

Intro to M68000 Assembler Programming...

- Assembler Instruction Format:

```
[Label]  Opcode Operand(s)  [Comment]
Example:
Loop:   move.b d0,d1      The rest of line is comment
```

- Effect of **processing** an **assembler instruction**:

- The **assembler** will **replace** the **mnemonic** `Opcode Operand(s)` with:
 - the **corresponding machine instruction code** (which is a **binary number** !!!)

The **machine instruction code** will be **stored** at the **next memory location** in the **assembly process**
- If a **label** is **included on the assembler code line**, then:
 - the **symbolic name label** is set **equal** to the **address of the instruction** in memory

- **NOTE:**

- **Never** start writing the **assembler instructions** at **column 1** !!!!!

Because the **assembler instructions** can be **mistaken** as a **label** !!!

- **Comment line:**

- If column 1 contains the character "*" (star), then the entire line is a comment line

Example:

```
123456789 <----- column number
```

```
* This whole line is a comment
```

```
* This line will cause a problem - because * did not start on column 1
```

- The **effect** of **labeling** assembling instructions:

- It is **crucial** to know that when you assign a label to an assembler instruction, the symbolic name of the label will be **equated** to the memory location at which the assembler instruction is stored.

(Because we will be labeling instructions like a madman when we write programs in assembler...)

- Here is an assembler program with 4 labels (L1, L2, L3 and L4) attached to instructions along with 3 labels (A, B, and C) attached to **ds** directives: [click here](#)

Assemble it and look at the assembler listing file **a.lst**

You should see the following:

```

1 000000      * Demonstrate the effect of DS directive
2 000000      * Assemble with: as255 instr
3 000000      * Look at the output file a.lst
4 001000                ORG $1000
5 001000 1200          move.b d0, d1
6 001002 1200  L1:     move.b d0, d1
7 001004 1200          move.b d0, d1
8 001006          A:   ds.b 10
9 001010 1200          move.b d0, d1
10 001012 1200 L2:     move.b d0, d1
11 001014 1200          move.b d0, d1
12 001016          B:   ds.w 10
13 00102A 1200          move.b d0, d1
14 00102C 1200 L3:     move.b d0, d1
15 00102E 1200          move.b d0, d1
16 001030          C:   ds.l 10
17 001058 1200          move.b d0, d1
18 00105A 1200 L4:     move.b d0, d1
19 00105C 1200          move.b d0, d1
20 00105E                end

                                SYMBOL TABLE
                                *****
A      001006          B      001016          C      001030
L1     001002          L2     001012          L3     00102C
L4     00105A

```

Notice that:

- The **symbol L1** is **equal** to the **address value (number) 001002₍₁₆₎**

- **In fact:**

- **Every label** will be **equated to** some **numeric value** when the assembler has **processed** the **assembler program** !!!

This will be **important** when we use **labels** to **access variable** and in the **branch instruction**:

- The **labels** will be **replaced** by a **numeric value** !!!

Overview M68000 assembler instructions

- M68000 instructions can be subdivided into 5 broad categories:
 - Data movement instructions (data copy is a better name)
 - arithmetic operations (add, sub, mult, div)
 - logic operations (and, or, not, shift, rotate)
 - branch and jump instructions (include subroutine call & return)
 - system control instructions - will not be covered here
- M68000 instructions can have:
 - 0, 1, or 2 operands
 - In binary operations (instructions with 2 operands), the second operand doubles as destination


```
ADD D0, D1      - operands are D0 and D1 (D0 + D1)
                 - The second operand doubles as destination
                   i.e., the result of the operation is stored
                   in D1
```
- Categories of M68000 instructions
 - Data movement (MOVE)
 - Arithmetic (ADD, SUB, MULS, DIVS)
 - Logic (AND, OR, NOT)
 - Branching (BRA, Bcc, BSR, JSR, RTS)
 - Controlling the CPU (not covered)
- Mapping of High Level Language constructs in M68000 Assembler
 - Program in High Level Language is first translated into assembler and then into computer code !

So there is always a way to express any construct found in any High Level Language in assembler language

- Variable definitions
 - DS and DC directives
- Constant definitions
 - EQU directive
- Statements
 - Assignment statement
 - MOVE
 - ADD, SUB, MULS, DIVS (to evaluate the expression)
 - AND, OR, NOT
 - Conditional statements (if, if-else)
 - BRA
 - Bcc
 - Loop statements (while, for, do-while)
 - BRA
 - Bcc
- Subroutine call
 - BSR, JSR
 - RTS

-
- Handout two sheets of papers describing M68000 instructions...

- M68000 instruction mnemonic codes and their meaning: [click here](#)
 - M68000 instruction mnemonic codes and the allowed addressing modes: [click here](#)
-

- How to read allowed effective addresses:

Instruction	Size	#	Dn	An	...
.....					
move <ea>, <ea>	BWL	*X	*	*X	

Size = BWL instruction can use sizes B, W and L
So you may use:

```

move.b .. , ..
move.w .. , ..
move.l .. , ..

```

The other columns list the "addressing modes"

```

#   represents a constant as operand
Dn  represents a data register as operand
An  represents an address register as operand
...

```

<ea> is the "effective address"

The marking in the columns give you information about

whether certain type of operands may be used for <ea>

- * means allowed, unless restricted by other comments
- X means not allowed in destination

So:

- * under Dn means: data registers can be used both in source and in destination
 - *X under # means: constants can be used in source but not in destination (make sense, you can't change a value of a constant...)
 - *X under An means: address registers can be used in source but not in destination (recall that we must use movea for address registers !)
- i.e.: movea ..., An

- I found an online M68000 instruction sheet that is not so descriptive, but at least it's in electronic form that you can do a "search" on: [click here](#)