Atmega328P

Registers

- General Purpose Registers (r0 - r31)
  - Start using at r16
  - X = r27:r26
  - Y = r29:r28
  - Z = r31:r30
- I/O Registers
  - PORTx / PINx / DDRx
- Not directly accessible Registers
  - SREG
  - Stack Pointer

PORTx / PINx / DDRx

- Set input : 0x00
- Set output : 0xff

register bits → pin function ↓	DDRx.n	PORTx.n	PINx.n
tri stated input	0	0	read data bit x = PINx.n;
pull-up input	0	1	read data bit x = PINx.n;
output	1	write data bit PORTx.n = x;	n/a

Loading Instruction

Some simple instructions

- Loading values into the general purpose registers

LDI Rd, k ; Rd = k

Example:

```
LDI R16,53          ; R16 = 53
LDI R19,$27         ; R23 = 0x27, '$' prepends a numerical
                      hexadecimal value
LDI R23,0x27        ; R23 = 0x27
LDI R23,0b11101100  ; R23 = 0b11101100
```

Atmega328P

Loading Instruction

Some simple instructions

- Copy values from register to another register

  MOV Rd, Rs ; Rd ← Rs

  Example:

  MOV R16,R0 ; Copy R0 to R16

Arithmetic Operation

Some simple instructions

- 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*

  Example:

  ADD R25, R9 ; R25 = R25+9

  ADD R17,R30 ; R17 = R17 + R30

Some simple instructions

- MUL (Multiply Unsigned)

  This instruction performs 8-bit 8-bit 16-bit unsigned multiplication.

  MUL Rd,Rs *; R1:R2 ← Rd x Rs*

  Example:

  MUL R5,R4 ; Multiply unsigned R5 and R4

  MOVW R4,R0 ; Copy result back in R5:R4

Some simple instructions

- AND (Logical AND)

  Performs the logical AND between the contents of register Rd and register Rs

  AND Rd,Rs *; Rd ← Rd and Rs*

  Example:

  AND R2,R3 ; Bitwise and R2 and R3, result in R2

  LDI R16,0x1 ; Set bitmask 0000 0001 in R16

  and R2,R16 ; Isolate bit 0 in r2

*** Division Operation ***

Some simple instructions

- ROR (Rotate Right through Carry)
- ASR (Arithmetic Shift Right)
- LSR (Logical Shift Right)

Shift right for 1 bit means dividing that value by 2

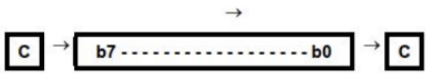
Atmega328P

Shifting

Some simple instructions

- **ROR (Rotate Right through Carry)**

  Shifts all bits in Rd one place to the right. The C Flag is shifted into bit 7 of Rd. Bit 0 is shifted into the C Flag.



Some simple instructions

- **ASR (Arithmetic Shift Right)**

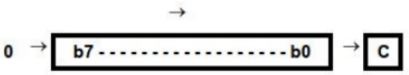
  Shifts all bits in Rd one place to the right. Bit 7 is held as constant. Bit 0 is loaded into the C Flag of the SREG

  Or in other words, it is used to divide a signed value by two without changing its sign.

Some simple instructions

- **LSR (Logical Shift Right)**

  Shifts all bits in Rd one place to the right. Bit 7 is cleared. Bit 0 is loaded into the C Flag



Directives

Assembler-Directives: Header Section

Directive	Description
.DEVICE	Defines the type of the target processor and the applicable set of instructions (illegal instructions for that type trigger an error message, syntax: .DEVICE AT90S8515)
.DEF	Defines a synonym for a register (e.g. .DEF MyReg = R16). Fixes the value of a symbole (later redefinition is not possible)
.EQU	Defines a symbol and sets its value (later changes of this value remain possible, syntax: .EQU test = 1234567, internal storage of the value is 4-byte- Integer)
.SET	Work same as .DEF but the value of the label can be changed later
.INCLUDE	Includes a file and assembles its content, just like its content would be part of the calling file (typical e.g. including the header file: .INCLUDE "C:\avrtools\appnotes\8515def.inc")

Atmega328P

Assembler-Directives: Code Section

Directive	Description
.CSEG	**Start** of the code segment (all that follows is assembled to the code segment and will go to the program space)
.DB	**Inserts one or more constant bytes in the code segment** (could be numbers from 0..255, an ASCII-character like 'c', a string like 'abcde' or a combination like 1,2,3, 'abc'. The number of inserted bytes must be even, otherwise an additional zero byte will be inserted by the assembler.)
.DW	Insert a **binary word** in the code segment (e.g. produces a table within the code)
.LISTMAC	Macros will be listed in the .LST-file. (Default is that macros are not listed)
.MACRO	Beginning of a macro (no code will be produced, call of the macro later produces code, syntax: .MACRO macro-name parameters, calling by: macroname parameters)
.ENDMACRO	End of the macro

Assembler-Directives: EEPROM Section

Directive	Description
.ESEG	Assemble to the EEPROM-segment (the code produced will go to the EEPROM section, the code produces an .EEP-file)
.DB	Inserts one or more constant bytes in the EEPROM segment (could be numbers from 0..255, an ASCII-character like 'c', a string like 'abcde' or a combination like 1,2,3,'abc'.)
.DW	Inserts a binary word to the EEPROM segment (the lower byte goes to the next address, the higher byte follows on the incremented address)

Assembler-Directives: SRAM Section

Directive	Description
.DSEG	Assemble to the data segment (here only BYTE directives and labels are valid, during assembly only the labels are used)
.BYTE	Reserves one or more bytes space in the data segment (only used to produce correct labels, does not insert any values)

Assembler-Directives: Everywhere

Directive	Description
.ORG	**Defines the address** within the respective segment, where the assembler assembles to (e.g. .ORG 0x0000)
.LIST	Switches the listing to the .LST-file on (the assembled code will be listed in a readable text file .LST)
.NOLIST	Switches the output to the .LST-file off, suppresses listing.
.INCLUDE	Inserts the content of another source code file, as if its content would be part of the source file (typical e.g. including the header file: .INCLUDE "C:\avrtools\appnotes\8515def.inc")
.EXIT	End of the assembler-source code (stops the assembling process)

Atmega328P

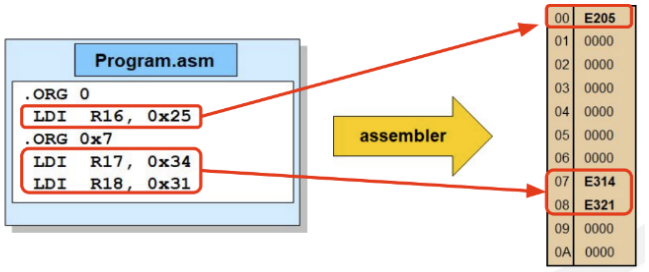
- .EQU name = value

```
.EQU COUNT = 0x25
LDI   R21, COUNT          ;R21 = 0x25
LDI   R22, COUNT + 3      ;R22 = 0x28
```

- .SET name = value

```
.SET COUNT = 0x25
LDI   R21, COUNT          ;R21 = 0x25
LDI   R22, COUNT + 3      ;R22 = 0x28
.SET COUNT = 0x19
LDI   R21, COUNT          ;R21 = 0x19
```

- .ORG address



Code structure

```
;
; *********************************************
; * [Add Project title here]                  *
; * [Add more info on software version here]  *
; * (C)20xx by [Add Copyright Info here]      *
; *********************************************
;
; Included header file for target AVR type
.NOLIST
.INCLUDE "tn13def.inc" ; Header for ATTINY13
.LIST
;
; =============================================
;   H A R D W A R E   I N F O R M A T I O N
; =============================================
;
; [Add all hardware information here]
;
; =============================================
;      P O R T S   A N D   P I N S
; =============================================
;
; [Add names for hardware ports and pins here]
; Format: .EQU Controlportout = PORTA
;         .EQU Controlportin = PINA
;         .EQU LedOutputPin = PORTA2
;
```

```
; =============================================
;   H A R D W A R E   I N F O R M A T I O N
; =============================================
;
; [Add all hardware information here]
;
; =============================================
;      P O R T S   A N D   P I N S
; =============================================
;
; [Add names for hardware ports and pins here]
; Format: .EQU Controlportout = PORTA
;         .EQU Controlportin = PINA
;         .EQU LedOutputPin = PORTA2
;
; =============================================
;    C O N S T A N T S   T O   C H A N G E
; =============================================
;
; [Add all constants here that can be subject
;  to change by the user]
; Format: .EQU const = $ABCD
;
; =============================================
;  F I X + D E R I V E D   C O N S T A N T S
; =============================================
;
; [Add all constants here that are not subject
;  to change or calculated from constants]
; Format: .EQU const = $ABCD
;
```

```
; =============================================
;   R E G I S T E R   D E F I N I T I O N S
; =============================================
;
; [Add all register names here, include info on
;  all used registers without specific names]
; Format: .DEF rmp = R16
.DEF rmp = R16 ; Multipurpose register
;
; =============================================
;       S R A M   D E F I N I T I O N S
; =============================================
;
.DSEG
.ORG  0X0060
; Format: Label: .BYTE N ; reserve N Bytes from Label:
;
; =============================================
;   R E S E T   A N D   I N T   V E C T O R S
; =============================================
;
.CSEG
.ORG $0000
        rjmp Main ; Reset vector
        reti ; Int vector 1
        reti ; Int vector 2
        reti ; Int vector 3
        reti ; Int vector 4
        reti ; Int vector 5
        reti ; Int vector 6
        reti ; Int vector 7
        reti ; Int vector 8
        reti ; Int vector 9
;
; =============================================
;     I N T E R R U P T   S E R V I C E S
; =============================================
;
; [Add all interrupt service routines here]
;
; =============================================
;     M A I N     P R O G R A M     I N I T
; =============================================
;
Main:
; Init stack
        ldi rmp, LOW(RAMEND) ; Init LSB stack
        out SPL,rmp
; Init Port B
        ldi rmp,(1<<DDB2)|(1<<DDB1)|(1<<DDB0) ; Direction of Pc
        out DDRB,rmp
; [Add all other init routines here]
        ldi rmp,1<<SE ; enable sleep
        out MCUCR,rmp
```


Important Examples

Example

```
        ldi ZL, low(led_on)          ; load Z with address to call
        ldi ZH, high(led_on)
        icall                        ; call led_on

led_on:
        ldi r16, 0b11111110          ; this is where Z points at and therefore the
        out PortA, r16               ; address to call
        ret
```

shifting

Shifting and Rotating

Shifting and rotating registers is good for handling serial data.

Example shows rotating in a bit into r16 if a serial data stream is present at PinD, 0:

```
shift_in:           ;when datastream bit is valid, go here
    clc             ;clear carry flag
    sbic PinD, 0    ;if PinD, 0 is cleared skip next
    sec             ;set carry flag
    rol r16         ;shift in carry flag
```

rol / ror

C <- b7 <------ b0 <- C (Rotate left, rol)

C -> b7 ------> b0 -> C (Rotate right,ror)

lsl / lsr

C <- b7 <------ b0 <- 0 (Shift left, lsl)

0 -> b7 ------> b0 -> C (Shift right, lsr)

asr -> arithmetic shift right

Subroutine call example

```
.org 0x00
ldi SPL low(RAMEND)
ldi SPH, high(RAMEND)

rcall subrtn_1

.org 0x100
subrtn_1:
rcall subrtn_2
ret

.org 0x140
subrtn_2:
ret
```

	Stack value	SP value	Comment
layer 0:	---	SP = ???	Stack before init
layer 1:	---		
layer 2:	---		

Then the SP is set to RAMEND:

	Stack value	SP value	Comment
layer 0:	---	<-SP	Stack after init
layer 1:	---		
layer 2:	---		

Stack state after rcall subrtn_1:

	Stack value	SP value	Comment
layer 0:	0x01		return address
layer 1:	---	<- SP	SP=SP-1
layer 2:	---		

Stack state after rcall subrtn_2:

	Stack value	SP value	Comment
layer 0:	0x01		return address
layer 1:	0x0101		return address
layer 2:	---	<- SP	SP=SP-1

Example

```
    ldi ZL, low(led_on)          ; load Z with address to call
    ldi ZH, high(led_on)
    icall                        ; call led_on

led_on:
    ldi r16, 0b11111110          ; this is where Z points at and therefore the
    out PortA, r16               ; address to call
    ret
```

Bit Manipulation

Bit Manipulation

cbr and *sbr* clear or set one multiple bit(s) in a register. These instructions only work on registers r16 to r31

```
    sbr r16, (1<<5)+(1<<3)       ;set bits 5 and 3 in register 16
    cbr r16, 0x03                ;clear bits 1 and 0 in register 16
```

Math Operation

Adding and Subtracting

add is adding the registers to the first register and appropriate flags (without setting the carry flag) in the Status register are set.

```
    add r16, r17                 ;r16 = r16 + r17
```

Adding and Subtracting

adc is add instruction which is using the Carry flag of the previous operation to increase the result if it is set.

```
    adc r16, r2                  ;r16 = r16 + r2 + C
```

Adding and Subtracting

adiw is the only add instruction which takes a constant as an argument. It only works on the low bytes of register pairs (24, 26, 28, 30) and adds the constant to the pair.

```
adiw YH:YL, 3          ;YH:YL = YH:YL + 3
adiw r24,0x0A          ; Add 0x0A to r24:r25 (result = 0x100A)
adiw r25:24,1          ; Add 1 to r25:r24
```

Adding and Subtracting

The subtract instructions work like the add instructions.

- sub r16, r17 ;r16 = r16 - r17
- sbc r16, r2 ;r16 = r16 - r2 - C
- sbiw ZL, 5 ;ZL:ZH = ZL:ZH - 5
- subi r16, 30 ;r16 = r16 - 30
- sbci r16, 4 ;r16 = r16 - 4 - C

Multiplying

- mul r16, r17 ;r1:r0 = r16 * r17
- muls r16, r17 ;r1:r0 = r16(signed) * r17(signed)
- mulsu r16, r17 ;r1:r0 = r16(signed) * r17(unsigned)

Multiply byte math

Multiple byte maths

```
ldi r16, 1          ; load r16 with 1
ldi r17, 0          ; load r17 with 0 (r17:r16 = 1)
ldi r18, 255        ; load r18 with 255
ldi r19, 0          ; load r19 with 0 (r19:r18 = 255)
add r16, r18        ; add low bytes (= 256 => r16 = 0, C = 1)
adc r17, r19        ; r19:r18 = 255 add low bytes (= 256 => r16 = 0)
                    ; add high byte with carry (= 0 + 1 (from carry) = 1)
                    ;=> r17:r16 = 256
```

Conditional branch

Conditional branches are branches based on the micro's Status Register. If the result of a previous operation left a status (for example "Zero"), this can be used to jump to code handling this result

Example:

```
    subi r16, 5          ; r16 = r16 - 5
    breq r16_is_0        ; goto r16_is_0 if result is r16 = 0
    brlo r16_is_lower0   ; goto r16_is_lower0 if r16 was lower than 5
r16_is_greater5:         ; come here when r16 was higher than 5
```

Example:

```
    ldi r16, 5           ; load desired loop count into r16
Loop:                    ; loop label
    dec r16              ; decrease loop count
    brne Loop            ; if not equal (result=0), loop again
```

```
    clr r16              ; clear counter
Loop:                    ; loop label
    inc r16              ; increase loop count
    cpi r16, 5           ; compare to desired loop count
    brne Loop            ; if not reached, loop again
```

Case Structures

The case structure compares a value to various case values

```
in r16, UDR      ;get UART data
cpi r16, 0       ;compare with case_0 value (0)
breq case_0      ;if case_0, jump there
cpi r16, 1       ;compare with case_1 value (1)
breq case_1      ;if case_1, jump there
cpi r16, 2       ;...and so on
breq case_2
```

Atmega328P

Uncondition jmp

Jump tell the PC to be changed to a specific code address (aka. line of code).

- **rjmp (relative jump)**: This instruction performs a jump within a range of +/- 2k words of program memory
  - rjmp go_here
- **ijmp (indirect jump to Z):** This instruction performs a jump to the address pointed to by the Z index register pair. As Z is 16 bits wide, ijmp allows jumps within the lower 64k words range of code space
  - ldi ZL, low(go_there)
ldi ZH, high(go_there)
ijmp
- **jmp (jump):** , This instruction can be used to jump anywhere within the code space. The address operand of jmp can be as big as 22 bits, resulting in jumps of up to 4M words
  - jmp go_far

Call

Subroutine calls require a proper stack setup and use of the return instructions.

- **rcall (relative call subroutine):** rcall can reach addresses within +/- 2k words. When rcall is executed, the return address is pushed onto the stack
  - rcall my_subroutine
- **icall (indirect call to Z)**: The subroutine pointed to by the Z index register pair is called
  - ldi ZL, low(my_subroutine)
ldi ZH, high(my_subroutine)
icall

Subroutine calls require a proper stack setup and use of the return instructions.

- **call (all subroutine):** This instruction can reach the lower 64k words of code space. It works just like rcall (regarding the stack) and needs 2 words of code space.
  - call my_subroutine

Return

Return instructions have to placed at the end of any subroutine or interrupt service routine (ISR). . The return address is popped from the stack and program execution goes on from there.

- **ret :** This instruction is used after finish the subroutine function to return PC back to the address after call instruction.
- **reti:** This instruction is used after ISRs. Basically it works like ret, but it also sets the I Flag (Global Interrupt Enable Flag) in the status register

Loops

For Loops

It is a counting loop from zero up to the required number of iterations

```
        ldi r16, 0              ;clear counting register
loop:
        out PortB, r16          ;write counting register to PortB
        inc r16                 ;increase counter
        cpi r16, 10             ;compare counter with 10
        brne loop               ;if <>10, repeat
```

While Loops

The while()...do{}-loop checks if a certain test result is true and performs the loop instructions.

```
while1:                 ;while(PinD = 1) rcall port_is_1
        in R16, PinD    ;perform check by reading pin value and
        cpi r16, 1      ;comparing it to 1
        brne while1_end ;if not true, end the loop
        rcall port_is_1 ;rcall port_is_1
        rjmp while1     ;and repeat loop
while1_end:
```

Macro

Example

```
.macro ldi16                ;create a macro for loading two registers with a 16-bit immediate

ldi @0, low(@2)             ;load the first argument (@0) with the low byte of @2
ldi @1, high(@2)            ;same with second arg (@1) and high byte of @2

.endmacro                   ;end the macro definition

ldi16 r16, r17, 1024        ; r16 = 0x00 r17 = 0x04
```

Atmega328P

Prog mem

Program Memory Directives

Example

```
.org 0x0100              ;set program memory address counter to 0x0100
.db 128                  ;place the number 128 at low byte of address 0x0100 in program memory
.db low(1000)            ;place the low byte of 1000 at low byte of address 0x0101
.db 128, low(1000)       ;place 128 at the low byte and the low byte of 1000 at the high byte of address 0x0102 in program memory
.db "Hello World!", 0    ;place “Hello World!\0” at the address 0x0103 in program memory
```

SRAM

SRAM Directives

Example

```
array_5: .byte 5         ; array_5 is a 5-byte SRAM segment
my_word: .byte 2         ; and is followed by my_word
```

Def

Register and Constant Directives

Example

```
.def temp = r16
.equ max_byte = 255
.set counter = 1
.set counter = 2    ;can occur in the same piece of code and they're each valid until a new .set is found, so .set counter = 1 is overridden by .set counter = 2.
```

Clock is here

Coding Directives

Example

```
.def byte_max = 255
.def clock = 8000000
.exit                    ; Assembler stops assembling at this line
```

Arithmetic Instructions

- ADD - Addition
- ADC - Addition with Carry (D = D+ S+ carry)
- SUB - Subtraction
- SBC - Subtraction with Carry (D = D- S- carry)
- MUL - Multiplication
- MULS - Signed Multiplication
- MULSU - Signed and Unsigned Multiplication
- INC - Increase by 1
- DEC - Decrease by 1

Logic Instructions

- AND - And operation
- ANDI - And with constant
- OR - Or operation
- ORI - Or with constant
- EOR - EX-OR operation
- COM - 1's complement operation

Branch instructions

- RJMP/RCALL – Relative Jmp (+/-k)
- IJMP/ICALL – Indirect Jmp (Z Reg)
- RET/RETI – Return from call/interrupt
- CP* - Compare
- SB* - Skip if Bit in Register or I/O is set/clr
- BR* - Branch if condition is met

Data Transfer

- MOV – Move between registers
- LD/LDI – Load / Load Immediate
- ST/STI – Store / Store Immediate
- LPM – Load Program Memory
  – Hardwired to load R0 with (Z) in code.
- IN/OUT – In and Out Ports
- PUSH/POP – On and off stack

Data Transfer

- RET/RETI – Return from call/interrupt
- CP* - Compare
- SB* - Skip if Bit in Register or I/O is set/clr
- BR* - Branch if condition is met

Bit and Bit test

- SBI/CBI – Set / Clear Bit in register
- LSL/LSR – Logical Shift Left / Right
- ROL/ROR – Rotate Left / Right (thru Carry bit)
- ASR – Arithmetic Shift Right
- SWAP – Swap Nibbles
- BST/BLD – Bit Store / Load
- BSET/BCLR – Set / Clear Status Bits by number
- SE*/CL* - Set / Clear Status Bits by name

Others

- NOP – Do nothing for 1 cycle
- SLEEP – Sleep until reset or interrupted
- WDR – Watchdog Reset

Branching and condition

Conditional Jump in AVR

SREG: | I | T | H | S | V | N | Z | C |

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 that if R20 is equal to R21 then R22 increases.

Solution:

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

Conditional Jump in AVR

Example 3: IF and ELSE

Solution:

```
        LDI   R17,5
        SUB   R21, R20          ;C is set when R20>R21
        BRCC  ELSE_LABEL        ;jump to else if CF is cleared
        INC   R22
        JMP   NEXT
ELSE_LABEL:
        DEC   R22
NEXT:
        INC   R17
```

Atmega328P

Stack memory [stack grows with decreased value]

Stack Memory

PUSH Rr

[SP] = Rr

SP = SP - 1

POP Rd

SP = SP + 1

Rd = [SP]

i/o programming

PORTx / DDRx	0	1
0	high impedance	Out 0
1	pull-up	Out 1

More about jmp

There are 3 ways to provide the jump address:

- PC = operand ⇒ JMP
- PC = PC + operand ⇒ RJMP
- PC = Z register ⇒ IJMP

They are actually give you a same appearance

- JMP Label
- RJMP Label

There are 3 ways to provide the jump address:

- PC = operand ⇒ JMP
- PC = PC + operand ⇒ RJMP
- PC = Z register ⇒ IJMP

```
LDI  ZL, low(main)
LDI  ZH, high(main)
IJMP
```

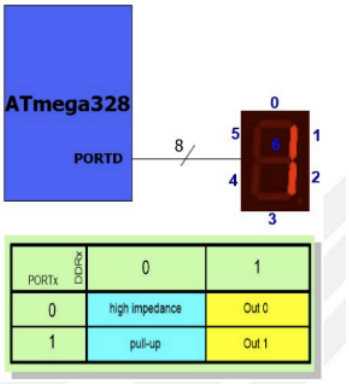
7 segment

Example 1

A 7-segment is connected to PORTD. Display 1 on the 7-segment.

DDRD: 1 1 1 1 1 1 1 1
PORTD: 0 0 0 0 0 1 1 0

```
    LDI   R20,0x06   ;R20 = 00000110 (binary)
    OUT   PORTD,R20  ;PORTD = R20
    LDI   R20,0xFF   ;R20 = 11111111 (binary)
    OUT   DDRD,R20   ;DDRD = R20
L1: RJMP L1
```



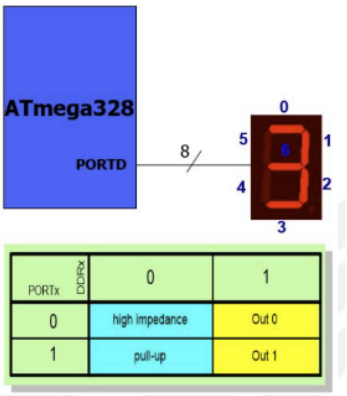
PORTx \ DDRx	0	1
0	high impedance	Out 0
1	pull-up	Out 1

Example 2

A 7-segment is connected to PORTD. Display 3 on the 7-segment.

DDRD: 1 1 1 1 1 1 1 1
PORTD: 0 1 0 0 1 1 1 1

```
    LDI   R20,0x4F   ;R20 = 01001111 (binary)
    OUT   PORTD,R20  ;PORTD = R20
    LDI   R20,0xFF   ;R20 = 11111111 (binary)
    OUT   DDRD,R20   ;DDRD = R20
L1: RJMP L1
```



PORTx \ DDRx	0	1
0	high impedance	Out 0
1	pull-up	Out 1

I/O bit manipulation programming

- SBI (Set Bit in IO register)

  SBI ioReg, bit ;ioReg.bit = 1

  Examples:

  SBI PORTD,0 ;PORTD.0 = 1

  SBI DDRC,5 ;DDRC.5 = 1

- CBI (Clear Bit in IO register)

  CBI ioReg, bit ;ioReg.bit = 0

  Examples:

  CBI PORTD,0 ;PORTD.0 = 0

  CBI DDRC,5 ;DDRC.5 = 0

Example 1

Write a program that toggles PORTB.4 continuously.

```
    SBI   DDRB,4
L1: SBI   PORTB,4
    CBI   PORTB,4
    RJMP  L1
```

Wtf is this i dont even understand

	Example
+	LDI R20,5+3 ;LDI R20,8
-	LDI R30,9-3 ;LDI R30,6
*	LDI R25,5*7 ;LDI R25,35
/	LDI R19,8/2 ;LDI R19,4

	Example
&	LDI R20,0x50&0x10 ;LDI R20,0x10
|	LDI R25,0x50|0x1 ;LDI R25,0x51
^	LDI R23,0x50^0x10 ;LDI R23,0x40

	Example
<<	LDI R16, 0x10<<1 ;LDI R16,0x20
>>	LDI R16, 0x8 >>2 ;LDI R16,0x2

Example 1

Write a program to copy the value $55 into memory locations $140 to $144

```
    LDI  R19,0x5      ;R19 = 5 (R19 for counter)
    LDI  R16,0x55     ;load R16 with value 0x55 (value to be copied)
    LDI  YL,0x40      ;load the low byte of Y with value 0x40
    LDI  YH,0x1       ;load the high byte of Y with value 0x1
L1: ST   Y,R16        ;copy R16 to memory location 0x140
    INC  YL           ;increment the low byte of Y
    DEC R19           ;decrement the counter
    BRNE  L1          ;loop until counter = zero
```

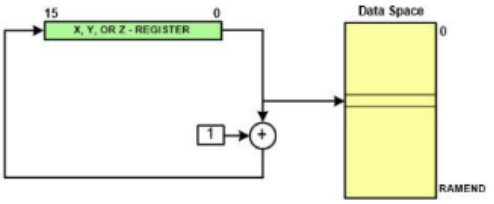
Bit Manipulation

cbr and *sbr* clear or set one multiple bit(s) in a register. These instructions only work on registers r16 to r31

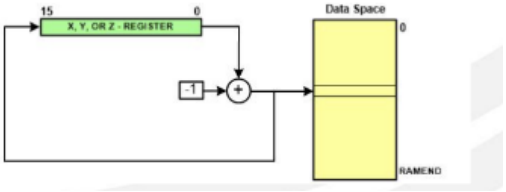
```
sbr r16, (1<<5)+(1<<3)      ;set bits 5 and 3 in register 16
cbr r16, 0x03               ;clear bits 1 and 0 in register 16
```

Auto-increment and Auto decrement

- Register indirect addressing with Post-increment
  - LD Rd, X+
    - LD R20,X+
  - ST X+, Rs
    - ST X+, R8



- Register indirect addressing with Pre-decrement
  - LD Rd, -X
    - LD R19,-X
  - ST –X,R31



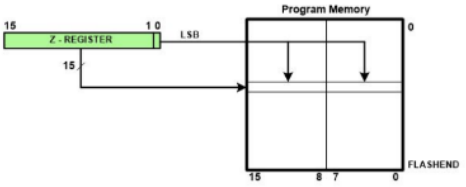
Read Data from Flash Memory

LPM Rd, Z ⇒ read from the low byte of address in Z

- LPM R15, Z

  Example:

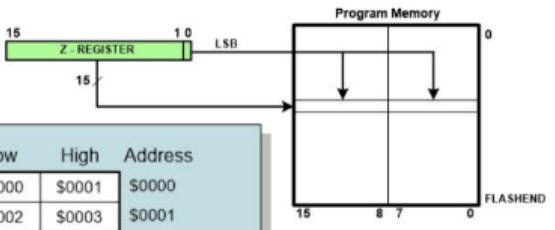
```
LDI R30,0x80
LDI R31,0
LPM R18,Z        ;read from the low byte of loc 0x40
```



LPM Rd, Z+ ⇒ read from the high byte of address in Z

- LPM R20,Z+

Read Data from Flash Memory



Low	High	Address
0000 0000 0000 0000	0000 0000 0000 0001	0000 0000 0000 0000
0000 0000 0000 0010	0000 0000 0000 0011	0000 0000 0000 0001
0000 0000 0000 0100	0000 0000 0000 0101	0000 0000 0000 0010
0000 0000 0000 0110	0000 0000 0000 0111	0000 0000 0000 0011
0000 0000 0000 1000	0000 0000 0000 1001	0000 0000 0000 0100
0000 0000 0000 1010	0000 0000 0000 1011	0000 0000 0000 0101
.....	.....	
1111 1111 1111 1100	1111 1111 1111 1101	0111 1111 1111 1110
1111 1111 1111 1110	1111 1111 1111 1111	0111 1111 1111 1111

Low	High	Address
$0000	$0001	$0000
$0002	$0003	$0001
$0004	$0005	$0002
$0006	$0007	$0003
$0008	$0009	$0004
$000A	$000B	$0005
$FFFC	$FFFD	$7FFE
$FFFE	$FFFF	$7FFF

Assembler macros can be powerful tools that allow you to create sequences of code as your single instruction

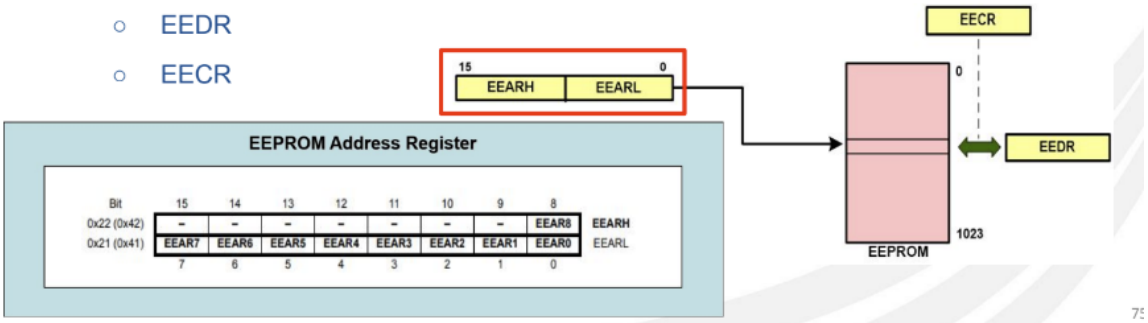
```
.MACRO   INITSTACK
         LDI   R16,HIGH(RAMEND)
         OUT  SPH,R16
         LDI   R16,LOW(RAMEND)
         OUT  SPL,R16
.ENDMACRO

INITSTACK
```

Assembler macros can be powerful tools that allow you to create sequences of code as your single instruction

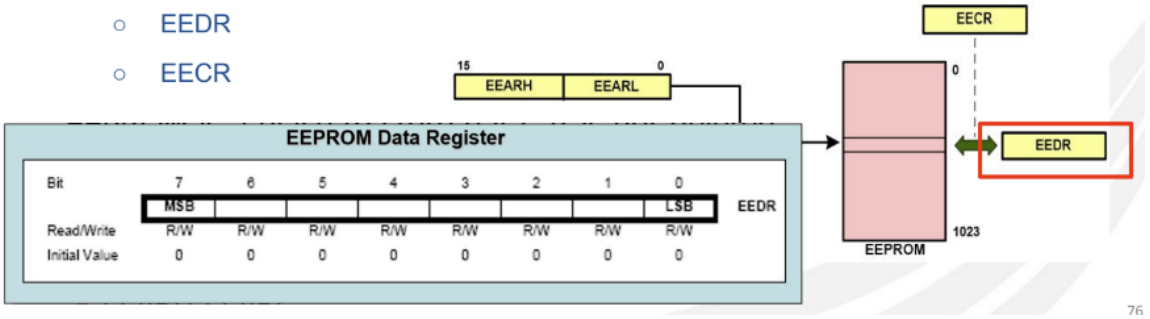
```
.MACRO LOADIO
       LDI   R20,@1
       OUT  @0,R20
.ENDMACRO

LOADIO  DDRB,0xFF
LOADIO  PORTB,0x55
```

eeprom

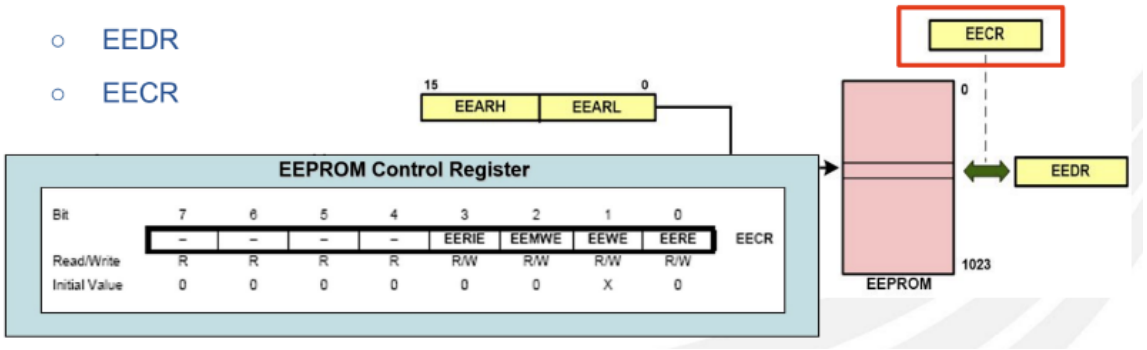
- EEPROM is a place to store data. It is not deleted when power is off
- ATmega32 has 1024 bytes of EEPROM
- In AVR 3 registers are dedicated to EEPROM
  - EEARH:EEARL
  - EEDR
  - EECR



- EEPROM is a place to store data. It is not deleted when power is off
- ATmega32 has 1024 bytes of EEPROM
- In AVR 3 registers are dedicated to EEPROM
  - EEARH:EEARL
  - EEDR
  - EECR



- EEPROM is a place to store data. It is not deleted when power is off
- ATmega32 has 1024 bytes of EEPROM
- In AVR 3 registers are dedicated to EEPROM
  - EEARH:EEARL
  - EEDR
  - EECR



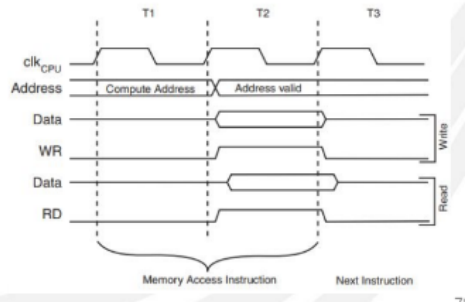
Reading / writing using eeprom

Reading from EEPROM

1. Wait until EEWE becomes zero.
2. Write new EEPROM address to EEAR
3. Set EERE to one.
4. Read EEPROM data from EEDR.

The following program reads the content of location 0x005F of EEPROM:

```
WAIT:  SBIC   EECR,EEWE   ;check EEWE to see if last write is finished
       RJMP   WAIT        ;wait more

       LDI    R18,0       ;load high byte of address to R18
       LDI    R17,0x5F    ;load low byte of address to R17
       OUT    EEARH, R18  ;load high byte of address to EEARH
       OUT    EEARL, R17  ;load low byte of address to EEARL
       SBI    EECR,EERE   ;set Read Enable to one
       IN     R16,EEDR    ;load EEPROM Data Register to R16
```

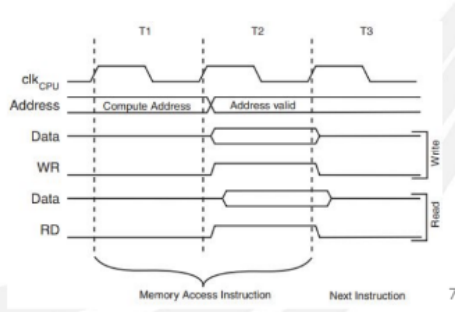


Writing to EEPROM

1. Wait until EEWE becomes zero.
2. Write new EEPROM address to EEAR (optional).
3. Write new EEPROM data to EEDR (optional).
4. Set EEMWE bit to one.
5. Within four clock cycles after setting EEMWE, set EEWE to one.

The program writes 'G' into location 0x005F of EEPROM:

```
WAIT:  SBIC   EECR,EEWE    ;check EEWE to see if last write is finished
       RJMP   WAIT         ;wait more
       LDI    R18,0        ;load high byte of address to R18
       LDI    R17,0x5F     ;load low byte of address to R17
       OUT    EEARH, R18   ;load high byte of address to EEARH
       OUT    EEARL, R17   ;load low byte of address to EEARL
       LDI    R16,'G'  ;load 'G' to R16
       OUT    EEDR,R16     ;load R16 to EEPROM Data Register
       SBI    EECR,EEMWE   ;set Master Write Enable to one
       SBI    EECR,EEWE    ;set Write Enable to one
```



Sreg

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

I: Global Interrupt Enable

1 = interrupts enable, when interrupt occurs I = 0

T: Bit Copy Storage

Space to store/load bit info for instructions BLD (Bit LoaD) and BST (Bit Store)

H: Half Carry Flag

Indicates a Half Carry in some arithmetic operations (carry of half bit) S: Sign Bit S = N xor V

V: Two's Complement Overflow Flag

Indicates whether a negative number overflows its maximum capability of 8-bit (Set to 1 when overflow)

N: Negative Flag

Indicates a negative result in an arithmetic or logic operation. (MSB of result)

Z: Zero Flag

Indicates a zero result in an arithmetic or logic operation (Z = 1 when zero result)

C: Carry Flag

Indicates a carry in an arithmetic or logic operation (Carry of MSB)

Atmega328P

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

```
        LDI     R17, 20
L1:     LDI     R16, 50
L2:     NOP
        NOP
        DEC     R16
        BRNE    L2
        DEC     R17
        BRNE    L1
```

machine cycle

1	
1	*20
1	*20 * 50
1	*20 * 50
1	*20 * 50
1/2	*20 * 50 (49 @ 2, 1 @ 1)
1	*20
1/2	*20 (19 @ 2, 1 @ 1)

Atmega328P

Stack

1) Upon reset, what is the value in the SP register? 0x29D
2) Upon pushing data onto the stack, the SP register is _decremented_ (decremented, incremented).
3) Upon popping data from the stack, the SP register is _incremented_ (decremented, incremented).
4) Can you change the value of the SP register? If yes, explain why you would want to do that.

Yes, we can. Generally, the stack pointer is used to point to the location in the memory where last element was added. By changing the stack pointer value you can indicate the specific location of the last item you add when you want to manipulate the stack memory

Stack pointer setup

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 R20
0008	PUSH R21
0009	PUSH R22
000A	POP R21
000B	POP R0
000C	POP R20
000D	L1: RJMP L1