EE2016: Microprocessers Lab - Assignment 4 Report

Aadarsh Ramachandiran (EE23B001), Aditya Kartik (EE23B003), Rohita Datta (EE23B065)

September 14, 2024

Abstract

This is a detailed report on an assignment based on programming the AVR ATmega8 micro-controller from Atmel for finding extremes, sum and sorted order of a given set of numbers in the assembly language using the Microchip Studio software. The report contains the details of the solution code, observation and comments on the programming ande debugging experience.

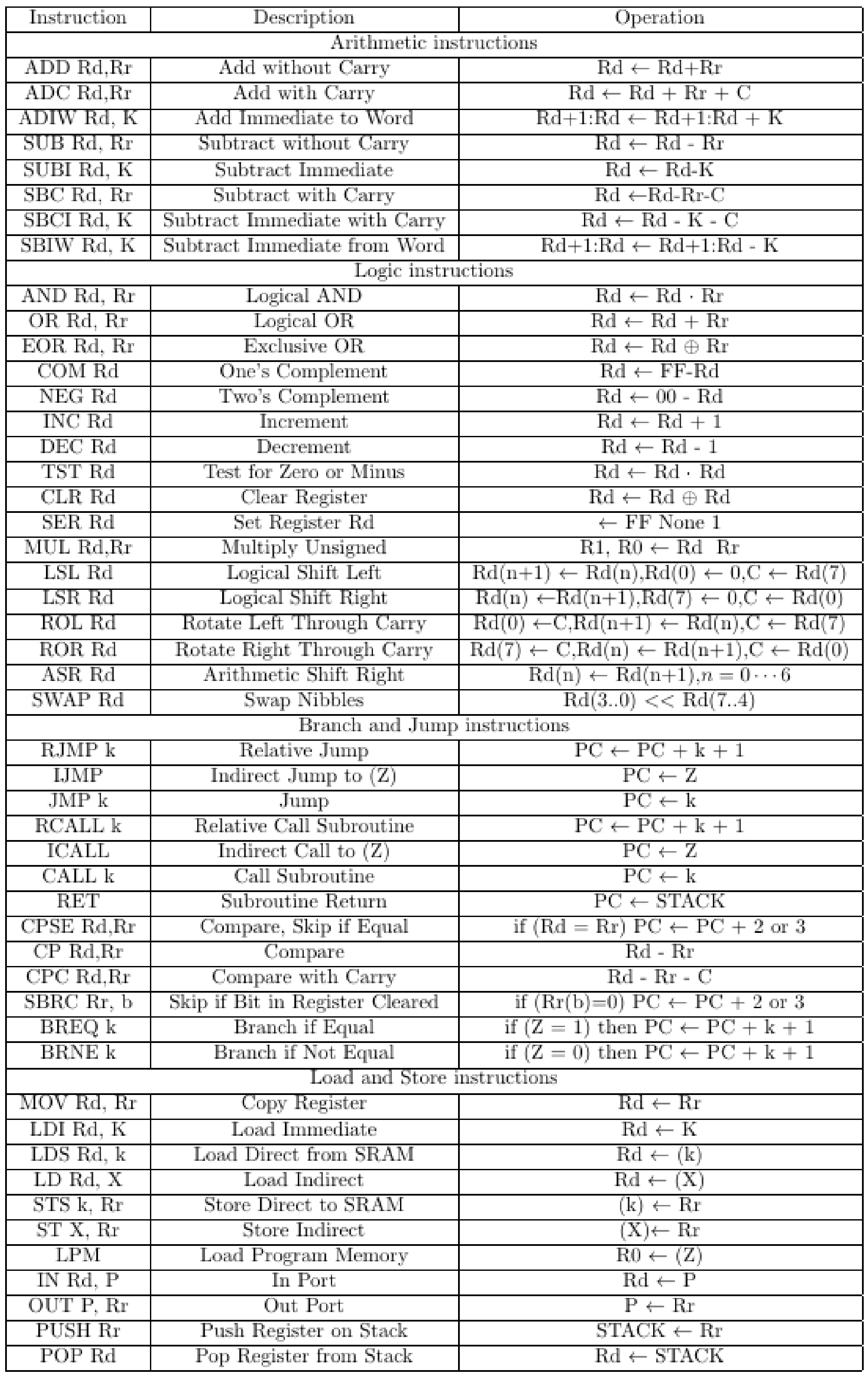
1 Introduction

This assignment involves: (i) writing a program to find the maximum and minimum of 10 numbers stored in flash memory and storing the output in registers suitably, (ii) adding 10 numbers from flash memory and storing the result in a register (Assuming that the register can hold the sum) and (iii) sorting 5 numbers initially stored arbitrarily in flash memory in ascending order and writing the sorted numbers to data memory (SRAM) whose contents are displayed in the memory window.

1.1 The AVR ATmega8

The AVR ATmega8 is an 8-bit Reduced Instruction Set Computer (RISC) single-chip microcontroller developed by Atmel. AVR was one of the first microcontroller families to use on-chip flash memory for program storage. Some of its features include general-purpose I/O ports, internal instruction flash memory, internal SRAM upto 16KB, timers etc. Since flash memory, SRAM and other units are all integrated onto a single chip, many applications can be run without dedicated external memory. Program instructions are stored in flash memory.

There are various instructions that are allowed in AVR assembly ranging from addition, subtraction and comparisons to branching and jumping that facilitates easy implementation of programs in the microcontroller. Some of the common instructions and their format can be found in **Figure 1**.



Instruction	Description	Operation
Arithmetic instructions		
ADD Rd,Rr	Add without Carry	$Rd \leftarrow Rd+Rr$
ADC Rd,Rr	Add with Carry	$Rd \leftarrow Rd + Rr + C$
ADIW Rd, K	Add Immediate to Word	$Rd+1:Rd \leftarrow Rd+1:Rd + K$
SUB Rd, Rr	Subtract without Carry	$Rd \leftarrow Rd - Rr$
SUBI Rd, K	Subtract Immediate	$Rd \leftarrow Rd-K$
SBC Rd, Rr	Subtract with Carry	$Rd \leftarrow Rd-Rr-C$
SBCI Rd, K	Subtract Immediate with Carry	$Rd \leftarrow Rd - K - C$
SBIW Rd, K	Subtract Immediate from Word	$Rd+1:Rd \leftarrow Rd+1:Rd - K$
Logic instructions		
AND Rd, Rr	Logical AND	$Rd \leftarrow Rd \cdot Rr$
OR Rd, Rr	Logical OR	$Rd \leftarrow Rd + Rr$
EOR Rd, Rr	Exclusive OR	$Rd \leftarrow Rd \oplus Rr$
COM Rd	One's Complement	$Rd \leftarrow FF-Rd$
NEG Rd	Two's Complement	$Rd \leftarrow 00 - Rd$
INC Rd	Increment	$Rd \leftarrow Rd + 1$
DEC Rd	Decrement	$Rd \leftarrow Rd - 1$
TST Rd	Test for Zero or Minus	$Rd \leftarrow Rd \cdot Rd$
CLR Rd	Clear Register	$Rd \leftarrow Rd \oplus Rd$
SER Rd	Set Register Rd	$\leftarrow$ FF None 1
MUL Rd,Rr	Multiply Unsigned	$R1, R0 \leftarrow Rd\ \ Rr$
LSL Rd	Logical Shift Left	$Rd(n+1) \leftarrow Rd(n), Rd(0) \leftarrow 0, C \leftarrow Rd(7)$
LSR Rd	Logical Shift Right	$Rd(n) \leftarrow Rd(n+1), Rd(7) \leftarrow 0, C \leftarrow Rd(0)$
ROL Rd	Rotate Left Through Carry	$Rd(0) \leftarrow C, Rd(n+1) \leftarrow Rd(n), C \leftarrow Rd(7)$
ROR Rd	Rotate Right Through Carry	$Rd(7) \leftarrow C, Rd(n) \leftarrow Rd(n+1), C \leftarrow Rd(0)$
ASR Rd	Arithmetic Shift Right	$Rd(n) \leftarrow Rd(n+1), n = 0 \cdots 6$
SWAP Rd	Swap Nibbles	$Rd(3..0) << Rd(7..4)$
Branch and Jump instructions		
RJMP k	Relative Jump	$PC \leftarrow PC + k + 1$
IJMP	Indirect Jump to (Z)	$PC \leftarrow Z$
JMP k	Jump	$PC \leftarrow k$
RCALL k	Relative Call Subroutine	$PC \leftarrow PC + k + 1$
ICALL	Indirect Call to (Z)	$PC \leftarrow Z$
CALL k	Call Subroutine	$PC \leftarrow k$
RET	Subroutine Return	$PC \leftarrow STACK$
CPSE Rd,Rr	Compare, Skip if Equal	if $(Rd = Rr)$ $PC \leftarrow PC + 2$ or 3
CP Rd,Rr	Compare	$Rd - Rr$
CPC Rd,Rr	Compare with Carry	$Rd - Rr - C$
SBRC Rr, b	Skip if Bit in Register Cleared	if $(Rr(b)=0)$ $PC \leftarrow PC + 2$ or 3
BREQ k	Branch if Equal	if $(Z = 1)$ then $PC \leftarrow PC + k + 1$
BRNE k	Branch if Not Equal	if $(Z = 0)$ then $PC \leftarrow PC + k + 1$
Load and Store instructions		
MOV Rd, Rr	Copy Register	$Rd \leftarrow Rr$
LDI Rd, K	Load Immediate	$Rd \leftarrow K$
LDS Rd, k	Load Direct from SRAM	$Rd \leftarrow (k)$
LD Rd, X	Load Indirect	$Rd \leftarrow (X)$
STS k, Rr	Store Direct to SRAM	$(k) \leftarrow Rr$
ST X, Rr	Store Indirect	$(X) \leftarrow Rr$
LPM	Load Program Memory	$R0 \leftarrow (Z)$
IN Rd, P	In Port	$Rd \leftarrow P$
OUT P, Rr	Out Port	$P \leftarrow Rr$
PUSH Rr	Push Register on Stack	$STACK \leftarrow Rr$
POP Rd	Pop Register from Stack	$Rd \leftarrow STACK$

Table 1: Instructions and Their Format

Figure 1: Common instructions and formats

2 Design

2.1 Finding maximum and minimum of 10 numbers

2.1.1 Description

The goal for this task in the assignment is to find the maximum and minimum of 10 numbers which are hard-coded in the flash memory.

Explanation for the commands used:
.cseg : The definition of the segment
.org 0x00 code segment 0x00 tells the assembler to start the code at the address 0x00.
The **label** 'numbers' points to the location in the flash memory where the ten numbers are stored.
.db tells the assembler to define bytes (10 bytes in this case) for storage in the flash memory.
ldi r30, low(numbers) and ldi r31, high(numbers) does immediate load of the address pointed to by the label 'numbers' into the **Z pointer**(associated with r30 and r31)

We start by loading the first number into r18, incrementing Z pointer, and initialising maximum and minimum to this first number which is stored in r20 and r19 respectively. We decrement loop counter and continue loading, comparing the loaded number in r18 with r20 (which holds the current minimum) and r19 (which holds the current maximum) and replacing the corresponding values appropriately. The comparisons and appropriate actions are implemented using the concept of conditional branching, wherein, we use the status register flags and code labels to navigate across the program based on different conditions. We loop until the counter becomes 0, that is, when all the numbers have been loaded and compared with. The following subsection shows the code with comments written on every line to improve the understanding of its working.

2.1.2 Code

Finding Maximum and Minimum of 10 given numbers:

```
; Program to find maximum and minimum of 10 numbers stored in a flash
   and store the resulting output in some registors

; Code segment
.cseg
; Start address for the 10 numbers
.org 0x00
; Store in flash memory
numbers: .db 5, 12, 7, 9, 18, 4, 15, 11, 8, 6 ; .db tells directive to
    define byte. Used for storage

; Initialize Z pointer to number
ldi r30, low(numbers) ; Load the low byte of the number's address into
    ZL
ldi r31, high(numbers) ; Load the high byte of the number's address
   into ZH
; We use two registers, low() and high() to allow storage of 16-bit
   addresses in our 8-bit registers
; Z is like a pointer to the program memory

; We use r18 for storage of all accessed numbers, r19 for storage of
   the maximum and r20 for storage of the minimum
lpm r18, Z+ ; Load the first number into r18 registor and increment Z
   pointer
mov r19, r18 ; Set the initial maxmimum as the first number
mov r20, r18 ; Same with minimum
```

```
ldi r21, 9 ; We store the loop counter in r20 to fetch sequentially
   all 10 numbers. Initialize with 9 since 1 number already fetched.

LOOP:
    lpm r18, Z+ ; Load next number
    cp r18, r19 ; Compare with current maximum and modify carry flag
       in status register
    brlo SKIP_MAX ; If r18 < r19, then don't update maximum. Go to
       label SKIP_MAX (conditional branching)
    mov r19, r18 ; If no skip, then r18>r19. Update maximum in r19

SKIP_MAX:
    cp r18, r20 ; Compare for minimum
    brsh SKIP_MIN ; Branch if same or higher
    mov r20, r18 ; Update minimum in r20

SKIP_MIN:
    dec r21 ; Decrement loop counter and modify the zero flag in
       status register
    brne LOOP ; If counter != 0 continue the loop (conditional
       branching)

nop ; No operation

; r19 holds the maximum value and r20 holds the minimum
```

2.2 Addition of 10 numbers

2.2.1 Description

The goal for this task in the assignment is to add 10 numbers which are hard-coded in the flash memory and store the resultant sum in a register. We begin by loading the address of 'numbers' into the Z pointer, initialising the sum value to 0 (which is stored in r16) and initialize a loop counter to 10 (in r17) for loading all the numbers from flash memory.
As each number is loaded (into r18) using the Z pointer, we sum with the value of r16 and store the resultant back in r16. The following subsection demonstrates the code for the program.
Note: We assume that the resultant sum is small enough to be stored in a single register. If not, we must use another register and handle the carry-over's appropriately.

2.2.2 Code

```
; Assembly code to add 10 numbers stored in flash memory and store the
    resultant output in a register. The numbers are assumed to be
   small enough that the final output is small enough to be stored in
   a regsiter

.cseg
.org 0x00

numbers: .db 1, 2, 3, 2, 1, 3, 1, 2, 1, 4

ldi r30, low(numbers)
ldi r31, high(numbers)

ldi r16, 0 ; Clear r16. It will hold the sum
ldi r17, 10 ; Loop counter
```

```
LOOP:
    lpm r18, Z+ ; Load a number
    add r16, r18 ; Add number to r16 (cumulative sum so far) and store
        final sum back in r16
    dec r17 ; Decrement loop counter
    brne LOOP ; Continue loop if counter != 0

; r16 holds the sum of the 10 numbers

nop ; No operation
```

2.3 Sorting of 5 numbers

2.3.1 Description

The goal for this task in the assignment is to sort five numbers which are hard-coded in the flash memory with the program and output the sorted numbers suitably. In order to make efficient use of the limited general purpose registers available and to allow for scalability to sort more numbers if needed, we decide to store and fetch the numbers in the Static RAM using appropriate pointer registers. We use the Z pointer to point to the initialized numbers in the flash memory for loading, and Y pointer to point to the addresses in the SRAM which we use to work with. The data memory starts from the address of 0x0060, hence, we decide to use five addresses for our five numbers starting from 0x0100 which the Y pointer is initialized to point to. The choice of 0x0100 is arbitrary and any address beyond 0x060 and within the capacity of the SRAM may be used.
We first load the numbers from the flash memory onto the SRAM in the initialised order. We, then, perform a sorting algorithm called **Bubble Sort** to sort the numbers in SRAM. The logic behind bubble sort is simple and is as follows.

```
1. Iterate through the list of numbers one by one.
2. In each iteration, compare the current and the subsequent number.
3. If the subsequent number is smaller than the current one, swap the two numbers.
   Else, do nothing.
4. Repeat the iteration n-1 times where n is the total number of numbers.
```

The core idea is to keep swapping adjacent numbers until all of them get sorted. While, this is not a very efficient algorithm in terms of time complexity, we believe it is one of the simplest ones for a hardware implementation. We use appropriate number of general purpose registers for the counter variables for iteration and temporary storage of the numbers concerned at any step. Conditional branching facilitates logic flow between different parts of the code.

2.3.2 Code

Sorting 5 numbers:

```
; Assembly program to sort 5 numbers and store the final output in
   data memory
; We shall use bubble sort for our purposes

.cseg
.org 0x00
numbers:
        .db 1, 2, 4, 3, 5

; Initialize Z pointer
ldi r30, low(numbers)
ldi r31, high(numbers)
ldi r20, 0 ; Counter for loading numbers into SRAM / and for outer
   loop in bubble sort
```

```
ldi r22, 0 ; Inner loop counter for the bubble sort algorithm
; Initialize Y pointer
ldi r28, 0x00
ldi r29, 0x01 ; SRAM data address starts from 0x0060. Any address in
   use should be >

LOAD_NUMBERS:
    lpm r21, Z+ ; Load a number
    st Y+, r21 ; Store number at location using (store indirect -
       using pointers)
    inc r20
    cpi r20, 5
    brlo LOAD_NUMBERS ; Keep loading until 5 numbers are fetched
    ldi r20, 0 ; Reset register with value 0

; Bubble sort
OUTER_LOOP:
    ldi r22, 0 ; Reset inner loop index to 0
    ; Reset Y pointer
    ldi r28, 0x00
    ldi r29, 0x01
INNER_LOOP:
    ld r24, Y ; Load number at index r22
    inc r28
    ld r25, Y ; Load next immediate number
    cp r24, r25 ; Compare
    brlo NO_SWAP ; If number less than next immediate- no swap
    ; Else swap
    ; Use r26 as temporary registor for swapping
    mov r26, r24
    mov r24, r25
    mov r25, r26
    ; Store
    st Y, r25
    dec r28
    st Y, r24
    inc r28
    inc r22
    cpi r22, 5 ; Stop if > 4 (= sets zero flag, but we need to set
       carry. Hence, compare with 5)
    brlo INNER_LOOP ; Repeat until r22 = 4
    rjmp INC_OUTER
NO_SWAP:
    inc r22
    cpi r22, 4 ; Stop if > 4
    brlo INNER_LOOP ; Repeat until r22 = 4
    rjmp INC_OUTER
INC_OUTER:
    inc r21 ; Increment outer loop
    cpi r21, 4 ; Stop if > 4
    brlo OUTER_LOOP ; Repeat until r21 = 4, ie, until all numbers are
       sorted

nop ; No operation

; The sorted numbers are stored in sorted_numbers in SRAM
```

3 Implementation in Microchip Studio

Once the codes are ready, we proceed to compile and run them in Microchip Studio. The following lines show the steps to navigate through the software and to compile and run the code.

- Open Microchip Studio and create a new **Assembler Project**.
- Choose the appropriate target microcontroller, in this case, **ATmega8**.
- This creates a sample template for `main.asm`.
- Paste the concerned assembly code into `main.asm`.
- To compile, go to **Build** and select **Build Solution**.
- Set the device to **Simulator** in the Device options.
- For simulation, click on **Debug** and then use **Step Into (F11)** to execute each assembly instruction step by step. This helps in double-checking the code line by line.
- Observe the changes in the **Processor Status Window**, where you can watch how the register values and flag values change for every instruction.
- In the **Memory View Window**, observe the contents of both **SRAM** and **Flash Memory**.
- As instructions fetch values from Flash Memory, these values are loaded into registers, and the effects on register and memory content can be observed in real-time.

4 Observation

4.1 Finding Maximum and Minimum Values of 10 Numbers

- At each step the Z pointer loads the number from flash memory.
- As each number is fetched, it is compared with the current max and min, and the latter is updated appropriately.
- At the end of the program, r19 holds the maximum value and r20 holds the minimum value.
- The approach of using comparison instructions (`cp`) with conditional branching (`brlo`, `brsh`) allows for efficient max/min determination.

4.2 Addition of Ten Numbers

- The numbers are sequentially fetched from flash memory into register r18, and then added to register r16 using the `add` instruction.
- The sum of the numbers is stored in r16, which can be observed in the register view.
- In our code, we add the numbers 1, 2, 3, 2, 1, 3, 1, 2, 1, 4 resulting in 20, which is stored in r16.

4.3 Sorting 5 Numbers

- The numbers are fetched from flash memory and stored in SRAM using indirect addressing.
- The bubble sort algorithm compares adjacent numbers and swaps them if they are out of order.
- During each iteration, the values in SRAM are modified to reflect the swapping process, and the final sorted numbers are stored back in SRAM.
- The sorted numbers can be observed in the IO View window.

4.4 Processor Status Window

- The Processor Status Window in AVR Microchip Studio allows you to observe register values and flags such as carry (C), zero (Z), and negative (N).
- During debugging, you can monitor how these flags change after comparison instructions, like `cp`, and arithmetic instructions, like `add`.

4.5 Observing SRAM and Flash Memory

- In Microchip Studio, you can observe both flash memory and SRAM using the **Memory View** window. Flash memory is non-volatile and stores the program code and constant data, while SRAM is volatile and holds variables and intermediate data during program execution.
- Flash memory can be viewed by selecting the appropriate address range in the **Memory View**, such as the area where the numbers are stored.
- SRAM changes dynamically as the program runs. In the sorting program, you can watch the contents of SRAM being updated with each swap in the bubble sort algorithm.
- Using the **Step Into (F11)** feature, you can observe how numbers are transferred from flash memory to SRAM, manipulated, and written back to SRAM in real-time.

5 Conclusion

The experiment successfully demonstrated the programming of an AVR ATmega8 microcontroller in assembly language by attempting various problems such as finding max/min, addition and sorting. All the code written for this assignment can also be found here