5/27/2014

Learning From Assembly:

A report on the Motorola 68000 assembly language design project

Josh Desmond

Learning From Assembly:

A report on the Motorola 68000 assembly language design project

Background

Figure 1: A segment of assembly code from the M68k

A small piece of the code:

```
logicLoop
    ;applies gravity
    move.w  (BALL1+yVel),d1
    addq    #2,d1
    move.w  d1,(BALL1+yVel)

    ;moves based on velocity in the Y
    ;(D1 still   contains yVel + grav)
    add.w   d1,(BALL1+yPos)

    ;move based on velocity in the X
    move.w  (Ball1+xVel),d1
    add.w   d1,(BALL1+xPos)

    ;returns to mainLoop
    rts
```

In 1984, the original Apple Macintosh personal computer was released for the public for the price of just $2,500 ("US$5,595 adjusted for inflation" according to Wikipedia). The computer was one of many using the now affordable Motorola 68000 processor; by the late 1980's the price of a 68000 had substantially dropped, and the chip was being used for more specialized tasks than a home-computer, such as in arcade machines. Today, the M68k is still in use, and is the CPU used in the TI-89 series (released in 1998).

For my project, I found a 68k simulator, and went about learning the language through a series of programs. To learn the basics, I translated what were basic concepts in Java into assembly. My very first program of writing a "method" was more complicated than anticipated, and force me to rethink what I thought were inherit concepts in computer science. The program passed parameters via the stack, represented by an address register; a "method" itself was called by storing the current location in the code onto the stack, and jumping to the beginning of the method, and return back to the saved location in code at the end of the method.

Program Descriptions:

I wrote a total of five programs before making my way to the final program. Each of these had a focus or idea behind them that was necessary to understand before designing the final program. To start, I wrote a hello world program, and went line by line through a tutorial understanding exactly what each letter or command was doing. I did this again for the second program, which branched to different sections of code based on user input. This forced me to learn how to use subroutines, and to read user input through trap tasks.

Next on the list was writing "methods". This program was focusing on a similar concept to input branching, however, it forced me to do a little more planning. The program would take three separate inputs, first a number, either one or zero, represent adding or multiplying; then another two numbers that would be added or multiplied, and the output would be displayed at the end. The key design functionality however was that I did not create four different branches for the program. Instead of having two different sections of code for multiplying, and two more for adding, there was a subroutine after numbers were input for multiplying and another for adding. Thus I only had to write one section of code for the numbers input.

The last two concepts were dealing with mouse input, and dealing with linked lists. Creating a linked list was an extremely difficult concept to understand in assembly, and is explained in further detail below.

Figure 2: A segment of memory in the M68k

Data Structures:

Perhaps the most difficult part of the physics program was writing a linked list of objects. In assembly, the concepts that are fundamental to object oriented program have to be built from the ground up; making the design of a ball itself is quite complicated. *Figure 2* is an example of how three balls are represented (and accessed) in memory. To actually dissect what is being represented, we need to have the ball structure and the linked list structure first; this is shown in Figure 3

Figure 3: The Ball and Link data structures in assembly

Ball:

```
        OFFSET  0       ;"comments"
length  ds.w    1       ;This is 0
xPos    ds.w    1       ;2 (bytes)
yPos    ds.w    1       ;4 (bytes)
xVel    ds.w    1       ;6 (bytes)
yVel    ds.w    1       ;8 (etc.)
        ds.w    0       ;word alignment
B_SIZE  equ     *       ;10
        ORG     *       ;End of Ball
```

Link:

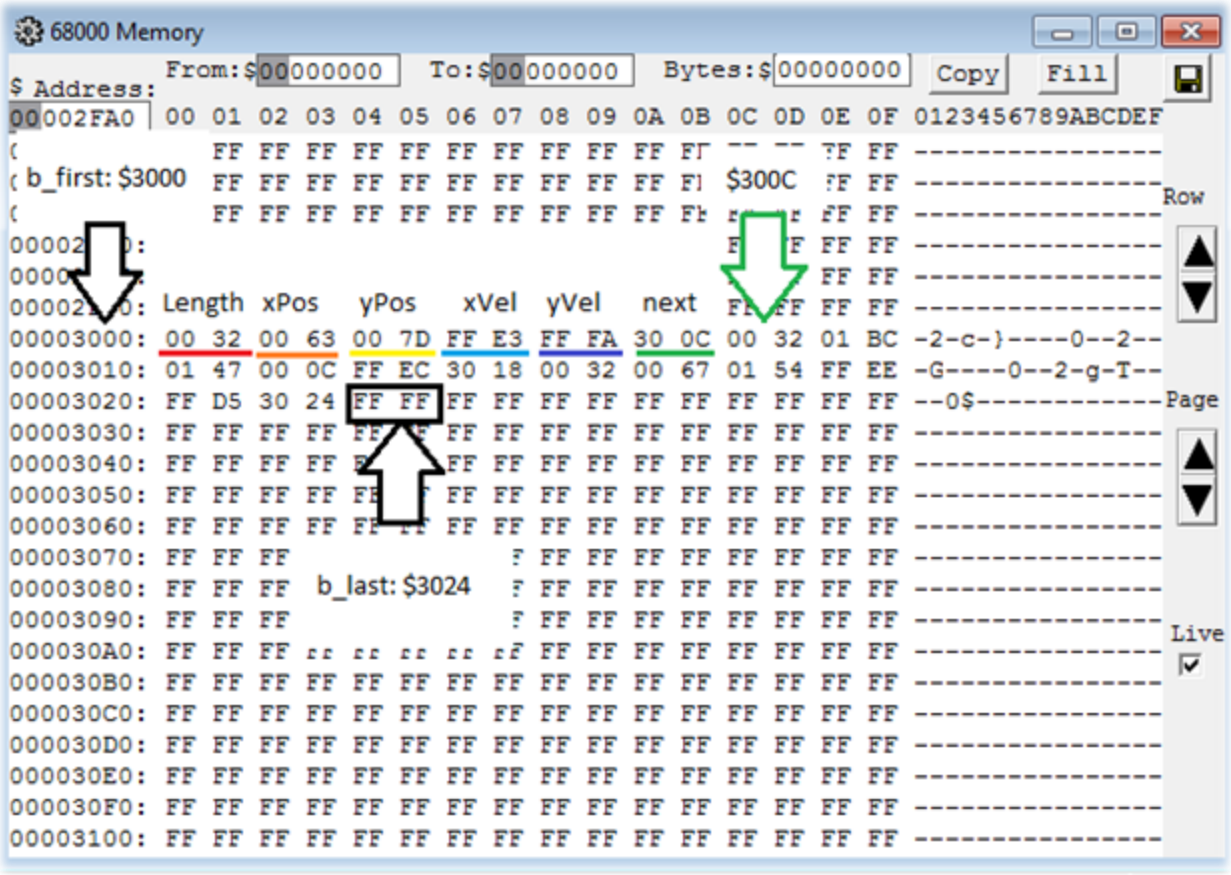
```
        OFFSET  0       ;
node    ds.b    B_SIZE  ;0
next    ds.w    1       ;10
        ds.w    0       ;word alignment
N_SIZE  equ     *       ;12
        ORG     *       ;End of Link
```

The OFFSET directive is used to declare relative locations in memory. The way these values are used to define an object is by using the labels *xPos,* or *next,* as a way to call numbers in memory a certain distance from a starting point. If for example, we had a ball start at memory location $3000 ($ is used for hexadecimal in the M68k assembly), then to retrieve its xPos, we would write *xPos($3000),* which is $3002-$3003 (as it is a word long). Having the code to the right can be used to decode our block of memory in Figure 2.

The section or structure titled Link is necessary for creating a linked list of balls. There is a node that is equal in size to the ball data structure, and a word sized section of memory titled *next* that is

used to point to the next node in the list. *Next* is referenced and treated in the exact same manner as calling the *xPos* of a ball. An annotated diagram of the linked list is shown on the next page in Figure 4.

Figure 4: The annotated version of the linked list



Running the programs:

To run the programs (feel free not to; like seriously, I'm only including this section in case you feel you need to), either run the installer for windows included in this folder, or go to http://easy68k.com/ and scroll to the bottom for installations. Run Easy68k, open one of the files in my project folder, and press F9 or click the assemble arrow/button at the top of the menu once open. This will bring you to the simulator, in which you can scroll through the memory by opening the menu under view, or run the program (again by pressing F9).