

## Flow Control

---

### Announcements

- Lab #0 printout due now.
- Office hours are back to regularly scheduled times.

### Last time

- DC and DS
- Addressing modes galore
  - What are addressing modes?
  - Data/Address Register Direct
  - Address Register Indirect
  - Absolute

### This time

- Misc. points
  - Definition of assembly language
  - What is assembly language good for (*skip*)
  - EQU
- More addressing modes
  - PC Relative
  - Immediate
- Random stuff
  - MOVE extension words
  - A5-relative globals
  - LEA
  - Labels—what do they mean?
- Flow control (hopefully)
  - Unconditional flow control
    - BRA
    - JMP
  - Introduction to conditional flow control

### Miscellaneous points

Last Wednesday, we didn't quite finish all the lecture notes. On Monday, Cameron covered the important parts, but there were a few things that he didn't go over. A bunch of it (talking about the history of assembly language) isn't really that important and we're going to skip it. If you are interested in this stuff, check out the handout and stop by my office hours. Although this accounts for a large portion of the material we didn't get to, there are a few other parts that I want to make sure you've seen.

## *The definition of an assembly language*

Just a quick definition for you: an assembly language is a programming language that has a one-to-one mapping to machine code. That is, a programmer writing in assembly language can figure out exactly what the machine code for his/her program is. The assembler has no leeway. This is different from high-level languages (like C), where the compiler has a choice about what the actual machine representation will be.

## *DC/DS/EQU*

Cameron already talked about `DC` and `DS`. Just as a reminder, `DC` allows you to place a data value in memory directly. `DS` (in theory) sets aside storage someplace in memory for global storage. Metrowerks doesn't deal with `DS` correctly.

What is `EQU` though? It turns out that you will never use `EQU` in this class—I simply want to mention it so you've seen it. `EQU` is the standard assembly syntax for doing a `#define`. It works something like:

```
Label      EQU      expression
```

The reason you will never use `EQU` is that CodeWarrior doesn't support it. I guess they figured that there was no reason to have two identical features in their assembler, since you can just use `#defines`.

## **More addressing modes**

On Monday, we started on the (somewhat tedious) catalog of the 68000's addressing modes. Today, we will finish up with a few more modes and then get on to some (hopefully) more interesting material.

## *PC relative addressing modes*

The next addressing modes that we'll cover are the `PC` relative addressing modes. These are addressing modes that let you access data that is located someplace relative to the `PC`. `PC` relative addressing allows you to easily write code that is *position independent*. *Position independent code (PIC)* will run no matter where it is located in memory. `PIC` is important for a few reasons:

- Subroutines can be written once and then put into an arbitrary location without having to change the code.
- Multiple programs can run at once easily. If programs weren't position independent and two programs both wanted the same location in memory, you couldn't run both at once.

Note that `PIC` isn't strictly needed to write programs for the 68K—it just makes things easier organizationally. There are also other solutions that don't require you to write `PIC`. One alternate solution is to fill in addresses at program load time. Another is to use specialized memory-management hardware to map addresses.

`PC` relative addressing modes work in a fairly obvious way once you're familiar with address register indirect with displacement modes. Let's take a look:

## PC relative with displacement

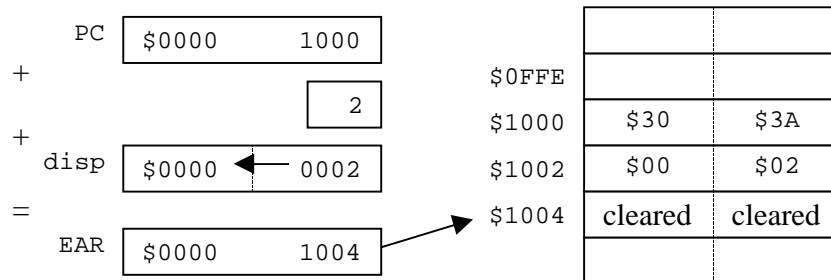
The first PC relative addressing mode is PC relative with displacement. Just like address register indirect with displacement, this figures out the effective address by adding a 16-bit offset to the current value of the PC. The syntax is pretty familiar:

Syntax:  $d_{16}(PC)$  or  $(d_{16}, PC)$

In order to get a feel for this mode, let's look at an example. We will look at the following instruction (assuming that the PC starts at \$0000 1000):

CLR.W 2(PC)

This example looks like:



A few important things to note:

- As you can see from our diagram, a mysterious value of 2 is added to the PC before the displacement is added. This value is explained by looking back to our simulation of the 68K CPU. If you remember, when we fetched an instruction, we updated the PC immediately by 2 to account for the fact that we'd already read the instruction. Thus, when we do PC relative addressing, we need to realize that the PC we're talking about is actually 2 greater than the location that the instruction itself is located at. In the words of the Motorola manual (page 2-13), "The value in the PC is the address of the extension word." Note that the address of the extension word is  $PC + 2$  for all instructions.
- As in address indirect with displacement, the displacement is sign-extended before addition. This allows for negative displacements.

We can hand-assemble this instruction:

```
00 11 000 000 111 010    ==> 0011 0000 0011 1010
                               $  3   0   3   A
                               $  0   0   0   2
```

**Important:** It appears that CodeWarrior attempts to be smart and does things a little differently. When you write:

CLR.W 2(PC)

...CodeWarrior interprets it as:

CLR.W 0(PC)

...in other words, it is attempting to make the +2 added to the PC transparent to the programmer. In class, we will stick to the above convention, but if you ever directly specify a displacement in CodeWarrior, realize that it may be a little different than you anticipated.

PC relative with index and displacement

If you know PC relative with displacement and you know address register indirect with index and displacement, this is an obvious addressing mode:

Syntax:  $d_8(PC, Xi\{.W, .L\})$  or  $(d_8, PC, Xi\{.W, .L\})$

Here's our example this time:

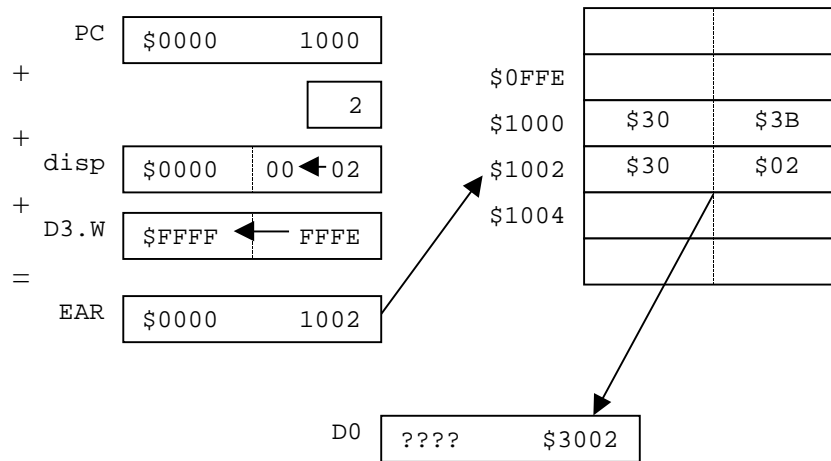
`MOVE.W 2(PC, D3.W), D0`

Let's first assemble this instruction:

```

00 10 000 000 111 011  ==> 0010 0000 0011 1011
                        $ 2 0 3 B
0 011 0 000 00000010  ==> 0011 0000 0000 0010
                        $ 3 0 0 2
    
```

Given that PC starts at \$0000 1000 and D3 starts at \$1000 FFFE, we have:



Differences between PC relative and Address register indirect

As already noted, the two PC relative addressing modes that we covered are very similar to address register indirect with displacement (and index). It seems like the PC can be used in place of an address register in any of the address register indirect addressing modes. *This is not true.* First, there are several address register indirect modes that just don't work with the PC:

- $(PC)$  isn't supported. Why would it be? It would simply give you back the value of the next instruction—why would you need that?
- $-(PC)$  and  $(PC)+$  aren't supported for the obvious reasons. You can't just change the PC.

The second difference is that PC relative addressing cannot be used as the destination of any operation. Thus, `CLR.L 10(PC)` is not allowed, nor is `MOVE.W D0, 10(PC)`. This is to make it difficult for programmers to write self-modifying code: a bad programming practice that we will look at later.

## *Immediate addressing*

The last addressing mode of the 68K is immediate addressing. Immediate addressing allows you to specify a constant value as the source operand of an operation.

Syntax:       #<number>

For instance:

```
MOVE.W       #1, D0
```

...puts the value 1 into the lower word of D0. Let's look at its assembly:

```
00 11 000 000 111 100   ===> 0011 0000 0011 1100
                              $  3    0    3    C
                              $  0    0    0    1
```

The immediate data follows the instruction. This is a fairly obvious addressing mode, so we won't diagram in memory what happens. However, it is instructive to look at a few different examples:

```
MOVE.L       #1, D0
```

...puts the value 1 into D0. Let's look at its assembly:

```
00 10 000 000 111 100   ===> 0010 0000 0011 1100
                              $  2    0    3    C
                              $  0    0    0    0
                              $  0    0    0    1
```

The important observation is that the extension needs to be 32 bits long, since we are moving 32 bits of data. This is sort of a waste (it would be nice to be able to specify only 16 bits in this case), but it's the way the 68K works.

One more example:

```
MOVE.B       #1, D0
```

...puts the value 1 into the lowest byte D0. Let's look at its assembly:

```
00 01 000 000 111 100   ===> 0001 0000 0011 1100
                              $  1    0    3    C
                              $  0    0    0    1
```

Notice that even though we only have one byte of immediate data, the immediate data still takes up a word. This is because all instructions must be an integral number of words long (you can't have a 1.5 word instruction). Thus, we pad the immediate data with 0's to balance it out.

## “Quick” instructions

This is probably a good place to bring to light a couple of neat instructions provided by the 68K to allow you to avoid multi-word instructions if your immediate data is fairly compact. These “quick” instructions are MOVEQ, ADDQ, and SUBQ. Let's assemble: ADDQ.L #1, D0:

```
ADDQ.L       #1, D0
```

```
0101 001 010 000 000   ===> 0101 0010 1000 0000
                              $  5    2    8    0
```

The other quick instructions are similar. One interesting note: the destination of the quick instructions *can* be an address register. There is no `ADDAQ`. However, when the destination is an address register, the condition codes aren't affected (just as if there was an `ADDAQ`). This is sort of an inconsistency in the 68K naming conventions.

### Immediate instructions

Another good thing to bring up here is the concept of immediate instructions. You may have noticed that for the `ADD` instruction, either the source or the destination must be a register. Thus, if you wanted to add some constant value to a location in memory (specified by an arbitrary addressing mode), you simply can't use the `ADD` instruction.

To solve this problem, the 68K has a number of *immediate* instructions, such as `ADDI`. These instructions allow you to specify an immediate source operand and an arbitrary destination operand. An example (that I won't assemble):

```
ADDI.L    #1, 0(A0)
```

### **Random stuff**

Now that we've seen all the addressing modes, there are a few topics that don't really fit into any other category. We'll slam through them here and then move onto the next section.

#### ***MOVE extension words***

As you may have noticed, the `MOVE` instruction is special. It is the only instruction that allows you to specify an arbitrary address for both the source and the destination. One question arises here though: which order do the extension words go in. As this isn't anyplace obvious in the 68K manual, I'll just tell you. Extension words are listed source first, then destination. A good way to remember this is that the `PC` in `PC`-relative addressing always contains `PC+2`, or *equivalently*, the address of the extension word. That means that the extension word for `PC`-relative addressing must come immediately after the first word of the instruction. Since `PC`-relative addressing can only be used for the source, the source must come first.

#### ***A5-Relative globals***

Earlier, we mentioned that global variables are allocated relative to `A5`. Now that we understand address register indirect addressing, we can see what that means.

On the Palm (and on 68000 Macintoshes), the operating system has a memory area set aside to hold global variables. Programmers can use this memory space to store data. This block is not at a fixed location in memory. Its location may change for different runs of an application. In order to allow applications to figure out where the location is, the OS puts a pointer to the memory in the register `A5`. The application programmer can then access memory locations in the block using address register indirect addressing.

An assembly language programmer could theoretically assign pieces of this memory to various variables by him/herself. However, this quickly gets tedious and invariably leads to bugs. To help out the programmer, the assembler usually provides some way to automatically allocate

memory from the global space. In some assemblers, DS will do this (not CodeWarrior). In CodeWarrior, if you want to allocate memory from the global space, you can just declare a global variable. When you access this global variable in code, CodeWarrior will fill in the location it assigned to you. As an example from the Toys stub code:

```
static VoidPtr gBase;
static Word gWidth, gHeight;
...
MOVE.L      kLCDBaseLoc, gBase          MOVE.L      $FFFFFFA0, -$006A(A5)
MOVE.W      D0, gWidth                  MOVE.W      D0, -$006C(A5)
MOVE.W      D1, gHeight                 MOVE.W      D1, -$006E(A5)
```

As you can see, the global variables were changed (at compile time) into references relative to A5. The compiler chose the offsets it did out of the pool of memory available in the application global space.

Note: as mentioned in the Toys stub code, if you want to access global variables, you must not modify A5.

### ***LEA***

LEA (Load effective address) can be used to figure out what the final value of an address is. An example is:

```
LEA          10(A0, D0.L), A1
```

This will store the value  $10 + A0 + D0$  into A1, because that's the address specified by  $10(A0, D0.L)$ . If there's interest (and time), we will assemble this in class. There's another (more realistic) example of LEA below.

### ***Labels***

So far, we have been using labels in our code without really understanding what they were for. For instance, we might have:

```
LEA          DeltaXs, A4
MOVE.W      0(A4, D0.W), D4
...
DeltaXs:    DC.W      0, -1, 1, 0
DeltaYs:    DC.W      -1, 0, 0, 1
```

What does this code look like after it's been assembled (and disassembled and cleaned up)?

```
LEA          *+$002A, A4                | 49FA 0028
MOVE.W      $00(A4,D0.W), D4           | 3834 0000
...
DC.W        $0000, $FFFF, $0001, $0000
DC.W        $FFFF, $0000, $0000, $0001
```

Notice that the assembler has replaced the first operand in the LEA with a PC-relative access (the PalmDebugger disassembles PC-relative references in a slightly different syntax, but it should be obvious what it is doing). The label allowed us (as programmers) to refer to another location in

our program without having to know its exact address (or its exact distance from the current instruction).

Note that labels aren't part of the 68K specification. They are a nicety provided by the assembler (just like the assembler helps us to allocate global space relative to `A5`). However, just about every assembler provides the concept of labels for referring to other locations in your program.

## Flow control

So far in class, all of our code has just executed one instruction after another. This doesn't really allow us to get too much done (as you can tell, we had to jump ahead a little bit in lab #0/1 to do anything interesting. We need to find some way to execute out of order.

**Idea:** Just modify the `PC` like any other register:

```
ADD.L    #0x30, PC                                doesn't really work
```

That would be a really interesting (and powerful) way to provide flow control. However, this isn't how we really do things (too unstructured I guess). Instead, we have special instructions that provide us with limited abilities to modify the `PC`.

## *BRA*

The `BRA` (for branch) instruction allows us to modify the `PC` by essentially adding to it or subtracting from it. Let's look at a silly little example:

```
Top:     ADDQ.L    #1, D0
         BRA      Top
```

What does this code do? It infinitely loops, continually adding 1 to `D0`. Not very useful, but very simple. Let's look at the machine language for this code. We've already seen that the `ADDQ` is `$5280`. What about the `BRA`? Well, we want to subtract 4 from the `PC`, or add -4 (the reason it's 4 rather than 2 is that the `PC` starts at `PC + 2`):

```
0110 0000 1111 1100
$ 6 0 F C
```

It is interesting to note that for displacements between -128 and +126, this instruction is only 1 word long. If the displacement is longer, we must expand into 2 words (as described in the 68K manual).

## *JMP*

The `JMP` (for jump) instruction allows us to modify the `PC` in more powerful ways. `JMP` allows you to set the `PC` to the value of an address register and also to set it directly to a constant value. As an example, let's say that we wanted to jump to the location stored in `A0`. We can do that with:

```
JMP      A0
JMP      (A0)
```

...note that I have shown both the wrong way and the correct way to do this. The first way (the wrong way) seems the most obvious, at least to me. However, for some reason the syntax for

JMP behaves more like a LEA. It loads the effective address of its operand into the PC instead of loading the operand itself. Let's look at the assembly for the instruction:

```
0100 1110 11 010 000   ==> 0100 1110 1101 0000
                        $  4   E   D   0
```

Fairly straightforward. Note that address register indirect with displacement (and index) also work. Let's also assemble an example with absolute addressing.

```
JMP          $1000

0100 1110 11 111 000   ==> 0100 1110 1111 1000
                        $  4   E   F   8
                        $  1   0   0   0
```

Again, pretty simple. Note, I have used `(xxx).w` addressing here because I've specified an address that was only 16 bits. Using `(xxx.L)` works fine too.

If you're looking carefully at the description of JMP in the 68000 manual, you might notice that PC relative addressing is also available. Interestingly enough, this gives you the same ability that the BRA instruction gave us. We can jump to some location relative to the current PC. So why does BRA exist? That's a good question. Probably the best answer is that for 8-bit displacements, BRA is more efficient (it takes up less memory).

## Conditional flow control

Next lecture, we will get into conditional flow control: flow control that decides whether to execute or not depending on the state of the processor.