

## Simulation of the dsPIC(1)

### 1 Purpose

- use a simulator and to understand its utility
- learn the instruction set and the assembly language, starting from the code produced by the compiler. See the link between high-level language instructions and machine code
- be aware of the limitations of small micro-controllers

### 2 Useful Documents

In the folder \Useful Documents

- Introduction to MPLAB
- The Explorer16 board
- MPLAB C30 C Compiler User's guide
- dsPIC30F/33F Programmer's Reference Manual\_70157B.pdf
- dsPIC33FJXXXGPX06/X08/X10 Data Sheet\_70286C.pdf

### 3 Addressing modes

#### Work

1. In the "Programmer's Reference Manual", identify the various addressing modes
2. Give an example of instruction illustrating each addressing mode and instructions using multiple addressing modes, explain how they work.
3. What is the size of an instruction? Is this coherent with the CPU architecture found in the Datasheet?
4. Examine the notion of "format" of the instruction and show by examples that the size of the instruction introduces limitations in the range of each addressing mode or the size of constants.

### 4 Types of the variables

#### Configuration of the environment

- Launch the IDE MPLAB
- Open the workspace (File>Open Workspace) "Simul\_dsPIC1.mcw" 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 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

#### Header Files

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

- Have a look for instance at lines related to the I/O port PORTA and explain the utility of this header file
- Remark and explain the use of the adjective "volatile"

### initvar.c

- Add the source file **initvar.c** to the project if it is not and browse it
- What are the various types of variables accepted by the compiler and their min-max range? (cf MPLAB C30 C Compiler Users guides). Convert the min-max range in decimal.
- Are the integer types signed or unsigned if you do not specify it explicitly?

#### An unambiguous definition of the variable types

The size of certain types of variable can changes 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  $\mu$ P it can be 8 bits, but in the most cases it is 16 bits
- for 16-bit  $\mu$ P it is 16 bits
- for 32-bit  $\mu$ P it can be 16 bits, or 32 bits

Moreover, the fact that a char or an integer 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, all which depends on the  $\mu$ P in a few files*, so that only those well-identified files have to be modified for another processor. Here you will use the header file **Datatypes.h** to redefine the types of the variables to avoid any unambiguity.

- add **Datatypes.h** in the header files of your project, browse its contents and observe the improved legibility of the new definitions

### variables.c

- Replace the file **initvar.c** by **variables.c**
- Make the line numbers appear in the C source (right-click on window->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 "Ink" 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.
- Where are the different variables stored?
- Describe how variables are initialized? What are the difference between the different types and the addressing mode(s)
- Observe the number of instructions necessary to the initialize of each variable thanks to the "Stopwatch" window
- Observe attentively the values of the various variables in the "Watch" window. Comments ?
- Observe attentively the values of the various variables in the "File register" window. In particular see how the INT32U and INT64U are stored and the mental effort required if you want to analyse long variables in memory. Are all variables initialized correctly ?
- Pay attention to the warnings appearing in the "Output" window, are they justified ?
- The code defines a global variable *glob1*. Where is it defined ? Observe how to place this variable in a register (the grammar can vary depending on the compiler, but the keyword *register* is always used).

REM: A global cannot be initialized in its declaration

**variables2.c**

- Replace the file variables.c by **variables2.c**. Observe the warnings and the actual results in the watch window. What is the problem? What is the difference between signed and unsigned variables ?

**assign.c**

- replace variables.c by **assign.c** and compile.
- Observe and comment the way variables are accessed compared to variables.c
- Observe the way assignments work