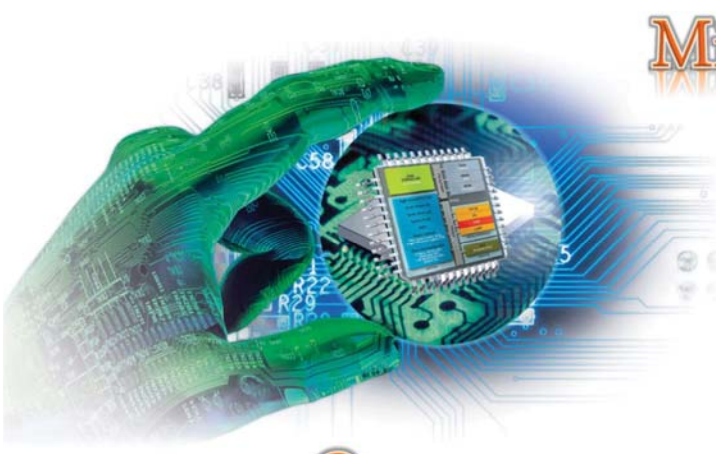
Microprocessors



3

Textbook: Ch03+Ch04

AVR Architecture and Assembly Language Programming

Dr. Hassan SHARABATY
EEE Department, Faculty of Engineering
University of Turkish Aeronautical Association

The general purpose registers (GPRs) in AVR

In the CPU, registers are used to **store information temporarily**. That information could be a byte of data to be processed, or an address pointing to the data to be fetched.

The vast majority of AVR registers are 8-bit registers.

MSB	D7	D6	D5	D4	D3	D2	D1	D0	LSB

The 32 GPRs of AVR (R0–R31) are located in the lowest location of memory address. All of these registers are 8 bits.

GPRs
R0
R1
R2
⋮
R15
R16
R17
⋮
R30
R31

The general purpose registers in AVR can be used by all arithmetic and logic instructions.

Some simple instructions

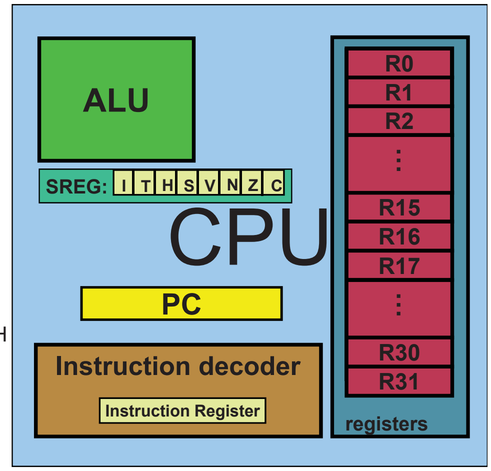
1. Loading values into the general purpose registers

LDI instruction (**L**oad **I**mmediate)

the LDI instruction copies 8-bit data into the general purpose registers. It has the following format:

LDI R_d , k ≡ ($Rd = k$)

- **K** is an 8-bit value that can be 0-255_d or 00-FF_H
- **R_d** (**d**estination) is R16 to R31.



Example:

LDI R16,53 ; loads the R16 with the value 53 in decimal

LDI R23,0x27 or LDI R23, $27 ; loads the R23 with the value 27_H

LDI R05,0x99 ; invalid instruction

Some simple instructions

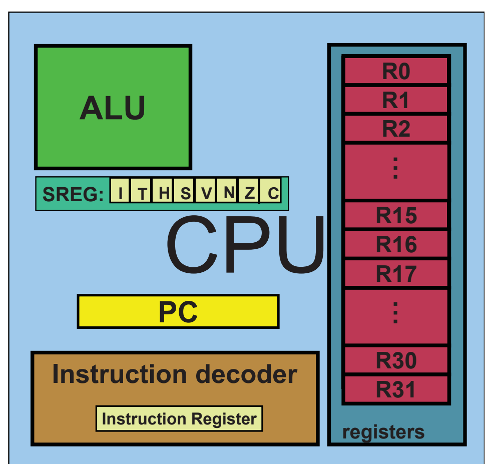
2. Arithmetic calculation

- There are some instructions for doing Arithmetic and logic operations; such as:

  ADD, SUB, MUL, AND, etc.

- **ADD Rd , Rs** ➔ Rd = Rd + Rs

  (ADD Rs to Rd and store the result in Rd)



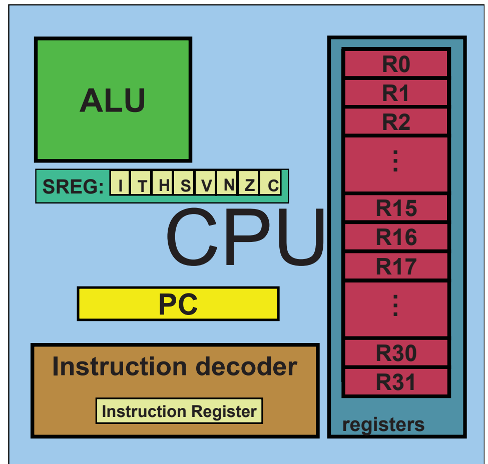
Example:

- ADD R25, R9 ➔ R25 = R25 + R9
- ADD R17,R30 ➔ R17 = R17 + R30

A simple program 1

- Write a program that calculates 19 + 95

```
LDI R16, 19     ;R16 = 19
LDI R20, 95     ;R20 = 95
ADD R16, R20    ;R16 = R16 + R20
```



A simple program 2

- Write a program that calculates 19 + 95 + 5

```
LDI     R16, 19         ;R16 = 19
LDI     R20, 95         ;R20 = 95
LDI     R21, 5          ;R21 = 5
ADD     R16, R20        ;R16 = R16 + R20
ADD     R16, R21        ;R16 = R16 + R21
```

Or

```
LDI     R16, 19         ;R16 = 19
LDI     R20, 95         ;R20 = 95
ADD     R16, R20        ;R16 = R16 + R20
LDI     R20, 5          ;R20 = 5
ADD     R16, R20        ;R16 = R16 + R20
```

The 2nd way is recommended, why?

A simple program 3

- Write a program to add the following numbers:
  - 25_H, 34_H

```
LDI R16,0x25  ;load 0x25 into R16
LDI R17,0x34  ;load 0x34 into R17
ADD R16,R17   ;add value R17 to R16 (R16 = R16 + R17)
```

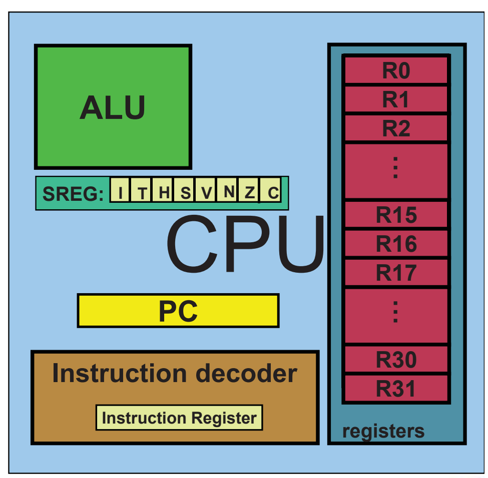

Some simple instructions

2. Arithmetic calculation

- SUB Rd , Rs

  ➔ Rd = Rd – Rs

- Example:
  - SUB R25, R9 ➔ R25 = R25 - R9
  - SUB R17,R30 ➔ R17= R17 - R30



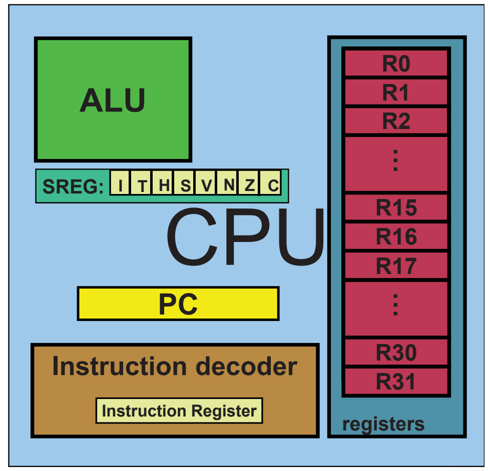
Some simple instructions

2. Arithmetic calculation

- **INC Rd** (Rd = Rd + 1)

Example:

INC R25 ➔ R25 = R25 + 1



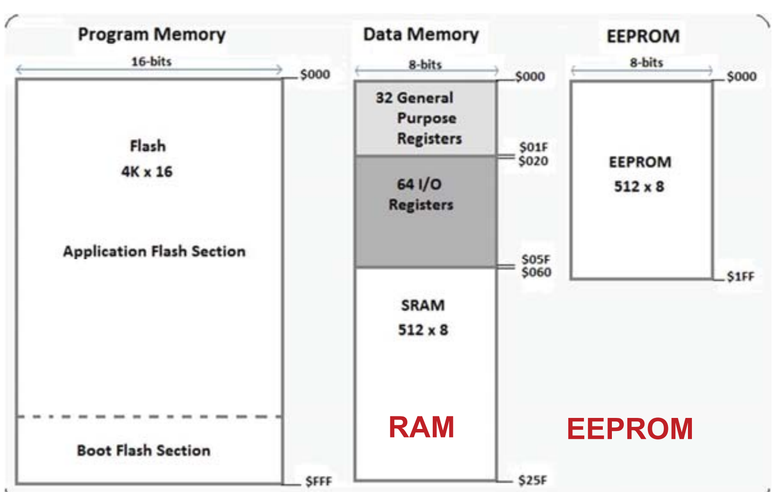
- **DEC Rd** (Rd = Rd – 1)

Example:

DEC R23 ➔ R23 = R23 - 1

AVR Memory Organization

In AVR microcontrollers there are tree kinds of memory space, removing the need for external memory in most applications:

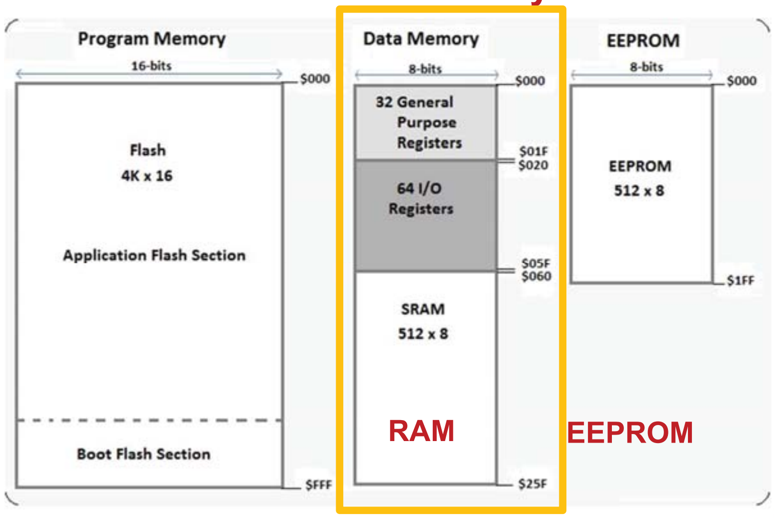


Code memory, **Data memory**

AVR Memory Organization

➢ Our program is stored in **code memory** space,

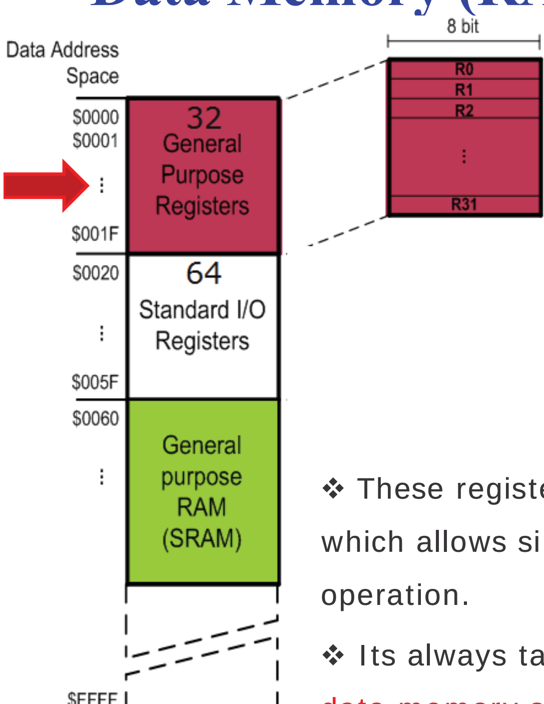
whereas the **data memory** stores data.



Code memory, **Data memory**

AVR Memory Organization

Data Memory (RAM)



➢ The **data memory** space is composed of three parts:

- **GPRs** (general purpose registers),
- **I/O memory**
- **Internal data SRAM**.

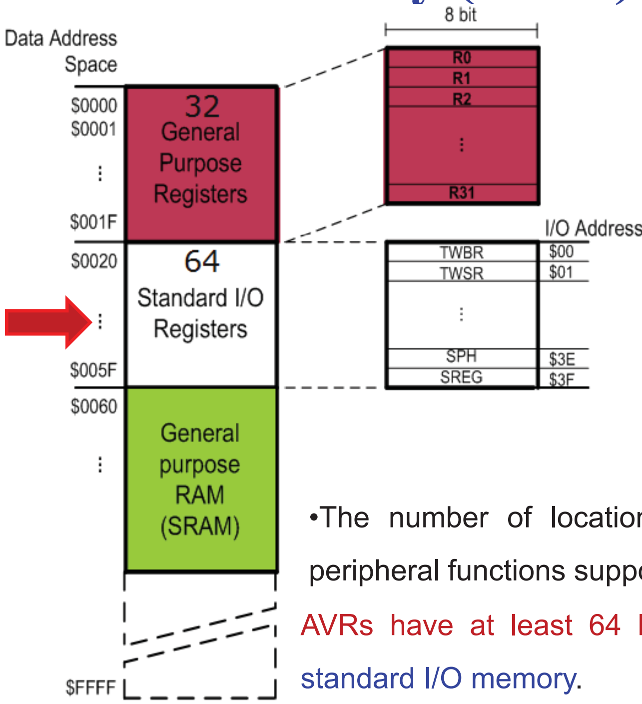
• **GPRs space** consists of 32 **general purpose 8-bit registers** (R0-R31) (the GPRs do not have any specific function).

❖ These registers have the shortest (fastest) access time, which allows single-cycle Arithmetic Logic Unit (ALU) operation.

❖ Its always take the **address location $00–$1F** in the data memory space, regardless of the AVR chip number.

AVR Memory Organization

Data Memory (RAM)

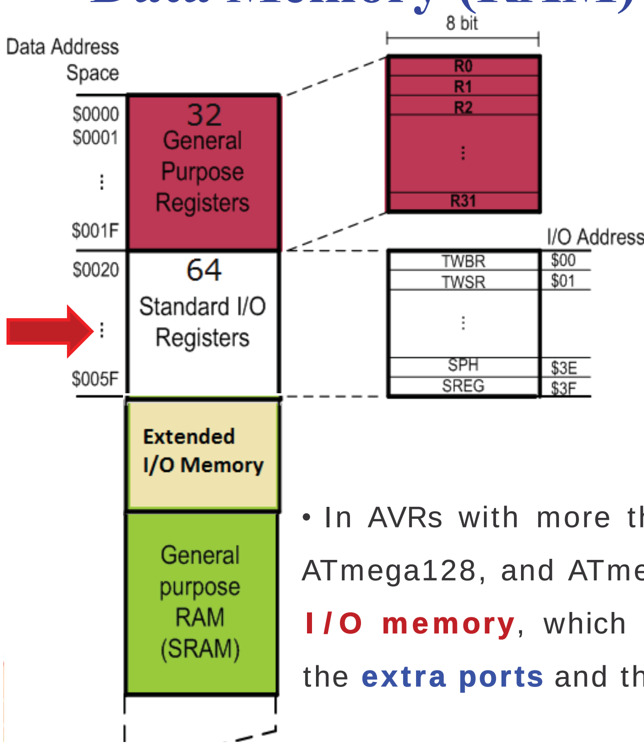


- **I/O Memory** or **Specific Function Registers (SFRs) :** These registers control the CPU peripherals functions, such as status register, timers, serial communication, I/O ports, ADC, and so on.
- The AVR I/O memory is made also of 8-bit registers.

•The number of locations depends on the pin numbers and peripheral functions supported by that chip. However, all of the AVRs have at least 64 bytes of I/O memory locations, called standard I/O memory.

AVR Memory Organization

Data Memory (RAM)



- **I/O Memory** or **Specific Function Registers (SFRs) :** These registers control the CPU peripherals functions, such as status register, timers, serial communication, I/O ports, ADC, and so on.
- The AVR I/O memory is made also of 8-bit registers.

- In AVRs with more than **32** I/O **pins** (e.g., ATmega64, ATmega128, and ATmega256) there is also an **extended I/O memory**, which contains registers **for controlling** the **extra ports** and the **extra peripherals**.

I/O Registers & their Data Memory Address Locations

Each location in I/O memory has two addresses:

Address I/O	Address Mem.	Name
$00	$20	TWBR
$01	$21	TWSR
$02	$22	TWAR
$03	$23	TWDR
$04	$24	ADCL
$05	$25	ADCH
$06	$26	ADCSRA
$07	$27	ADMUX
$08	$28	ACSR
$09	$29	UBRRL
$0A	$2A	UCSRB
$0B	$2B	UCSRA
$0C	$2C	UDR
$0D	$2D	SPCR
$0E	$2E	SPSR
$0F	$2F	SPDR
$10	$30	PIND
$11	$31	DDRD
$12	$32	PORTD
$13	$33	PINC
$14	$34	DDRC
$15	$35	PORTC

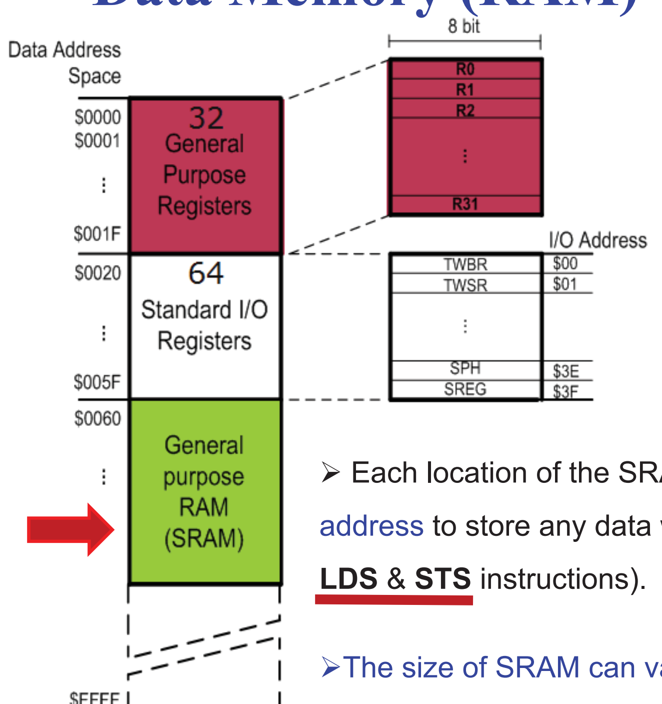
Address I/O	Address Mem.	Name
$16	$36	PINB
$17	$37	DDRB
$18	$38	PORTB
$19	$39	PINA
$1A	$3A	DDRA
$1B	$3B	PORTA
$1C	$3C	EECR
$1D	$3D	EEDR
$1E	$3E	EEARL
$1F	$3F	EEARH
$20	$40	UBRRC UBRRH
$21	$41	WDTCR
$22	$42	ASSR
$23	$43	OCR2
$24	$44	TCNT2
$25	$45	TCCR2
$26	$46	ICR1L
$27	$47	ICR1H
$28	$48	OCR1BL
$29	$49	OCR1BH
$2A	$4A	OCR1AL

Address I/O	Address Mem.	Name
$2B	$4B	OCR1AH
$2C	$4C	TCNT1L
$2D	$4D	TCNT1H
$2E	$4E	TCCR1B
$2F	$4F	TCCR1A
$30	$50	SFIOR
$31	$51	OCDR OSCCAL
$32	$52	TCNT0
$33	$53	TCCR0
$34	$54	MCUCSR
$35	$55	MCUCR
$36	$56	TWCR
$37	$57	SPMCR
$38	$58	TIFR
$39	$59	TIMSK
$3A	$5A	GIFR
$3B	$5B	GICR
$3C	$5C	OCR0
$3D	$5D	SPL
$3E	$5E	SPH
$3F	$5F	SREG

- **Data memory address: (0000_H - $FFFF_H$)** ➔ for I/O memory **(20_H - $5F_H$)**.
- **I/O address : (00_H - $3F_H$)** which is a relative address in comparison to the beginning of the I/O memory

AVR Memory Organization

Data Memory (RAM)



- **Internal data SRAM** is used for **storing temporary data** and parameters by AVR programmers and C compilers. Generally, this is called scratch pad.

➢ Each location of the SRAM can be accessed directly by its address to store any data we want as long as it is 8 bit. (Using **LDS** & **STS** instructions).

➢ The size of SRAM can vary from chip to chip, even among members of the same family.

Using instruction with Data Memory.

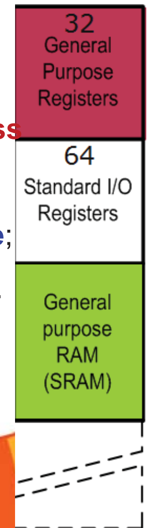
LDS (Load direct from data space)

LDS Rd, k ;Rd = [k]
 	
 Example:
 LDS R1, 0x60

; load Rd with the contents of location K

; (0 ≤ d ≤ 31)

; K is an address between \$0000 to \$FFFF



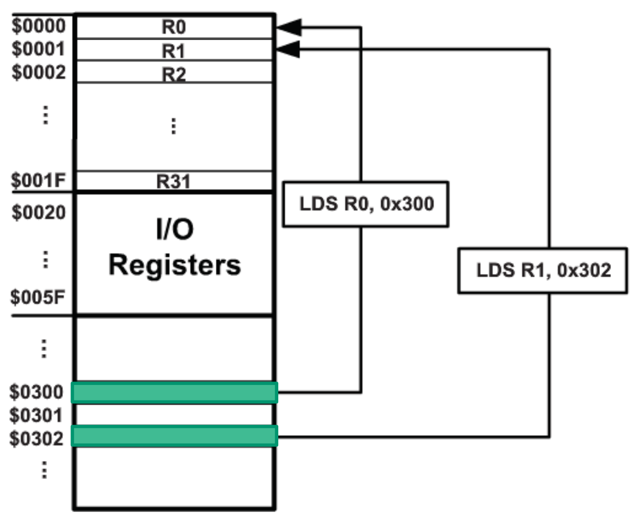
- The LDS instruction tells the CPU to load (copy) one byte from an **address** in the data memory to the GPRs.
- The **location k** in the data memory **could be any part of the data space**; it can be one of the I/O registers, a location in the internal SRAM, or a GPR.
- For example:

LDS R20,0x1 ; will copy the contents of location 1
; (which is the address of R1) into R20.
; So, the instruction copies R1 to R20.

Using instruction with Data Memory.

Unleash your Creativity!

Write program to add the contents of location 0x300 of SRAM to location 0x302.



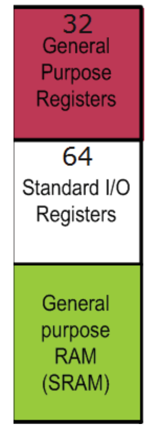

```
LDS    R0, 0x300    ;R0 = the contents of location 0x300
LDS    R1, 0x302    ;R1 = the contents of location 0x302
ADD    R1, R0       ;add R0 to R1
```

Using instruction with Data Memory.

STS (Store direct to data space)
STS k , Rs ; [k]=Rs Example: STS 0x60,R15 ; [0x60] = R15

;store register Rs into location K

;K is an address between $0000 to $FFFF



- The STS instruction tells the CPU to store (copy) the contents of the GPR to an address location in the data memory space.
- The **location** K **could be any part of the data memory space**; it can be one of the I/O registers, a location in the SRAM, or a GPR.
- **Example:** Write program to add the contents of location 0x220 to location 0x221, and stores the result in location 0x221:

```
LDS   R30, 0x220  ;load R30 with the contents of location 0x220
LDS   R31, 0x221  ;load R31 with the contents of location 0x221
ADD   R31, R30    ;add R30 to R31
STS   0x221, R31  ;store R31 to data space location 0x221
```

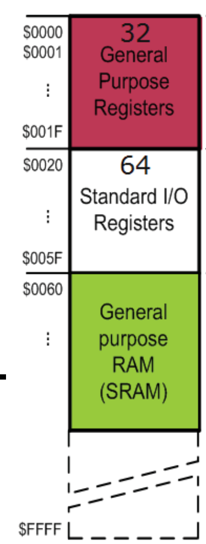
Using instruction with Data Memory.

- *Example: Write a program that stores CA_H into location 0x35 of RAM.*

Solution:

```
LDI  R20, 0xCA        ;R20 =CAH   = 11001010
STS  0x35, R20        ;[0x35] = R20 = CAH
```

Notice that you cannot copy (store) an immediate value directly into the SRAM location in the AVR. This must be done via the GPRs.



- *Example: Write a program that copies the contents of location 0x80 of RAM into location 0x81.*

Solution:

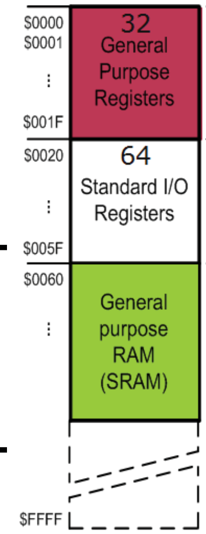
```
LDS  R20, 0x80     ;R20 = [0x80]
STS  0x81, R20     ;[0x81] = R20 = [0x80]
```

Using instruction with Data Memory.

- ***Example: Add contents of location 0x90 to contents of location 0x95 and store the result in location 0x313.***

Solution:

```
LDS  R20, 0x90        ;R20 = [0x90]
LDS  R21, 0x95        ;R21 = [0x95]
ADD  R20, R21         ;R20 = R20 + R21
STS  0x313, R20       ;[0x313] = R20
```



- ***Example: What does the following instruction do?*** **LDS R20, 2**

Answer:

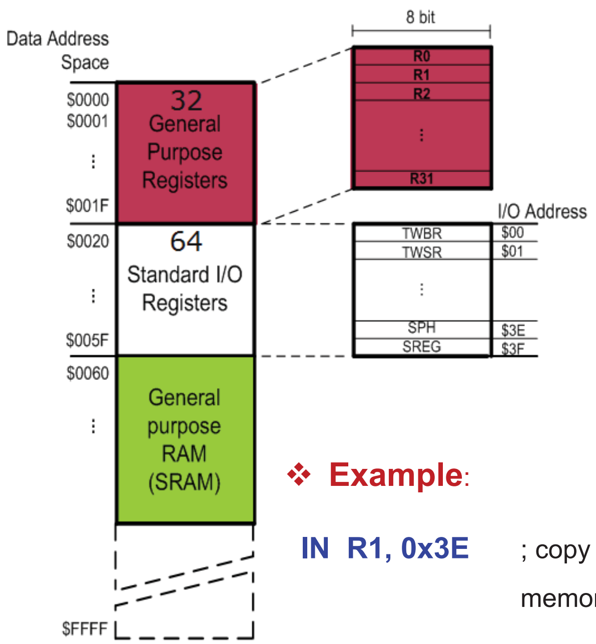
It copies the contents of R2 into R20; as 2 is the address of R2.

- **Example:** ***Store 0x53 into the*** **PORTC** ***. The address of*** **PORTC** ***is 0x35***

Solution:

```
LDI        R20, 0x53        ;R20 = 0x53
STS        0x35, R20        ;PORTC = R20
```


IN instruction (IN from I/O location)



$\boxed{\textbf{IN } \mathbf{R_d}\textbf{, IO}_{\textbf{addr}} \rightarrow R_d = [addr]}$

; load an I/O location to the GPR

$(0 \le d \le 31)$, $(00 \le addr \le 63)$

or $(00_H \le addr \le 3F_H)$

- The IN instruction tells the CPU to load one byte from an I/O register to the GPR.

❖ **Example:**

IN R1, 0x3E ; copy the contents of location $3E_H$ (whose data memory address is 0x5E) of the I/O memory into R1

➔ R1 = SPH

IN instruction (IN from I/O location)

Example:

IN R19, 0x10 ;load R19 with the contents of location $10 (R19 = PIND)

; write the equivalent LDS instruction?

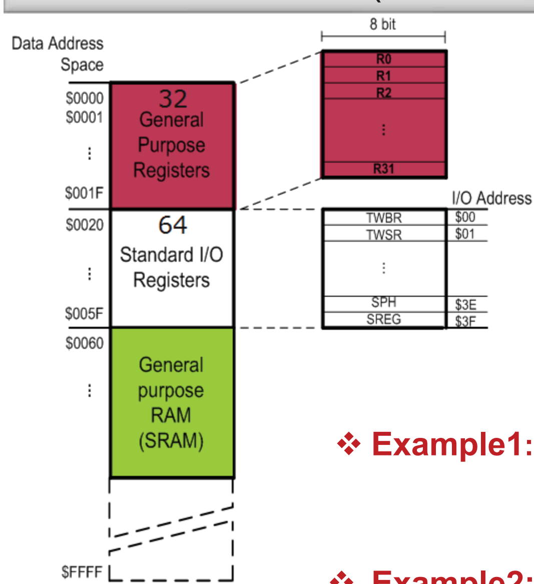
To work with the I/O registers more easily, **we can use their names instead** of their **I/O addresses**.

✓ **IN R19, PIND** ;load R19 with PIND

Example: Write a program to adds the contents of PIND to PINB, and stores the result in location 0x300 of the data memory:

```
IN  R1,PIND       ;load R1 with PIND
IN  R2,PINB       ;load R2 with PINB
ADD R1, R2        ;R1 = R1 + R2
STS 0x300, R1     ;store R1 to data space location $300
```


OUT instruction (Out to I/O location)



OUT IO_{Addr}, Rs ; [addr]=Rs

; store register to I/O location

($0 \le s \le 31$) , ($0 \le Addr \le 63$)

- **OUT** instruction tells the CPU to **store the GPR content** to **the I/O register**.

❖ **Example1: OUT 0x3E, R15** ;SPH = R15

❖ **Example2:** The following program copies PINB to PORTC:

```
IN   R20, PINB    ; load R20 with the contents of I/O reg PINB
OUT  PORTC, R20   ; out R20 to PORTC
```

OUT instruction (Out to I/O location)

Read also p67: **MOV Rd, Rs**

Unleash your Creativity!

Example:

Write a program that **adds** the contents of the **PINC** IO register to the contents of **PIND** and **stores the result** in **location 0x90** of the SRAM

Solution:

```
IN     R20,PINC   ;R20 = PINC
IN     R21,PIND   ;R21 = PIND
ADD    R20,R21    ;R20 = R20 + R21
STS    0x90,R20   ;[0x90] = R20
```

Example: Write a program to **get data from the PINB** and **send it to** the I/O register of **PORT C** **continuously**.

Solution:

```
AGAIN: IN   R16, PINB      ;bring data from PortB into R16
       OUT  PORTC,R16      ;send it to Port C
       JMP  AGAIN          ;keep doing it forever
```


COM - Complement instruction

Unleash your Creativity!

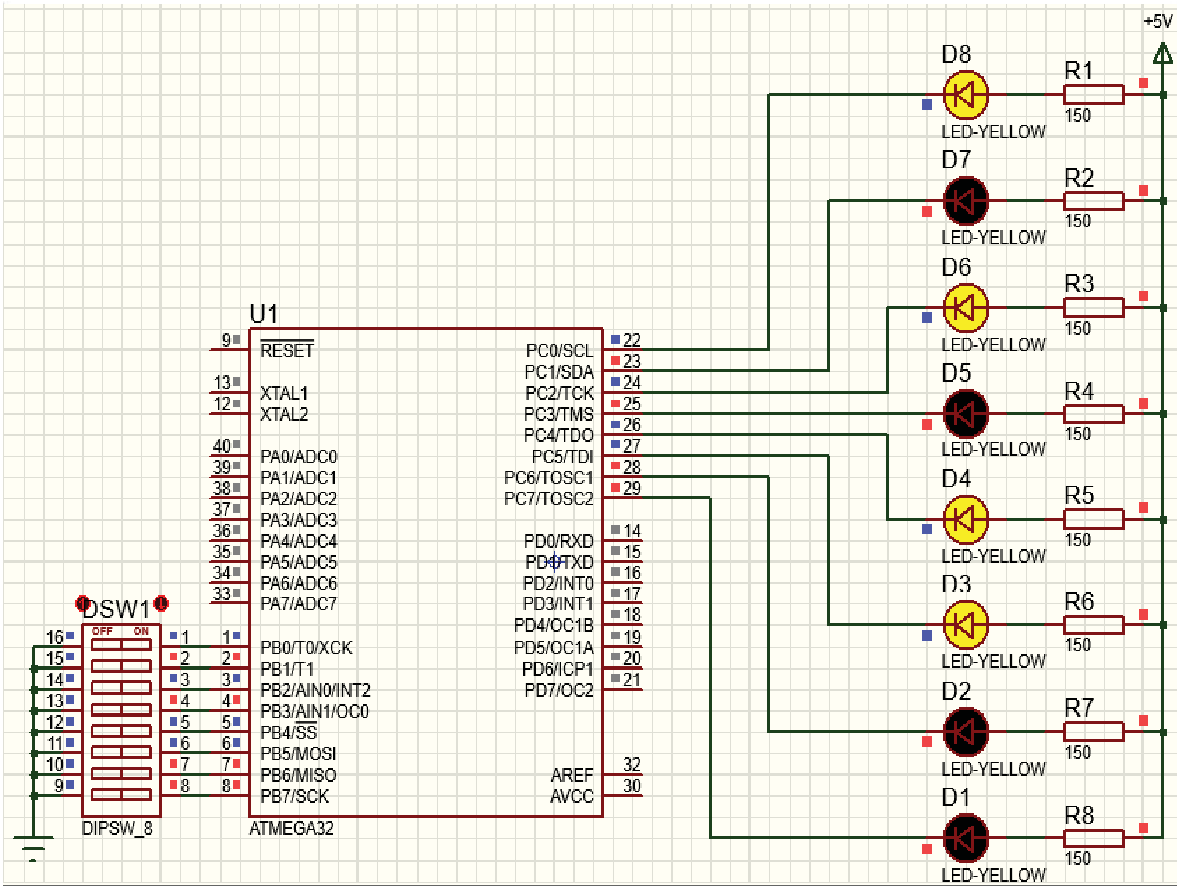
COM Rd

This instruction **complements** (inverts) **the contents of Rd** and places the **result** back **into the same register**.

Example

Send the value 01010101_2 into PORTC. Then invert it before sending it again to PORTC.

```
LDI    R16 , 0x55      ;R16 = 0x55
OUT    PORTC, R16      ;copy R16 to Port C (PC = 0x55)
COM    R16             ;complement R16        (R16 = 0xAA)
OUT    PORTC, R16      ;copy R16 to Port C (PC = 0xAA)
```



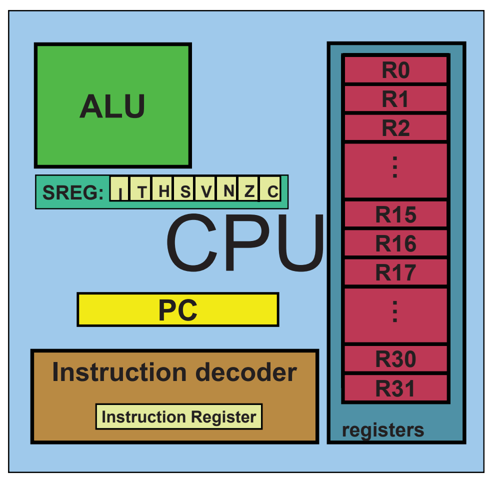
AVR Status Register (SREG)

Unleash your Creativity!

- The **status register** (SREG) in AVR is an 8-bit register. It **contains information about the state** of the processor.

It is also referred to as the **flag register**.

The bits C, Z, N, V, S, and H are called **conditional flags**, meaning that they indicate some conditions that result after an instruction is executed.



Bit	D7							D0
SREG	I	T	H	S	V	N	Z	C

C – Carry flag
Z – Zero flag
N – Negative flag
V – Overflow flag

S – Sign flag
H – Half carry
T – Bit copy storage
I – Global Interrupt Enable

AVR Status Register (SREG)

Unleash your Creativity!

Carry Flag (C): Bit 0

D7							D0
I	T	H	S	V	N	Z	C

This flag is **set** whenever there is a **carry out or borrow out from MSB (D7) after an 8bit addition or subtraction.**

Examples:

```
LDI   R19,0x01
LDI   R20,0xFF
ADD   R20, R19
```

R20	C
	X
1111 1111	X
1 0000 0000	1

Flag not affected (keeps the value corresponding to the **previous** arithmetical operation).

```
LDI   R19,0x01
LDI   R25,0x00
SUB   R25, R19
```

R25	C
	X
0000 0000	X
1 1111 1111	1

AVR Status Register (SREG)

Unleash your Creativity!

Zero Flag (Z): Bit 1

D7							D0
I	T	H	S	V	N	Z	C

The zero flag reflects the result of the last arithmetic or logic operation.

If the result is **zero**, then **Z = 1**.

If the result is **not zero**, then **Z = 0**.

Example:

	R20	Z
LDI R20, 0x05	0000 0101	X
DEC R20	0000 0100	0
DEC R20	0000 0011	0
DEC R20	0000 0010	0
DEC R20	0000 0001	0
DEC R20	0000 0000	1

result not 0

result = 0

Flag not affected (keeps the value corresponding to the previous arithmetical operation).

- INC + DEC affect N-Z-V-S

AVR Status Register (SREG)

Unleash your Creativity!

Negative flag (N): Bit 2

D7							D0
I	T	H	S	V	N	Z	C

Represent the **sign** of the last arithmetical or logical operation.

If the **MSB** (D7 bit) of the result is **zero** (**positive result**), then N = 0.

If the **MSB** (D7 bit) the result is **one** (**negative result**), then N = 1.

Examples:

		R20	N
LDI	R20,0x3F	0011 1111	X
INC	R20	0100 0000	0

Flag not affected (keeps the value corresponding to the previous arithmetical operation).

The processor doesn't knows if the result is to be interpreted as "**signed**" or "**unsigned**". **N flag is always generated**. It is the **programmer responsibility to check or not** N flag.

AVR Status Register (SREG)

Unleash your Creativity!

Overflow flag (V): Bit 3

D7							D0
I	T	H	S	V	N	Z	C

This flag is **set** whenever the **result** of a **signed number** operation **is too large**, causing the **high-order bit** to **overflow into the sign bit**.

$$\begin{array}{r} 01000000 = +64 \\ 01000001 = +65 \\ \hline 10000001 = \cancel{-127} \end{array} \qquad \begin{array}{r} 10000001 = -127 \\ 10000001 = -127 \\ \hline 100000010 = \cancel{+2} \end{array}$$

Wrong! The answer is incorrect
and **the sign bit has changed**.

In general, the **carry flag** is used to detect errors in **unsigned arithmetic operations** while the **overflow flag** is used to detect errors in **signed arithmetic operations**.

AVR Status Register (SREG)

D7							D0
I	T	H	S	V	N	Z	C

Sign flag (S): Bit 4

Sign flag is the result of Exclusive-ORing of N and V flags.

$$S = N \oplus V$$

V	N	S
0	0	0
0	1	1
1	0	1
1	1	0

Half carry flag (H): Bit 5

If there is a **carry from D3 to D4** during an ADD or SUB operation, this bit is **set**; otherwise, it is cleared.

D7	D6	D5	D4	D3	D2	D1	D0

		R20	H
LDI	R20,0x3F	0011 1111	X
INC	R20	0100 0000	1

This flag bit is used by instructions that perform BCD (binary coded decimal) arithmetic. In some microprocessors this is called the AC flag (Auxiliary Carry flag).

AVR Status Register (SREG)

Example: Show the status of the C, H, and Z flags after the addition of 0x38 and 0x2F in the following instructions:

```
LDI   R16, 0x38      ;R16 = 0x38
LDI   R17, 0x2F      ;R17 = 0x2F
ADD   R16, R17       ;add R17 to R16
```

Solution:

```
                 1
   $38       0011 1000
 + $2F       0010 1111
   $67       0110 0111        R16 = 0x67
```

C = 0 because there is no carry beyond the D7 bit.
H = 1 because there is a carry from the D3 to the D4 bit.
Z = 0 because the R16 (the result) has a value other than 0 after the addition.

AVR Status Register (SREG)

Unleash your Creativity!

Example: Show the status of the C, H, and Z flags after the addition of 0x9C and 0x64 in the following instructions:

```
LDI     R20, 0x9C
LDI     R21, 0x64
ADD     R20, R21        ;add R21 to R20
```

Solution:

```
                 1
    $9C      1001 1100
  + $64      0110 0100
   $100    1 0000 0000        R20 = 00
```

C = 1 because there is a carry beyond the D7 bit.
H = 1 because there is a carry from the D3 to the D4 bit.
Z = 1 because the R20 (the result) has a value 0 in it after the addition.

AVR Status Register (SREG)

Unleash your Creativity!

Example: Show the status of the C, H, and Z flags after the subtraction of 0x23 from 0xA5 in the following instructions:

```
LDI     R20, 0xA5
LDI     R21, 0x23
SUB     R20, R21        ;subtract R21 from R20
```

Solution:

```
    $A5      1010 0101
  - $23      0010 0011
    $82      1000 0010        R20 = $82
```

C = 0 because R21 is not bigger than R20 and there is no borrow from D8 bit.
Z = 0 because the R20 has a value other than 0 after the subtraction.
H = 0 because there is no borrow from D4 to D3.

AVR Status Register (SREG)

Unleash your Creativity!

Example: Show the status of the C, H, and Z flags after the subtraction of 0x73 from 0x52 in the following instructions:

```
LDI     R20, 0x52
LDI     R21, 0x73
SUB     R20, R21        ;subtract R21 from R20
```

Solution:

	$52	**0101 0010**	
-	**$73**	**0111 0011**	
	$DF	**1101 1111**	**R20 = $DF**

C = 1 because R21 is bigger than R20 and there is a borrow from D8 bit.
Z = 0 because the R20 has a value other than zero after the subtraction.
H = 1 because there is a borrow from D4 to D3.

AVR Status Register (SREG)

Unleash your Creativity!

Example: Show the status of the C, H, and Z flags after the subtraction of 0x9C from 0x9C in the following instructions:

```
LDI     R20, 0x9C
LDI     R21, 0x9C
SUB     R20, R21        ;subtract R21 from R20
```

Solution:

	$9C	**1001 1100**	
-	**$9C**	**1001 1100**	
	$00	**0000 0000**	**R20 = $00**

C = 0 because R21 is not bigger than R20 and there is no borrow from D8 bit.
Z = 1 because the R20 is zero after the subtraction.
H = 0 because there is no borrow from D4 to D3.

Flag bits and decision making

Unleash your Creativity!

How these flag bits are useful to make decision?

Some instructions in AVR make a **conditional jump** (branch) **based on the status of the flag bits.**

Ex:

SUB R17,R30

Table 2-5: AVR Branch (Jump) Instructions Using Flag Bits

Instruction	Action
BRLO	Branch if C = 1
BRSH	Branch if C = 0
BREQ	Branch if Z = 1
BRNE	Branch if Z = 0
BRMI	Branch if N = 1
BRPL	Branch if N = 0
BRVS	Branch if V = 1
BRVC	Branch if V = 0

One jump instructions

Branch if Lower.
Branch if Same or Higher.
Branch if Equal.
Branch if Not Equal.
Branch if Minus.
Branch if Plus.
Branch if Overflow Flag is Set.
Branch if Overflow Flag is Cleared.

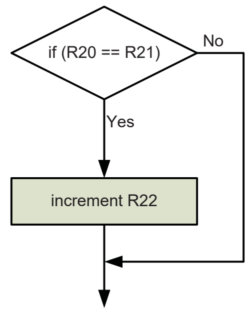
AVR Conditional Jump instructions

Unleash your Creativity!

Instruction	Abbreviation of	Comment
BREQ *lbl*	Branch if Equal	Jump to location *lbl* if Z = 1,
BRNE *lbl*	Branch if Not Equal	Jump if Z = 0, to location *lbl*
BRCS *lbl* BRLO *lbl*	Branch if Carry Set Branch if Lower	Jump to location *lbl*, if C = 1
BRCC *lbl* BRSH *lbl*	Branch if Carry Cleared Branch if Same or Higher	Jump to location *lbl*, if C = 0
BRMI *lbl*	Branch if Minus	Jump to location lbl, if N = 1
BRPL *lbl*	Branch if Plus	Jump if N = 0
BRGE *lbl*	Branch if Greater or Equal	Jump if S = 0
BRLT *lbl*	Branch if Less Than	Jump if S = 1
BRHS *lbl*	Branch if Half Carry Set	If H = 1 then jump to *lbl*
BRHC lbl	Branch if Half Carry Cleared	if H = 0 then jump to lbl
BRTS	Branch if T flag Set	If T = 1 then jump to lbl
BRTC	Branch if T flag Cleared	If T = 0 then jump to lbl
BRIS	Branch if I flag set	If I = 1 then jump to lbl
BRIC	Branch if I flag cleared	If I = 0 then jump to lbl

Example 1

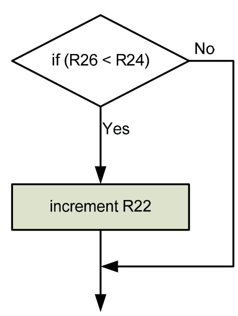
- Write a program to **increases R22**, **if** R20 = R21.
- **Solution**:




```
    SUB R20,R21      ;Z will be set if R20 == R21
    BRNE  NEXT       ;if Not Equal jump to next
    INC R22
NEXT:
```


Example 2

- Write a program **that increases R22**, **if** R26 < R24 .
- **Solution**:




```
    SUB R26,R24      ; C will be set if R26 < R24
                     ; C =0 if R26 >= R24
    BRCC    L1       ;if Carry cleared jump to L1
    INC R22
L1:
```

Example 3

Unleash your Creativity!

- Write a program that **increases R22**, **if** R26 >= R24.

- **Solution**:

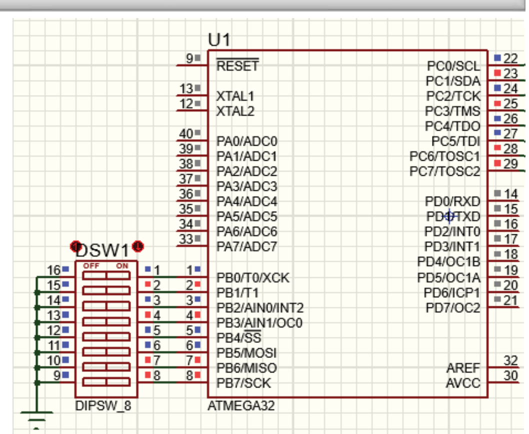
```
    SUB R26,R24        ;C will be set if R26 < R24
                        ; C =0 if R26 >= R24
    BRCS    L1         ;if Carry set jump to L1
    INC R22
L1:
```


Example 4

Unleash your Creativity!

- Write a program to **stay in the loop testing PINB** until it has a value other than zero.

- **Solution**:




```
OVER:  IN  R20, PINB       ; read PINB to R20
       TST  R20            ; set the flags according to R20
       BREQ  OVER          ; jump if R20 is zero (Z=1)
```

TST instruction: **Examine** a register **and set** the flags (**Z, N, V & S**) according to the contents of the register **without performing any arithmetic instruction.**

Example 5

Unleash your Creativity!

- Write a program to determine if RAM location 0x200 contains the value 0. If so, put 0x55 into it.
- **Solution**:

```
        .EQU  MYLOC=0x200
        LDS  R30, MYLOC
        TST  R30                ;set the flags Z & N
                                ;(Z=1 if R30 has zero value)
        BRNE  NEXT              ;branch if R30 is not zero (Z=0)
        LDI  R30, 0x55          ;put 0x55 if R30 has zero value
        STS  MYLOC,R30          ;and store a copy to loc $200
NEXT:   ....
```


Example 6

Unleash your Creativity!

Find the sum of the values 0x79, 0xF5, and 0xE2. Put the sum into R20 (low byte) and R21 (high byte).

	R21 (high byte)	R20 (low byte)
At first	$0	$00
Before LDI R16,0xF5	$0	$79
Before LDI R16,0xE2	$1	$6E
At the end	$2	$50

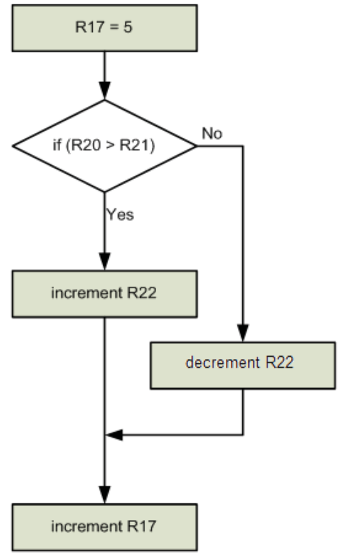
Solution:

```
        LDI  R21, 0         ;clear high byte (R21=0)
        LDI  R20, 0x79      ;clear low byte (R20 = 0)
        LDI  R16, 0xF5
        ADD  R20, R16       ;R20 = 0x79 + 0xF5 = 0x6E and C = 1
        BRCC  next          ;branch if C = 0
        INC  R21            ;C = 1, increment (now high byte = 1)
next :  LDI  R16, 0xE2
        ADD  R20, R16       ;R20 = 0x6E + 0xE2 = 0x50 and C = 1
        BRSH  OVER          ;branch if C = 0
        INC  R21            ;C = 1, increment (now high byte = 2)
OVER:                       ;now low byte = 0x50, and high byte = 02
```

Example 7: IF and ELSE

Unleash your Creativity!

```
R17 = 5;
if (R21 < R20)
        R22++;
else
        R22--;
R17 ++;
```




```
            LDI     R17,5
            SUB     R21,R20
            BRCS    IF_YES
            DEC     R22
            JMP     NEXT
IF_YES:     INC     R22
NEXT:       INC     R17
```


Looping in AVR

Unleash your Creativity!

In the AVR, there are several ways to **repeat an operation many times**.

One way is to use a **decreasing counter** with **BRNE** instruction.

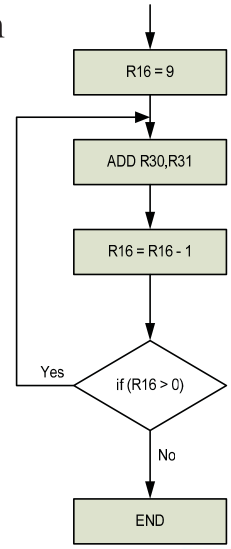
```
        LDI     Rn, number of repetitions.
BACK :  .........           ;start of the loop
        .........           ;body of the loop
        .........           ;body of the loop
        DEC     Rn          ;decrease the counter Rn (Z = 1  when  Rn = 0)
        BRNE    BACK        ; Branch to (BACK) if Z = 0   i.e Repeat the loop
```

Prior to the start of the loop, the **Rn** is loaded by the needed number of repetitions.

Looping in AVR

- Write a program that executes the instruction "ADD R30,R31" **9 times**.
- **Solution:**

```
        LDI     R16 , 9     ;R16 = 9
L1:
        ADD     R30 , R31
        DEC     R16         ;R16 = R16 - 1
        BRNE    L1          ; if Z = 0 jump to L1
        .....
```



Note that the last loop can be repeated a maximum of 255 times. (why?)

Looping in AVR

Example: Write a program to:
- (a) clear R20.
- (b) add 3 to R20 ten times.
- (c) send the sum to PORTB.

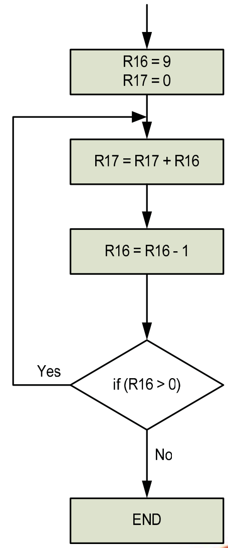
Solution:

```
        LDI  R16, 10        ;R16 = 10 (decimal) for counter
        LDI  R20, 0         ;R20 = 0
        LDI  R21, 3         ;R21 = 3
AGAIN:  ADD  R20, R21       ;add 03 to R20 (R20 = sum)
        DEC  R16            ;decrement R16 (counter)
        BRNE AGAIN          ;repeat until COUNT = 0
        OUT  PORTB,R20      ;send sum to PORTB
        ......
```

Looping in AVR

- Write a program that calculates the result of 9+8+7+…+1
- **Solution**:

```
        LDI     R16, 9          ;R16 = 9
        LDI     R17, 0          ;R17 = 0
L1:     ADD     R17, R16        ;R17 = R17 + R16
        DEC     R16             ;R16 = R16 - 1
        BRNE    L1              ;if Z = 0
L2:     RJMP    L2              ;Wait here forever
```

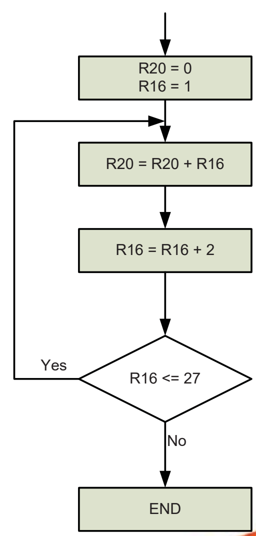


Looping in AVR

- Write a program that calculates the odd numbers 1+3+5+…+27

Solution:

```
      LDI R20,0
      LDI R16,1
L1:   ADD R20,R16
      LDI R17,2
      ADD R16,R17      ;R16 = R16 + 2
      LDI R17,27       ;R17 = 27
      SUB R17,R16
      BRCC L1          ;if R16 <= 27 jump L1
```



Looping in AVR

Loop inside a loop

Example: Write a program to:

- (a) load the "PORTC" register with the value 0x55.
- (b) Complement "PORTC" **700** times.

Solution:

```
        LDI  R16, 0x55          ;R16 = 0x55
        OUT  PORTC, R16         ;PORTC = 0x55
        LDI  R20, 10            ;load 10 into R20 (outer loop count)
LOP_2:  LDI  R21, 70            ;load 70 into R21 (inner loop count)
LOP_1:  COM  R16                ;complement R16
        OUT  PORTC, R16         ;load PORTC SFR with the complemented value
        DEC  R21                ;dec R21 (inner loop)
        BRNE LOP_1              ;repeat it 70 times
        DEC  R20                ;dec R20 (outer loop)
        BRNE LOP_2              ;repeat it 10 times
```


Calling a Function

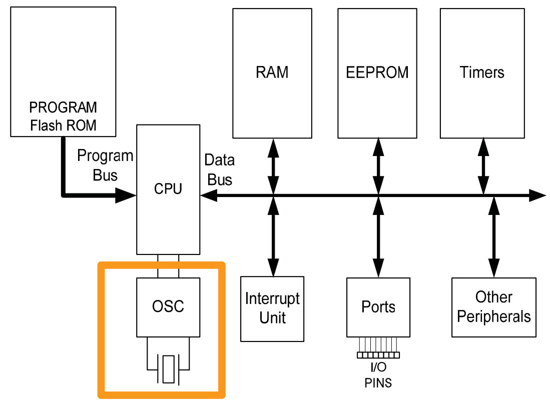
Example Toggle all the bits of Port B **every 1sec** by sending to it the values $55 and $AA continuously.

```
BACK:  LDI   R16,0x55        ;load R16 with 0x55
       OUT   PORTB,R16       ;send 55H to port B
       CALL  DELAY_1sec      ;time delay
       LDI   R16,0xAA        ;load R16 with 0xAA
       OUT   PORTB,R16       ;send 0xAA to port B
       CALL  DELAY_1sec      ;time delay
       RJMP  BACK            ;keep doing this indefinitely

DELAY_1sec:
             ....
       RET                   ;return to caller
```

Time delay

Unleash your Creativity!



		machine cycle
LDI	R16, 19	1
LDI	R20, 95	1
LDI	R21, 5	1
ADD	R16, R20	1
ADD	R16, R21	1
		5

1 MHZ ➔ instruction cycle =1μsec ➔ Delay = 5 x 1μs

8 MHZ ➔ instruction cycle =125nsec ➔ Delay = 5 x 125ns

10 MHZ ➔ instruction cycle =100nsec ➔ Delay = 5 x 100ns

Time delay

Unleash your Creativity!

Delay 0.25 sec

NOP = No operation just wastes clock cycles

			machine cycle		
	LDI	R17, 200	1		
L1:	LDI	R16, 250	1		x200
L2:	NOP		1	x 250	x200
	NOP		1	x 250	x200
	DEC	R16	1	x 250	x200
	BRNE	L2	2	x 250	x200
	DEC	R17	1		x200
	BRNE	L1	2		x200

Crystal frequency =1 MHZ ➔ instruction cycle =1μsec

Delay = {1+ [1+ (1+1+1+2)x250 +1+2] x200 } x 1μs=250.8 msec

Calling a Function

Example Toggle all bits of Port B by sending to it the values $55 and $AA continuously. Put a time delay between each transmitting of data to Port B.

```
        LDI   R16,HIGH(RAMEND)   ;load SPH
        OUT   SPH,R16
        LDI   R16,LOW(RAMEND)    ;load SPL
        OUT   SPL,R16
BACK:   LDI   R16,0x55           ;load R16 with 0x55
        OUT   PORTB,R16          ;send 55H to port B
        CALL  DELAY              ;time delay
        LDI   R16,0xAA           ;load R16 with 0xAA
        OUT   PORTB,R16          ;send 0xAA to port B
        CALL  DELAY              ;time delay
        RJMP  BACK               ;keep doing this indefinitely
DELAY:
        LDI   R20,0xFF           ;R20 = 255,the counter
AGAIN:
        NOP                      ;no operation wastes clock cycles
        NOP
        DEC   R20
        BRNE  AGAIN              ;repeat until R20 becomes 0
        RET                      ;return to caller
```

Calling many subroutines from the main program

```
MAIN:
        CALL  Delay

        CALL  SUBR_1
                     ; end of MAIN

Delay:
        ....
        ....
        RET
                     ; end of Delay subroutine

SUBR_1:
        ....
        ....
        RET
                     ; end of subroutine 1
```

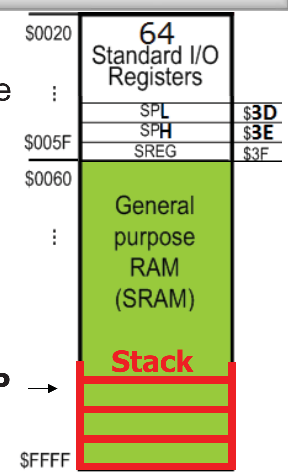
how the CPU knows **where to resume** when it returns from the called subroutine**?**

CPU store the needed information in **Stack memory**.

Stack memory and Stack Pointer SP in AVR

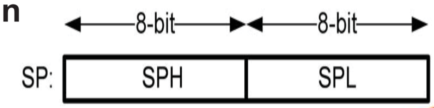
➢ **Stack** memory is a section of RAM (usually defined by the programmer at end of SRAM)

➢ It's **mainly used** to store some needed information **for calls** and **interrupts**.



❖ Stack location in SRAM?

➢ **SP (stack pointer) register**, which is implemented in I/O memory, is used to **point the stack section**



Note:

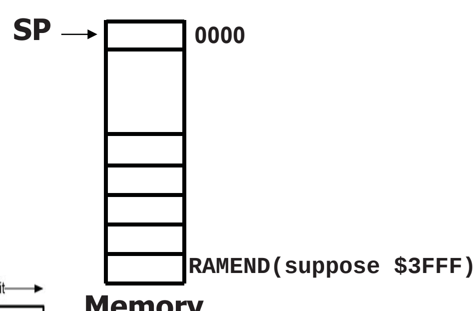
- The **S**tack **P**ointer (**SP**) points to the **Top Of Stack** (**TOS**).
- The stack is **LIFO memory (Last-In-First-Out)**.

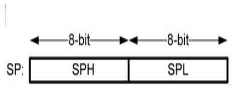
Initializing the stack pointer

➢ When the AVR is powered up, the SP register contains the value 0, therefore, we must initialize the SP at the beginning of the program.

➢ It is common to initialize the SP to **"RAMEND"** which represents the **address** of the last RAM location.

R20:	$10	**R22:**	$30
R21:	$20	**R0:**	$00





Address	Code
	ORG 0
0000	LDI R16,HIGH(RAMEND)
0001	OUT SPH,R16
0002	LDI R16,LOW(RAMEND)
0003	OUT SPL,R16
0004	LDI R20,0x10
0005	LDI R21, 0x20
0006	LDI R22,0x30
0007	PUSH $10
0008	PUSH $20
0009	PUSH $30
000A	POP R21
000B	POP R0
000C	POP R20
000D	L1: RJMP L1

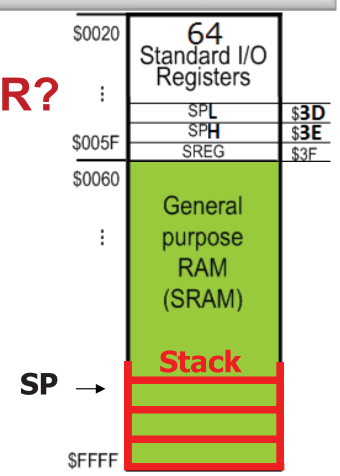
PUSH and POP instruction

❖How stacks are accessed in the AVR?

- Storing information on the stack is called a **PUSH**.

PUSH Rs ; Push the register Rs onto stack memory.

;Rs can be any of the GPRs (R0-R31).



1) `[SP] = Rs` (**Content of** the register **Rs** is saved in the memory location where the SP points **)**

2) `SP = SP - 1` **(**SP is decremented by one**)**

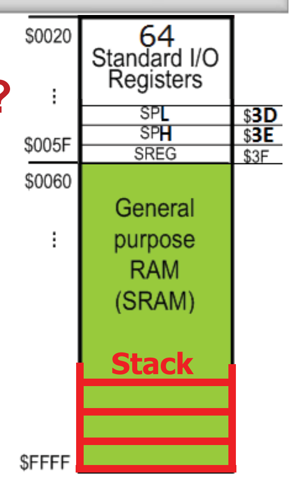
PUSH and POP instruction

❖How stacks are accessed in the AVR?

- Loading of stack content back into one register is called a **POP**. (opposite process of **pushing**).

POP Rd ; retrieve the data from stack memory back into Rd

;Rd can be any of GPRs (R0-R31).



1) `Rd = [SP]` (**Content of** the **top location in the stack** is copied back into the Rd**)**

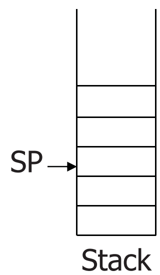
2) `SP = SP + 1` **(**SP is incremented by one**)**

Calling a Function

- To execute a call:
  - **1)** Address of the next instruction is saved into stack
  - **2)** PC is loaded with the appropriate value

Machine code:

940E	000A
opCode	operand



PC: 0009

Address	Code
0000	LDI R16,HIGH(RAMEND)
0001	OUT SPH,R16
0002	LDI R16,LOW(RAMEND)
0003	OUT SPL,R16
0004	LDI R20,15
0005	LDI R21,5
0006	**CALL** FUNC_NAME
00 08	INC R20
0009	L1: RJMP L1
000A	FUNC_NAME:
000A	ADD R20,R21
000B	SUBI R20,3
000C	**RET**
000D	

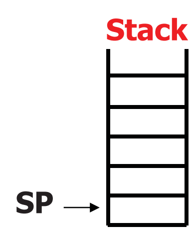
CALL, RET instructions and the role of the stack

➢ Call instruction:

1- Push the address of the next instruction onto the stack,

2- Decrement the stack pointer.

3- Transfers control to that subroutine.




```
MAIN:      :
           :
           CALL   Delay
           :
           :              ; end of MAIN
Delay:     ....
           ....
           RET            ; end of subroutine 1
```

➢ RET instruction:

1- Copy back the top 2 locations of the stack to the Program Counter PC

(they should contain the address of the instruction below the CALL)

2- Increment the stack pointer.

Some Instructions Using a GPR as Operand

Unleash your Creativity!

Table 2-3: Some Instructions Using a GPR as Operand

Instruction		
CLR	Rd	Clear Register Rd
INC	Rd	Increment Rd
DEC	Rd	Decrement Rd
COM	Rd	One's Complement Rd
NEG	Rd	Negative (two's complement) Rd
ROL	Rd	Rotate left Rd through carry
ROR	Rd	Rotate right Rd through carry
LSL	Rd	Logical Shift Left Rd
LSR	Rd	Logical Shift Right Rd
ASR	Rd	Arithmetic Shift Right Rd
SWAP	Rd	Swap nibbles in Rd

These instructions operate on a single GPR register and place the result in the same register.

ALU Instructions Using Two GPRs

Unleash your Creativity!

Table 2-2: ALU Instructions Using Two GPRs

Instruction		
ADD	Rd, Rr	ADD Rd and Rr
ADC	Rd, Rr	ADD Rd and Rr with Carry
AND	Rd, Rr	AND Rd with Rr
EOR	Rd, Rr	Exclusive OR Rd with Rr
OR	Rd, Rr	OR Rd with Rr
SBC	Rd, Rr	Subtract Rr from Rd with carry
SUB	Rd, Rr	Subtract Rr from Rd without carry

Rd and Rr can be any of the GPRs.

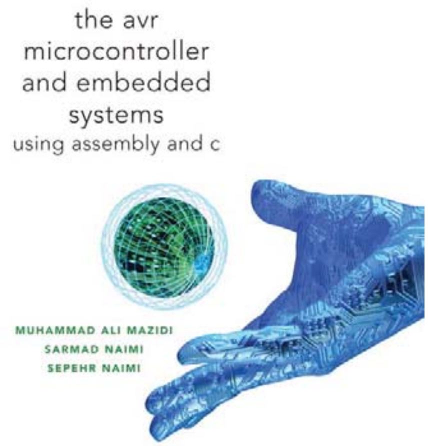
These instructions operate on two GPR registers of source (Rr) and destination (rd) and then place the result in the destination register (Rd)

References:

Unleash your Creativity!

For further reading students are referred to:

- The AVR Microcontroller and Embedded Systems: Using Assembly and C, Prentice Hall, 2011.



END