Labo n° 5
Microprocessor Architectures [ELEC-H-473]
dsPIC33: simulation

v1.0.0

Purpose

- Use a simulator and to understand its utility.
- Understand the instruction set and the assembly language, starting from the code produced by a C compiler. See the link between high-level language instructions and assembly/machine code.
- Discover the limitations of small micro-controllers.
- Understand the time spent in basic computations and other limitations.
- Understand how parameters are passed to a function.
- Understand `string` initialisation in the `dsPIC33`.

Useful Documents

A set of useful documents can be found in the network share "Labo/ELEC-H-473/dsPIC33":
- Introduction to MPLAB
- MPLAB C30 C Compiler User's guide
- dsPIC30F/33F Programmer's Reference Manual
- dsPIC33FJXXXGPX06/X08/X10 Data Sheet
- dsPIC30F/33F CPU reference manual

1 Architecture analysis

1.1 CPU architecture

Question 1. Detail the architecture of the CPU used in this microcontroller. Focus on the CPU, not on the peripherals.

Question 2. Explain the difference between a microprocessor and a microcontroller.

2 MPLAB IDE – Simulation workspace

You can configure your environment to suit your needs:
- Launch the MPLAB IDE.
- Open the workspace **"Simul_dsPIC1.mcw"** (File>Open Workspace) which already includes:
  - the choice of the microcontroller.
  - the configuration of the debugger in the *simulation mode* which enables you to execute and debug the code in your development environment, *without* any microcontroller hardware.
- Display the following windows in your IDE
  - C source code
  - Output for note, warning and error messages
  - Disassembly listing (right-click in the window and check "Symbolic Disassembly")
  - Memory Usage: amount of memory used for program and data
  - Stopwatch: number of cycles required to execute each instruction
  - Watch window to observe the registers and variables
  - File Register: to dump the Data Memory
- Save your workspace under another name (File>Save Workspace as) to be able to reuse it any time.

3 Project organisation

Using the MPLAB IDE, you now will compile some code for the `dsPIC33` using the C30 compiler and analyse it.

3.1 Header Files

Notice the presence of the file `p33FJ256GP710.h` in the header files. Double-click on it and read its content.

Question 3. Look at lines related to the I/O port `PORTA`, for example. Explain the utility of this header file.

Question 4. Explain the use of the adjective "volatile" in this header file.

3.2 Types of the variables

Add the source file `initvar.c` to the project and read it.

Question 5. Are the integer types signed or unsigned if you do not specify it explicitly?

Variable types definition ambiguity

The size of certain types of variable can change from one processor to another; for example the default size of the `int` standard can be puzzling because it depends on the size of the data bus (but not always):

- for 8-bit µP it can be 8 bits, but is often 16 bits
- for 16-bit µP it is 16 bits
- for 32-bit µP it can be 16 bits, or 32 bits

Moreover, the fact that a `char` or an `int` is `signed` or `unsigned` by default depends on the compiler and on the options of the compiler. It is always a good practice to improve the *portability* of the code *i.e.* to minimize the modifications when you want to migrate to another processor. For this reason it is wise to gather *everything which depends on the µP* in a few files, so that only those well-identified files have to be modified for another processor.
In this lab, you will use the header file `Datatypes.h` to redefine the types of the variables to avoid any ambiguity.

Question 6. Add `Datatypes.h` in the header files of your project, explain the improvement given by these new definitions.

3.3 Variables initialisation

- Replace the file `initvar.c` by `variables.c`.
- Make the line numbers appear in the C source (right-click on Edit->Properties->C File Type ->line numbers)
- Click on "Build all" to compile and link every file of the project. Ignore the warnings for the moment.

REM : the assembly code begins with the "`lnk`" instruction which will allocate a "stack frame" for the local variables of the function `main()`. Refer to 4.7.4 of the "dsPIC30F/33F Programmer's Reference Manual_70157B.pdf" for the explanation of stack frames and the use of `W15` as a Stack Pointer and `W14` as a Frame Pointer.

Observe in the assembly code the initialization of the different variables; advance in the program using breakpoints and step-by-step mode (both in C and in ASM) and observe the state of the variables in the "Watch" window (trick: add `W14` to the Watch Window); you can change the properties of the Watch Window to observe the values of the variables in different bases.

Question 7. Observe the number of instructions necessary to initialize each variable, use the "Stopwatch" window.

Question 8. The code defines a global variable `glob1`. Where is it defined and initialized?

Question 9. Explain how to place this variable in a register (the grammar can vary depending on the compiler, but the keyword `register` is always used).

3.4 Variable assignations

Replace `variables.c` by `assign.c` and compile.

Question 10. Observe and comment the way variables are accessed compared to `variables.c.`

Question 11. Observe the way assignations work. Explain any unexpected result.

4 Arithmetics

Open the workspace **"Simul_dsPIC2.mcw"** (File>Open Workspace).

4.1 Basics

Replace, in your current project, the source file by `arithm.c` and compile the project.

Question 12. How are the different additions and multiplications carried out?

Question 13. For each operation, indicate if the result 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`.

Question 14. Compare time necessary to execute the multiplications. Are all results correct?

Rules of thumb

- To spare **memory**, use the smallest size variable, according to the maximum value of the variable, of course. It might lead to longer execution time.
- Arithmetical operations are faster with variables of the native size of operands for the ALU (16 bits with the `dsPIC33`).
- Pay attention to Carry and Overflows.
- Do typecasting yourself if you want to be sure of the behaviour of the compiler.

4.1.1 Multiplication, division

Replace the file `arithm.c` by `exemple1.c` and compile the project;

Question 15. Observe the various manners of computing $a \times b/c$: What are the differences? Explain why the multiplication should be done first.

Question 16. Compare the computing time of the operations. Which one is the longest operation?

4.2 Fixed point *vs* Floating point

Replace the file `exemple1.c` by `exemple2.c`; compile the project;

Question 17. Compare, for the 3 methods, the result and the number of clock cycles.

Question 18. Explain how rounding works in the 3rd method.

Question 19. 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 and:
- 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).

Question 20. How would you change the program if `f1` had to be multiplied by 2^4 and `f2` by 2^6? Check that the saving of computation time of calculation is still significant in this case.

Fixed point *vs* Floating point

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 if the CPU does not have a FPU. 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. Three 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 $2^4 \times 2^4 = 2^8$ 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.

5 Parameter passing

5.1 Passing by value

Replace the file `exemple2.c` by `param1.c` and compile the project.

Question 21. Where are stored the variables of `main()` (a, b, c...)?

Question 22. Where are copied, in the `main()` function, the different parameters of `Add3()` and `Mul2()` before those function are called?

Question 23. In `Add3()` where are stored the local variable `a`?

Question 24. In `Add3()` where are stored the parameters passed to `Add3()?`

Question 25. How `Add3()` and `Mul2()` return their results?

5.2 Passing by reference

Replace the file `param.c` by `param2.c` and compile the project.
In this file, in `function1()`, an array is passed as a parameter.

Question 26. How is this parameter passed?

Question 27. How is the array used in `function1()`?

The function `swapnum()` illustrates how you can pass parameters that are not arrays by reference.

Question 28. Explain the mechanism used with `swapnum()`.

6 Character strings

A *character string* 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, popular and compact code is the ASCII code (see Useful documents for a table). A more recent code is Unicode, that may use more than one byte if needed to represent letters in any language[1].

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 `dsPIC33` is a processor with an Harvard architecture[2]. 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

[1]This lab focuses on the classic ASCII code, for further reference about Unicode and the various possibilities, read the official documentation on the Official Unicode website

[2]This is just a short summary, read the CPU documentation for more exhaustive explanations.

constants requires a datapath instruction bus to data bus. In the `dsPIC` this path links the 16 least significant bits of the instruction 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.

- Replace the file `param2.c` by `carac.c` and compile the project.
- Put a breakpoint at the first line of code and run the program.
- 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 Memory 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;

Question 29. How is this area seen from the Program point of view? Why is it so.

Observe the initialization of the different variables and constants, and the mechanism of a for loop.