Addressing modes

- Addressing mode = how to specify the effective (~~or actual~~ address (or actual location) of the operand.

- M68000 provides those basic addressing modes found in all computers:

(1) immediate #n
(2) direct n (also known as "absolute")
(3) indirect

(An)
n(An)

M68000: addr. register indirect with displacement.

indirect with displacement

(4) indexed

(An, Dm)

n(An, Dm)

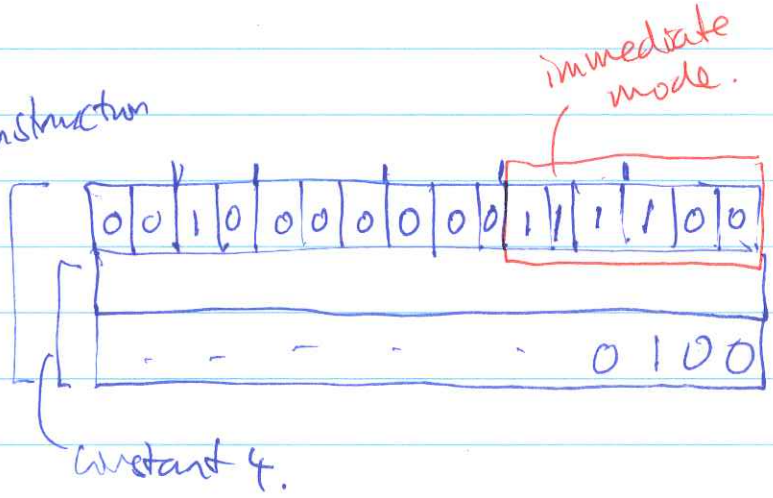
Immediate addressing mode : #<constant>

- used to specify constant operands.

eg: move.b #4, D0
move.w #4, D0
move.L #4, D0.

encoding:

move.L #4, D0 :



The constant is stuck inside the instruction and it is "immediately" available.

- "immediate" means: the operand is immediately available. in the instruction.

Examples:

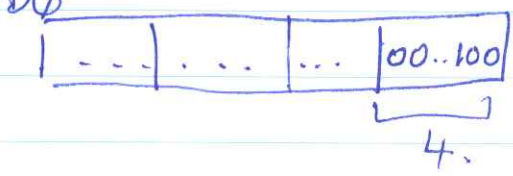
Simple

```
move.l #0, D0
move.l #-4, D1
move.b #'H', D2
```

result:

Symbolic constant:

```
MAX   EQU  4
move.B  #MAX, D0
```



Advance usage:

```
move.l #A, A0
move.l #B, A1
```

```
A: ds.l 10
B: ds.l 10
```

- A is equated to started address of the array!!!

Q: what's the effect?

- move the base addr. of array A in A0
- and move the base addr. of array B in A1 !

DEMO: imm.s

(Absolute)

Direct addr. mode : An, Dn, MEM_ADDR

- Used to specify operand in a register or in memory location.
- "Direct" means: the location is immediately available in the instruction.

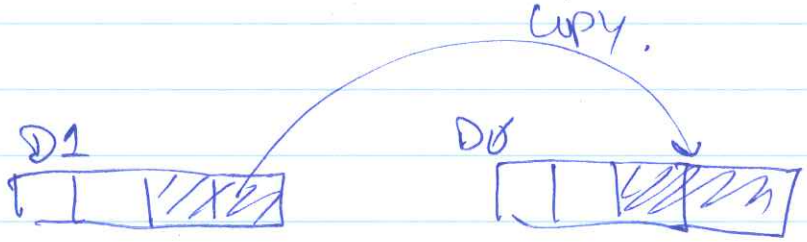
  (But NOT the operand! you have to go to the location to find the operand).

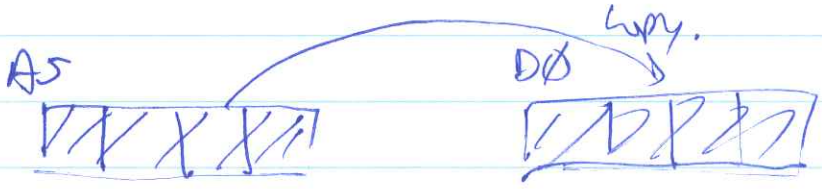
notation:

Descriptions.

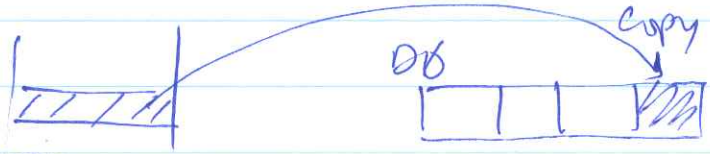
```
move.W   D1, D0
move.L   A5, D0.
move.b   4, D0
```

Descriptions:



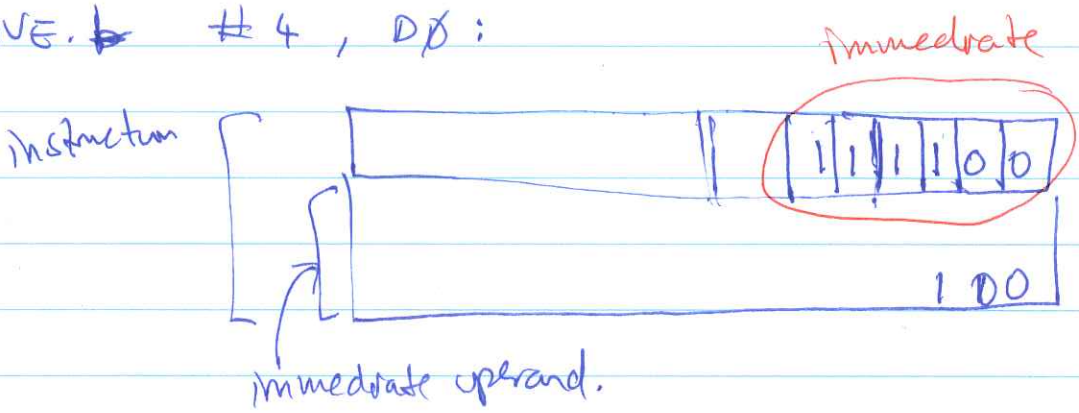


Memory:

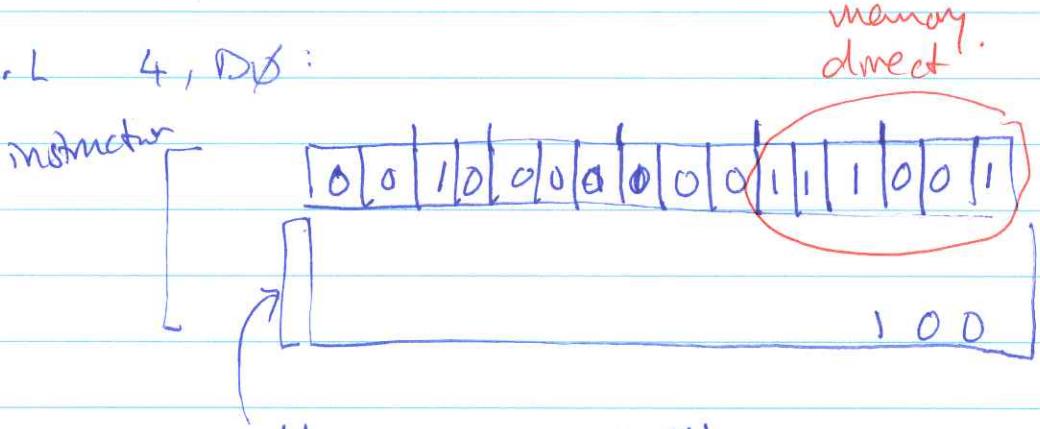


Instruction encoding: difference between move.l #4, D0 and move.l 4, D0.

MOVE.L #4, D0:



move.L 4, D0:



- Typical use of direct modes:

(1) memory direct: access simple variables (int, float, char, short etc)

(2) register direct: get/save data or temp result in registers.

absolute.s

Example use of direct addr. mode

C: `int i;`

Assembler:
```
i: ds.l 1
```

C: `i = 4;`

```
move.l #4, i
```

C: `int x, y, z`

Assembler:
```
x: ds.l 1
y: ds.l 1
z: ds.l 1
```

C: `z = x+y`

```
move.l x, d0
add.l  y, d0
move.l d0, z
```

or:
```
move.l x, d0
move.l y, d1
add.l  d0, d1
move.l d1, z
```

C: `int i;`

```
i: ds.l 1
```

C: `i++;` (`i++ = i = i+1`)

```
addq.l #1, i
```

or
```
addi.l #1, i
```

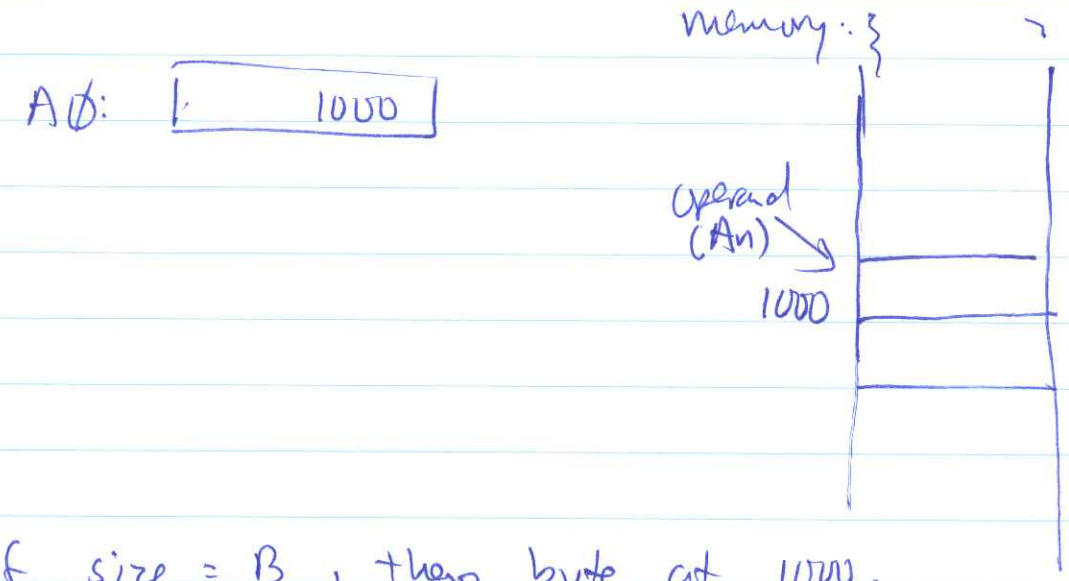
Skip this. Included in indexed!!!

Indirect addressing mode: (An)

- M68000 only have a register indirect mode.
- The operand (of the instruction) is located at memory location address given by An

eg: if A0 contains 1000
then the notation (A0)
means: the operand is at memory location 1000.

Picture:



if size = B, then byte at 1000.
W word
L long word.

Demo: indirect.S

What is the indirect mode good for ?

Indirect mode is primarily used to access simple variables (integer, float) through a pointer.

Explain what is a pointer variable!!!

eg:

```
int  i;
int  *ptr;

ptr = &i;
*ptr = 4;
```

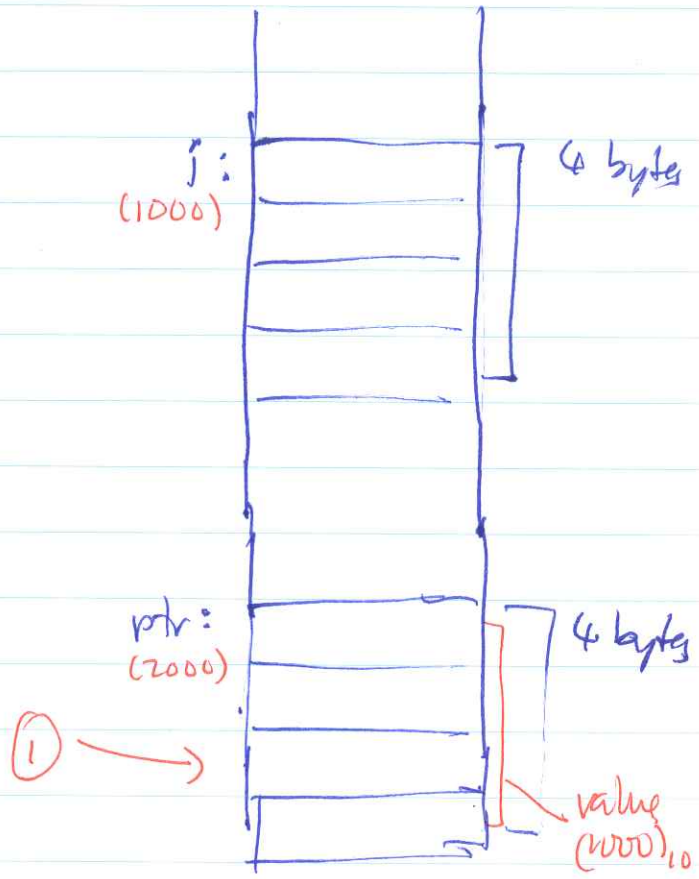
is equivalent to:

```
i:   ds.l  1
ptr: ds.l  1

move.l  #i, ptr.        (put addr. of i in ptr variable)

move.l  ptr, A0
move.l  #4, (A0)
```

Pictorially:

```
i : ds.l 1
ptr : ds.l 1
```

result



(1) move.l #i, ptr

↳ after replacing the label by the address values

move.l #1000, 2000 - put value 1000 in loc. 2000 in memory

(2) move.l ptr, A0

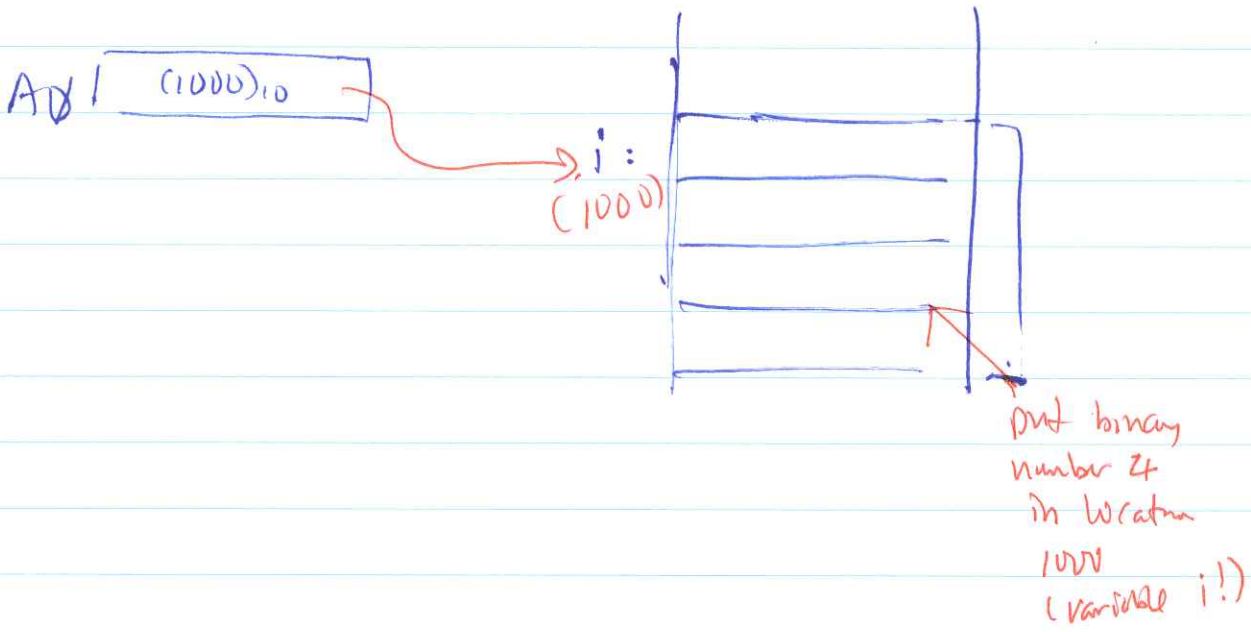
↳ after replacing label ptr by 2000

mc68k reads: move.l 2000, A0.

result: A0 | | | | | | | | | | | | | | | | |

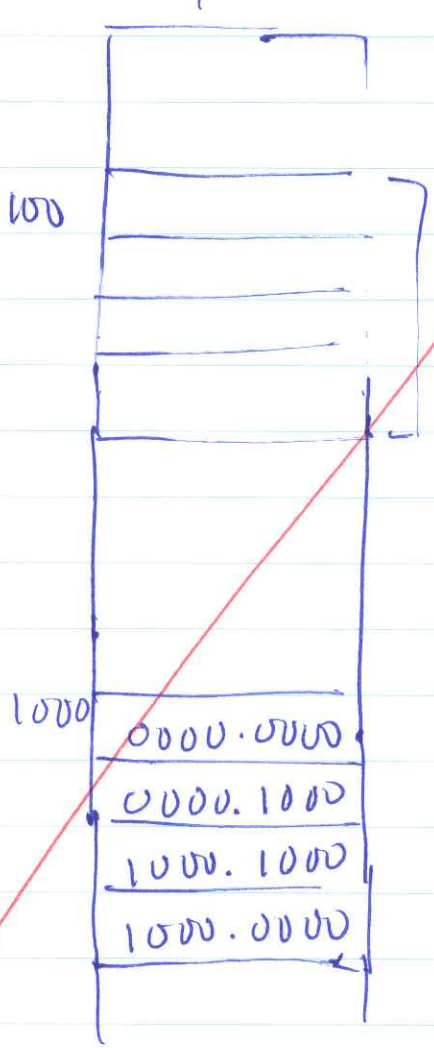
$(1000)_{10}$.

(3) move.l #4, (A0)



- Note: some computers (eg. VAX 11) have memory indirect addressing mode.

eg:



SKIP

4 bytes containing binary number $(1000)_{10}$

Then the instruction:

"move.L (100), D0"

will ~~not~~ update D0 to: | 0000.0000 | 0000.1000 | 1000.1000 | 1000.0000 |

memory indirect mode is NOT available in M68000.

Another use: access array elements.

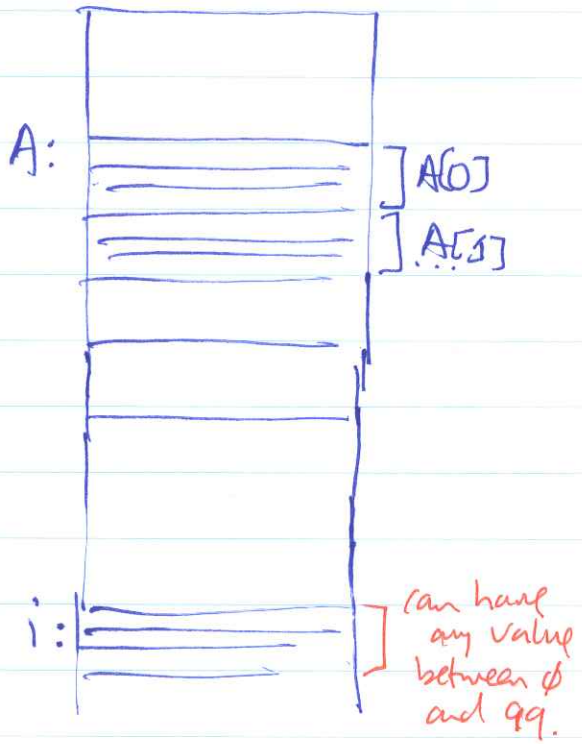
In C:

```
int A[100];
int i;

A[i] = 4;
```

In assembler:

```
A:  ds.l  100     (array A)
i:  ds.l  1.      (int i).
```

Situation after definition:



A[i] = 4; means: put the value 4 into memory location reserved for variable A[i].

Note: A[i] depends on current value of i.

The "ultimate" instruction is:

```
move.l  #4, <ea>
```

↑ this is the address of variable A[i].

The address of A[i] = computed!

= base address of array A + 4 * i

(1) get the base address of array A

```
move.l  #A, A0
```

(2) compute offset:

Compute offset 4 * i

```
move.l   i, D0

add.l    D0, D0        (2*i)
add.l    D0, D0        (4*i).
```

(use multiply later).

(3) compute address of variable A[i]:

```
adda.l   D0, A0
```

← now A0 "points" to A[i].

(4) put 4 in that memory location:

```
move.l   #4, (A0).
```

Summary:

```
move.l   #A, A0

move.l   i, D0
add.l    D0, D0
add.l    D0, D0
adda.l   D0, A0
move.l   #4, (A0).
```

Better way: indexed addr. mode !!!

Exercise in book :

Initial situation :

Registers :

D1 : A874 BF56

A1 : 00003004

Byte..

3000	52
3001	FA
LAB1 : 3002	4B
3003	AC

↑ Hex number address

Memory :

	Address	Content
	3000	52FA
LAB1 :	3002	4BAC
	3004	1234
	3006	FFEE
	3008	F467

↑ in hex !

Result :

(a) MOVE.W D1, 3000H

	3000	BF56
LAB1 :	3002	4BAC
	3004	1234
	3006	FFEE
	3008	F467

Result :

(b) MOVE.B LAB1, D1

D1 : A8 74 BF 4B

(c) MOVE.L (A1),D1 Result:

D1 : | 12 | 34 | FF | EE |

(d) MOVE.W #12,D1 Result

D1 : | A8 | 74 | 00 | 0C |

Hex. decimal # for 12.

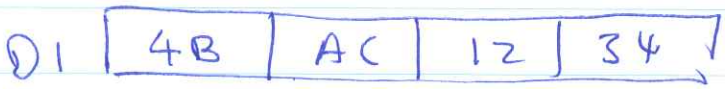
(e) MOVE.W #$12,D1 Result. ($12 = $(18)_{10}$)

D1 : | A8 | 74 | 00 | 12 |

(f) MOVEA.L #LAB,A1 Result

A1 : | 00 | 00 | 30 | 02 |

(g) MOVE.L LAB1, D1 Result:



(h) MOVE.W (A1), D1 Result

Address register indirect with displacement : d16(An)

- This mode is an extension of the address register indirect mode.

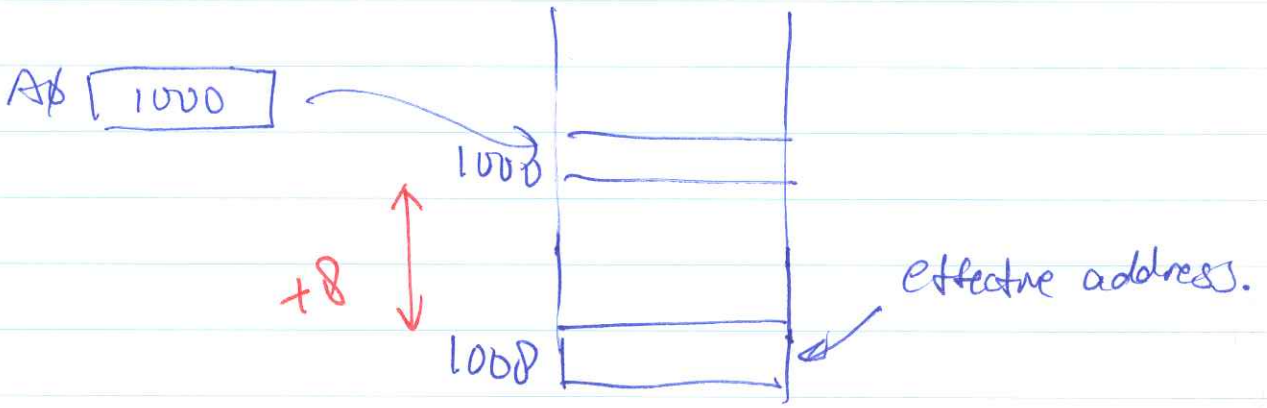
- The addressing mode d16(An) adds the displacement d16 to the content of the address reg An to get the effective memory address of the operand :

memory effective addr = d16 + An

↑

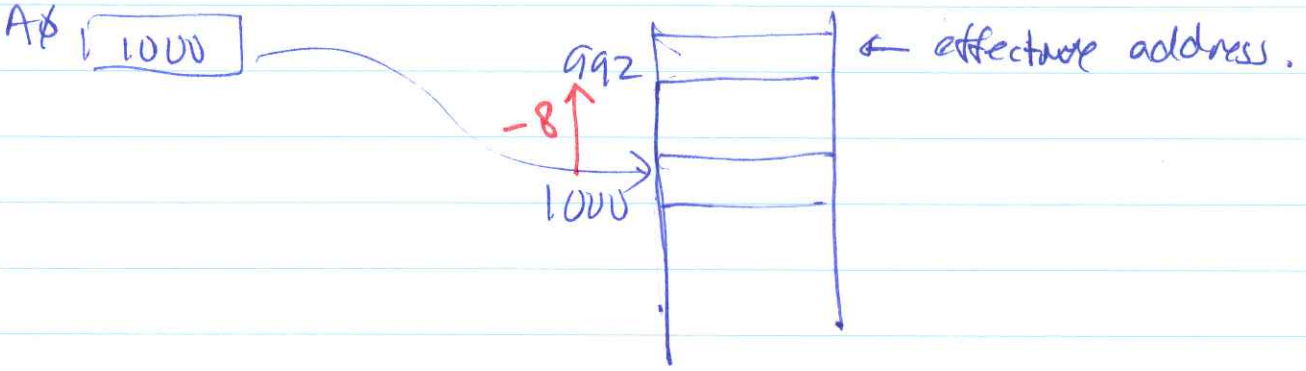
16 bit 2's Comp. Constant.

eg: 8(A0) = address 8 further than where A0 is pointing.



Neg. displacement is allowed.

eg: $-8(A\emptyset)$ = address -8 further than where $A\emptyset$ is pointing.



Usage:

Demo: addr mode-indirect.s

- ~~Address indirect w~~
  Address register indirect ~~mode~~ with displacement are used in:
  (0) Access Array element indexed by Constant: A[4]
  (1) accessing elements in structures.
  (2) accessing parameters of functions that are passed on the stack (later).
  (4) accessing local variables of (recursive!) functions on the stack.

Example usage:

```
struct my_rec
    { int x, y, z; };

struct my_rec A;

    A.x = 10;
    A.y = 20;
    A.z = A.x + A.y;
```

```
A: ds.l 3

move.l  #A, Aϕ
move.l  #10, ϕ(Aϕ).

move.l  #A, Aϕ
move.l  #20, 4(Aϕ)

move.l  #A, Aϕ

move.l  ϕ(Aϕ), Dϕ
add.l   4(Aϕ), Dϕ

move.l  Dϕ, 8(Aϕ).
```

```
class MyClass
 { int x;
   int y;
   int z;
 }

MyClass A = new MyClass();

    A.x = 10;
    A.y = 20;
    A.z = A.x + A.y.
```

Example: linked list

```
struct linked_list
    { int value1, value2;
      struct Linked_list *next;
    };

struct linked_list *head;

struct linked_list A, B, C;

A.next  = &B;

B.next  = &C;

C.next  = NULL;

head    = &A;

head -> value2 = 1;

head -> next -> value2 = 7;

head -> next -> next -> value2 = 10;
```

int = 4 bytes

pointer = 4 bytes

head: ds.l 1

A: ds.l 3
B: ds.l 3
C: ds.l 3

A.next = &B:

```
move.l  #A, A0
move.l  #B, 8(A0)
```

B.next = &C:

```
move.l  #B, A0
move.l  #C, 8(A0)
```

C.next = NULL

```
move.l  #C, A0
move.l  #0, 8(A0).
```

head = &A

```
move.l  #A, head.
```

head->value2 = 1

```
move.l  head, A0
move.l  #1, 4(A0).
```

head->next->value2 = 4;

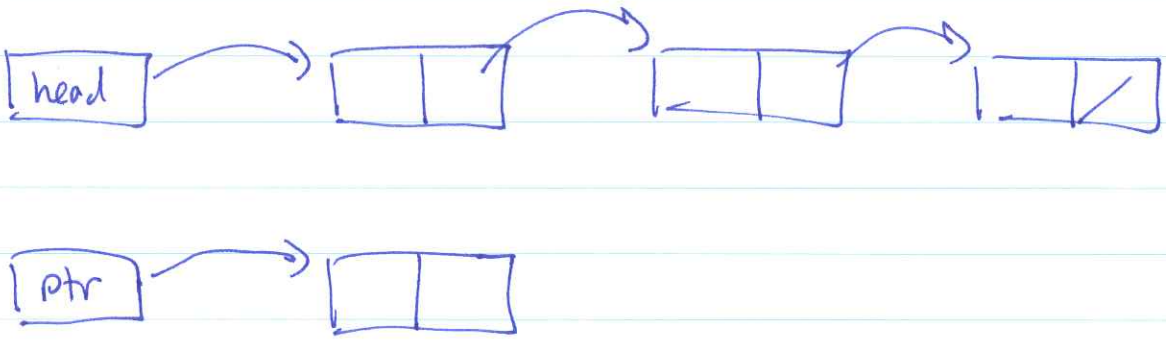
```
move.l  head, A0
move.l  8(A0), A0
move.l  #4, 4(A0).
```

head → next → next → value2 = 10

```
move.l  head, A0
move.l  8(A0), A0
move.l  8(A0), A0
move.l  #10, 4(A0).
```

Exercise:

Suppose we have a linked list of a list element:

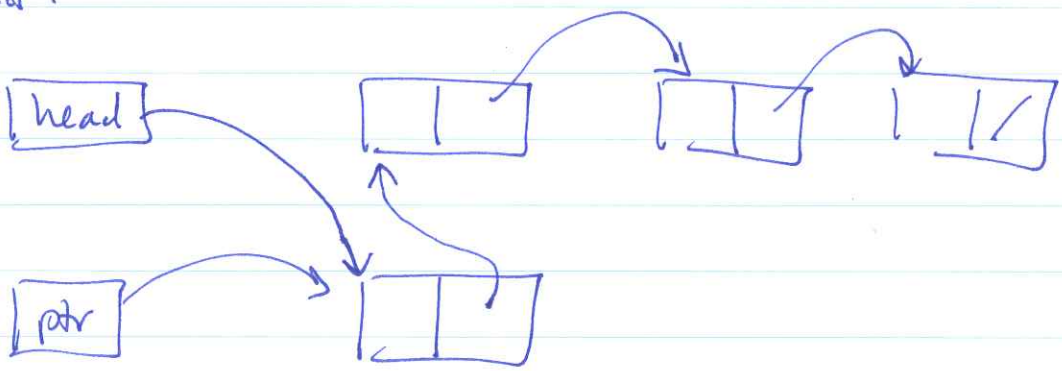


and we want to insert the list element at the start of the list.

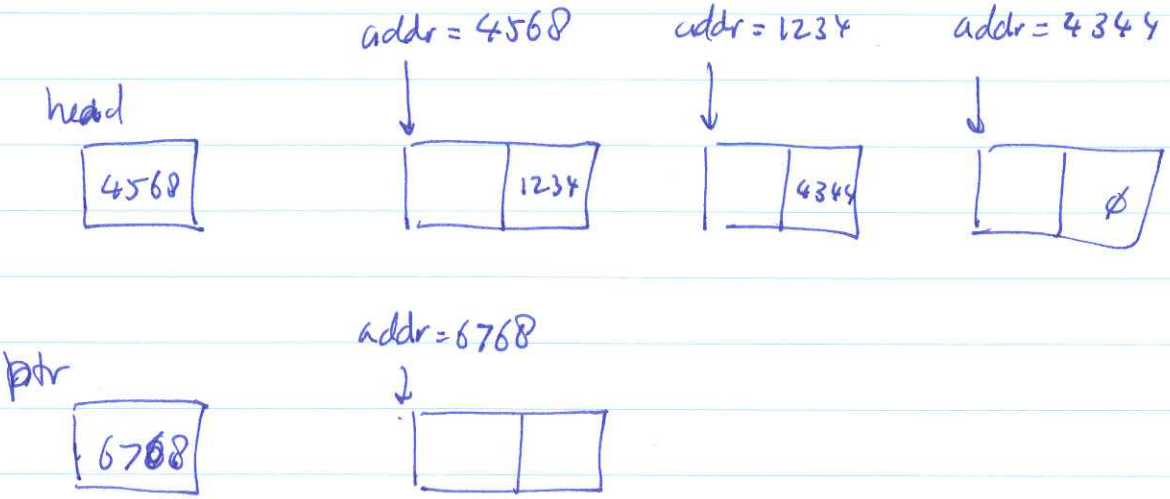
In C:

```
ptr → next = head;
head = ptr;
```

Result:



What is really happening? The linked list is really like this:



When you follow the addresses fields:

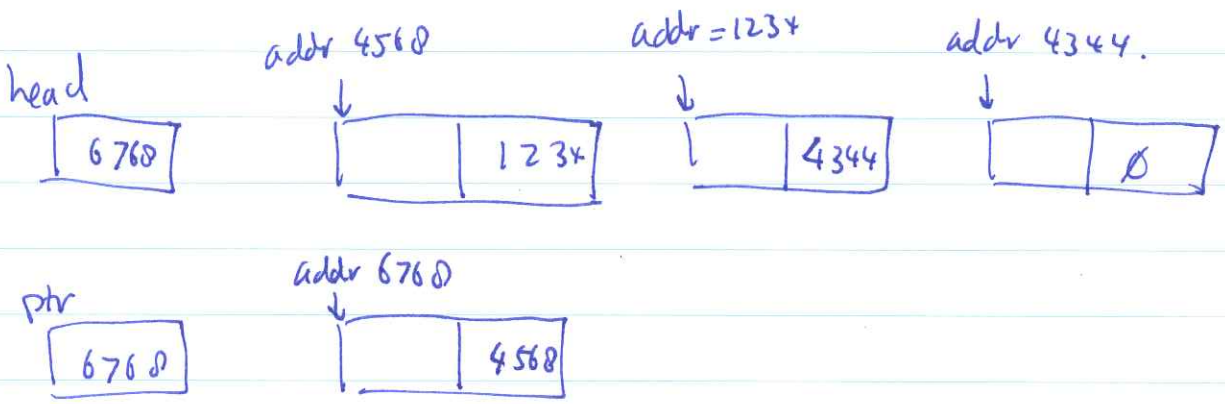
head = 4568.

at 4568 : next = 1234. — next element

at 1234 : next = 4344 — next element

at 4344 : next = ∅ end of list.

If you want to insert the list element at the start of the list, you need to change the content of the variables to:



In assembler:

ptr → next = head

```
move.l  ptr, A0
move.l  head, 4(A0)
```

head = ptr

```
move.l  ptr, head.
```

Exercise:

Insert list element at end of linked list:

Can't do it yet
We have not covered while loops !!

M68000 can only use .w

Indexed addressing mode: d8(An, Xm.W)

- The indexed addressing mode uses 2 registers
  (1) A "base" register
  (2) An "index" register.
- It is the addressing mode used to access array elements
- M68000 has an extended indexed addr. mode: called:

  "Indirect addressing mode with index and displacement".

  (page 150 Problem)
- The effective address is specified as:

  8 bit 2's compl. number

  d8(An, Xm.W)

The operand is located at memory address:

d8 + An + Xm.W

d8: 8 bit (extend to 32)
An: 32 bits
Xm.W: 16 bit (extend to 32)

Example:

suppose: A0 | ← 4000 →

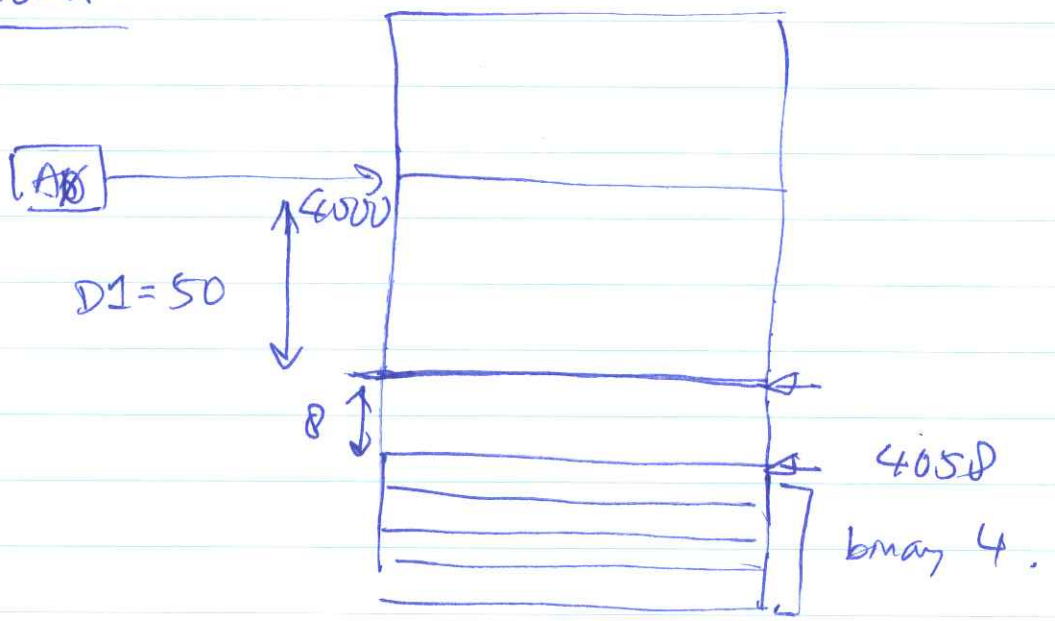
D1 | ~~~ | ← 50 →

Then:

move.l #4, 8(A0, D1.W)

will: ~~plant~~ 4 at memory location (8 + 4000 + 50)
= 4058.

Result:



* Usage of indirect addr. mode with index & displacement

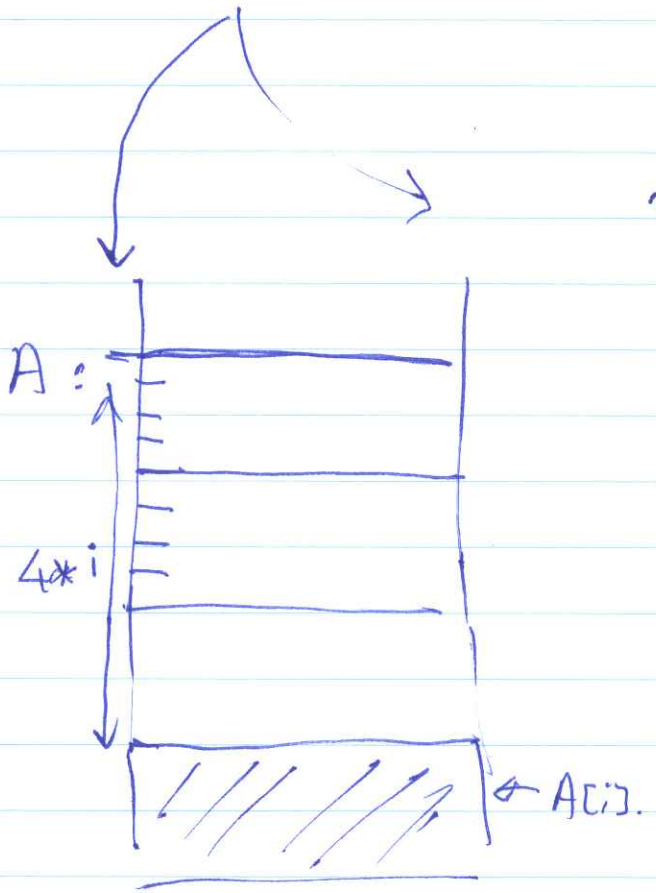
(1) Access away of simple variables
(use displacement = ∅)

(2) Access away of structures.

Example usage: access array of simple variables.

```
int A[100];
int i

A[i] = 4;
```

```
A:  ds.l  100
i:  ds.l  1.
```




```
move.l  #A, A0

move.l  i, D0
add.l   D0, D0
add.l   D0, D0

move.l  #4, 0(A0, D0.w)
```

We only use the lower 16 bits

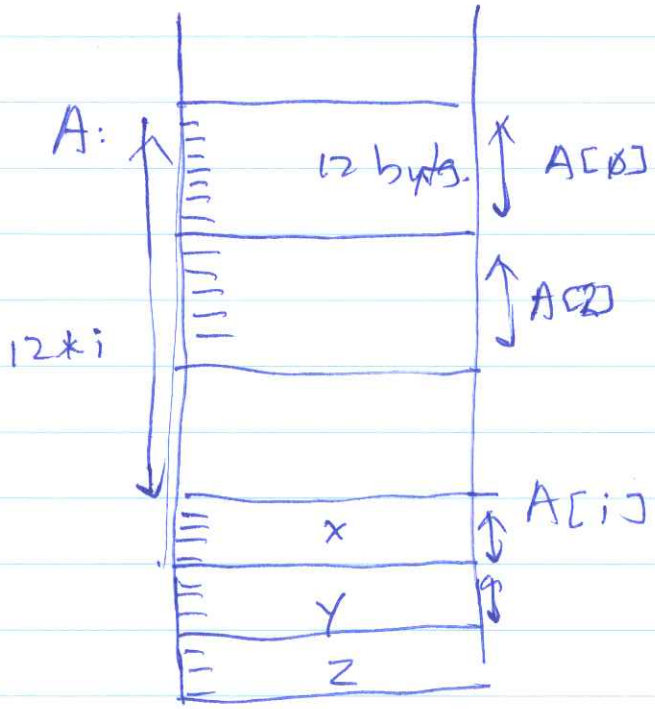
Example usage: access array of structures.

```
struct { int x;
         int y;
         int z;
       } A[100];
int i;

A[i].y = 14;
```

```
A: ds.l 300
i: ds.l 1
```

```
move.l  #A, A0
move.l  i, D0
add.l   D0, D0
add.l   D0, D0
move.l  #14, 4(A0,D0.w)
```

Pictorally:



(cover (An)+ and -(An) later!).

first: head towards project.