Laboratory 2 Assembly Programming and Debugging

Instructor: Dr. Lihong Zhang

1 Objectives

With the completion of this lab, you should be able to:

- Understand the major functionalities available in AVR Studio
- Use AVR Studio to debug your assembly language programs
- Write simple assembly language programs

2 Introduction

In this lab, you will first use the provided correct assembly code to debug instructions including certain I/O operations, in order to gain familiarity with the AVR Studio software. Then you need to debug another program and fix any existing bugs using AVR Studio. You can download the hex file to the target microcontroller to test your program. Finally you will write a simple assembly language program to read the switches and output the reading to the LEDs.

You do not need to submit any comments or code for section 3; you do for sections 4 & 5.

3 Compiling and Debugging Code

In AVR Studio, make a new project and name it *lab2a*, then write the following assembly code.

```
.INCLUDE "M32DEF.INC"

                LDI     R17, LOW(RAMEND)        ; Initialize stack pointer
                OUT     SPL, R17
                LDI     R17, HIGH(RAMEND)
                OUT     SPH, R17

                SER     R17                     ; Set R17
                OUT     DDRD, R17               ; PORTD as output

MAIN:           OUT     PORTD, R17              ; Output R17 to PORTD
                CALL    DELAY                   ; Call DELAY subroutine
                ROL     R17                     ; Rotate leftward R17
                RJMP    MAIN                    ; Relative-jump to MAIN

DELAY:          LDI     R18, 0XFF               ; Delay subroutine
COUNT1:         SER     R19                     ; Set R19
COUNT2:         DEC     R19                     ; Decrement R19
                BRNE    COUNT2                  ; If R19 is not equal to 0, jump to COUNT2
                DEC     R18                     ; Decrement R18
                BRNE    COUNT1                  ; If R18 is not equal to 0, jump to COUNT1
                RET                             ; End of the subroutine
```

Before proceeding further, notice a few important points in the code above. The first line is the header file needed for operating on a specific microcontroller. For any type of microcontrollers, you should import appropriate header file. In the header file, some useful information, such as registers and addresses, is defined

so that the assembler can link your code to the right registers and addresses properly. Make sure you have used the proper header file for your target microcontroller.

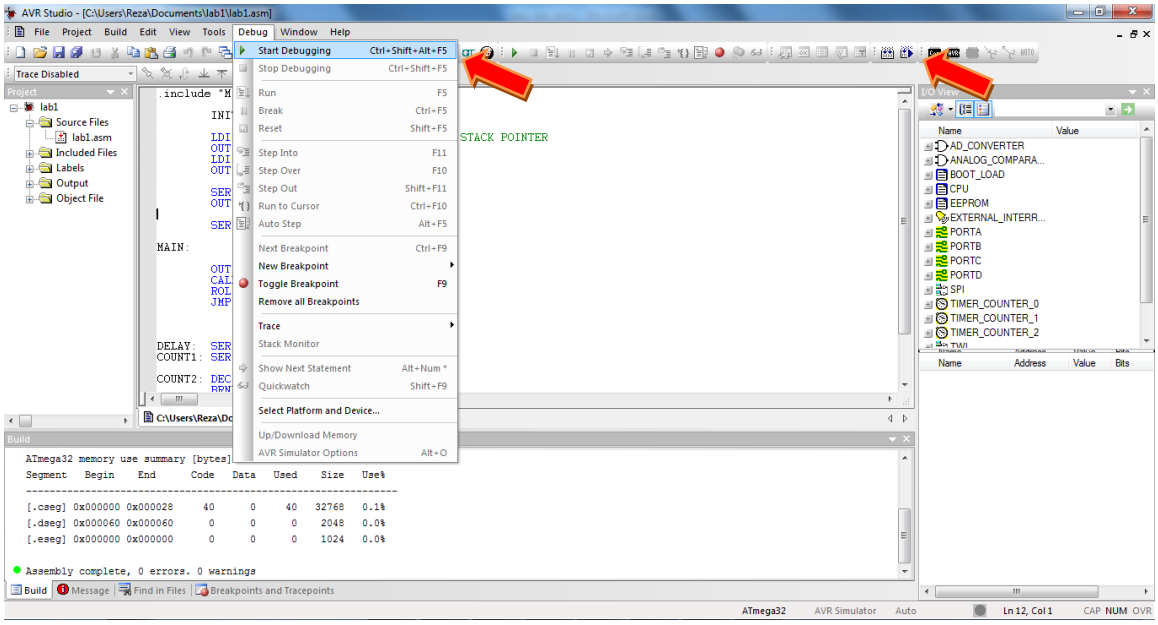
3-1 Building Program

You should now compile the program with the AVR Studio. For this purpose, click on the *Build* icon or on the menu Build -> Build. Make sure your program does not have any syntax errors. If your program does not have any errors, you are ready to debug it.

4-1 Tracing

Gaining experience with debugging is very important; if a program does not work, you need to be able to determine why. By simply looking at an assembly language program, it is often hard to understand what the bugs are. In this part of the lab, you will use several debugging features available to you in AVR Studio.

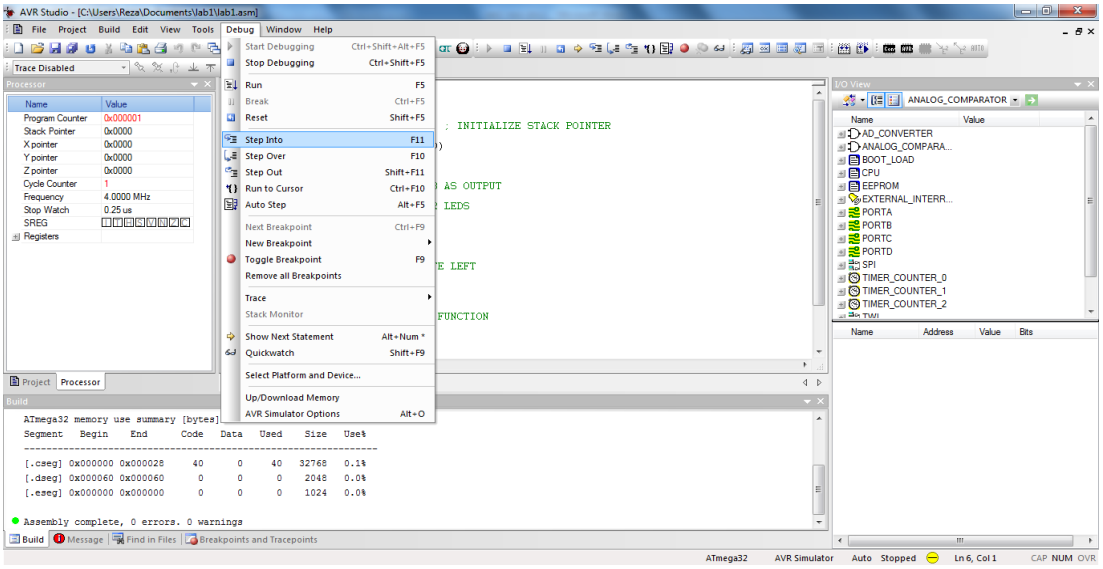
Select ***Start Debugging*** from the ***Debug*** menu or click on the ***Assemble and Run*** icon on the toolbar. A yellow arrow appears next to the first instruction of the program, which shows the next instruction to be executed.



To execute the next instruction, press ***F11*** or select the ***Step Into*** from the ***Debug*** menu. There are also other tracing tools in the ***Debug*** menu, as well:

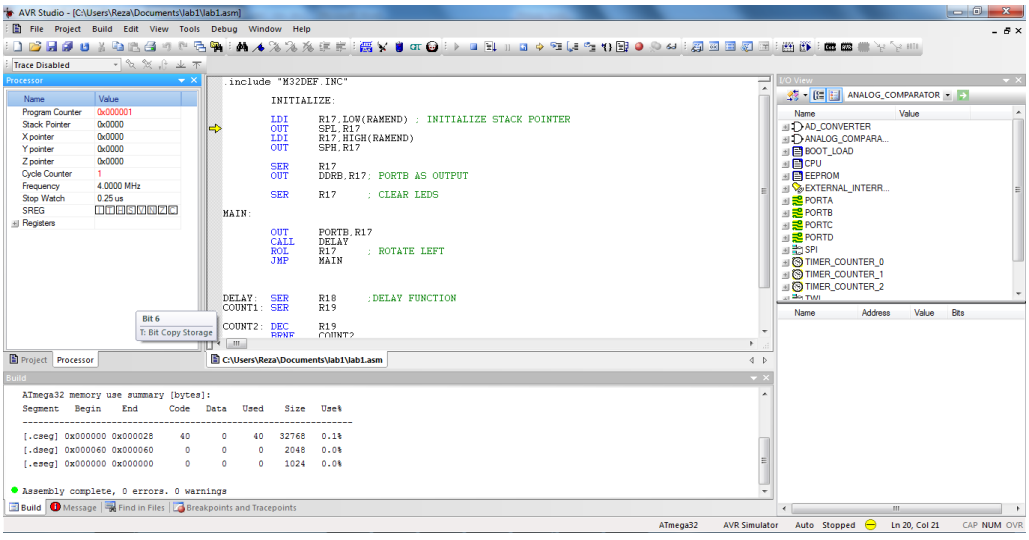
- ***Step Over***: It is executes the next instruction, like ***Step Into***. The only difference between them is that, if the next instruction is a function call, the Step Into, goes to the function; but Step Over executes the function completely and goes to the next instruction.
- ***Step Out***: If you are in a function, Step Out executes the program up to the end of the function.

- ***Breakpoint***: instead of tracing your code instructions one by one, you can use breakpoints and run your code as usual. Program will stop on the breakpoint and at that time you can check the content of the registers and memory to evaluate your program. You can put or remove a breakpoint for each line by clicking on the ***Toggle Breakpoint*** (***F9*** Button) in the ***Debug*** menu or ***Breakpoint*** icon on the toolbar.
- For more information about the Tracing tools, you can check the AVR Studio's help.



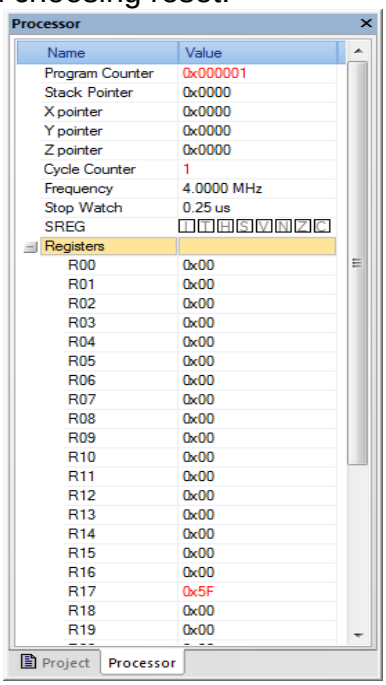
3-3 Watching

By going through the following figures in this part, you will learn how to use the different tools to watch the program.

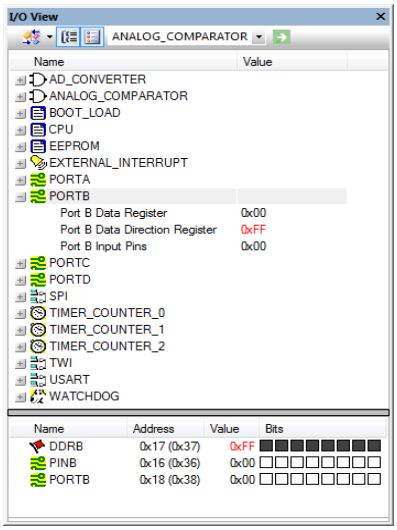


- **Processor:** By default this window is located on the left side. It shows the contents of the registers that are related to the CPU: general purpose

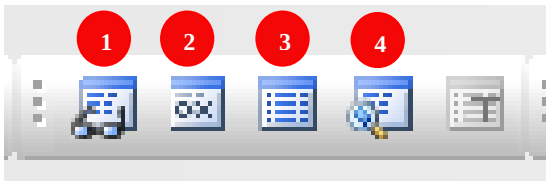
registers (R0 to R31), PC (Program Counter), SP (Stack Pointer), Status Register (SREG), X, Y, and Z registers. Cycle Counter counts the number of machine cycles that have been passed and the Stop Watch represents how much time has elapsed. You can use the parameters to measure the execution time of your program. You can reset them as well, by right clicking on them and choosing reset.



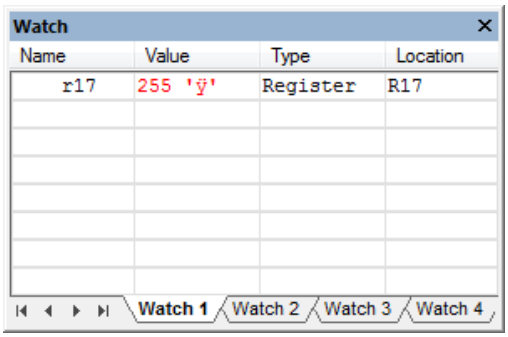
- **I/O View**: By default this window is located on the right side. In this window, you can see the value of the different I/O registers. In the upper box, the related I/O registers are grouped. For instance, click on PORTB and see the values of DDRB, PINB, and PORTB.



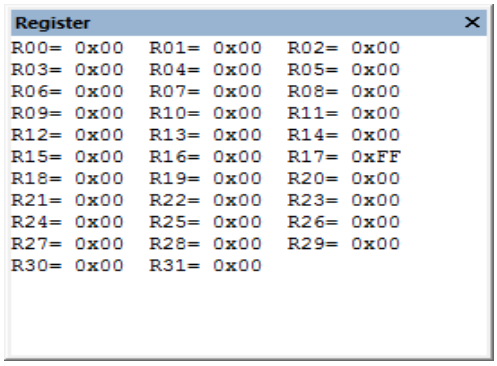
In the figure below, you can see some icons in the toolbar that are numbered as 1 to 4. The use of them is discussed as follows.



1- **Watch**: Click on the tool icon which is numbered as 1. The watch window appears; in this window you can see the value of different variables at the current time. Double click under the ***Name*** title and type R17 and then press ***Enter***; the value of the R17 will be displayed, and if you continue tracing by pressing the F11 button (Step Into), the changes of R17 will be displayed.

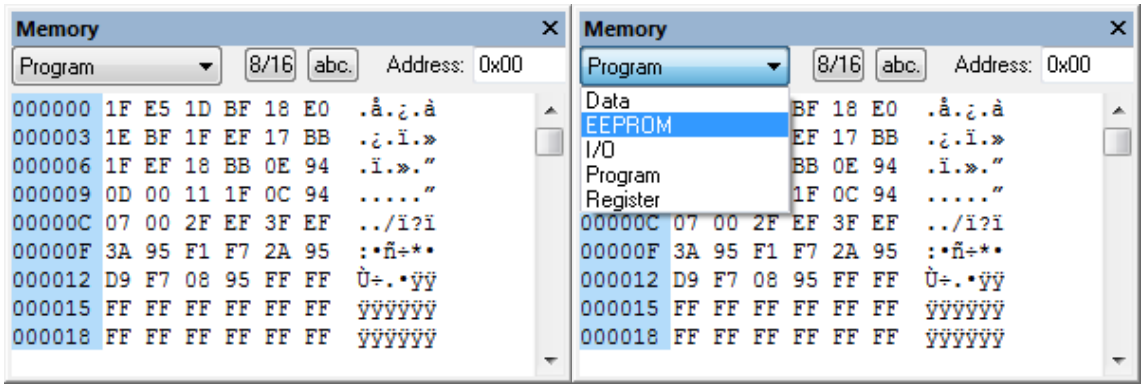


2- **Register**: Click on the tool icon which is numbered as 2, the ***Register*** window will be displayed. This window shows the contents of all of the general purpose registers, at the current time. You can close the window by clicking on the X which is displayed on the top right corner of the window.



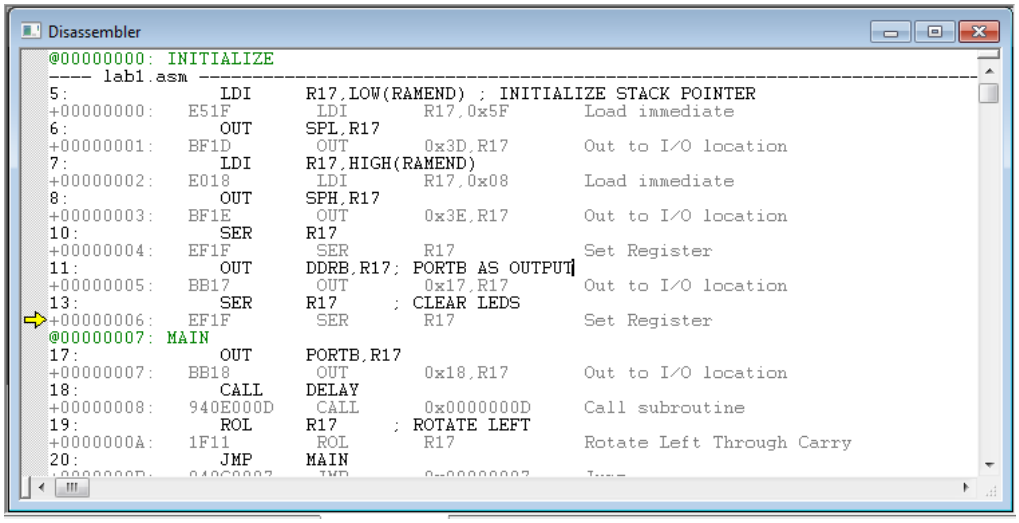
3- ***Memory***: Click on the tool icon which is numbered as 3. The Memory window appears; in this window you can see the contents of different locations of memory, at the current time. In the window:

   a. The blue column shows the address of the first location in each row. For example, in the picture, location $000000 contains $E51F, $000001 contains $BF1D and so on.
   b. We can choose which of the memories to be displayed using the combo box on the top left corner of the window:
      - ***Data***: SRAM memory
      - ***EEPROM***: EEPROM memory
      - ***I/O***: I/O registers
      - ***Program***: Flash memory
      - ***Register***: general purpose registers



4- **Disassembler**: Click on the tool icon which is numbered as 4. This window shows the contents of the flash memory. In the window:

1) The black texts display our program.
2) Below each of the instructions of our program, its assembly language equivalent is displayed. As our program is in assembly language, our instructions and their equivalent are the same.
3) The gray numbers in the middle of each line list the machine code of each instruction. For example, according to following figure, the machine code equivalent of SER R17 is EF1F.
4) The last column describes what the assembly instruction does. For example, as you see in the following figure, LDI is to Load immediate, or RJMP is a Relative Jump.
5) The gray numbers at the beginning of each line describe at which location of flash memory each of the instructions is located. For example, in the following figure, “SER R17” is located in address 0006.
6) The yellow arrow points to the next instruction, which will be executed.



4 Fixing Syntax Errors

Now you have already learnt how to trace your program in order to fix any errors. In this part of the lab you should fix any existing errors in a given program using the methods learnt above.

1- Make a new project and name it lab2b.
2- Use a web browser to log in to your account at D2L (**https://online.mun.ca/)**, and download the file called **lab2b.asm** to your directory.
3- Add the file to the project or copy and paste the code to your project assembly file using a text editor.
4- Build program and check for errors. Note that the **lab2b.asm** file has several errors, which include misspelled, invalid assembly mnemonics, mistaken labels and operands, as well as logic errors.

Question-1: How many syntax errors do you get when you build the project? Mark the errors within the program and discuss how to fix them based on your debugging process.

5 Logic-Error Debugging and Programming

When you finished debugging your code, you can program the compiled hex file to the target microcontroller (i.e., ATmega32) using the instructions you learned in Lab1. For this lab, you should use a 10-wire cable to connect PORTB header and SWITCHES header, and another 10-wire cable to connect PORTD header and LEDS header.

Question-2: Have you fixed all of the errors? Does pressing push-button SW0 or SW7 stop execution? What happens to the LEDs when you press any of the other buttons?

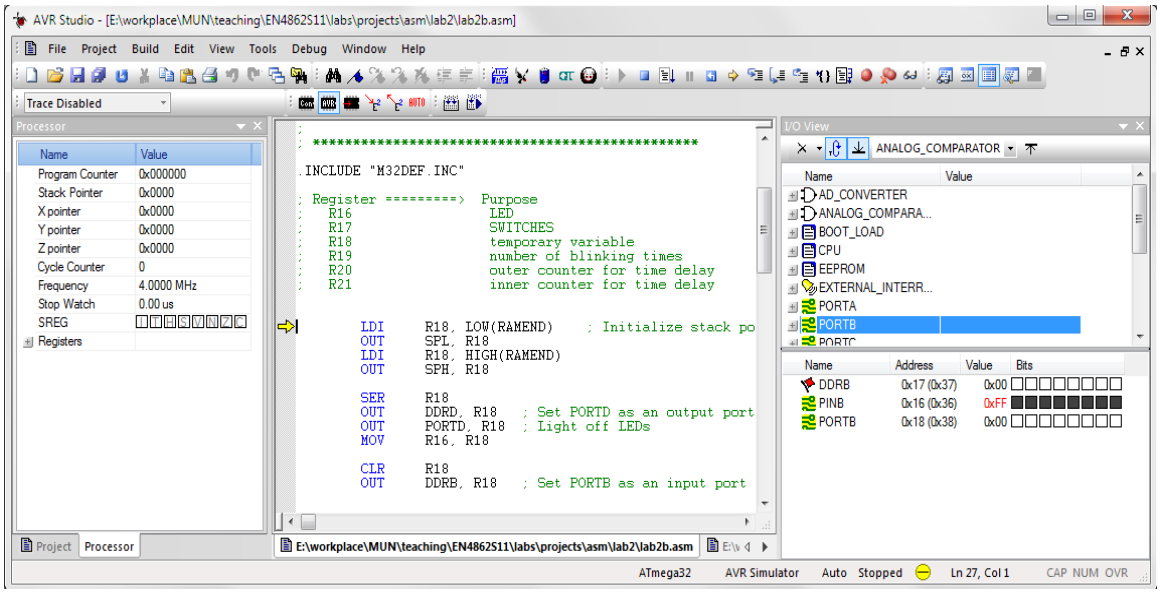
Until you press push-button SW0 or SW7, the program will likely run for a very long time before the LEDs will stop blinking. For some reason, the program is not finishing the loop where the lights blink.

The STK600 does not support on-chip debugging, which means once a program's hex file is downloaded to the target microcontroller, the program just runs from the beginning to the end and we cannot stop the running by setting a breakpoint. Therefore, we have to go back to the AVR Studio and use its simulator to debug the program and find any logic bugs. Then we can download the modified hex file to the STK600 to verify the correctness of our modification.

Within the AVR Studio, set a breakpoint at the following instruction:

```
DEC     R19
```

Click Build -> Build and Run. You can observe that the yellow arrow marker stops at the first instruction of the program (i.e., “LDI R18, LOW(RAMEND)”). On the I/O View Window (normally on the right side of the AVR Studio), click PORTB icon. Then in the bottom sub-window of I/O View, three registers (including DDRB, PINB, and PORTB) are displayed. Mark all the bits of PINB (marked bit as logic 1 and unmarked bit for logic 0) by clicking as shown in the picture below. This is to simulate the input of push-button switches (initially no button is pressed, so all the PINB bits should be marked). Note that if pressing one of the switches, the corresponding SW pin on the SWITCHES header of the STK600 board will be pulled low (i.e., logic 0).



Quesiton-3: Click Debug -> Run, where does the program stop executing? Report the value of register R19? Click Debug -> Run a few more times, and report the results. Does the program seem to be working as expected at this point (in particular, the initial value of R19)? When will the LEDs finally stop blinking?

Based on what you have found out from the breakpoints, fix the ASM file by loading the appropriate register with a value before the line labeled REP2. A value between 20 and 40 is recommended. Assemble and download this file. Run the program again, and check that the program is now running correctly. Test the correctness of the program by pressing both push buttons SW0 and SW7 before and after the LEDs have stopped blinking. Then challenge some classmates to a duel.

Qustion-4: Include the final (correct) version of your code within your lab report. Briefly describe what happens in the game.

Question-5: Comment on the purpose of the following instruction:
EOR R16, R18

Question-6: Write a simple assembly program to read anything from the switches and output the reading to the LEDs on the STK600. You may want to create another project (e.g., lab2c) to hold your program. After programming to the target microcontroller (i.e., ATmega32), demonstrate its correctness to a TA who will sign on your printed version of the code. Make sure you place appropriate comments in your code.

6 Submission

Before the announced lab report due time, submit your answers to the questions above and signed print-outs of the code with your comments on the success of each program. Note that it will be penalized if you fail to obtain a TA's signature.