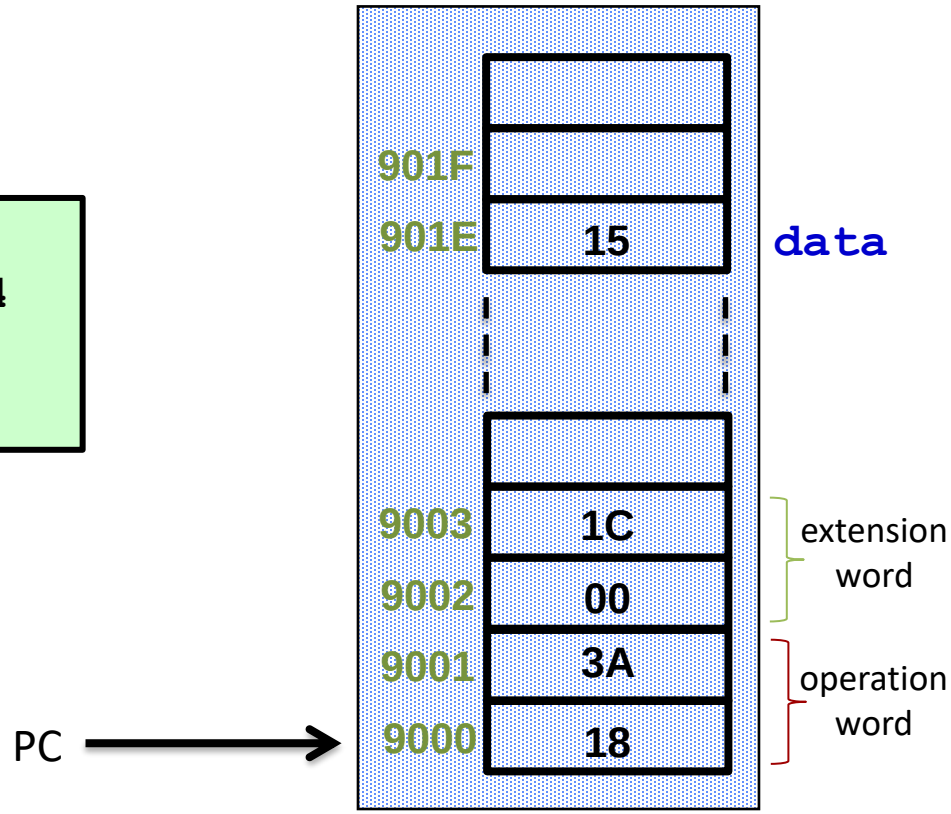


PC with Displacement – Assembler Perspective

- Address of operand is computed by adding a 16-bit signed displacement (in extension word) to the PC
  - Symbol: d16(PC)

```
        org     $9000
        move.b  data(pc),d4
        org     $091E
data    dc.b    $15
```

```
data      00901E
  PC    - 009002
--------------
DISP      00001C
```



Summary

- Address modes
  - used to identify the location of the operand that will be operated on by the instruction
- Register Direct
  - Fastest address mode, shortest instructions, used for accessing frequently accessed program variables, implements REGISTER keyword in C
- Absolute Addressing
  - Slowest address mode, slightly faster (short) mode exists, used to access simple global variables in C
- Immediate Addressing
  - Used for true constants, only makes sense for source operand
- Indirect Addressing
  - Implements pointers in C
- Indirect with post-increment / pre-decrement
  - Used to traverse memory in forward/reverse directions
  - Implements PUSH and POP stack operations
  - Implements auto-increment and auto-decrement pointer operations in C

Summary

- Indirect with Displacement
  - Used to access a *static* location in an array or field in a structure
- Indirect with Index and Displacement
  - Used to access a *random* location in an single or multi-dimensional array, structures, etc.
- PC Indirect with Displacement
  - Used to write relocatable code
- Only certain address modes can be used with certain instructions
  - Check your textbook if you are not sure!
- Address registers have their own instructions
  - MOVEA, ADDA, SUBA, CMPA
    - None of which affect the flags in the condition-code register
  - Don’t use A7 (for now)
  - Be careful performing .W operations on address registers due to the effects of sign-extension