M68000 Direct Addressing Mode

- Operand specified with the direct addressing mode is the value contained in a location given "directly"
- Two direct modes:
  - Register direct: operand is in register (whose name is given directly)
  - Memory direct: operand is in memory, whose address is iven directly)
- Recall the context that the address mode is used within the MOVE instruction:

```
MOVE <EA>,<EA>
      ^    ^
      |    |
      |    +--- Source operand 2 and Destination
      +-------- Source operand 1
```

- **Syntax to specify the *direct* mode:**

```
Dn   - Data register direct (n = 0, 1, 2, 3, 4, 5, 6, 7)
An   - Addr register direct (n = 0, 1, 2, 3, 4, 5, 6, 7)
N    - Memory direct:       (N = a constant number (= address in memory))
```

NOTE: **labels** can be (and are often) used instead of a constant number, because labels are **always** equated to a constant number by the assembler !

- Semantics (meaning):
  - `Dn`: The operand is located in the data register Dn
  - `An`: The operand is located in the address register An
  - `N`: The operand is located in the memory at address N
  - **NOTE:** the **size** of the operand is given in the **instruction** !!!

- **Examples:**

```
(1) MOVE.B D1, D0     move byte from D1 into D0
(2) MOVE.B 0, D0      move byte at memory address 0 into D0
(3) MOVE.W 0, D0      move word (short) at memory address 0 into D0
(4) MOVE.L 100, D0    move long word (int) at memory addr 100 into D0
```

- **Advanced example:**

```
        MOVE.L A,D0      move the first element of array A (A[0])
        ...              into D0
        ...
A:      DS.L 10          int A[10]
```

Contrast the result with the **immediate** mode:

```
        MOVE.L #A,D0     move the address of the array A
                         (which is equal to the address of the first
                          element of A (A[0]) in D0
        ...
        ...
A:      DS.L 10          int A[10]
```

- Here is a demo to illustrate the difference between the **direct** mode and the **immediate** mode:
  - **DEMO:** click here

- Difference between "MOVE.L #A,D0" and "MOVE.L A,D0" in **assembler code**

```
000000 203C          MOVE.L #A,D0
       0000
       000C
000006 2039          MOVE.L A,D0
       0000
       000C
00000C       A:      DS.L 1
000010               END
```

  - In **both** cases, the label **A** is translated into the constant **0000 000C**
  - The different is in the **instruction code** (203C vs. 2039)!

    - In the first case, the instruction code **203C** tells the CPU to use **0000 000C** (hex) as the **operand** (a constant number)
    - In the second case, the instruction code **2039** tells the CPU to **get the operand in memory** at **address 0000 000C** (hex)

- **Direct mode using memory operands**
  - There are **two ways** to store **a serie of bytes** in memory:

    - **Big endian**: stores the serie of bytes from **left to right**

Example:

```
  11110000  00001111  10101010  01010101

Stored in memory as:

           +-----------+
           |   ....    |
           +-----------+
           | 11110000  |
           +-----------+
           | 00001111  |
           +-----------+
           | 10101010  |
           +-----------+
           | 01010101  |
           +-----------+
           |   ....    |
           +-----------+
           |   ....    |
```

- **Little endian**: stores the serie of bytes from **right to left**

  Example:

```
  11110000  00001111  10101010  01010101

Stored in memory as:

           +-----------+
           |   ....    |
           +-----------+
           | 01010101  |
           +-----------+
           | 10101010  |
           +-----------+
           | 00001111  |
           +-----------+
           | 11110000  |
           +-----------+
           |   ....    |
           +-----------+
           |   ....    |
```

- Only **Intel** CPU (AMD included) used **little endian**

  M68000 uses **big endian**

- When using **memory operands**, be careful that you use the **correct size** or you may retrieve the **wrong value**

Example:

```
move.l #-5, d0
move.l d0, 5672

move.l 5672, d1          d1 = -5

move.w 5672, d2          word operand in d2 = -1 !!!
move.b 5672, d3          byte operand in d3 = -1 !!!
```

- **Example Program:** (Demo above code) 

  - Prog file: click here

- Now you should understand that the following high level programming language construct

```
(static) int A, B, C;

C = A + B;
```

is equivalent the following in M68000 assembler language:

```
    MOVE.L A,D0          Get value in memory location A into D0
    MOVE.L B,D1          Get value in memory location B into D1

    ADD.L  D0,D1         Now D1 has the sum

    MOVE.L D1,C          Put the sum (D1) in memory location C
    ...
    ...
A:  DS.L 1
B:  DS.L 1
C:  DS.L 1
```

NOTE: we put the variables at the END to prevent them from being "fetched and executed as instructions"

- **Repeated Warning**:
  - Make sure you use the **proper size** especially with memory variables:

```
    move.l  A, d0          is correct
    ...
A:  dc.l  -4
```

But don't be surpised to be the wrong value if you use a different operand size:

```
    move.w  A, d0          is INCORRECT !!!
```

```
    ...
A: dc.l  -4
```

- Here is a program to show you what was happening: **DEMO** click here

---