Module 4 - Arithmetic

From memory access, addressing modes, the SREG, to arithmetic operations in AVR assembly,

1. Memory in AVR Architecture

AVR Architecture is an 8 bit single-chip RISC microcontroller with a modified Harvard Architecture that is organized as the following which causes it to behave certain ways when handling memory.

eclipse.umbc.edu - AVR Architecture

> “ You don't have to memorize all this don't worry.

There are several memory spaces in this organization, but for now we will look at the two most important ones to know.

Program/Flash Memory

rhjcoding.com - Flash memory When you compile and upload codes (flashing), it's stored in the **Flash Memory** section semi-permanently. During runtime, the control unit fetches instructions from this memory section.

This portion of the memory is non-volatile which means the data stored won't be erased after power loss.

Data Memory - DS (Data Space)

rhjcoding.com - Data Memory The **Data Memory** consists of several memory parts mapped into one contiguous memory addresses.

- **32 General Purpose Registers** (0x0000 - 0x001F)
- **64 I/O Registers** (0x0020 - 0x005F)
- **160 Extended I/O Registers** (0x0060 - 0x00FF)
- **SRAM** (Implementation Spesific)

The **Static RAM** (SRAM) is a memory block used to store data used during runtime. One part of this is the stack that may come in handy for temporary data (more on this later).

thecoderscorner.com - SRAM

2. Addressing Modes

Due to how the AVR Architecture and its memory are organized, AVR instructions, including arithmetic instructions, must follow certain addressing methods. Addressing methods are how the control unit may access different data locations according to the instruction which are usually 16 - 32 bits in length. Those data locations may include general purpose registers, I/O registers, extended I/O registers, and the SRAM.

Single Register Direct (Rd)

eclipse.umbc.edu - Single Register Direct

Operates on a single general purpose register **Rd** with d being values 0 - 31 (R0 - R31). Data is read from the register **Rd**, operated on, then stored back into the same register.

Some instructions with this addressing mode are `ROL Rd` and `COM Rd` which rotates the bits in Rd and inverts the bits in Rd respectively.

Double Register Direct (Rd, Rr)

eclipse.umbc.edu - Double Register Direct

Unlike Single Register Direct, Double Register Direct operates on two general purpose registers: source **Rr** and destination **Rd**. Data is read from Rr which are then operated alongside Rd to be stored back into Rd.

Instructions such as `ADD Rd, Rr` and `AND Rd, Rr` operates on both Rd and Rr which then stores the results (Addition and bitwise AND) in Rd.

Immediate Mode (Rd, K)

A constant value **K** alongside a register **Rd** is provided in the instruction itself, therefore the data itself is stored as a constant inside the Flash Memory. The constant value is limited to 255 (0xFF, 8 bits) and the register used is limited to R16 - R31 (4 bits with an offset).

Instructions such as `ORI Rd, K` and `LDI Rd, K` operates with a constant value K then storing it into Rd (bitwise OR and Loading).

> Notice how the instructions have 'I' in its names (stands for Immediate).

Some instructions that have **W** (Word) in its name operates on 16 bit words which are stored in two consecutive registers **R[d + 1]:Rd**. One instruction that operates like this is the `ADIW Rd, K` that adds a constant K to the 16 bit data in R[d + 1]:Rd. Word Immediate instructions can only operate on 4 even-indexed GP registers {R24, R26, R28, R30} and are limited to 6 bits constant values (0 - 63).

> The 16 bit opcode for ADIW is [1001 0110 KKdd KKK] which limits K to 6 bits and d to 2 bits (4 different registers max).

I/O Direct (Rd/Rr, A)

eclipse.umbc.edu - I/O Direct

Operates with I/O memory address **A**. Headers like `avr/io.h` defines these memory addresses into readable constants such as `PORTB`, `PINC`, and 'DDRA'. This **doesn't** include Extended I/O addresses such as `UBBR0H`.

Instructions such as `IN Rd, A` and `OUT A, Rr` stores IO(A) into Rd and Rr into IO(A) respectively.

Data Direct (Rd/Rr, k)

eclipse.umbc.edu - Data Direct

Instructions with this addressing utilizes 16 bit value **k** (0 - 65535) which is an address that corresponds to a space in memory, including the Extended I/O addresses scuh as `UBRR0H` and `UCSR0A`.

One instruction with Data Direct addressing is `LDS Rd, k` which loads the value in address k to register Rd.

Data Indirect (Rd/Rr, X/Y/Z +/- q)

eclipse.umbc.edu - Data Indirect

Registers X, Y, and Z corresponds to certain 16 bit register pairs in the memory.

Register	Higher Byte	Lower Byte
X	R27	R26
Y	R29	R28
Z	R31	R30

Instructions with Data Indirect addressing operates on data stored **in the address stored in the X/Y/Z registers**. For example, if X contains the value `0x01FF`, then `LD Rd, X` would load the data value stored in address `0x01FF` into Rd.

A feature of Data Indirect addressing is pre/post increment/decrement operators. By adding '+' or '-' next to the X/Y/Z register (eg: `X+`), the address stored would be automatically incremented or decremented respectively. This operation can be pre-ordered or post-ordered meaning the address value change would happen **before or after** the instruction itself.

As an example, assume `Y = 0x02`. Instruction `LD Rd, Y+` would store the value DS(Y) into Rd **THEN** increments Y to be `0x03`. As opposed to this, putting the increment operator **before** Y (`LD Rd, +Y`) would first increment the value of Y to be `0x03` **THEN** loads the data in the newly updated address into Rd.

> This may come in handy when operating with arrays stored as blocks of consecutive memory.

Certain instructions also allows accessing memory with displacement **q** on pointer registers Y/Z (X gak diajak). These instructions, usually ending with **D**, such as `STD Y+3, Rd` would offset the address stored in Y by 3 which stores the value in Rd into address DS(Y+3).

To see which instructions uses which memory addressing methods, refer to the AVR Instruction Set Manual. Knowing which instructions utilizes which memory addressing method would help you choose the best instruction to use in different situations when handling arithmetic and logic operations.

3. The Status Register

The Status Register (SREG) is a special 8 bit register that saves different operational status flags in each bit. Different operations may affect different flags (bit) of the register which then would be useful to create decisions after.

In AVR architecture, SREG is an I/O register meaning that it can be operated with instructions such as `OUT` and `IN`.

Carry Flag (C) [0]

- Indicates a Carry after addition or a Borrow after substraction.
- Usually happens when adding up numbers that results in a result greater than 8 bits (255) or when substracting numbers that results a negative analytically.
- Substracting a smaller number by a bigger one would result to a negative. This can be used to test whether a number is smaller than the other.

Zero Flag (Z) [1]

- Indicates that the previous operation results a 0.
- Can be set by different arithmetic operations such as `SUB` or `DEC` to logical operations such as `AND`.
- Substracting two equal values would result in a 0. This property can be used to test if two numbers are equal.

Negative Flag (N) [2]

- Indicates that the previous number results a negative.
- Under the hood works by testing the most significant bit (bit 7, leftmost) which indicates a 2s complement.

Two's Complement Overflow Flag (V) [3]

- Indicates that the previous operation is outside the range of signed values -128 to 127. These two values are the lowest and highest 8 bit signed values.
- Useful for testing overflow during signed 8 bit integer operations.

Sign Flag (S) [4]

- XOR of N and V flag.
- Indicates the sign of the result of the last operation. S = 1 means the last operation resulted in a negative signed number.

Half Carry (H) [5]

- Functions like the Carry flag but for only the lower nibbles of the last operation.
- Can come in handy when operating with 4 bit values such as BCD.

Bit Copy Storage (T) [6]

- Unlike the others, can be freely used by the programmer anyway they like.
- Set to 1 with `SET` and 0 with `CLT` instructions.
- Can be used by other registers with `BST Rr, b` and `BLD Rd, b` that moves the 1 bit value of T into or out of bit **b** of register Rr/Rd.
- Technically a free 1 bit storage.

Global Interrupt Enable (I) [7]

- Enables interrupt when set to 1 with `SEI` or disables it when cleared with `CLI`.

4. Advanced Arithmetic Operations

As a refresher, here are some fundamental AVR arithmetic and logical instructions.

Mnemonic	Operand	Description	Example	Notes
ADD	Rd, Rr	Add	`ADD R1, R2`	Rd = Rd + Rr
ADC	Rd, Rr	Add with Carry	`ADC R1, R2`	Rd = Rd + Rr + C (Carry flag)
ADIW	Rd, K	Add Immediate to Word	`ADIW R24, 40`	R[d+1]:Rd = R[d+1]:Rd + K
SUB	Rd, Rr	Subtract	`SUB R16, R17`	Rd = Rd - Rr
SBC	Rd, Rr	Subtract with Carry	`SBC R16, R17`	Rd = Rd - Rr - C
SUBI	Rd, K	Subtract Immediate	`SUBI R16, 67`	Rd = Rd - K
SBIW	Rd, K	Substract Immediate from Word	`SBIW R24, 40`	R[d+1]:Rd = R[d+1]:Rd - K
SBCI	Rd, K	Subtract Immediate with Carry	`SBCI R17, 0`	Rd = Rd - K - C
INC	Rd	Increment	`INC R16`	Rd = Rd + 1
DEC	Rd	Decrement	`DEC R16`	Rd = Rd - 1
MUL	Rd, Rr	Multiply Unsigned	`MUL R16, R17`	R1:R0 = Rd × Rr (16-bit result)
MULS	Rd, Rr	Multiply Signed	`MULS R16, R17`	R1:R0 = Rd × Rr (signed)
NEG	Rd	Negate (Two's Complement)	`NEG R16`	Rd = 0x00 - Rd
AND	Rd, Rr	Logical AND	`AND R1, R2`	Rd = Rd AND Rr
ANDI	Rd, K	AND Immediate	`ANDI R16, 0x0F`	Rd = Rd AND K
OR	Rd, Rr	Logical OR	`OR R1, R2`	Rd = Rd OR Rr
ORI	Rd, K	OR Immediate	`ORI R16, 0x80`	Rd = Rd OR K
EOR	Rd, Rr	Exclusive OR	`EOR R16, R17`	Rd = Rd XOR Rr
COM	Rd	One's Complement	`COM R16`	Rd = 0xFF - Rd (inverts all bits)
NEG	Rd	Two's Complement	`NEG R16`	Rd = -Rd (Signed)

Mnemonic	Operand	Description	Example	Notes
CLR	Rd	Clear Register	`CLR R16`	Rd = 0
SER	Rd	Set Register	`SER R16`	Rd = 0xFF (R16-R31 only)

Register Pair Arithmetic

To perform arithmetic operations with 16 bit words (0 - 65535), we can utilize the Carry Flag to perform ripple carry arithmetic operations.

16 Bit Word Addition

Suppose you want to add two big 16 bit numbers 26983 and 4882 stored in two register pairs R16:R17 and R18:R19. You can add the lower byte first then add the higher byte with a carry addition.

```
; DEC 26983 = HEX 0x6967
LDI R16, 0x69   ; upper byte
LDI R17, 0x67   ; lower byte

; DEC 4882 = HEX 0x1312
LDI R18, 0x13   ; upper byte
LDI R19, 0x12   ; lower byte

ADD R17, R19    ; add lower byte
ADC R16, R18    ; add upper byte + carry from lower

; R16:R17 = 31865
```

16 Bit Word Substraction

A similar thing can be done for substracting two 16 bit numbers.

```
; DEC 26983 = HEX 0x6967
LDI R16, 0x69   ; upper byte
LDI R17, 0x67   ; lower byte

; DEC 4882 = HEX 0x1312
```

```
LDI R18, 0x13   ; upper byte
LDI R19, 0x12   ; lower byte

SUB R17, R19    ; substract lower byte
SBC R16, R18    ; substract upper byte - borrow (C) from lower

; R16:R17 = 22101
```

Immediate Arithmetic

There are 3 immediate arithmetic instructions: `ADIW`, `SBIW`, and `SUBI`. In AVR there is no "Add with immediate" instruction. To do so, you can utilize the `SUBI Rd, K` instruction to substract a negative immediate number to achieve addition.

Suppose you want to add 67 to 61 stored in R16. You can do the following

```
LDI     R16, 61
SUBI    R16, -67 ; - -61 = + 61
; R16 = 128
```

Notice how there are only immediate instructions for word addition and substraction? Recall that word immediate addressing (`ADIW` & `SBIW`) can only operate on 6 bit values (0 - 63). How can we perform immediate arithmetic operations on bigger numbers like 26983?

Word Immediate Arithmetic

To perform addition/substraction operation with immediate numbers greater than 63, we simply split the word operation into two byte immediate operations similar to the previously explained 16 bit operations.

16 Bit Immediate Addition

```
; DEC 4882 = HEX 0x1312
LDI R16, 0x13       ; upper byte
LDI R17, 0x12       ; lower byte

.EQU num, 0x5457    ; immediate constant DEC 21591
SUBI R17, lo8(-num) ; 0x12 - -0x57 = 0x12 + 0x57         ; add lower byte
```

```
SBCI R16, hi8(-num) ; 0x13 - -0x54 - C = 0x13 + 0x54 + C ; add upper byte
; R16:R17 = 26473
```

16 Bit Immediate Substraction

```
LDI R16, 0x67       ; yeah you get the idea atp
LDI R17, 0x69

.EQU num, 0x1312
SUBI R17, lo8(num)  ; this time we simply just substract it
SBCI R16, hi8(num)  ; no need to be negative :)
; R16:R17 = 26983
```

Multiplying Numbers

There are multiple multiplication instructions in AVR, each supporting different data formats such as `MUL Rd, Rr` for unsigned numbers, `MULS Rd, Rr` for signed numbers, and `MULSU Rd, Rr` for multiplying a signed with an unsigned number.

Generally, the way they behave are similar: they multiply Rd with Rr then storing a 16 bit result in R1:R0.

```
LDI R16, 23
LDI R17, 9
MUL R16, R17
; R1:R0 = 23 * 9 = 207 (16 bits wide)
```

5. The Stack

The Stack is a memory section of the SRAM that follows First-In-First-Out (FIFO) principles. It is generally used to store temporary data for quick and easy access.

The top of the stack is tracked by the 16 bit (adjusting to memory addressing) **Stack Pointer** that stores the address. It increments and decrements accordingly.

rjhcoding.com - Stack pointer

The stack grows downwards meaning that newer data is placed on lower memory address.

> The top, is in fact, the bottom

The stack is used during subroutine (think of functions) calling to store the previous program counter when called with `RCALL` or `ICALL`. The program then restores the program counter to return to the previous location with `RET`.

Aside from subroutines, you can utilize the stack for your own use. The following are instructions that affects the stack.

Instruction	Stack Pointer	Description
PUSH Rd	SP = SP - 1	Value in Rd is pushed onto the top of the stack.
POP Rr	SP = SP + 1	The top of the stack is popped into Rr
RCALL ICALL	SP = SP - 2	PC is stored onto the stack. Program jumps to the subroutine.
RET RETI	SP = SP + 2	The top of the stack is popped back into PC. The program returns to the caller address.

Push Operation

rjhcoding.com - Push

Pop Operation

rjhcoding.com - Pop

Using Stacks

A very useful use case for stacks is as temporary storage without using up temporary registers to reduce mental overhead. For example, you might swap the contents of two registers `R16` and `R17`.

```
PUSH R16        ; temporarily store R16
MOV  R16, R17   ; overwrite R16 with R17
POP  R17        ; return the temporary data to R17
```

This form of temporary container may become handy as your program grows. Even though AVR provides 32 general purpose registers, keeping track each and every single one of them may become harder as your program grows with more subroutines. One subroutine might unintentionally modify a register that you were using during its execution which might produce unexpected results.

For example, given a subroutine that interact with USART using R24

```
SER_send_byte:
  LDS   R24, UCSR0A     ; R24 is filled with the content of UCSR0A
  SBRS  R24, UDRE0
  ...
  RET
```

Calling it from main would unexpectedly modify R24

```
main:
  LDI   R24, 0b00001010
  ...
  RCALL SER_send_byte
  OUT   TCCR0B, R24      ; this behaviour might be unexpected since R24 is no longer the same
```

Aside from giving proper documentations to subroutines which provides information on affected registers, the stack can be used to store values in affected register and retrieve it before returning.

```
; Sends R16 to USART
; Blocks while UDRE0 is not ready.
; Registers Affected: R24
SER_send_byte:
  PUSH  R24             ; stores R24
  LDS   R24, UCSR0A
  SBRS  R24, UDRE0
```

```
RJMP  SER_send_byte
STS   UDR0, R16
POP   R24             ; retrieve the value back to R24
RET
```

Do keep in mind that this would add an overhead by using additional cycles and memory for storing into the stack.

6. Printing Bytes as Hexadecimal Values

To easily debug and view numbers, we can create a subroutine that outputs numbers into serial. Since our architecture uses 8 bits, it is easier to print bytes as two hexadecimal values 0x00 - 0xFF by splitting the upper and lower nibbles. In this page, we will create a subroutine that prints R16 as hexadecimal to serial.

Given the following subroutines to use the serial:

```
; Initializes USART
; Registers Affected: R24
SER_init:
  CLR   R24
  STS   UCSR0A, R24                    ; clear UCSR0A register
  STS   UBRR0H, R24                    ; clear UBRR0H register
  LDI   R24, 103                       ; store in UBRR0L 103 value
  STS   UBRR0L, R24                    ; to set baud rate 9600
  LDI   R24, 1 << RXEN0 | 1 << TXEN0   ; enable RXB & TXB
  STS   UCSR0B, R24
  LDI   R24, 1 << UCSZ00 | 1 << UCSZ01  ; asynch, no parity, 1 stop, 8 bits
  STS   UCSR0C, R24
  RET
```

```
; Prints character in R16 to USART
; Blocks while UDRE0 is not ready.
; Registers Affected: R24
SER_send_byte:
  LDS   R24, UCSR0A
  SBRS  R24, UDRE0                     ; test data buffer if data can be sent
  RJMP  SER_send_byte                  ; loop back if not ready
  STS   UDR0, R16                      ; sends R16 to USART
  RET
```

Printing Nibbles

A 4 bit nibble, the lower parts of a byte, can be mapped into hexadecimal values 0 - F. Adding '0' to the nibble would map values 0 - 9 to its respective '0' - '9' ASCII characters.

```
SER_nibble:
  ANDI  R16, 0x0F ; mask that removes the higher nibble
  SUBI  R16, -'0' ; add '0' to R16 to represent ASCII '0' - '9'.
  ...
```

Just simply doing this, however, wouldn't work on values A - F since they are not continuously mapped right after '9' which would instead prints ':' for 10. To do so, we can check whether the resulting ASCII is greater than '9'. If so, we can then add it by 7 since 'A' is located 7 characters after '9'.

```
SER_nibble:
  PUSH  R16           ; preserves R16
  ANDI  R16, 0x0F     ; mask that removes the higher nibble
  SUBI  R16, -'0'     ; add '0' to R16 to represent ASCII '0' - '9'.
  CPI   R16, '9' + 1  ; compare with '9' + 1 = ':'
  BRLT  print_nibble  ; if no problem just simply print the character
  SUBI  R16, -7       ; otherwise add with 7 first to adjust for 'A'
print_nibble:
  RCALL SER_send_byte
  POP   R16           ; retrieves R16
  RET
```

With this subroutine, we can print the lower nibble of R16

```
main:
  RCALL SER_init

  LDI   R16, 0x0C
  RCALL SER_nibble      ; prints C

  LDI   R16, 0x14       ; prints 4
  RCALL SER_nibble
```

Print Nibbles

Printing Bytes

We can expand this further by printing entire bytes by first printing the upper nibble followed by the lower nibble. In AVR there is an instruction `SWAP Rd` that swaps the upper 4 bits with the lower 4 bits of Rd.

```
LDI R16, 0x14
SWAP R16
RCALL SER_nibble       ; prints 1 instead
```

With this, we can complete the hexadecimal printing subroutine.

```
; Prints R16 as HEX to USART
; R16 is preserved.
SER_hex:
  SWAP R16            ; swap to get upper nibble first
  RCALL SER_nibble
  SWAP R16            ; revert the nibbles back for the lower one
  RCALL SER_nibble
  RET

main:
  RCALL SER_init
  LDI   R16, 0x3C
  RCALL SER_hex      ; prints 3C
```

Print Bytes

7. ??? ????

Literally me

```
#define __SFR_OFFSET 0x00
#include "avr/io.h"

.global main
main:
  RCALL SER_init

  LDI   ZH, hi8(opening_msg)
  LDI   ZL, lo8(opening_msg)
  RCALL SER_print

  ; 16 Bit Addition
  LDI   ZH, hi8(addition_msg)
  LDI   ZL, lo8(addition_msg)
  RCALL SER_print

  ; DEC 26983 = HEX 0x6967
  LDI R16, 0x69   ; upper byte
  LDI R17, 0x67   ; lower byte

  ; DEC 4882 = HEX 0x1312
  LDI R18, 0x13   ; upper byte
  LDI R19, 0x12   ; lower byte

  ADD R17, R19    ; add lower byte
  ADC R16, R18    ; add upper byte + carry from lower

  ; R16:R17 = 31865
  RCALL SER_hex
  MOV R16, R17  ; print the lower byte this time
  RCALL SER_hex

  ; 16 Bit Substraction
  LDI   ZH, hi8(substraction_msg)
```

```
LDI   ZL, lo8(substraction_msg)
RCALL SER_print

; DEC 26983 = HEX 0x6967
LDI R16, 0x69   ; upper byte
LDI R17, 0x67   ; lower byte

; DEC 4882 = HEX 0x1312
LDI R18, 0x13   ; upper byte
LDI R19, 0x12   ; lower byte

SUB R17, R19    ; substract lower byte
SBC R16, R18    ; substract upper byte - borrow (C) from lower

; R16:R17 = 22101
RCALL SER_hex
MOV R16, R17  ; print the lower byte this time
RCALL SER_hex

; 16 Bit Immediate Addition
LDI   ZH, hi8(iaddition_msg)
LDI   ZL, lo8(iaddition_msg)
RCALL SER_print

LDI R16, 0x13       ; upper byte
LDI R17, 0x12       ; lower byte

.EQU num, 0x5457    ; immediate variable directive
SUBI R17, lo8(-num) ; 0x12 - -0x57 = 0x12 + 0x57
SBCI R16, hi8(-num) ; 0x13 - -0x54 - C = 0x13 + 0x54 + C

RCALL SER_hex
MOV R16, R17  ; print the lower byte this time
RCALL SER_hex

; 16 Bit Immediate Substraction
LDI   ZH, hi8(isubstraction_msg)
LDI   ZL, lo8(isubstraction_msg)
RCALL SER_print
```

```
  LDI R16, 0x67       ; yeah you get the idea atp
  LDI R17, 0x69

  .EQU num, 0x1312
  SUBI R17, lo8(num)  ; this time we simply just substract it
  SBCI R16, hi8(num)  ; no need to be negative :)

  RCALL SER_hex
  MOV R16, R17  ; print the lower byte this time
  RCALL SER_hex

  ; Multiplication
  LDI   ZH, hi8(multiplication_msg)
  LDI   ZL, lo8(multiplication_msg)
  RCALL SER_print

  LDI R16, 23
  LDI R17, 9
  MUL R16, R17

  MOV R16, R1
  RCALL SER_hex
  MOV R16, R0
  RCALL SER_hex

loop:
  RCALL loop

; Initializes USART
; Registers Affected: R24
SER_init:
  CLR   R24
  STS   UCSR0A, R24                    ; clear UCSR0A register
  STS   UBRR0H, R24                    ; clear UBRR0H register
  LDI   R24, 103                       ; store in UBRR0L 103 value
  STS   UBRR0L, R24                    ; to set baud rate 9600
  LDI   R24, 1 << RXEN0 | 1 << TXEN0    ; enable RXB & TXB
  STS   UCSR0B, R24
  LDI   R24, 1 << UCSZ00 | 1 << UCSZ01  ; asynch, no parity, 1 stop, 8 bits
  STS   UCSR0C, R24
```

```
  RET

; Prints character in R16 to USART
; Blocks while UDRE0 is not ready.
; Registers Affected: R24
SER_send_byte:
  LDS   R24, UCSR0A
  SBRS  R24, UDRE0                    ; test data buffer if data can be sent
  RJMP  SER_send_byte                 ; loop back if not ready
  STS   UDR0, R16                     ; sends R16 to USART
  RET

; Prints entire message of data pointed in Z until string end (0)
; To fill Z with string message:
;   LDI   ZH, hi8(message)
;   LDI   ZL, lo8(message)
; Registers Affected: R16, R24 (SER_send_byte)
SER_print:
  LPM   R16, Z+                       ; load char of string onto R18
  CPI   R16, 0                        ; check if R16 = 0 (end of string)
  BREQ  exit_SER_print                ; if yes, exit
  RCALL SER_send_byte                 ; send the character byte
  RJMP  SER_print                     ; loop back & get next character
exit_SER_print:
  RET

; Prints the lower nibble of R16
; Registers Affected: R24 (SER_send_byte)
SER_nibble:
  PUSH  R16           ; preserves R16
  ANDI  R16, 0x0F     ; mask that removes the higher nibble
  SUBI  R16, -'0'     ; add '0' to R16 to represent ASCII '0' - '9'.
  CPI   R16, '9' + 1  ; compare with '9' + 1 = ':'
  BRLT  print_nibble  ; if no problem just simply print the character
  SUBI  R16, -7       ; otherwise add with 7 first to adjust for 'A'
print_nibble:
  RCALL SER_send_byte
  POP   R16           ; retrieves R16
  RET
```

```
; Prints R16 as HEX to USART
; R16 is preserved.
SER_hex:
  SWAP R16            ; swap to get upper nibble first
  RCALL SER_nibble
  SWAP R16            ; revert the nibbles back for the lower one
  RCALL SER_nibble
  RET

opening_msg:
    .byte 10,13 ; new line, carriage return
    .ascii "== Literally Einstein and Tesla =="
    .byte 0

addition_msg:
    .byte 10,13 ; new line, carriage return
    .ascii "16 Bit Addition: "
    .byte 0

substraction_msg:
    .byte 10,13 ; new line, carriage return
    .ascii "16 Bit Addition: "
    .byte 0

iaddition_msg:
    .byte 10,13 ; new line, carriage return
    .ascii "16 Bit Immediate Addition: "
    .byte 0

isubstraction_msg:
    .byte 10,13 ; new line, carriage return
    .ascii "16 Bit Immediate Substraction: "
    .byte 0

multiplication_msg:
    .byte 10,13 ; new line, carriage return
    .ascii "Multiplication: "
    .byte 0
```

Output

8. References

“AVR ® Instruction Set Manual AVR ® Instruction Set Manual.” Available: https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-InstructionSet-Manual-DS40002198.pdf

“Lecture 02 – AVR Architecture,” Umbc.edu, 2025. https://eclipse.umbc.edu/robucci/cmpe311/Lectures/L02-AVR_Archetecture/

“How the Arduino memory model works - for AVR · The Coders Corner,” Thecoderscorner.com, 2018. https://www.thecoderscorner.com/electronics/microcontrollers/efficiency/how-arduino-avr-memory-model-works/

“AVR Tutorials - Working With Registers R0 - R31,” www.rjhcoding.com. http://www.rjhcoding.com/avr-asm-registers.php

“Lecture 04 – AVR CPU Registers,” eclipse.umbc.edu. https://eclipse.umbc.edu/robucci/cmpe311/Lectures/L05-AVR_Addressing_Modes/

“AVR Tutorials - The Status Register,” www.rjhcoding.com. http://www.rjhcoding.com/avr-asm-sreg.php

“Assembly via Arduino - Unsigned Arithmetic Operations.” https://akuzechie.blogspot.com/2021/10/assembly-via-arduino-unsigned.html

“AVR Tutorials - Assembly Subroutines,” Rjhcoding.com, 2018. http://www.rjhcoding.com/avr-asm-functions.php

M. Reynolds, “AVR® Stack Register - Developer Help,” Microchip.com, 2023. https://developerhelp.microchip.com/xwiki/bin/view/products/mcu-mpu/8-bit-avr/structure/stack/ (accessed Feb. 25, 2026).

“Lecture 04 – AVR CPU Registers,” Umbc.edu, 2025. https://eclipse.umbc.edu/robucci/cmpe311/Lectures/L04-AVR_CPU_Registers/