Lab 2: Memory, Arithmetic and Logical Operations

EE222: Microprocessor Systems

Contents

1 Acknowledgements

This lab exercise is prepared by Mohammad Azfar Tariq and Muhammad Usman under the supervision of Dr. Rehan Ahmed for the course EE-222 Microprocessor Systems. Reporting any error or discrepancy found in the text is appreciated.

2 Administrivia

2.1 Objectives

By the end of this lab you will be able to;

- Appropriately use assembler directives in assembly code
- Implement basic arithmetic and logical operations in assembly language
- Differentiate, interpret and implement signed and unsigned arithmetic operations
- Critically analyze status register for insights in arithmetic manipulations
- Understand data memory of ATmega16
- Use data memory to store and load data

2.2 Deliverable

You are required to submit a report including;

1. Tables of observations and inferences
2. Source codes with **"proper comments"**

in the beginning of next lab.

3 Hardware Resources

No hardware implementation or resources are required for this lab. ☺

4 Introduction

4.1 Assembler Directives

The assembler needs some directions to assemble the code in a proper way as desired by the programmer. These directions are given through assembler directives. They are not part of the code.

Directive	Description	Example
`.EQU`	Define a symbol/name with a fixed value	`.EQU Time=33`
`.CSEG`	Instructions written after this directive are placed in code memory. If not declared explicitly, everything is placed in code memory by default	`.CSEG` `;your code`
`.ORG`	Place the upcoming instructions at the memory address declared	`.ORG 0x0F`

We expect you to **use assembler directives** in your codes for lab exercises.

4.2 Labels

Labels are used to point the instructions or data written after it. Compiler replaces them by respective address of instruction after compilation.

4.3 Data Space

Data space or data memory in AVR is primarily divided in three (or at times four) spaces.

1. Register File
2. Special Function Registers (SFRs)
3. SRAM
4. EEPROM

4.4 Memory Addressing Modes

Five Addressing Modes exist specifically for AVR architectures.

1. Data Direct Addressing
2. Input/Output Direct Addressing
3. Data Indirect Addressing
4. Data Indirect Addressing with Post-Increment or Pre-Decrement
5. Data Indirect Addressing with Displacement

4.5 Status Register

Status Register (`SREG`) is a regular 8 bit register with each bit representing crucial indications about operations being performed in the Arithmetic Logic Unit (ALU).

7	6	5	4	3	2	1	0
I	T	H	S	V	N	Z	C

For the moment, ignore I & T while a thorough analysis of the other flag bits is intended in this lab.

4.6 Important Instructions

In this lab you will be using instructions mentioned below. If you haven't studied them yet, please refer to the Microchip AVR ISA Manual[1].You are free to use other instructions too, if you find them useful.

Instruction	**Description**	**Example**
LDI Rd, K	Load immediate to register Rd	LDI R20, 0x01
LDS Rd, K	Load from RAM location K to Rd	LDS R20, 0x300
STS K, Rr	Store Rr to RAM location K	STS 0x300, R20
CPI Rd, K	Compare Rd with K (Rd - K)	CP R20, 0x01
BREQ K	Jump to label K if Zero flag is set to 1 by previous instruction	BREQ LABEL
BRNE K	Jump to label K if Zero flag is set to 0 by previous instruction	BRNE LABEL
ROL Rd	Rotate Left through carry	ROL R20
LSL Rd	Logical Shift Left	LSL R20
ADD Rd, Rr	Add Rr to Rd	ADD R20, R21
ADC Rd, Rr	Add Rr to Rd along-with the carry (Rd = Rd+Rr+C)	ADC R20, R21
SUB Rd, Rr	Subtract Rr from Rd	SUB R20, R21
SUBC Rd, Rr	Subtract Rr from Rd along-with carry (Rd = Rd - Rr - C)	SUBC R20, R21
INC Rd	Increment Rd by unity	INC R20
MUL Rd, Rr	Multiply Rd and Rr assuming unsigned	MUL R20, R21
MULS Rd, Rr	Multiply Rd and Rr considering them signed	MULS R20, R21
MULSU Rd, Rr	Multiply considering Rd signed and Rr unsigned	MULSU R20, R21

[1] https://www.microchip.com/webdoc/avrassembler/avrassembler.wb_instruction_list.html

5 Lab Tasks

5.1 Data Space

The address ranges of sections in data space are critical for implementation of a system involving Microcontrollers.

5.1.1 Task A

Consult the Datasheet[2] for ATmega16A and jot down the initial and final addresses of each address space in your report.

Example: Address Range of Register File begin from `0x000` and ends at `0x01F`.

5.2 Status Register, Arithmetic & Logical operations

5.2.1 Task B

Complete the code given below such that it subtract `1`, eight times from `R20`. Observe Status register on each decrement and record your readings on each instruction in the table(or doc[3]). In the first row of table write the value (1/0) of `CSR`. Write Yes/No in the second(alteration) row if the value of flag changes from previous one and jot down the reason for change in the third row.

```
.ORG 0x00
    LDI   R20, 4

    ;- - - -Your code here - - -


    ;- - - - - - - - - - - - - - -

End:
    RJMP  End
```

5.2.2 Task C

Multiply `0x011` and `0x0AB`

1. Considering both of them unsigned: `mul`.
2. Considering both of them signed: `muls`.
3. Considering `0x011` signed and `0x0AB` unsigned: `mulsu`.

Record and analyze the output in the table[4](or doc[5]).

[2]http://ww1.microchip.com/downloads/en/devicedoc/atmel-8154-8-bit-avr-atmega16a_d*atasheet.pdf*
[3]https://github.com/Uthmanhere/EE222/blob/master/02_taskA.docx
[4]https://github.com/Uthmanhere/EE222/blob/master/02_taskE.pdf
[5]https://github.com/Uthmanhere/EE222/blob/master/02_taskE.docx

5.2.3 Task D

In digital systems, packed binary coded decimal (BCD) is a type of binary code in which each decimal digit is represented by four bits. A small algorithm called "Double Dable"[6] is used to convert binary numbers to BCD representation. Binary numbers from $0 - 99$ will have a BCD representation in 8-bits. Follow the algorithm explained below and write an assembly program that converts binary numbers $(0 - 99)$ to 8-bit BCD. Keep the result in R20.
The lower 4-bits of 8-bit BCD represents the "Unit digit" and upper 4-bits represents "Tens digit".

1. Clear 8-bit BCD to zero.
2. Check (separately) if "Unit digit" and "Tens digit" is less than or equal to four (100_2).
3. If not, add three (11_2) to them (separately).
4. Shift MSB of binary number to LSB of BCD representation and repeat from step 2 until all the bits of binary number are shifted to BCD.
5. Stop after last shift.

For example consider the conversion of 99 (01100011_2),

BCD			
Tens	**Unit**	**Binary Number**	
0000	0000	01100011	
0000	0000	11000110	Shift left one bit
0000	0001	10001100	Shift left one bit
0000	0011	00011000	Shift left one bit
0000	0110	00110000	Shift left one bit
0000	1001	00110000	Add 3 to Unit because greater than 4
0001	0010	01100000	Shift left one bit
0010	0100	11000000	Shift left one bit
0100	1001	10000000	Shift left one bit
0100	1100	10000000	Add 3 to Unit because greater than 4
1001	1001	00000000	Shift left one bit
9	**9**		

Complete the code below such that it works for any value of "Num" between $0 - 99$.

```
.EQU Num=23
.ORG 0x00
    LDI   R20, Num
    ;- - - - - Your code here- - - - -

    ;- - - - - - - - - - - - - - - - -
End:
    RJMP End
```

[6] https://en.wikipedia.org/wiki/Double_dabble

5.3 Memory Operations

5.3.1 Task E

When we program AVR microcontrollers in C, the "int" data type is of 2-bytes. But as you have seen till now, in assembly, all the instructions handle data in form of a single byte and the data chunks greater than a byte are first broken down into bytes and then processed separately. Suppose we have two integers 1284_{10} and 775_{10}. Write an assembly program that;

- Step 1: Saves them in memory location `0x200` and `0x300`.
- Step 2: Calculates sum `(1284 + 775)` in `R20` and `R21`.
- Step 3: Subtracts them `(1284 - 775)` and keep result in R22 and R23.

Note that the data must be held in **"Little Endian Format"**.