The 68000

Lesson 3 – First instructions

First Instructions

- Move
- Arithmetics
- Integers vs reals
- Additional information

Move : How to use

- We need to know how to move data to/from memory
  - move D2, D4 ; this will copy data from D2 to D4
- First parameter corresponds to the source
- Second parameter corresponds to the destination
- Must precise the data form
  - .B for a BYTE
  - .W for a WORD
  - .L for a LONGWORD

Move: Syntax

- move.B D2,D4 : byte
- move.W D2,D4 : word
- move.L D2,D4 : longword

- THIS IS CAPITAL !!!

Move : Different Uses

- May be used to store a value in a register
  - Example: To store 1270_{10} in the D4 register do "move.w #1270, D4"
  - # stands for immediate value
  - To store an hexadecimal number use "$" "move.w #$4F6, D4"
  - To store a binary number use "%" "move.b #%00000001, D4"
- Go through all the uses from lecture 2

Arithmetics

- Addition
- Subtraction
- Negative numbers

Addition

- Let's calculate 100 + 25
  - 100 can be stored in 8 bits so we will use byte operations
  - ```
    move.b  #100, D0
    move.b  #25, D1
    add.b   D0, D1
    ```
  - This will add D0 and D1, storing the result into D1. The result will be stored in binary, of course.
    ```

Subtraction

- Let's do 100 – 25
  - ```
    move.b   #100, D0
    move.b   #25, D1
    sub.b    D1, D0
    ```
  - This stores D0 – D1 into D0
    ```

Subtraction – Warning!

- What happens when you do
```
move.b  #100, D0
move.b  #25, D1
sub.b   D0, D1
```
- This will save 25 – 100 into D1 so you will get a negative number!

Using Negative numbers

- In the 68000, negative numbers are stored in the 2's complement format.
- You should know what that means, but just in case...
  - You take the absolute value of the number, turn it into binary, do the 1's complement and add 1.
  - -4 would be written %11111100

The importance of data size

- Let's take an example: 4 – 1
  - In binary this will be (if using a byte)

```
carry     1 1 1 1 1 1
          %00000100
+         %11111111
          ---------
          %00000011
```

  - If we keep the one byte data size, the result is 3, but let's suppose we use a word (16bits) The result would be %0000000100000011 = 259 !!!

Processor's point of view

- It calculates your addition on the data specified and stores the carry bit elsewhere (in the status register)
- You should always check your carry bit!
- You may want to use the "ext" opcode

The sign problem...

- Let' s imagine we have got an array in which all entries are bytes and in which numbers may be as well positive as negative.
- The first number stored is -1 : $FF=%11111111 (it is a byte!) Now, we want to add this to a number which is stored in D0 but this number is a word...
- If we did:

```
clr.w     D1; clear the D1 register's lower word
move.b    (A0), D1
add.w     D1, D0
```

This would be completely false, let’s see why...

The problem is out there...

- If we clear the register D1, we will obtain the value \$0000 in the lower word of D1
- If we then move our byte into D1, we will have the value \$00FF in the lower word of D1.
- So adding D1 will be adding 255 (\$00FF) instead of –1 (\$FFFF)

The truth is out there

- In order to avoid the sign problem we can use the opcode “ext”
- ext.W will extend the sign bit of a byte into a word and therefore will transform $00FF into $FFFF
- The right code becomes:

```
clr.w    D1
move.b   (A0),D1
ext.w    D1          ; transforms $00FF to $FFFF
add.w    D1,D0
```

Multiplications

- There are two possible multiplication instructions:
  - MULU if you are using unsigned numbers
  - MULS if you are using signed numbers
- "muls.x D2,D4" will multiply the lower x of D2 and D4, and store the 32 bit result in D4.

Divisions

- As in multiplications 2 divisions exist
  - DIVU for unsigned numbers
  - DIVS for signed numbers
- "divs.x D2,D4" will calculate D4/D2 and store the quotient in the lower word of D4 and the rest in the upper word of D4.

Integers Vs. Reals

- The 68000 processor has no build in instructions to handle floating point arithmetics (reals)
- We can simulate the float numbers by:
  - Write software routines which will handle the float calculus
  - Use integers (fixed numbers)
- Software routines are quite complex to make and rather slow, so we mainly use fixed point arithmetics (mode “fix” in calculators)

Some examples

- If we are in "fix2" mode, Π = 3.1416... Is represented by 3.14
- The number will be stored in memory as 314
- Numbers are multiplied by 100 (in this example)
- The problems are:
  - High calculus errors
  - Low accuracy

Additional information

- The move instruction has some limitations:
  - When you move one word or one byte into a 32 bits register, the upper word or the upper byte is not affected
  - When move is used to send or get something from memory, you cannot access to odd addresses when moving words or longwords. It would result in an "address error".
  - Never forget the parentheses when necessary