LAB 1

AVR ASSEMBLER & SIMULATOR

OBJECTIVES:

- To examine and use an AVR assembler.
- To examine and use an AVR simulator.
- To examine the flag bits of the SREG

MATERIAL:

- Atmel Studio or the assembler of your choice.

WEB SITES:

- www.microchip.com for Atmel Studio Software

ACTIVITY 1

Write and assemble the following program. Use the simulator to single-step to examine the value that is contained in the registers.

```
        LDI     R16,0xFF
        OUT     DDRB,R16

L1:     OUT     PORTB,R16
        LDI     R20,0
        OUT     PORTB,R20
        RJMP    L1
```

1) What is the value of DDRB after executing OUT DDRB, R16?

   0xFF

2) What happens to PORTB immediately after:
   - OUT PORTB, R16 : PORTB = 0xFF
   - OUT PORTB, R20 : PORTB = 0x00

LAB 1

AVR ASSEMBLER & SIMULATOR

ACTIVITY 2

Write and assemble a program to add all the single digits of your ID number using R0 and R24 and save the result in R16. Pick 7 random numbers (all single digits) if you do not want to use your ID number. Then use the simulator to single-step the program and examine the registers of each step.

1) Indicate the size (8- or 16-bit) of each of the following registers for each step.
   R0 = 8-bit R24 = 8-bit R16= 8-bit
   Data in the memory pointed by current PC = 16-bit

ACTIVITY 3

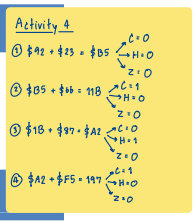
Write and assemble a program to load register R20 with value $99. Then from register R20 move it to R0, R12, and R31. Use the simulator to single-step the program and examine the registers.

1) Can LDI load an immediate value directly into below registers? If it is not, please give the reason.
   - R0: No
   - R12: No
   - R20: Yes
   - R31: Yes

The reason: The LDI (Load Immediate) instruction in AVR assembly is restricted to working only with the upper 16 general-purpose registers, specifically R16 through R31. It cannot load values directly into registers R0 through R15.

ACTIVITY 4

Write and assemble a program to add the following data and then use the simulator to examine the C, H and Z flags after the execution of each addition. $92, $23, $66, $87, $F5



ACTIVITY 5

Write and assemble the following program. Use the simulator to single-step and examine the flags and register content after the execution of each instruction.

```
LDI     R20,$27 → R20 = $27
LDI     R21,$15 → R21 = $15
SUB     R20, R21 → R20 = $12

LDI     R20,$20 → R20 = $20
LDI     R21,$15 → R21 = $15
SUB     R20, R21 → R20 = $0B

LDI     R24,95 → R24 = $5F
```

Activity 5

Block 1: Flags → C = 0, Z = 0

Block 2: Flags → C = 0, Z = 0

Block 3: Flags → C = 0

Block 4: Flags → C = 1, Z = 0

LAB 1

AVR ASSEMBLER & SIMULATOR

```
        LDI     R25,95  → R25 = $5F
        SUB     R24, R25 → R24 = $00

        LDI     R22,50 → R22 = $32
        LDI     R23,70 → R23 = $46
        SUB     R22, R23 → R22 = $EC

L1:     RJMP    L1
```

ACTIVITY 6

a) Write and assemble a program to load a value into each location of R20 – R23. Use the COM instruction to complement the value in each register. Use the simulator to single-step and examine the flags and register content after the execution of each instruction.

```
        LDI     R20,0 → R20 = $00
        LDI     R21,0xFF → R21 = $FF
        LDI     R22,0x11 → R22 = $11
        LDI     R23,0x22 → R23 = $22

        COM     R20 → R20 = $FF , Flags: C=1, Z=0
        COM     R21 → R21 = $00, Flags: C=1, Z=1
        COM     R22 → R22 = $EE, Flags: C=1, Z=0
        COM     R23 → R23 = $DD, Flags: C=1, Z=0

L1:     RJMP    L1
```

b) Find the value of the C flag after the execution of the following codes.

(1)
```
LDI   R20, $85
LDI   R21, $92
ADD   R20, R21
```
∴ C = 1

(2)
```
LDI   R16, $15
LDI   R17, $72
ADD   R16, R17
```
∴ C = 0

(3)
```
LDI   R25, $F5
LDI   R26, $52
ADD   R25, R26
```
∴ C = 1

LAB 1

AVR ASSEMBLER & SIMULATOR

(4)
```
LDI   R25, $FE
INC   R25
INC   R25
```
∴ C = 0

ACTIVITY 6

Write and assemble the following program. Use the simulator to single-step to examine how the program works. Give explanations line by line as well as the data in status register (SREG)

```
.EQU SUM = 0x300 → Define a constant symbol named SUM with the value 0x300
.ORG 00 → Tells the assembler to start placing the code at Flash Memory address 0x0000
        LDI     R16, 0x25 → Loads the hexadecimal value $25 into Register R16 (SREG: No flags affected)
        LDI     R17, $34 → Loads $34 into R17
        LDI     R18, 0b00110001 (hex $31) → Loads binary value into R18
        ADD     R16, R17 → Adds R17 ($34) to R16 ($25) / Result: $59 (SREG: H=0, S=0, V=0, N=0, Z=0, C=0)
        ADD     R16, R18 → Adds R18 ($31) to R16 ($59) / Result: $8A (SREG: N=1, S=1, others 0)
        LDI     R17, 11 → Loads decimal 11 into R17
        ADD     R16, R17 → Adds R17 ($0B) to R16 ($8A) / Result: $95 (SREG: N=1, S=1, others 0)
        STS     SUM, R16 → Stores the value in R16 ($95) directly into SRAM at address $0300
HERE:   JMP     HERE → An infinite loop where the program jumps back to the label HERE
```

a) After building the program, check the "Memory" windows with "0x00,prog". Find the last address of the program segment. Ans 0x000A (in Word Address) or 0x0014 (in Byte Address)

b) Change .ORG 0x00 to .ORG0x100. Find the start and last address of program.

```
.EQU SUM = 0x300
.ORG 0x00
        LDI     R16, 0x25
        LDI     R17, $34
        LDI     R18, 0b00110001
```

b) Ans
- Start Address: 0x0000 (the second part starts at 0x0100)
- Last Address: 0x0107 (Word Address)

LAB 1

AVR ASSEMBLER & SIMULATOR

```
.ORG 0x100
        ADD     R16, R17
        ADD     R16, R18
        LDI     R17, 11
        ADD     R16, R17
        STS     SUM, R16
HERE:   JMP     HERE
```

c) What happens to our compiled program when we add .ORG 0x100 as above code.
d) Why is the address not 0x100?

c) Ans The compiled program is split into two separate segments in the Flash memory.

① The instructions before the directive remain at the beginning (starting at 0x0000)

② The .ORG 0x100 directive forces the assembler to place the subsequent instructions starting specifically at address 0x0100

③ This creates a gap of unused memory between the end of the first segment and the start of the second segment.

d) Ans It appears as 0x200 because of the difference between Word and Byte addressing

The simulator's Memory window displays addresses in Bytes (8-bit), but the AVR Program Memory (Flash) is organized in 16-bit Words to store instructions. The .ORG directive uses Word addresses. Since 1 Word equals 2 Bytes, the address 0x100 (Word) translates to 0x200 (Byte) in the simulator view (0x100×2=0x200).

```
.SET RAM_START = 0x0100
.CSEG
.ORG 0x0000
hello_data:
        .DB "HELLO WORLD!"
main:
        LDI YL, LOW(RAM_START)
        LDI YH, HIGH(RAM_START)
        LDI ZL, LOW(2*hello_data)
        LDI ZH, HIGH(2*hello_data)
        LDI R18, 12
copy_loop:
        LPM R16, Z+
        ST Y+, R16
        DEC R18
        BRNE copy_loop
END:
        rjmp END
```

e) What does this program do? Explain line-by-line of the code by debugging and observing the changes in the SRAM.

This program copies a string of data ("HELLO WORLD!") from the Program Memory (Flash) to the Data Memory (SRAM).
It initializes pointers to the source (Flash) and destination (SRAM) addresses, uses a loop to read bytes one by one from Flash using the LPM instruction, and writes them to SRAM using the ST instruction.

Line-by-Line Explanation

1. .SET RAM_START = 0x0100
   - Defines a constant named RAM_START with the value 0x0100. This will be the starting address in the SRAM where the data is copied.
2. .CSEG
   - Tells the assembler that the following code belongs to the Code Segment (Program Memory).
3. .ORG 0x0000
   - Sets the location counter to 0x0000. The code execution will begin from this address.
4. hello_data: .DB "HELLO WORLD!"
   - Stores the constant string "HELLO WORLD!" into Program Memory byte-by-byte starting at the current address. The label hello_data marks this location.
5. main:
   - A label marking the start of the main program code.
6. LDI YL, LOW(RAM_START)
7. LDI YH, HIGH(RAM_START)
   - These two lines load the 16-bit address 0x0100 (defined by RAM_START) into the Y Pointer Register (R28:R29). The Y pointer will track where to write data in the SRAM.
8. LDI ZL, LOW(2*hello_data)
9. LDI ZH, HIGH(2*hello_data)
   - These lines load the address of the "HELLO WORLD!" string into the Z Pointer Register (R30:R31).
   - Note: The address is multiplied by 2 because Program Memory is addressed by words (16-bit), but the LPM instruction accesses bytes. The byte address is always 2× the word address.
10. LDI R18, 12
    - Loads the value 12 into register R18. This acts as a loop counter because "HELLO WORLD!" has 12 characters.
11. copy_loop:
    - A label marking the beginning of the copy loop.
12. LPM R16, Z+
    - Load Program Memory: Reads the byte pointed to by the Z register (from Flash) into register R16.
    - Z+: Automatically increments the Z pointer to the next byte address after the read,.
13. ST Y+, R16
    - Store Indirect: Writes the value held in R16 to the SRAM location pointed to by the Y register.
    - Y+: Automatically increments the Y pointer to the next SRAM address.
14. DEC R18
    - Decrements the loop counter R18 by 1.
15. BRNE copy_loop
    - Branch if Not Equal: Checks the Zero flag (Z). If R18 is not yet zero, it jumps back to copy_loop to process the next character.
16. END: rjmp END
    - An infinite loop to stop the program once copying is finished.

Observation in SRAM (Debugging)
If you debug this in the simulator and view the Memory window (selecting "Data" or "SRAM"):
• Before execution, SRAM addresses starting at 0x0100 will be empty (or random/zero).
• After execution, starting at address 0x0100, you will see the ASCII hex values for "HELLO WORLD!" appearing sequentially:
   - 0x0100: 48 ('H')
   - 0x0101: 45 ('E')
   - 0x0102: 4C ('L')
   - ...and so on.