Register-Level Programming

Introduction

Programming can be considered a set a instructions that are executed in a precise order. A programming simulator can evaluate how instructions store, move and calculate data. It can allow a user to visualize how a specific processor will respond to a program. For example, there is an instruction that can load a user-defined value in a register. When the code is executed, a representation of the value will appear in a register window. A simulator can step through a set of instructions one instruction at a time, in order to more easily follow the operation of a machine.

During your career, it's unlikely that you'll do any register-level programming. So, what's this all about? Why study it with a simulator? The answer: it's a learning tool. Understanding the fine details of programming will allow you to better understand more complex high-level programming languages and more complex hardware.

Objective

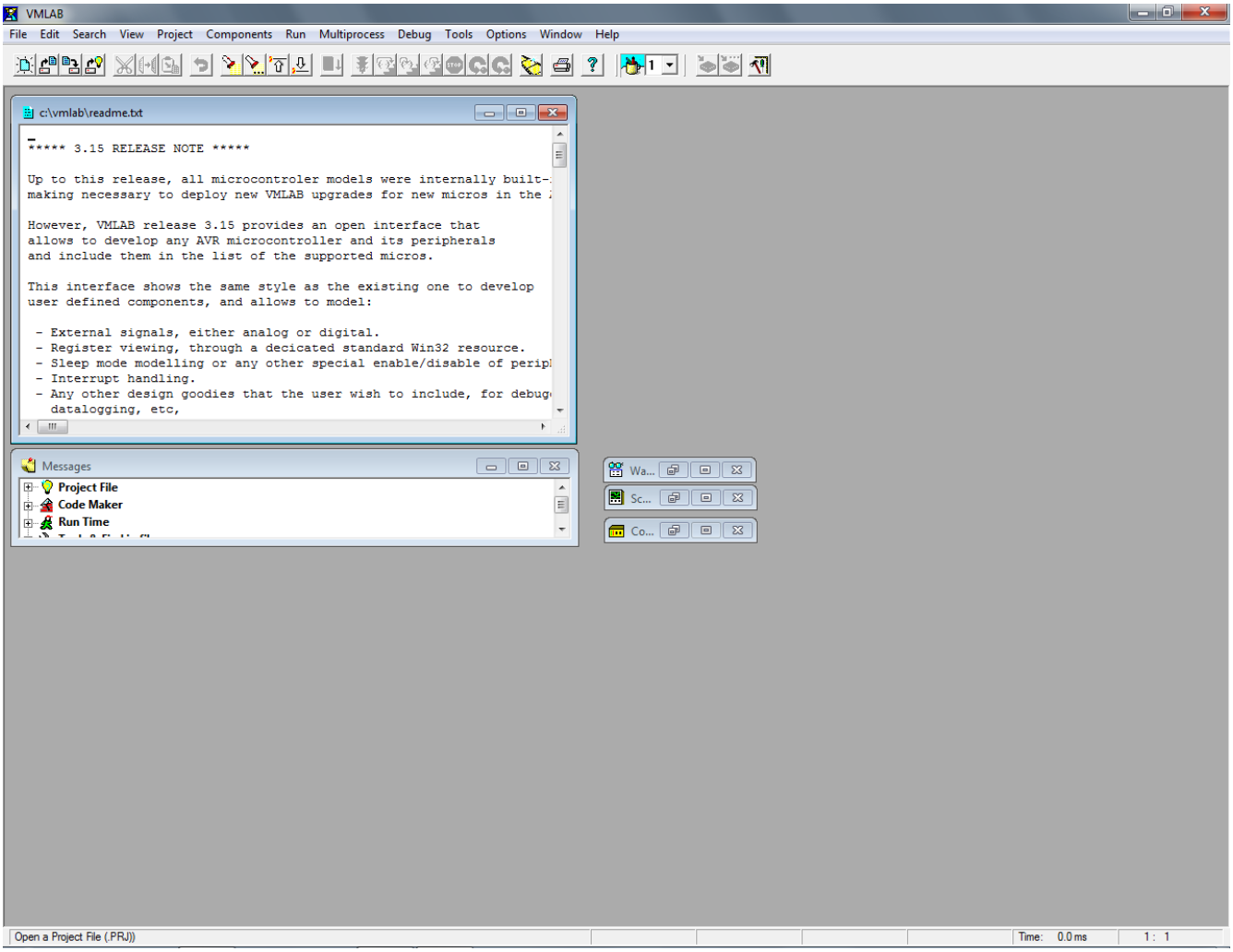
In this lab you will use a simulator called VMLAB (Visual Micro LAB), which simulates some microcontrollers from a company named Atmel. The text you write and modify is an assembly language called AVR. During this lab, keep in mind that the simulator uses a program called an assembler to change the text file into machine code, or the 1's and 0's that a machine understands.

Procedure 1: Getting Started

Open VMLAB in one of the following ways:

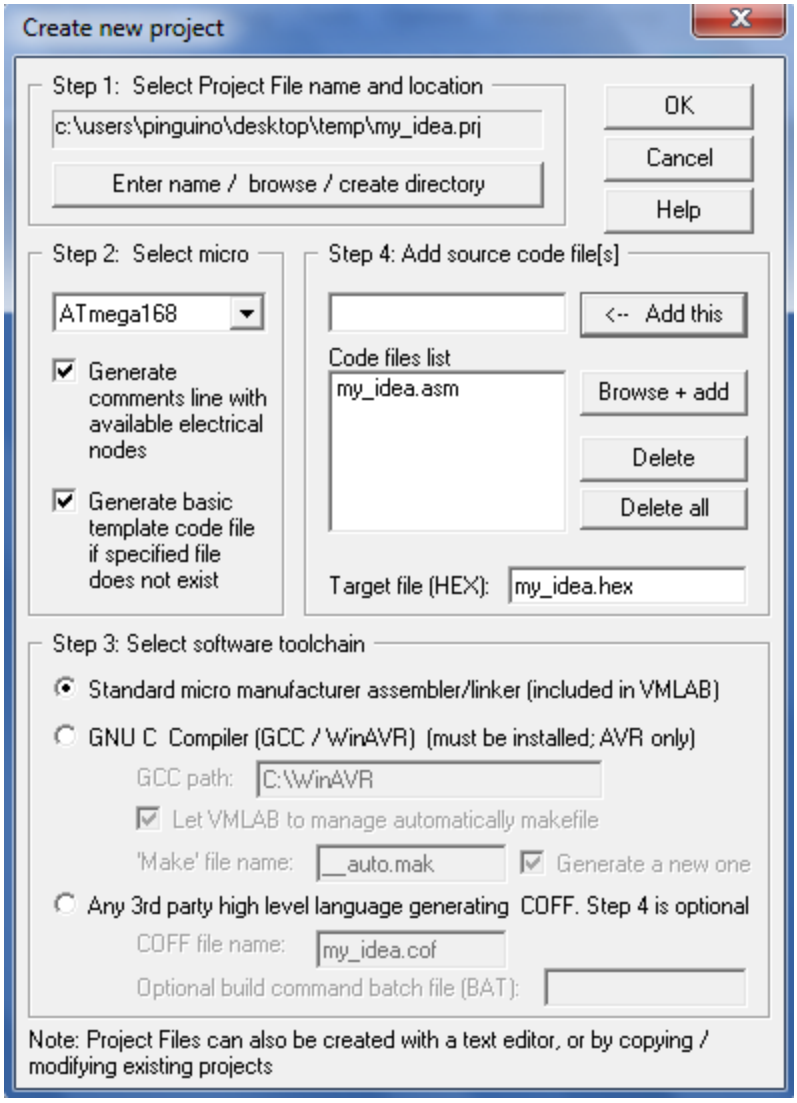
- Download and install VMLAB from the company's website and open it.
- Download the software from the course website, if available, and install and open it.
- If the software is available on the lab computers, open it.

The screen should look something like the following:



Create a new project. In the menu bar at the top of the interface, click “Project” and select “New Project”.

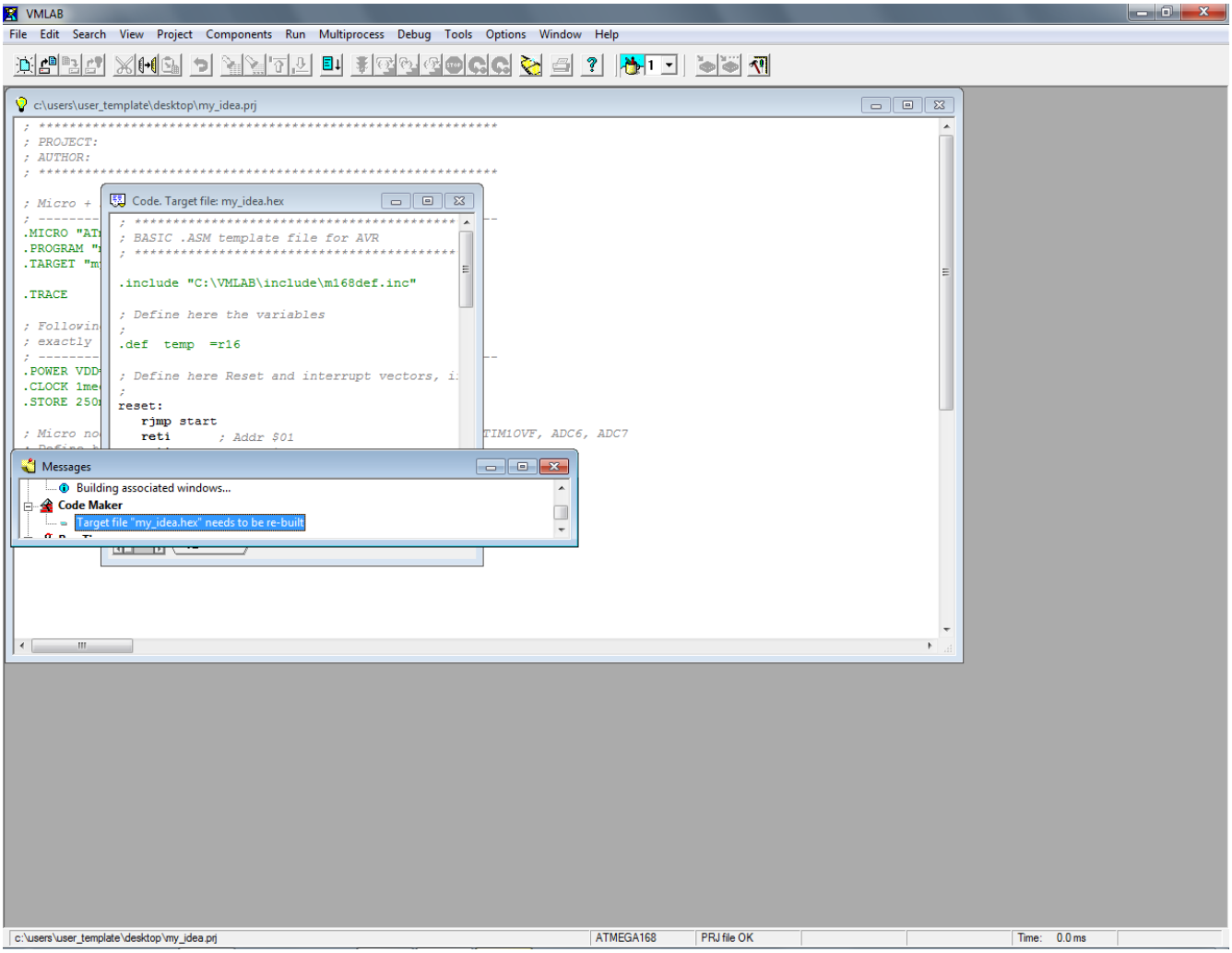
The following menu will open:



Follow the steps listed:

1. Select the file name and location.
2. Change the microcontroller to “ATmega168”.
3. Leave the software toolchain settings unchanged.
4. Click “<-- Add this”.

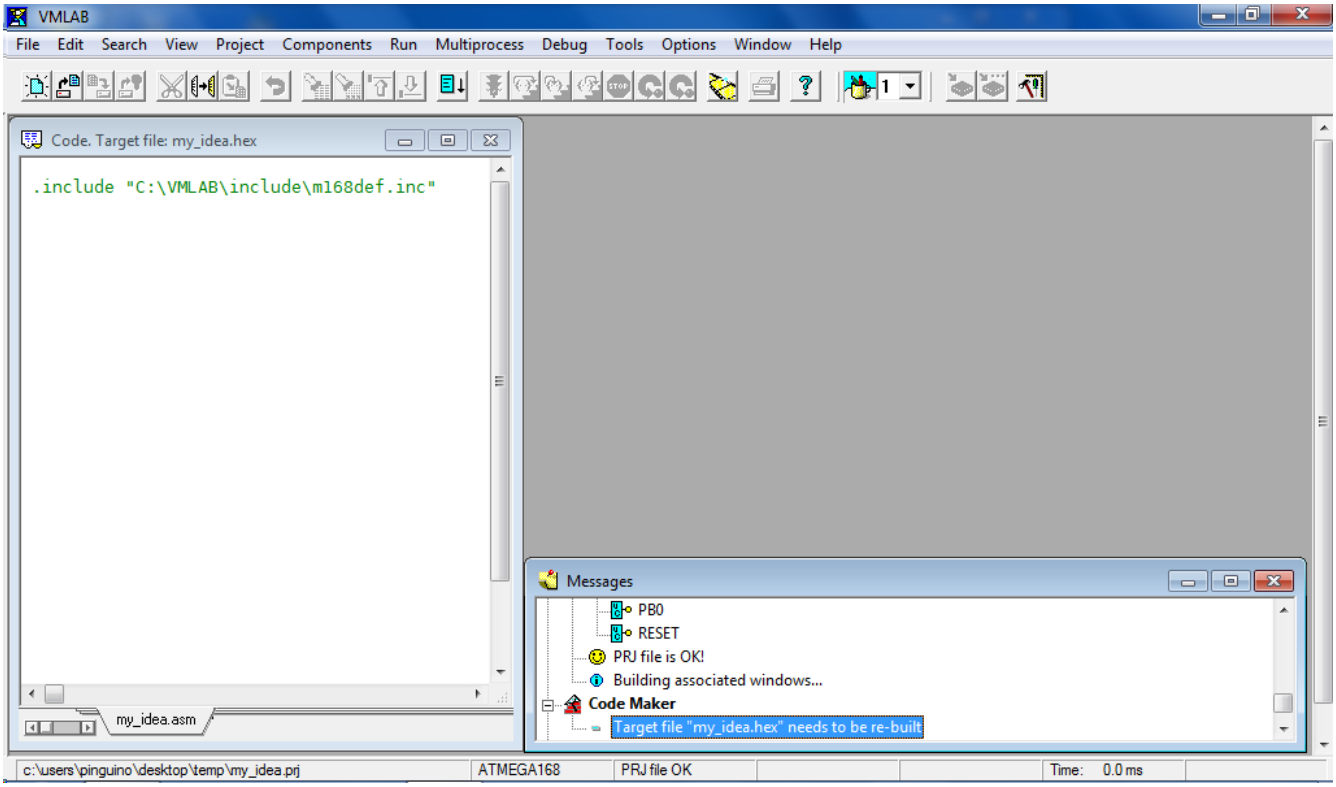
After clicking “OK”, a few windows are shown:



Here is what each window is for:

- A **Project File** window with the extension: “.prj”. This file will not be altered in this lab.
- A “**Messages**” window with feedback to the user. For example, if an error occurs during building a message will appear in this window.
- An **Assembly File** window. This is the text file where AVR code is written. The file type is “.asm”. But the title of this window is “Code. Target file...” The file has a “.hex” extension. This is because the “.asm” file will be assembled into machine code, which has a “.hex” file extension.

Minimize the project file window. Erase everything in the assembly file window except the line that starts with ".include". Move the message window out of the way. The screen should look something like this:



Enter the following line into the assembly file window:

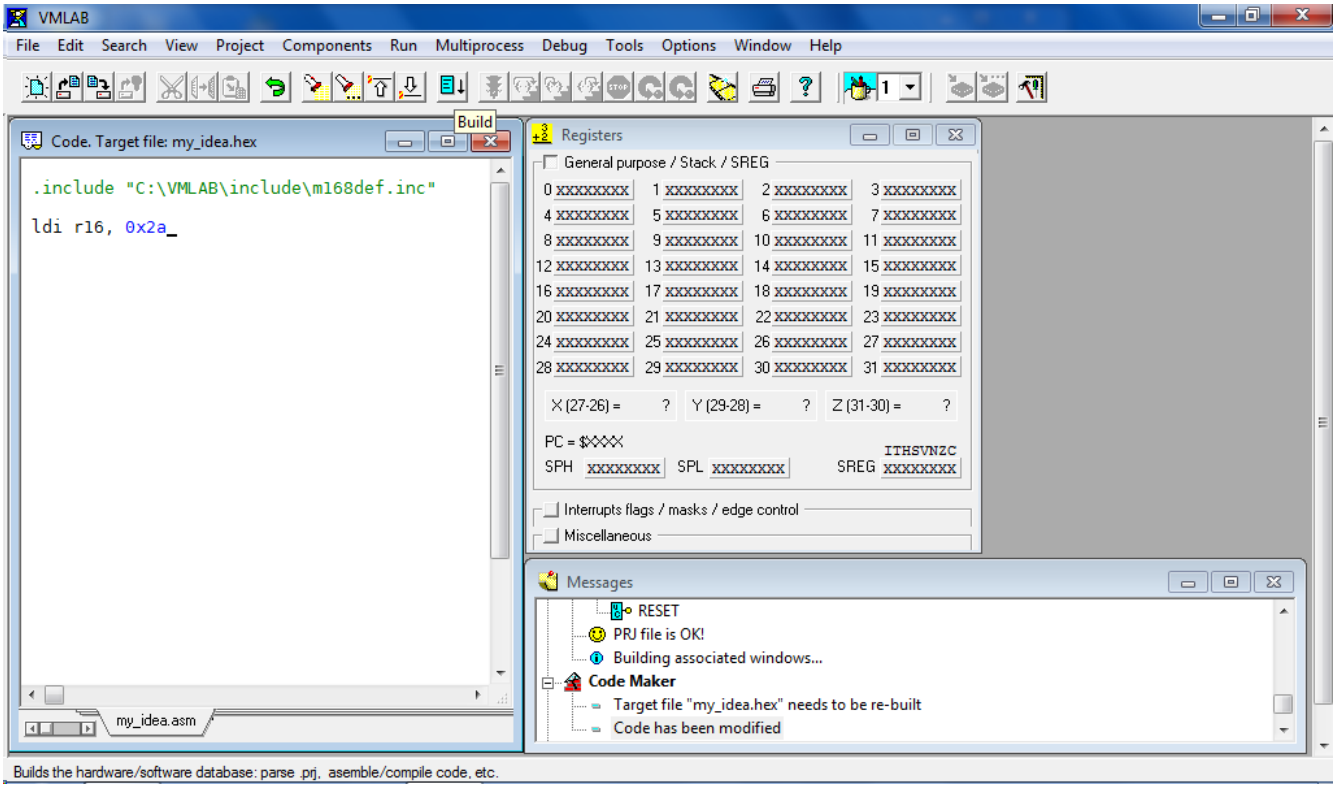
```
ldi r16, 0x2a
```

This instruction, LDI, (that can be referred to as "LoaD Immediate value") loads the value 0x2A into a general purpose register named R16. Note that assembly is not case sensitive.

To open a view of the registers do the following:

From the menu select View->Registers/Flags. Expand the "General Purpose / Stack / SREG" block by clicking on the little square to the left. Move the registers window out of the way.

The screen should look something like this:



Click on the “Build” icon, which looks like a small, light blue file with an arrow next to it, pointing down (shown above). If typed correctly, the message window will show “Success! All ready to run.” Note this step is the assembler taking your assembly file and assembling it into a file that can be loaded onto a microcontroller.

Next, click on the “Go / Continue” icon, which looks like a traffic signal with a green light.

The registers window should show the value in R16. To change the value click on the assembly file window and hit any key. A pop-up message may appear asking to restart the simulation. If so, click “Yes”.

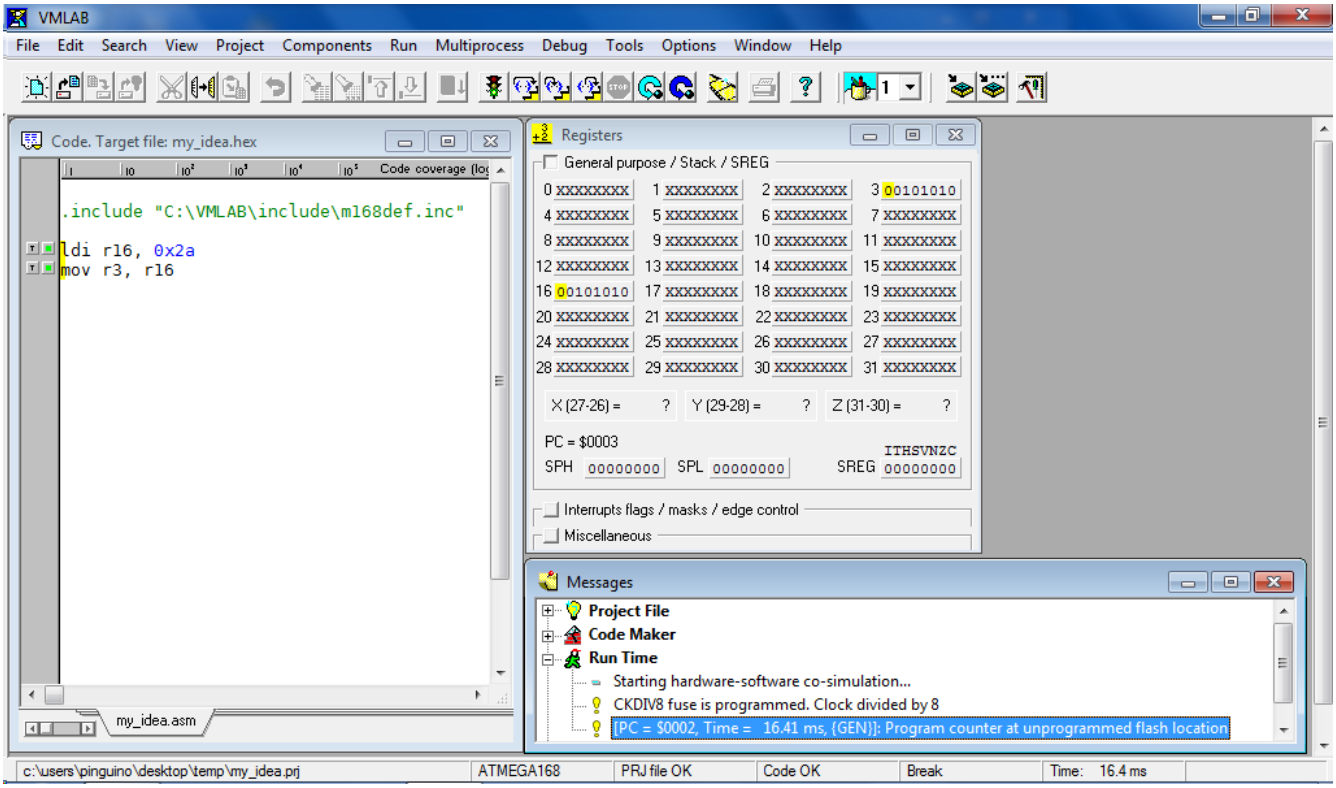
Repeat this process several times with different values and registers to get familiar with building and running. Note: the LDI instruction only works on registers 16 to 31 (R16 – R31).

To move values from one register to another use the following instruction: MOV. For example, type in the following instructions:

```
ldi r16, 0x2a
mov r3, r16
```

Build and run this code.

The screen should look something like this:



To restart the simulation (and change the register display to unknown values) click the “Restart (deep)” icon, which is a dark blue three-quarter arrow. (Tip to understanding each icon: hover over each icon until the help text appears.)

Don't worry if you see the message, “ Program counter at unprogrammed flash location”. This just means all of the code has been executed.

The instruction to clear a register has the following syntax: the mnemonic CLR (for “clear”) followed by the register number. This can be used to clear registers 0 to 31 (R0 – R31). For example:

```
clr r0
```

There is also a set register instruction that sets all the bits in the register to 1. Here is an example of the syntax:

```
ser r16
```

If you try to use the SER instruction on registers 0 to 15, it will not build? Why not? Use the full AVR instruction document to determine why.

To add one to a register use the INC instruction. To subtract one from a register use the DEC instruction. Here are examples:

```
inc r0
dec r0
```

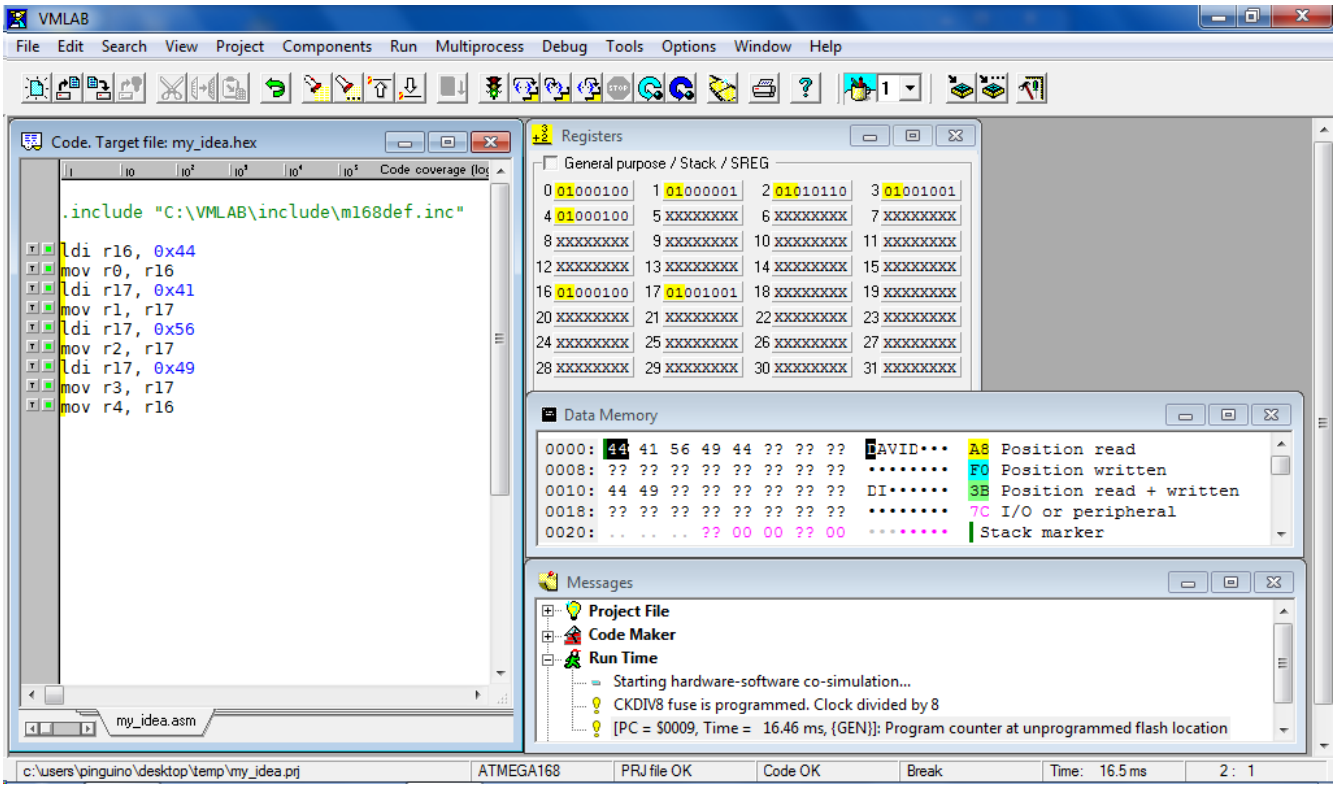
Experiment with these new instructions until you understand how they work and then move on to the next section with an alternate method to view the values of registers.

Procedure 2: Storing Values

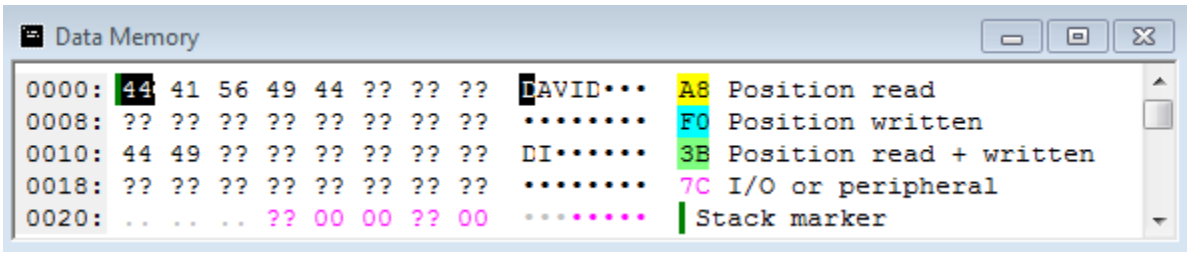
Enter the following code and run it one line at a time.

```
ldi r16, 0x44
mov r0, r16
ldi r17, 0x41
mov r1, r17
ldi r17, 0x56
mov r2, r17
ldi r17, 0x49
mov r3, r17
mov r4, r16
```

To open the data memory: From the menu select View->Data Memory. At this point the screen should look something like this:



Let's take a closer look at the data memory window...



The address of the first value in a row is shown as a four digit hex number on the left side with a gray background. The values (also in hex) are shown in a block to the right of the addresses. Each row has 8 bytes. To the right of the values are the representations in ASCII code. (If the ASCII value is not specified a period is shown.) Finally, to the right of the ASCII code is a key to the color highlighting used in the data memory.

The first address is 0x0000 or 0 (also known as R0). It has a highlighted value of 0x44 in the image above. The next address is 0x0001 and has the value 0x41. On the next address row, the addresses start with 0x0008, address 8 (or R8). Question marks indicate that no values are specified. Skipping to the third row, the address is 0x0010, or 16 (or R16). The first four rows represent the addresses R0 to R31.

Now it's time to write your name into the data memory. Find an ASCII table and look up the hex values that represent the upper and lower case letters in your last name. Then use the AVR instructions covered so far to store your name in the registers starting with the first letter in R0.

Save your ".asm" file. Now you know the basics and can enter and view values. The next section will change the flow of a program.

Procedure 3: Loops and Branches

By default, a program will execute line by line, from top to bottom. One way to change this program order is to use a jump instruction. To be able to jump to a line of code, the location (to jump to) must be labeled.

A label must begin with a letter or an underscore (_). After the first character a label can have letters, digits and underscores. A label must end with a colon (:) character.

To jump a relatively short "distance" (read: to a nearby line) use the RJMP instruction. Enter the following example code:

```
ldi r16, 0x07
loop_1:
     inc r16
     rjmp loop_1
```

This code creates an infinite loop always jumping back a couple of lines to the increment instruction. So, execute the code line-by-line just to get a feel for what it does.

Another method to change the flow of execution is called a branch instruction. A branch tests a condition. If the result of the test is TRUE, the program moves to the label. If the result of the test is FALSE, the program continues to the next line in the program.

Consider the instruction BRNE (branch if not equal). You might reasonably ask, "Branch if not equal to what?" In this case, BRNE will test if the last register changed equals zero. Here is some example code:

```
ldi r16, 0x07
loop_1:
     dec r16
     brne loop_1
```

As you execute this code line-by-line, you will see that the last register before the BRNE test is R16. As long as that register is not zero the program will branch to the label called “loop_1”. Only when this register is zero will the program continue to the next line.

Deliverables:

1. Use the datasheet to determine what the NOP instruction does. When would this instruction be useful?

2. List and summarize the function of each of the AVR commands used in this lab.