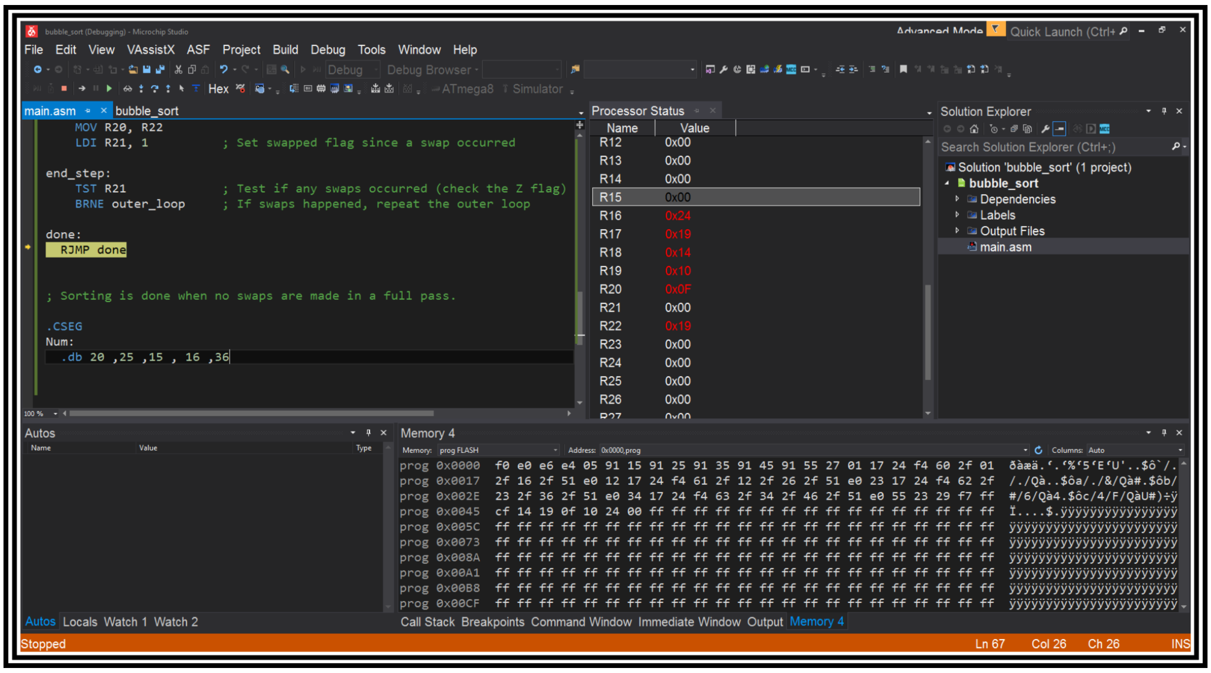


```
    MOV R20, R22
    LDI R21, 1              ; Set swapped flag since a swap occurred

end_step:
    TST R21                 ; Test if any swaps occurred (check the Z flag)
    BRNE outer_loop         ; If swaps happened, repeat the outer loop

done:
  RJMP done


; Sorting is done when no swaps are made in a full pass.

.CSEG
Num:
  .db 20 ,25 ,15 , 16 ,36
```

```
Name   Value
R12    0x00
R13    0x00
R14    0x00
R15    0x00
R16    0x24
R17    0x19
R18    0x14
R19    0x10
R20    0x0F
R21    0x00
R22    0x19
R23    0x00
R24    0x00
R25    0x00
R26    0x00
R27    0x00
```

```
Memory: prog FLASH    Address: 0x0000,prog    Columns: Auto
prog 0x0000  f0 e0 e6 e4 05 91 15 91 25 91 35 91 45 91 55 27 01 17 24 f4 60 2f 01  ðàæä.‘.‘%‘5‘E‘U'..$ô`/.
prog 0x0017  2f 16 2f 51 e0 12 17 24 f4 61 2f 12 2f 26 2f 51 e0 23 17 24 f4 62 2f  /./Qà..$ôa/./&/Qà#.$ôb/
prog 0x002E  23 2f 36 2f 51 e0 34 17 24 f4 63 2f 34 2f 46 2f 51 e0 55 23 29 f7 ff  #/6/Qà4.$ôc/4/F/QàU#)÷ÿ
prog 0x0045  cf 14 19 0f 10 24 00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  Ï....$.ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
prog 0x005C  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
prog 0x0073  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
prog 0x008A  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
prog 0x00A1  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
prog 0x00B8  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
prog 0x00CF  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

bubble_sort (Debugging) - Microchip Studio
File Edit View VAssistX ASF Project Build Debug Tools Window Help
Debug
Debug Browser
Hex
ATmega8
Simulator
main.asm
bubble_sort
Processor Status
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'bubble_sort' (1 project)
bubble_sort
Dependencies
Labels
Output Files
main.asm
Autos
Name
Value
Type
Memory 4
Autos Locals Watch 1 Watch 2
Call Stack Breakpoints Command Window Immediate Window Output Memory 4
Stopped
Ln 67 Col 26 Ch 26 INS

EE2016: Assignment-4

By Siva(EE23B151),
Bhavesh(EE23B017)
and Naveen(EE23B113)

Group 28

Date: 10/9/2024

Indian Institute of Technology Madras

Contents

1 Introduction

1.1 AVR Micro-controllers

AVR microcontrollers, developed by Atmel (now Microchip Technology), are known for their simplicity and efficiency. They are widely used in various applications due to their ease of use and broad support.

1. Architecture

- **RISC-Based:** Uses a Reduced Instruction Set Computing architecture for efficient execution.
- **8-Bit Core:** Primarily 8-bit, with some 16-bit and 32-bit variants.
- **Registers:** 32 general-purpose registers for arithmetic and data handling.

2. Memory

- **Flash Memory:** For program storage, typically ranging from a few kilobytes to hundreds.
- **RAM:** For data storage during execution.
- **EEPROM:** Non-volatile memory for data retention.

3. I/O Ports and Peripherals

- **I/O Ports:** Configurable as input or output for interaction with external devices.
- **Peripherals:** Include timers, ADCs, and support for UART, SPI, and I2C communication.

4. Development Tools

- **Languages:** Commonly programmed in C or assembly.
- **Environments:** Atmel Studio, Arduino IDE.
- **Programming Interfaces:** ISP and JTAG.

5. Popular Models

- **ATmega Series:** Includes ATmega328 and ATmega32.
- **ATTiny Series:** For simpler, low-power applications.

6. Applications and Community

AVR microcontrollers are used in hobby projects, education, and industry. The Arduino platform has fostered a large community and extensive resources.

1.2 Microchip Studio

Microchip Studio, formerly known as Atmel Studio, is an integrated development environment (IDE) used for developing applications with Microchip's microcontrollers and digital signal controllers.

1. Key Features

- **Support for AVR and SAM Devices:** Provides tools and libraries for programming AVR and ARM-based SAM microcontrollers.
- **Code Editor:** Features an advanced code editor with syntax highlighting, IntelliSense, and code navigation.
- **Debugger:** Integrated debugger with support for breakpoints, variable inspection, and step-through execution.
- **Project Management:** Tools for managing project files, dependencies, and build configurations.
- **Peripheral Drivers:** Includes libraries and drivers for accessing and controlling hardware peripherals.

2. Development Tools

- **C/C++ Compiler:** Supports compiling C and C++ code with optimization options.
- **Simulator:** Allows simulation of microcontroller code before hardware deployment.
- **Programmer Integration:** Compatible with various Microchip programmers and debuggers for uploading code to microcontrollers.

3. Getting Started

To get started with Microchip Studio:

- Download and install from the Microchip website.
- Create a new project or open an existing one.
- Write and compile your code using the integrated tools.
- Debug and test your application using the built-in debugger and simulator.

2 Tasks

2.1 Task 1: Finding the MAX & MIN of 10 numbers

Code explanation

1. Setup and Initialization

- **Registers and Constants**:
  - `largest` (r16) and `smallest` (r19) are defined to store the largest and smallest numbers found.
  - `temp` (r17) is used as a temporary storage for the current number being compared.
  - `i` (r18) is the loop counter, starting at `num_count - 1` since the first number is handled separately.
- **Pointer Initialization**:
  - The Z pointer (`ZH:ZL`) is loaded with the address of the list of numbers in program memory.

2. Initial Values

- The first number in the list is loaded into both `largest` and `smallest`, setting the initial values for comparison.

3. Comparison Loop

- The loop (`compare_loop`) processes each remaining number in the list:
  - The next number is loaded into `temp`.
  - This number is compared to `largest`. If it's greater, `largest` is updated.
  - Then, it's compared to `smallest`. If it's smaller, `smallest` is updated.
- The loop counter `i` is decremented after each iteration.
- The loop continues until all numbers have been compared.

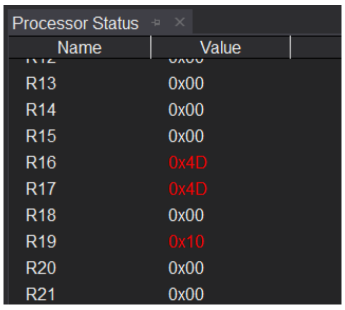
4. End of Program

- After all numbers are processed, the program enters an infinite loop (`done`) to halt execution.

5. AVR Code and Output

AVR Code: contains the '.asm' file for task 1. In the code, we have used the set of numbers for testing as $\{16, 36, 25, 50, 20, 60, 70, 46, 55, 77\}$ which is stored in the flash memory.

The code finds the highest (which is 77) and the lowest (which is 16) and stores them in registers `R16` and `R19` respectively.



Processor Status

Name	Value
R13	0x00
R14	0x00
R15	0x00
R16	0x4D
R17	0x4D
R18	0x00
R19	0x10
R20	0x00
R21	0x00

Figure 1: Largest number in `R16` and Smallest in `R19`

2.2 Task 2: Computing the sum of 10 numbers

1. Setting the Origin

```
.ORG 0x0000
```

This directive sets the starting address of the program in memory to 0x0000.

2. Initialization

```
LDI R16, 0
```

Loads the immediate value 0 into register R16. This register will be used to accumulate the sum of the numbers.

```
LDI ZH, HIGH(numbers << 1)
LDI ZL, LOW(numbers << 1)
```

Loads the high and low bytes of the address of the numbers array into the Z register pair (ZH and ZL). The address is shifted left by 1 bit to match the addressing mode.

```
LDI R17, 10
```

Loads the immediate value 10 into register R17. This value serves as a counter for the number of elements to sum.

3. Summation Loop

```
SUM_LOOP:
    LPM R18, Z+
    ADD R16, R18
    DEC R17
    BRNE SUM_LOOP
```

- **LPM R18, Z+:** Loads the next byte from the address pointed to by Z into R18 and increments Z. This fetches the next number in the sequence.
- **ADD R16, R18:** Adds the value in R18 to R16, accumulating the sum.
- **DEC R17:** Decrements the counter in R17.
- **BRNE SUM_LOOP:** Branches back to `SUM_LOOP` if R17 is not zero, thus continuing the loop until all 10 numbers are processed.

4. Halting Execution

```
NOP
rjmp $
```

- **NOP:** A no-operation instruction that does nothing and is used here to create a placeholder.
- **rjmp $:** An infinite loop that jumps to the current address, effectively halting further execution.

5. Data Segment

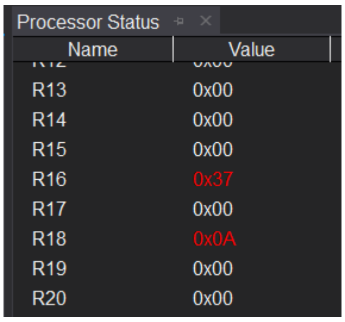
```
.CSEG
numbers:
    .DB 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Defines a constant segment (`.CSEG`) and initializes the numbers array with values 1 through 10. This is the data that will be summed.

6. AVR Code and Output

AVR Code: contains the '.asm' file for task 2. In the code, we have used the set of numbers for testing as $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ which is stored in the flash memory.

The code finds the sum of all the ten numbers and stores it in the register `R16`.



```
Processor Status
Name    Value
R13     0x00
R14     0x00
R15     0x00
R16     0x37
R17     0x00
R18     0x0A
R19     0x00
R20     0x00
```

Figure 2: Sum of 10 numbers in `R16`

2.3 Task 3: Sorting 5 numbers

The code implements a **bubble sort algorithm** to sort the numbers stored in registers `R16` to `R20`, with the **largest** number ending up in `R16` and the **smallest** in `R20`.

Loading Input Data:

The input data is loaded into registers `R16` to `R20` from **flash memory**. This is done using the `LPM` (Load Program Memory) instruction, where the **Z pointer** sequentially moves through the memory, transferring data into the registers.

Sorting Process:

The sorting process involves repeatedly **passing through the registers**, **comparing adjacent pairs**, and **swapping** them if necessary. The code uses an **outer loop** with four key steps (`step_1` to `step_4`), each of which compares adjacent register pairs and either swaps them or moves to the next step. Here's how each step works:

- **Step 1:** Compare `R16` and `R17`. If `R16 < R17`, **swap** them, otherwise **directly move to step 2**.
- **Step 2:** Compare `R17` and `R18`. If `R17 < R18`, **swap** them, otherwise **directly move to step 3**.
- **Step 3:** Compare `R18` and `R19`. If `R18 < R19`, **swap** them, otherwise **directly move to step 4**.
- **Step 4:** Compare `R19` and `R20`. If `R19 < R20`, **swap** them.

After each pass, the next largest value moves toward `R16`, with the largest number eventually ending up in `R16` and the smallest in `R20`.

Completion Condition:

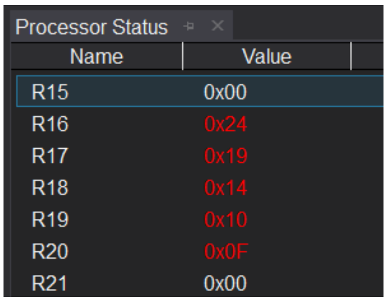
The loop continues until the sorting is complete, as determined by the **flag register** (`R21`). If a swap occurs during any of the steps, the flag is set, and the process repeats. Once a full pass is made with **no swaps** (`R21 = 0`), the outer loop terminates, indicating the registers are fully sorted.

This ensures the sorting process continues until no further swaps are needed, at which point the numbers are ordered from **largest (R16)** to **smallest (R20)**.

AVR Code and Output

Bubble Sort: Contains the '.asm' file for bubble sort. Flash memory inputs are hard-coded as $\{20, 25, 15, 16, 36\}$.

The code sorts it and stores them with `R16` containing 36 (the largest) and `R20` containing 15 (the smallest).



Name	Value
R15	0x00
R16	0x24
R17	0x19
R18	0x14
R19	0x10
R20	0x0F
R21	0x00

Figure 3: Sorted $\{20, 25, 15, 16, 36\}$ as $\{36, 25, 20, 16, 15\}$