

## Addressing Modes

---

### Announcements

- Lab 0 is due at midnight tonight. Refer to the electronic submission handout for details. Paper copies will be due on Wednesday.
- Lab 1 goes out today. A bug was fixed on Friday, so if you downloaded it before then, you should get a new copy. It's due 4/21/99 at midnight.
- Doug will miss his office hours today.

### Last Time

- Instruction Representation
  - A simple computer
    - Instruction Representation
    - Inner workings
    - Example
  - The real 68000
    - Differences from the simple model
    - 68000 Instructions
- Assembly Language
  - What is assembly language good for?
  - How do we write in assembly language?
    - Syntax of instructions
    - Other assembler features

### This Time

- Lab1
- DC and DS
- Addressing Modes
  - What are Addressing Modes?
  - Register Direct Addressing
  - Address Register Indirect
  - Absolute Addressing
  - PC Relative Addressing

## Lab1

Lab1 is the first substantial assignment. There is a lot more to do than in Lab0. It will be due on 4/21/99, so please try to start it as soon as possible.

In order to complete Lab1, you will have to use some instructions that we have not covered. Use the 68k reference manual as it contains all the information you will need. If you are unsure how to perform some operation in assembly, the 68k reference manual is the right place to look.

## DC and DS

Although both DC and DS are in the notes from last lecture, Doug did not get a chance to discuss them, so will cover them very briefly today. If you would like more information on them, please see the notes from last lecture.

Let us now take a look at the define constant (DC) assembler directive, which CodeWarrior actually supports. The syntax for DC is as follows:

```
[Label] DC.{B,W,L} expression [, expression2 [, expression3 [...]]]
```

It is important to note that DC actually sets aside memory and sets the value of that memory to the (compile-time) value of the given expression. Let us look at an example to help clarify things. Say we had the following code (written in traditional assembly):

```
CLR.L    D1
DC.L     $D01160A0
CLR.L    D1
```

What happens when we assemble this? We get:

```
$4281 D011 60A0 4281
```

Notice that the value specified by the DC gets stuck right in the middle of the code—no questions asked. Why would we ever want to do such a silly thing? Usually, we use DC for storing constant data that doesn't fit in 32 bits. For instance, as you will see in assignment 0, DC is great for allocating space for constant strings. You can do:

```
MyString DC.B "Hello World!"
```

You might also be tempted to use the space allocated by DC to store a variable. *This will not work.* Because DC allocates memory in the same space that the program resides, you cannot write to it (at least in the Palm, where the programs execute from write-protected memory).

Finally, we have the define storage (DS) directive. The DS directive allows you to allocate some number of bytes/words/longs that is uninitialized. This directive is supposed to actually let you define storage that you can use to store global variables. An example is:

```
GlobalArray DS.L 10
```

This example allocates space for 10 long words. It should be obvious that the memory allocated by the DS directive should be allocated somewhere else (not in the program space). On Cöder (the prior assembler for this class), the space is allocated relative to A5. This makes sense, since on the Macintosh, A5 points to the space for globals. However, for whatever reason, **Metrowerks does not get DS right**. They allocate storage in the program space—a bad idea.

In Codewarrior's "assembly in C", to get the same effect we can define a global variable outside of our assembly routine. These will be allocated A5 relative and can still be accessed in assembly.

## Addressing Modes

### What are Addressing Modes

Today we're going to go through a bunch of different addressing modes. There are quite a few of them, but most of them have common themes. Besides, after a few of them, it will start to come together. If you have questions about a particular addressing mode, see section 2 of the 68k reference manual.

What are addressing modes? In short, they allow us to specify where the operands are for instructions. For example:

```
CLR.L D0
```

The addressing mode tells this instruction to modify a data register and then, specifically, which data register to modify.

There are a lot of addressing modes. Here's a list of some of them:

1. Register Direct
  - Data register direct
  - Address register direct
2. Memory address modes
  - Address register indirect
  - Address register indirect with postincrement
  - Address register indirect predecrement

- Address register indirect displacement
  - Address register indirect index and displacement
3. Special Addressing
- Absolute short
  - Absolute long
  - PC with displacement
  - PC with displacement and index
  - Immediate

## Register Direct Addressing

### Data Register Direct

Register direct addressing allows to specify a register which contains the operand for the instruction. Let us take a look at data register direct addressing first. The syntax for it is as follows:

Syntax:     Dn                 {n: 0..7}

This means that to specify register direct addressing, we simply specify D followed by the register number. You've already used this mode several times, and will use it extensively throughout the course. Let's take a look at an example of data register direct addressing in action. Take the clear instruction, for example:

```
CLR.W         D5
```

This instruction, when assembled, is:

```
0100 0010 01     000   101
$4    2     4            5
```

After this instruction, the lower half of D5 is cleared. You should be pretty comfortable with this addressing mode by now. But how does data register direct compare with address register direct?

### Address Register Direct

Here is the syntax for address register indirect:

Syntax:     An                 {n: 0..7}

Okay, so syntactically they are the same. But they are not interchangeable. Let us look at the clear instruction again. For example:

```
CLR.L         A0               =>         0100 0010 10     ??????
```

If we try to assemble the instruction, everything will go fine until we have to specify the addressing mode of the operand. If you look in section 4-74, you'll see that the chart just has dashes where address register direct mode is specified. This means that address register direct mode cannot be used with this instruction. This goes back to the decisions the designers of the 68k made about "address" registers and for what they should be used. Does it ever make sense to zero out an address and then use that address? According to the designers, no it does not. Hence, the address register direct mode cannot be used with the clear instruction. Other invalid uses of address register direct mode include:

```
MOVE.L    D0, A0
MOVEA.B   D0, A0
```

If you try to hand assemble these instructions, you will find that it cannot be done. In the first example, the designers simply wanted to make sure you knew what you were doing by making you specify `MOVEA` rather than just `MOVE` when you wanted to put an address in an address register. But what about the second example? Because a single byte can never be used as an address by itself in the 68k, the `MOVEA` instruction is undefined for the byte size. Finally, let's look at an example that actually does work:

```
MOVEA.L   D0, A0
```

This instruction, when assembled is:

```
0010 000 001 000 000
$2   0   4         0
```

## Address Register Indirect

### Address Register Indirect

As exciting as register direct addressing is, you've all seen that before and it's time to move on to bigger and better things. Let us now examine address register indirect addressing, which has the following syntax:

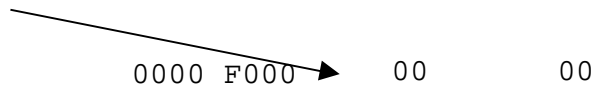
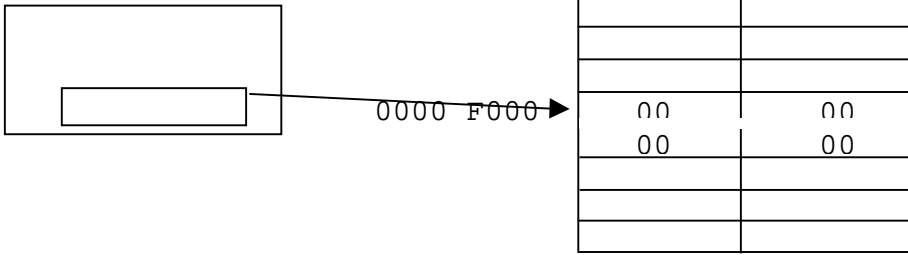
Syntax:      (An)            {n: 0..7}

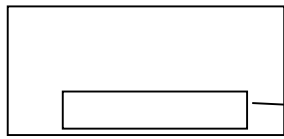
Now, let's look at an example, using, of course, our favorite example, clear.

```
CLR.L     (A3)
```

This instruction, when assembled, is:

```
0100 0010 10   010 011
$4   2   9         3
```

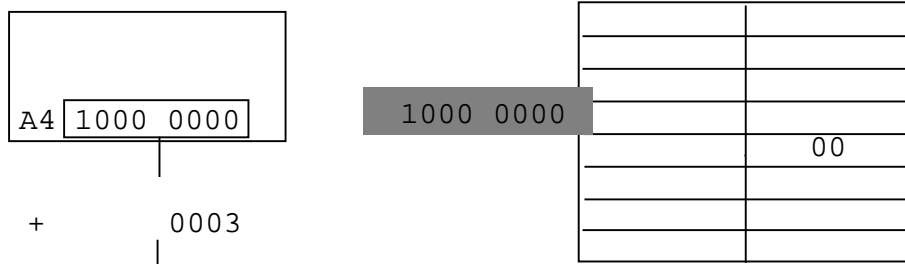




0000 0FFC →

00	00
00	00

If A4 is 1000 0000 before the instruction:



EA 1000 0003

Now, what is happening here, is that the displacement is added to the value in A4. However, it is very important to note that the displacement is a signed short integer (one word, 16 bits). This means, that a displacement of FFFF would be negative, which is perfectly acceptable. To allow this, the 16 bit displacement is sign extended before adding it to the 32 bit address. Sign extension involves taking the most significant bit of the 16 bit displacement and repeating that bit for the absent 16 bits in the second word. Again, remember that the value of A4 does not change. Let's compare displacement and postincrement addressing modes, assuming A4 is still 1000 0000:

		Cleared	A0 afterwards
CLR.W	2(A4)	1002	1000
CLR.W	(A4)+	1000	1002

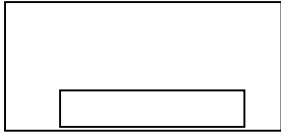
### Address Register Indirect with Index and Displacement

Building on displacement mode, is the index and displacement variant of the address indirect addressing mode. Here's the syntax:

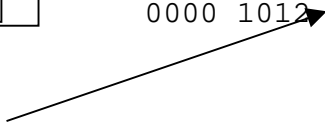
Syntax:  $d(A_n, X_i[.W, .L])$  {n: 0..7}  
 or  $(d, A_n, X_i[.W, .L])$  {d is 8 bits}  
 {X<sub>i</sub> is a register, i: 0..7}

Alright, this one is a little more complex than the those that we have looked at previously, but it is basically the same as all the rest. In contrast to the 16 bit displacement in address register indirect with displacement mode, the displacement in this mode is only 8 bits. X<sub>i</sub>, in this case, can be either an address or data register. The [.W, .L] part determines how much of X<sub>i</sub> gets used to calculate the index. Hopefully an example will make this a little clearer.

CLR.L 2(A3, D4.L)



0000 1012



00	00

kilobytes or last 32 kilobytes of memory. In general, it is important to know that absolute short addressing exists, but you probably will not use it very much.

## Absolute Long

The opposite of absolute short addressing, is, of course, absolute long addressing. Here's the syntax:

Syntax:     (XXX).L     or     XXX.L     {XXX is long address}

Absolute long addressing allows you to specify a full 32 bit address. You will probably use absolute long addressing more than absolute short. It is important to note that the size of the instruction can be different than the size of the address used to specify the operand. For example:

```
CLR.W       $F0002002.L
```

The above instruction is perfectly valid.

## PC Relative Addressing

### PC Relative with displacement

In addition to using address registers, the PC register can be used in some indirections. Be careful, however, because things relative to the PC are read only. Here's the syntax:

Syntax:     d(PC)     or     (d, PC)     {d is 16 bits}

As was mentioned earlier, memory relative the PC is read only and therefore, instructions such as clear cannot be specified. For example, if you tried to hand assemble the following instruction, you would be unable to do it.

```
CLR.W       2(PC)       =>       invalid
```

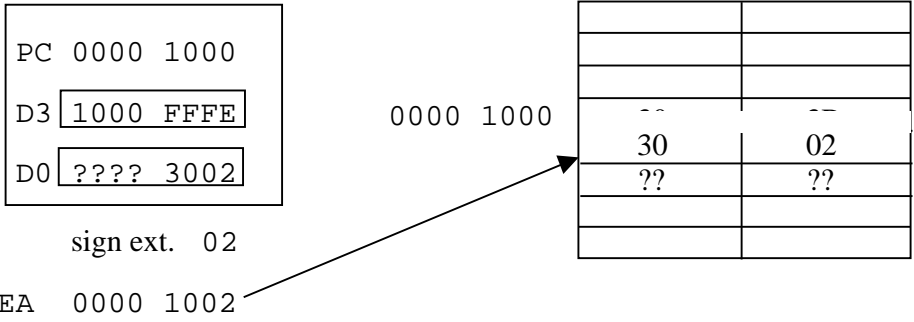
In addition to remembering that things accessed relative the PC are read only, remember that because the PC gets incremented whenever an instruction is read, it is always two bytes ahead. For example, let's look at the following operation:

```
MOVE.W     2(PC), D0
```

The assembly for the above instruction is:

```
00  11  000  000  111  010
$3      0      3  A
0000    0000    0000 0011
$0      0      0  2
```





Now, why is the effective address \$1002? Well, because the lower word of D3 is the hexadecimal representation of  $-2$  and the displacement is  $+2$ , they cancel out. That just leaves the PC automatic incrementation, which makes the PC \$1002 when calculating the effective address. Now, because the index and displacement cancelled out, \$1002 becomes the effective address and the contents of memory at \$1002 are moved into D0.

