M68K EXPERIMENT 3: MATH INSTRUCTIONS IN M68K

This lab will be about the extension of the mathematical capabilities of M68K. With 32-bit data and address bus, M68K is capable of performing complicated mathematical operations with ease and speed. And we can operate on larger numbers by combining these instructions.

3.1. Extending Multiplication Capability

In this first part, you are going to perform a multiple-precision (32-bit x 32-bit) multiplication by using MULU and MULS instructions from. Since the multiplicand and the multiplier will be 32-bit long the result will be 64-bit long, and to store this result a 64-bit register will be needed. However there are no 64-bit registers is available in 68K architecture. Instead, you can synthesize it by concatenating two 32-bit data registers with the use of X-bit (Check Status Register).

```
ORG $400600 ;data section
NUM1 DS.L 1
NUM2 DS.L 1
RES64 DS.L 2

ORG $400400 ;program section
MAIN
LEA NUM1, A0 ; pass addresses of numbers by address registers
LEA NUM2, A1 ; (it is called passing param’s by reference)
LEA RES64, A2
JSR MUL_EXT ; then call the subroutine
TRAP #11

MUL_EXT ; implement your extended mult. function
...
RTS
```

You can test your implementation by performing the following operation (the calculation of the length of a light year in kilometers):

299,792 x 31,558,464 = 9460975039488 or in hex: $49310 x $1E18B40 = $89ACE0E7400

3.2. Calculating the Mean of a Vector

[Before continuing to the following section, it is strongly recommended to read documents “68k_w05_subroutine_stack.pdf”, and “modular.pdf” to learn how to pass parameters to subroutines]

A vector hereby is a one-dimensional array of numbers. The size of the vector is the number of elements. The mean of a vector is the average of its elements.
You are required to write a function (subroutine) that will calculate the mean of a vector. This subroutine will get inputs (address and size of the array) by using some parameter passing methods. The address of the vector points to the start address of the data set. Assume that numbers in data set are 16-bit and size of the vector doesn’t exceed 65535 elements. You can use a data register to pass the vector size, or you can use the stack memory. If you directly define an example data set to a specific memory location in your assembly code, it will be easy to handle each trial.

```
ORG $400800 ; data area
VSIZE DC.W 5 ;
VECT DS.W 5 ; vector length is 5 (You may set VECT[] = {1, 2, 3, 4, 5})

ORG $400400 ; program area
MAIN LEA VSIZE, A0
    MOVE.W (A0), D0 ; pass vector size by value using reg. D0
    LEA VECTOR, A0 ; pass vector addr. by reference using addr. reg. A0
    JSR CALC_MEAN ; call subroutine
    TRAP #11

CALC_MEAN ; IMPLEMENT YOUR CODE HERE
...
RTS
```

As a conclusion of the parameter passing requirements above, it is obvious that you need to develop a function that is capable of handling a vector (array) of any size. You can also develop your program to handle vectors with fixed size (lower performance grade).

Another requirement for the function is that it must return a value according to the number of vectors and size. This return value must be held by a data register. If the size of the vector is entered as zero, the return value will be “-1”.

You will call the subroutines by an upper (parent/main) routine. Modify the sample code above and try to pass parameters to the stack before calling the subroutine. And force your subroutines to be transparent and reentrant (Look at lecture notes week05). Take care of your subroutines to be dynamic, transparent, and reentrant (search these terms). This will help you to be better programmers. For example, the C-version of multiplication may be as following;

```
long* mult1(int &number1, ,int &number2);  (1)
long mult2(int number1, int number2); (2)
```

These can be interpreted as following: two input parameters will be passed into the function (or subroutine). These are pointers (or addresses) of two numbers to be multiplied as in (1). Or we can pass two numbers themselves as input parameters as in (2). The output may be the address of result stored in a memory address or the number itself returned in two data registers or the number passed on the stack. For dynamic codes, passing the input/output parameters in registers or on stack is important.