Lab Assignment 3—Decisions, Decisions, Decisions!
Reading Data and Making Choices

Overview:
In this lab you will write an assembly language program that reads simulated sensor data, copies it to a safe location in memory, performs some comparisons using the copied data, and, depending on the comparison results, writes certain values to memory.

Administrative Information:
The **Atmel 8-bit AVR Instruction Set** document (on Canvas, in Files | Docs) will prove invaluable in selecting the proper types of assembly language instructions to use.

Background—Memory Maps:
As you know, the ReadyAVR board contains an ATmega128A microcontroller. Since it's a micro*controller*, both program memory and data memory are located inside the ATmega128A. But how much memory is there? And how is it partitioned? Manufacturers publish *memory maps* to answer these questions for program designers. For example, the *Data Memory* map for the ATmega128A microcontroller is shown in Figure 1 below. (Since Atmel processors use a modified **Harvard architecture**, a separate memory map would be needed to show how the *program* memory is laid out. But that's a topic for another day…)

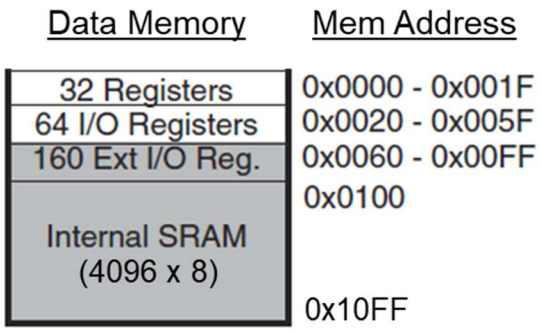

Figure 1. The ATmega128A Data Memory Map

One important point about memory maps—they typically show the low memory addresses at the top of the diagram and the high memory addresses at the bottom. But when we speak about the "top" of memory, we mean the highest address and when we speak about the "bottom" of memory we mean the lowest address. *Could they make it any more confusing?* Anyway, you need to be aware of this.

Atmel microcontrollers (and other brands as well) store their program code and constants in non-volatile FLASH memory, and use SRAM for temporary storage of intermediate values, program variables, and any other data that doesn't need to be saved when power is removed. The ATmega128A data memory map above shows that **Registers** occupy the first 0x0100 bytes, followed by the **Internal SRAM** area starting at address 0x0100 and going through address 0x10FF. This means your programs have 4096 bytes of SRAM to work with in addition to the various register locations.

Typically, the **top end** of SRAM (starting at address 0x10FF and working backwards toward 0x0100) is used for the ***stack***. The stack is a **special data structure** used by the microcontroller

when running programs. Its size isn't fixed—stack memory is allocated dynamically as the demands of the running program change. You'll be looking at the stack in detail in class, but for now, just understand that the stack exists and occupies some number of locations near the top of SRAM. To ensure our program doesn't accidentally overwrite any stack memory values, this lab will only use SRAM locations near the *bottom* of the data memory (near address 0x0100).

Preliminary Setup:

Begin by accepting the Lab03 assignment you received from your instructor. After the remote assignment repo has been created for you, open **Git Shell** and navigate to your local `D:\EECE3624Labs` solution directory. Enter the following command to clone the remote `Lab03` assignment repo to your local storage device.

```
D:\EECE3624Labs> git clone <URL-to-remote-assignment-repo> Lab03
```

Open your `EECE3624Labs` solution in **Atmel Studio**, R-click on the solution name, choose to add an existing project, and select the `Lab03.asmproj` file in the newly cloned local repo. Don't forget to set `Lab03` as the new startup project and `Lab03.asm` the entry file.

The Lab03 project repo contains an `AssemblyProgramTemplate.asm` file. This bare-bones assembly language program can be used as a starting point when writing your own assembly language programs. In fact, **`Lab03.asm` is simply a copy of this template**. Open the `Lab03.asm` file and **update the header information**, including the file name, lab name, author, created date and program description fields, and save everything via **File | Save All**.

Lab Procedure:

Your goal is to read **two 8-bit sensor values** from **SRAM locations 0x0100 and 0x0101**, perform some tests on the sensor values, and **write four test result values to SRAM locations 0x0110 through 0x0113**. In a typical microcontroller application, the sensor data would be placed into memory by some external I/O device, but this lab doesn't involve external I/O devices so you'll **simulate** this behavior by writing the two sensor values to locations 0x0100 and 0x0101 yourself at the start of your program. Then the rest of the program can read and process those values *as if they had been written there by some external I/O device*.

Speaking of simulation, **this week you won't need your actual ReadyAVR board at all**. Instead, you'll use Atmel Studio's **built-in code simulator** to run and debug your program. So be sure neither the ReadyAVR nor the JTAGICE3 is attached to the computer. Instead, configure Atmel Studio for simulation mode by selecting **Project | Lab03 Properties… | Tool** and then picking **Simulator** as the selected debugger/programmer.

> Using a full-featured simulator like that found in Atmel Studio can greatly reduce application development time while ensuring that code runs correctly and efficiently.

To make your code easier to read, you should use *equate* and *define* statements. They not only make your code more readable, but also help organize the code more efficiently. So, add these three statements **just after the header block** (i.e. just before the `.org 0x0000` statement):

```
.equ THRESHOLD = 0x90 ; Create a constant
.def Sensor1   = R20  ; Define a nickname for R20
.def Sensor2   = R21  ; Define a nickname for R21
```

Next, remove the `; your code starts here...` line and replace it with these:

```
;-----------------------------------------------------------------
; Write simulated sensor data into locations 0x0100 and 0x0101
;-----------------------------------------------------------------
; load the simulated sensor values into registers R16 and R17
LDI R16, 0x61        ; simulated sensor 1 value
LDI R17, 0x97        ; simulated sensor 2 value

; initialize 16-bit X-register to point to SRAM location 0x0100
LDI XH, high(0x0100) ; put high byte of 0x0100 into XH
LDI XL, low(0x0100)  ; put low byte of 0x0100 into XL

; store simulated sensor values at SRAM locations 0x0100 and 0x0101
ST X+, R16           ; (0x0100) <- (R16)
ST X,  R17           ; (0x0101) <- (R17)
```

As you can see, the simulated sensor values are first loaded into R16 and R17. Next, the X-register is initialized to ***point*** to the SRAM location where the first sensor value will be stored. Finally, both sensor values (in registers R16 and R17) are written to SRAM using two `ST` instructions. Notice the **post-increment operator** ("+") that **automatically increments the X register** *after* it's used in the first `ST` instruction. This is an important, time-saving feature.

Looking at the code above, answer the following questions in your lab report:

Q1: Register R16 could have easily been replaced with R25, **but not R26**. Explain why not.

Q2: Why is there **no plus sign** after the "X" in the `ST X, R17` instruction?

Coding the Program:

The code above simply puts two values into SRAM at locations 0x0100 and 0x0101, simulating the placing of these values there by an external I/O device. Now you must write the rest of the program. Appendix A contains a flowchart of the program's logic along with some programmer's notes. **There is a lot of great information in those notes and that flowchart, so study both carefully *before* you start writing any code**. And note that the code you write must use the 16-bit **Y-register** as a pointer when reading or writing SRAM memory.

Analysis:

Once your program is working, set a breakpoint at the `ldi R16, HIGH(RAMEND)` instruction. Run the debugger and **begin single-stepping through your code** (shortcut key F10). While debugging, open the "Memory 1" window, choose "**data IRAM**" from the drop-down memory window (see Figure 2 below), and scroll through the addresses until you get to 0x0100. Now you can see what is stored in memory—very handy! Observe that the simulated sensor values (0x61 and 0x97) are properly stored in memory locations 0x0100 and 0x0101, respectively. **If the values are not correct, go back and fix your code before going on**.

Continue single-stepping and watch your program read the sensor values from locations 0x0100 and 0x0101 and store them into registers R20 and R21. Continue single stepping and observe how your program tests the sensor values and, as necessary, writes result values into locations 0x0110 through 0x0113. *Is your program functioning correctly? Were the results what you expected (i.e. did each test properly "pass" or "fail")?* If not, go back and check your code, make the necessary changes, and repeat the test. **Discuss both <u>expected</u> and <u>actual</u> results in your report along with anything interesting about the instructions you chose to use**.

Once your code is operating correctly, take **screen shots** of the "Registers" section of the "Processor Status" window and the "Memory 1" window showing the values in memory locations 0x0100 and 0x0101 (sensor values), along with locations 0x0110 through 0x0113 (final results). Include these screen shots in the body of your lab report, and use the **drawing functions in MS Word** (circles, arrows, textboxes, etc.) to **cleanly and clearly** show the values stored in registers and memory. Also, be sure to **properly label the screenshots** with **figure numbers** and **descriptions**. Here's an example of what I mean (*NB*: results are **<u>not</u>** correct for this lab!):

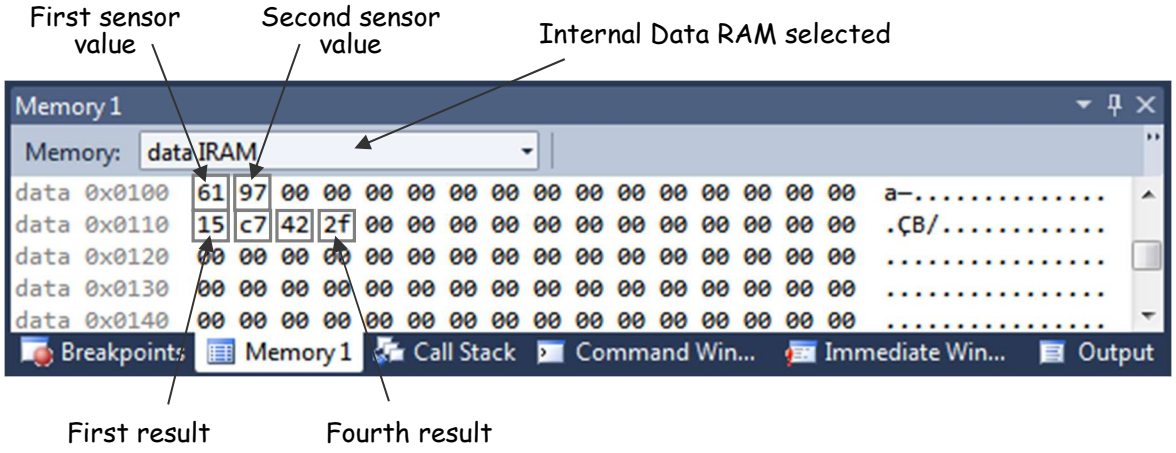

Figure 2. Memory1 window showing sensor values and final results

Report:

This is a **one-week lab**. Write up your report discussing what you learned and answer any questions posed in this handout. Include properly labeled and captioned screenshots, and an appendix containing your <u>sufficiently commented</u> code.

You must submit your lab report via Canvas (in **PDF** format), and your code in this lab's GitHub repo. **<u>Both must be submitted by noon a week after lab is completed</u>**.

Bonus Opportunity: I solved this problem with only **37 lines of executable code** <u>including the code that sets the reset vector, initializes the stack, stores the simulated data, and even the final `nop`!</u> If you solve the problem with fewer lines of code than me, I'll give you some extra points. **<u>To get the bonus, you must explicitly indicate the executable line count in your report</u>**.

Appendix A. Programmer's Notes and Flowchart

Programmer's Notes:

1. The flowchart does **not** show how sensor data gets into SRAM. In practice this is assumed to happen externally. The lab shows you how to simulate this.
2. THRESHOLD has the value of 0x90, and should be treated as signed if comparing with signed values and unsigned if comparing with unsigned values.
3. Use the 16-bit **Y register** as a pointer when reading (`LD` instruction) or writing (`ST` instruction) SRAM.
4. Define and use the symbolic names **Sensor1** and **Sensor2** for registers R20 and R21.
5. Remember that Sensor1 data is **unsigned**, while Sensor2 data is **signed**, so you may need different instructions when making comparisons. (***NB:*** When comparing Sensor1 to Sensor2, treat both as unsigned.)
6. While many instructions can be used to compare data and make program flow decisions, the CP and CPI instructions, along with branch instructions BRGE, BRLT, BRMI, BRPL, BRLO, BRSH, BREQ, and BRNE should be adequate. You'll also find the "Relative Jump" instruction, RJMP, very helpful.
7. When using the Y register as a pointer, take advantage of the assembler's **pre- and post-indexing** capabilities.
8. Place a **`nop`** instruction as the last line of code in your program. This ensures you can single step through the last program instruction without having the debugger close prematurely.