# Simulation of the dsPIC(2)

## 1 Arithmetics

**Purposes**

- Understand how to reduce computing time and what are the limitations in arithmetical operations in small microcontrollers
- understand how parameters are passed to a function

**arithm.c**

1. Replace, in your current project, the source file by **arithm.c** and compile the project
2. Observe how the different additions and multiplications are carried out
3. For each operation, indicate if it is correct; if it is not, explain the error and the result obtained. Explain the tests (if) based on the "Carry" C and on the "Overflow" OV.
4. Compare time necessary to execute the multiplications are they all correct ?

***Conclusions***

- To spare memory, use the smallest size variable, according to the maximum value of the variable, of course.
- Arithmetical operations work faster with variables of the size of normal operands of the ALU (16 bits here).
- Pay attention to Carry and Overflows
- Do typecasting yourself if you want to be sure of the behaviour of the compiler

**exemple1.c**

5. Replace, in your current project, the file arithm.c by **exemple1.c** and compile the project;
6. Observe the various manners of computing a.b/c: observe the differences in the precision of the result. What are the reasons for those differences ? Show why the multiplication should be done first.
   Compare the computing times of the operations. Which is the longer operation?

**exemple2.c**

7. Replace, in your current project, the file exemple.c by **exemple2.c** ; compile the project;

The great advantage of the **float** variables is that you have got few risk of overflow (check the largest value of a 32-bit floating number), but the price to pay is an increased computing time. This example shows how you can avoid the use of **float** variables to save computing time: this program multiplies f1 and f2 and extract the integer part to send it to an 8-bit digital-to-analog converter. Two methods are shown.

In the second method, f1 and f2 are initially multiplied by a power of 2 (here $2^4$), then rounded and placed in **char** variables. We know that the product is $2^4 * 2^4 = 2^8$ times too large; it is thus necessary to divide it by $2^8$ (i.e. keep the most significant byte)

In the third method, the last division gives a rounded result instead of a truncated on, which is an improvement.

8. Compare, for the 3 methods, the result and the number of clock cycles.

9. Explain how rounding works in the 3rd method

10. What is, in your opinion the greatest disadvantage of the second method ?
    The constants by which f1 and f2 are multiplied are chosen this way
    - we know that f1 will never exceed 15
    - if f1 is multiplied by 16, the result will not exceed 255 and can be placed in an INT8U (**a** in the program)

11. How will have your to change the program if f1 is multiplied by $2^4$ and f2 by $2^6$? Check that the saving of computation time of calculation is still significant in this case.

## 2 Parameter passing

**Passing by value: param1.c**

1. Replace, in your current project the file exemple2.c by **param.c** and compile the project
2. Determine
   - where the variables of main() (a,b,c,…) are stored
   - where are copied, in the main() function, the different parameters of Add3() and Mul2() before those function are called
   - in Add(3) where are stored
     - the local variable a (remark that the name a can be reused since it is local to main() and Add3(), even it does not improve the legibility of the program of Add3()
     - the parameters passed
   - how Add3() and Mul2() return their value

**Passing by reference: param2.c**

1. Replace, in your current project the file param.c by **param2.c** and compile the project In this file, in function1(), an array is passed as a parameter
2. Observe
   - how is this parameter passed :
   - how the array is used in function(1) :
3. swapnum() illustrates how you can pass parameters that are not arrays by reference.
   - What are the meaning of &a and *i ?
   - Explain the mechanism.

## 3 <u>Character strings</u>

A *character strings* is simply an array of char. The array size must be equal to the largest string that it shall store. To take into account the variable size of the string, it its terminated by a *null character (0x00)*. Since a microprocessor can only store binary numbers, letters are coded. The most ancient, polular and compact code is the ASCII code (see \Useful documents for a table). A more recent code is Unicode, that uses 2 bytes/character to represent letters in any language.

Strings are an opportunity to speak about *constants*, since many strings that you shall use in your programs (like prompts and error messages) are constants that you define when you write your source code. Since they do not change, *constants are stored in program memory.*

The dsPIC is a processor with an Harvard architecture. Program memory is in FLASH technology and stores 24-bit instructions extracted by a special instruction bus. Data memory is a static RAM and stores variables of different sizes extracted via a 16-bit data bus. Thus, accessing constants requires a datapath instruction bus to data bus. In the dsPIC this path links the 16 least significant bits of the intruction bus to the data bus. Now what about the addressing mode to access constants. in the dsPIC, the trick is called PSV (Program Space Visibility) and allows pages of the Program Memory to appear in the upper half (starting at 0x8000) of the Data Memory (see dsPIC33 Data Sheet §4.6.3). By default the Page is 0, so that the address 0x000000 of Program Memory is mapped at 0x8000 of Data Memory.

1. Replace, in your current project the file param2.c by **carac.c** and compile the project
2. Put a breakpoint at the first line of code and run the program
3. Open the *Program Memory* window (View->Program memory) and browse the beginning of the code in the various modes that you can choose by clicking on the buttons at the bottom of the window. In particular you can view the Program Memory seen from the Data Memory point of view by the *PSV Mixed* and *PSV Data*.
    - Note that the compiler produces a starting code (corresponding to the reset vector) to initialize lots of things in the processor, clear the Data Memory,….
    - Browse the Program Memeory in mode PSV Data until you find the constant strings "hello" and "world". Note their address in both Program and PSV spaces and length
    - Browse the same addresses in PSV Mixed mode; how is this area seen from the Program point of view ? Why is it so.
4. observe the initialization of the different variables and constants, and the mechanism of a for loop. What is the difference between using a INT8U index and a INT16U index ?