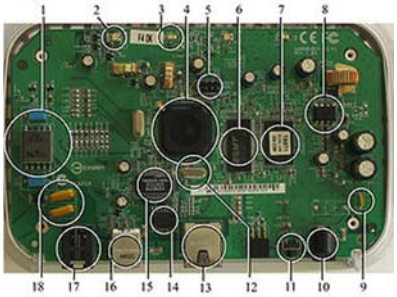
Embedded Systems

AVR Assembly Language Programming:
II) AVR instruction set



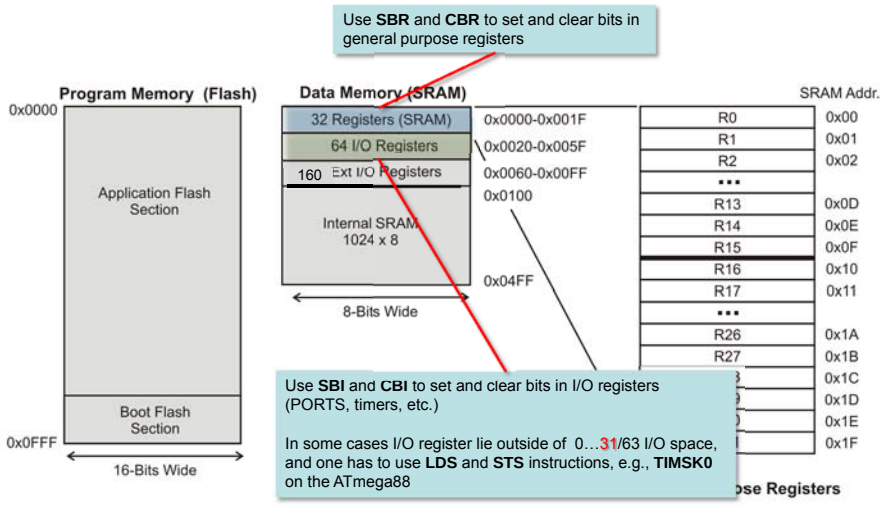
Inside an ASDL Modem/Router

AVR, ATmega88PA, ATtiny45, etc.

- **We will cover most of the AVR instructions**
- **Relevant Atmel documentation are under “Resources” on class website**

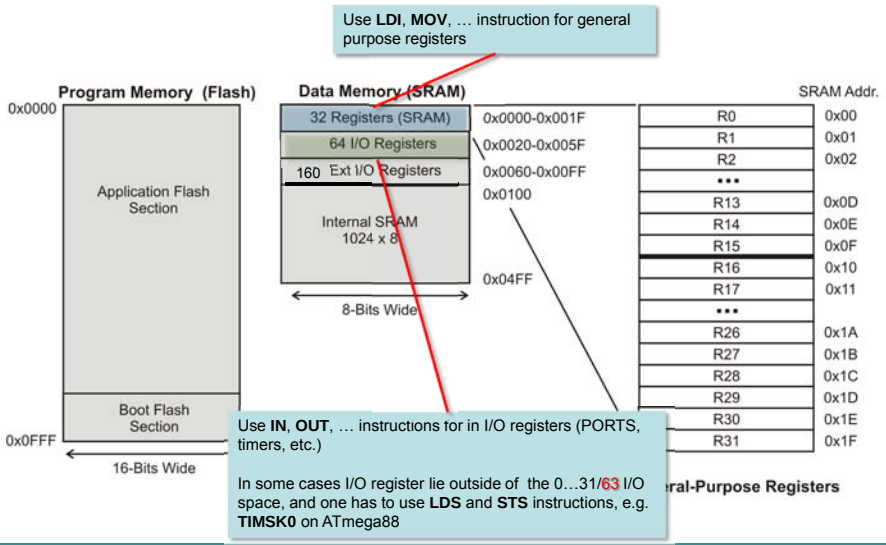
ATmega88 Memory Map

Some instructions are specific to certain memory areas



ATmega88 Memory Map

Some instructions are specific to certain memory areas



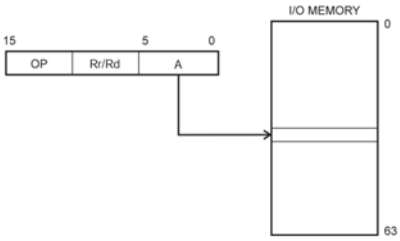
AVR Instructions - I/O Registers

Syntax	Description	Example
cbi A, b	**Clear Bit in I/O Register -** Clears a specified bit in an I/O Register. This instruction operates on the lower 32 I/O Registers (0 ≤ A ≤ 31).	cbi 0x12, 3
sbi A, b	**Set Bit in I/O Register -** Sets a specified bit in an I/O Register. This instruction operates on the lower 32 I/O Registers (0 ≤ A ≤ 31).	sbi 0x16, 5
in Rd, A	**Load an I/O Location to Register -** Loads data from the I/O Space (Ports, Timers, Configuration Registers etc.) into register **Rd** in the Register File (0 ≤ A ≤ 63).	in R25, 0x16
out A, Rr	**Store Register to I/O Location -** Stores data from register **Rr** in the Register File to I/O Space (Ports, Timers, Configuration Registers, …; 0 ≤ A ≤ 63).	out 0x18, R16

Data (I/O) Addressing Modes - Direct

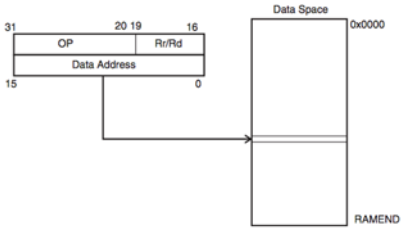
I/O - direct addressing

IN, OUT, …



Data – direct addressing

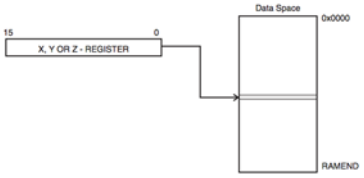
LDS, STS, …
[LDS (16bit), STS (16bit), …]



Data Addressing Modes – Data Indirect

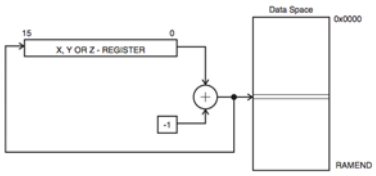
Data Indirect Addressing

e.g., LD Rd, X



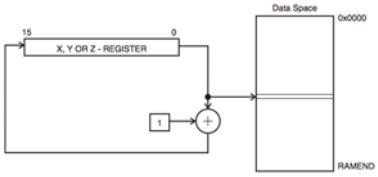
Data Indirect Addressing with pre-decrement

e.g., LD Rd, -Y



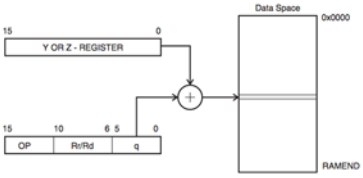
Data Indirect Addressing with post-increment

e.g., LD Rd, Z+



Data Indirect Addressing with displacement

e.g., LDD Rd, Y+q …



Example – What does this program do?

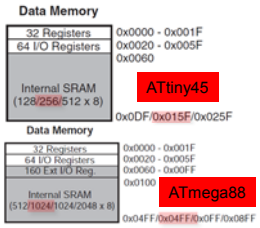
```
.cseg
.org 0
;---------------------
sbi 0x18,0
;---------------------
in  R16, 0x18
ori  R16,0b00000010
out 0x18, R16
;---------------------
lds R18,0x0038
ori R18,0b00000100
sts 0x0038,R18
;---------------------
ldi R26, 0x38
ldi R27, 0x00
LD R19, X
ori R19,0b00001000
ST X, R19
```

I/O Registers 0 to 31

I/O Registers 0 to 63

LDS – Load Direct from Data Space
Data space (register, I/O, (Ext.I/O), SRAM) 0 to 65535
STS – Store Direct to Data Space

Load Indirect from Data Space to Register using Index X
Data space (register, I/O, (Ext.I/O), SRAM) 0 to 65535
Store Indirect From Register to Data Space using Index X



Quiz

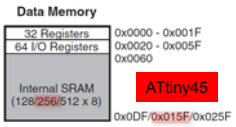
What will be the result of the following program lines?

```
ldi r16, 0xF0
sts 0x0001, r16
```

Answer:
R01 ← 0xF0

```
ldi r16, 0xF0
out 0x07, r16
```

Answer:
ADMUX ← 0xF0
(see I/O register summary in DS)



AVR Instructions

Syntax	Description	Example
ldi Rd, K	**Load Immediate -** Loads an 8-bit constant directly to register (16 <= d <= 31)	ldi R28, 0x5F
clr Rd	**Clear Register -** This instruction performs an Exclusive OR between a register and itself. This will clear all bits in the register.	clr R18
ser Rd	**Set all Bits in Register –** Loads $FF directly to register Rd.	ser R17
mov Rd, Rr	**Copy Register -** This instruction makes a **copy** of one register into another. The source register **Rr** is left unchanged, while the destination register Rd is loaded with a copy of **Rr**.	mov R0, R16

Example – Simple Loop

Example:

```
        clr   r18        ; clear r18
loop:   inc   r18        ; increase r18
        ...
        cpi   r18,$50    ; Compare r18 to $50
        brne  loop
```


Examples

Example:

```
clr   r31       ; Clear Z high byte
ldi   r30,$F0   ; Set Z low byte to $F0
lpm             ; Load constant from Program
                ; memory pointed to by Z
```

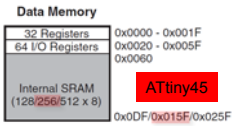
Example 2:

ldi R23, ~((1<<2) | (1<<4) | (1<<7)) & 0x0F

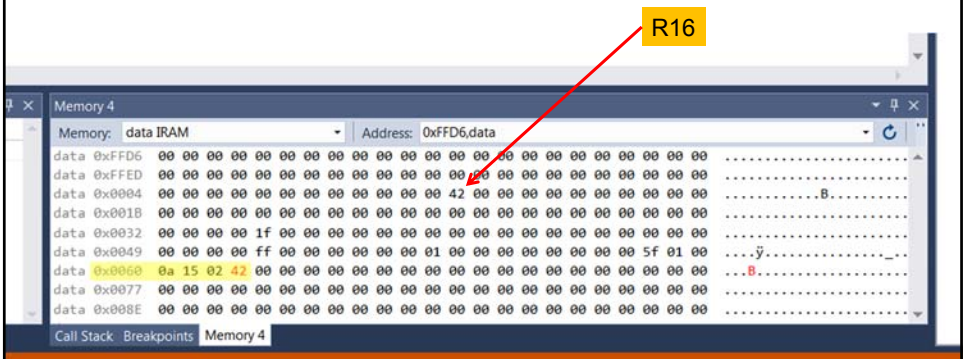
What will be the content of R23? **R23 = 0x0B**

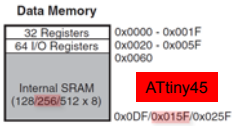
Example – Look Up Table (Initialization)

```
.dseg
.org 0x0060 ; start in SRAM - ATtiny45
lut: .byte 4

.cseg
.org 0x0000

ldi r16, 0x0A ; initialize 1st byte of LUT
sts lut, r16

ldi r16, 0x15  ; initialize 2nd byte of LUT
sts lut+1, r16

ldi r16, 0x02  ; initialize 3rd byte of LUT
sts lut+2, r16

ldi r16, 0x42  ; initialize 4th byte of LUT
sts lut+3, r16

; ....
```



Example – Look Up Table (Initialization)





Example – Look Up Table (Access)

```
; ....

ldi r26, low(lut)
ldi r27, high(lut)
ld r19, X+      ; read 1st byte of LUT
ld r19, X+      ; read 2nd byte of LUT
ld r19, X+      ; read 3rd byte of LUT
ld r19, X+      ; read 4th byte of LUT
```

Q: What will be the content of r26 and r27 after running this program?

Answer:
R26 = 0x64
R27 = 0x00

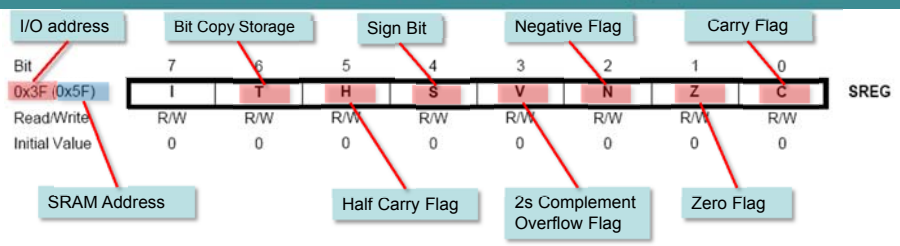
AVR Instructions

Syntax	Description	Example
sbr Rd, K	**Set Bits in Register -** Sets specified bits in register **Rd**. Performs the **logical ORI** between the contents of register **Rd** and a constant mask **K** and places the result in the destination register **Rd**. →not the same as SBI !	sbr R16, 3
cbr Rd, K	**Clear Bits in Register -** Clears the specified bits in register **Rd**. Performs the **logical AND** between the contents of register **Rd** and the **complement** of the constant mask **K**. The result will be placed in register **Rd**. → not the same as CBI !	cbr R16, 0xF0

Write a program that configures I/O lines PD7, PD6, PD3, and PD0 as output and make sure that pins PD7 and PD6 are set to 0.

```
in R16, DDRD
sbr R16, 0xC9
out DDRD, R16
in R17, PORTD
cbr R17, 0xC0
out PORTD, R17
```

AVR Instructions to Set/Clear Flags in SREG



SEC – Set Carry Flag in SREG	**CLC – Clear Carry Flag in SREG**
SEH – Set Half Carry Flag	**CLH – Clear Half Carry Flag**
SEI – Set Global Interrupt Flag	**CLI – Clear Global Interrupt Flag**
SEN – Set Negative Flag in SREG	**CLN – Clear Negative Flag**
SES – Set Signed Flag	**CLS – Clear Signed Flag**
SET – Set T Flag	**CLT – Clear T Flag**
SEV – Set Overflow Flag	**CLV – Clear Overflow Flag**
SEZ – Set Zero Flag	**CLZ – Clear Zero Flag**

(Bit-)Test & Branch Instructions

AVR Compare/Test Instructions

CPI – Compare with Immediate

This instruction performs a compare between register **Rd** and a constant. The register is not changed. All conditional branches can be used after this instruction.

Example:

```
        cpi     r19,3     ; Compare r19 with 3
        brne    error     ; Branch if r19<>3
        ...
error:  nop               ; Branch destination (do nothing)
```


AVR Compare/Test Instructions

CP – Compare

This instruction performs a compare between two registers **Rd** and **Rr**. None of the registers are changed. All conditional branches can be used after this instruction.

Example:

```
        cp    r4,r19    ; Compare r4 with r19
        brne  noteq     ; Branch if r4 <> r19
        ...
noteq:  nop             ; Branch destination (do nothing)
```

AVR Compare/Test Instructions

TST – Test for Zero or Minus

Tests if a register is zero or negative. Performs a logical AND between a register and itself. The register will remain unchanged.

Example:

```
        tst   r0        ; Test r0
        breq  zero      ; Branch if r0=0
        ...
zero:   nop             ; Branch destination (do nothing)
```


TST vs. AND

TST – Test for Zero or Minus

Description:

Tests if a register is zero or negative. Performs a logical AND between a register and itself. The register will remain unchanged.

Operation:
(i) Rd ← Rd • Rd

	Syntax:	Operands:	Program Counter:
(i)	TST Rd	0 ≤ d ≤ 31	PC ← PC + 1

16-bit Opcode: (see AND Rd, Rd)

0010	00dd	dddd	dddd

AND – Logical AND

Description:

Performs the logical AND between the contents of register Rd and register Rr and places the result in the destination register Rd.

Operation:
(i) Rd ← Rd • Rr

	Syntax:	Operands:	Program Counter:
(i)	AND Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

0010	00rd	dddd	rrrr

Summary – Compare/Test Instructions

Syntax	Operands	Action	Description
cp Rd, Rr	Two Registers (0-31)	Affects status flags the same as "sub Rd, Rr"* (Rd-Rr)	Compare Rd to Rr
cpc Rd, Rr	Two Registers (0-31)	Affects status flags the same as "sbc Rd, Rr"* (Rd-Rr-C)	Compare Rd to Rr with carry
cpi Rd, k	Register (16-31) and byte value	Affects status flag the same as "subi Rd, k"* (Rd-k)	Compare Rd with immediate constant
tst Rd	One Register (0-31)	Affects status flag the same as "and Rd, Rd"*	Test for zero or minus

*... does not update Rd!

AVR Instruction - BRNE

BRNE – Branch if Not Equal

Conditional relative branch. Tests the Zero Flag (**Z**) and branches relatively to **PC** if **Z** is cleared. If the instruction is executed immediately after any of the instructions **CP**, **CPI**, **SUB** or **SUBI**, the branch will occur if and only if the unsigned or signed binary number represented in Rd was not equal to the unsigned or signed binary number represented in **Rr**.

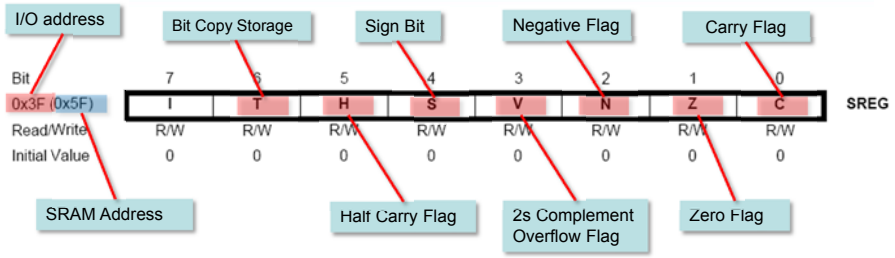
This instruction branches relatively to **PC** in either direction (**PC** - 63 ≤ destination ≤ **PC** + 64). The parameter **k** is the offset from **PC** and is represented in two's complement form. (Equivalent to instruction **BRBC 1,k**)

Example:

```
        eor     r27,r27   ; Clear r27
loop:   inc     r27       ; Increase r27
        ...
        cpi     r27,5     ; Compare r27 to 5
        brne    loop      ; Branch if r27<>5
        nop               ; Loop exit (do nothing)
```

Cycles: 1 if condition is false
2 if condition is true

Conditional Branch Instructions based on SREG



Syntax	Description	Example
brbc s, k	**Branch if Bit in SREG is Cleared -** Tests a single bit (s) in SREG and branches relatively to PC if the bit is cleared. • This instruction branches relatively to PC in either direction (PC - 63 ≤ destination ≤ PC + 64). • The parameter k is the offset from PC and is represented in two's complement form.	**cpi r20, 5** **brbc 1, noteq** **…** **noteq:**
brbs s, k	**Branch if Bit in SREG is Set -** Tests a single bit (s) in SREG and branches relatively to PC if the bit is set. • This instruction branches relatively to PC in either direction (PC - 63 ≤ destination ≤ PC + 64). • The parameter k is the offset from PC and is represented in two's complement form.	**brbs 6, bitset** **…** **bitset:**

Cond. Branch Instructions based on "brbc" & "brbs"

BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC←PC+k + 1	None	1/2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC←PC+k + 1	None	1/2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1/2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1/2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1/2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1/2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1/2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1/2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1/2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1/2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC + k + 1	None	1/2
BRLT	k	Branch if Less Than Zero, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1/2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1/2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1/2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1/2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1/2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1/2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1/2
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1/2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1/2

18 (!) branch instructions based on BRBS and BRBC

"Cheat Sheet" - Conditional Branch Instructions

cp Rd, Rr … or **cpi Rd, k**

\+

→ sub, subi

Instruction	Description	If	Then	Else
breq k	branch if equal	Rd = Rr (Z = 1)	PC ← PC + k + 1	PC ← PC + 1
brne k	branch if not equal	Rd ≠ Rr (Z = 0)	PC ← PC + k + 1	PC ← PC + 1
brsh k	branch if same or higher (unsigned)	Rd ≥Rr (C = 0)	PC ← PC + k + 1	PC ← PC + 1
brlo k	branch if lower (unsigned)	Rd < Rr (C = 1)	PC ← PC + k + 1	PC ← PC + 1
brge k	branch if greater than or equal (signed)	Rd ≥ Rr (N ⊕ V = 0)	PC ← PC + k + 1	PC ← PC + 1
brlt k	branch if less than (signed)	Rd < Rr (N ⊕ V = 1)	PC ← PC + k + 1	PC ← PC + 1

Need test/branch for Rd<=Rr? → use brsh/brge k and swap Rd and Rr

tst Rd … or **and Rd, Rr** with **Rd=Rr**

\+

Instruction	Description	If	Then	Else
breq k	branch if equal (zero)	Rd == 0 (Z = 1)	PC ← PC + k + 1	PC ← PC + 1
brne k	branch if not equal (not zero)	Rd ≠ 0 (Z = 0)	PC ← PC + k + 1	PC ← PC + 1
brmi k	branch if minus	N = 1	PC ← PC + k + 1	PC ← PC + 1
brpl k	branch if plus	N = 0	PC ← PC + k + 1	PC ← PC + 1

Example – Simple Loop

Example:

```
        clr   r18       ; clear r18
loop:   inc   r18       ; increase r18
        ...
        cpi   r18,$50   ; Compare r18 to $50
        brne  loop
```

Branch IF (Z==0) … brbc 1, loop

Example (1)

Generate the following IF THEN structure in AVR assembly language:

```
…C1…
if(b==3) {
   …Cif…
}
…C2…
```

```
.def b=R16
ldi b, 42

…C1…
cpi b, 3
breq if_start
if_end:
…C2…

…
if_start:
   …Cif…
rjmp if_end
…
```

branch IF (Z==1)

Make sure that this code portion can only be reached from the corresponding **breq** line

Conditional Branch Instructions – Examples (2)

Example 1

```
…
If (a==0) {
        …C1…
} else {
        …C2…
}
…
```

Example 2

```
…
If (b=>-42) {
        …C1…
} else {
        …C2…
}
…
```

Example 3

```
…
If (a<0) {
        …C1…
} else {
        …C2…
}
…
```

Solution → … on whiteboard

Conditional Skip Instructions

SBIC – Skip if Bit in I/O Register is Cleared

This instruction tests a single bit in an I/O Register and skips the next instruction if the bit is cleared. This instruction operates on the lower 32 I/O Registers – addresses 0-31.

Example:

```
e2wait:  sbic  $1C,1      ; Skip next inst. if EEWE cleared
         rjmp  e2wait     ; EEPROM write not finished
         nop              ; Continue (do nothing)
```

SBIS – Skip if Bit in I/O Register is Set

This instruction tests a single bit in an I/O Register and skips the next instruction if the bit is set. This instruction operates on the lower 32 I/O Registers – addresses 0-31.

Example:

```
waitset: sbis  $10,0      ; Skip next inst. if bit 0 in Port D set
         rjmp  waitset    ; Bit not set
         nop              ; Continue (do nothing)
```


Conditional Skip Instructions

CPSE – Compare Skip if Equal

This instruction performs a compare between two registers **Rd** and **Rr**, and skips the next instruction if **Rd = Rr**.

Example:

```
inc     r4        ; Increase r4
cpse    r4,r0     ; Compare r4 to r0
neg     r4        ; Only executed if r4<>r0
nop               ; Continue (do nothing)
```

Summary - Conditional Skip Instructions

Syntax	Operands	Action	Description
cpse Rd, Rr	Two registers (0-31)	PC <-- (Rd==Rr) ? PC+2 (or 3) : PC+1	Compare Rd with Rr and skip if equal
sbic A, b	I/0 register (0-31 only) and bit number (0-7)	PC <-- (IO(A,b)==0) ? PC+2 (or 3) : PC+1	Skip if bit in I/O register is clear
sbis A, b	I/0 register (0-31 only) and bit number (0-7)	PC <-- (IO(A,b)==1) ? PC+2 (or 3) : PC+1	Skip if bit in I/O register is set
sbrc Rr, b	Register (0-31) and bit number (0-7)	PC <-- (Rr(b)==0) ? PC+2 (or 3) : PC+1	Skip if bit in register is clear
sbrs Rr, b	Register (0-31) and bit number (0-7)	PC <-- (Rr(b)==1) ? PC+2 (or 3) : PC+1	Skip if bit in register is set

Branch Instructions

RJMP – Relative Jump

Relative jump to an address within **PC** - 2K +1 and **PC** + 2K (words). For some AVR microcontrollers, this instruction can address the entire memory from every address location. See also **JMP**.

Example:

```
        cpi     r16,$42   ; Compare r16 to $42
        brne    error     ; Branch if r16 <> $42
        rjmp    ok        ; Unconditional branch
error:  add     r16,r17   ; Add r17 to r16
        inc     r16       ; Increment r16
ok:     nop               ; Destination for rjmp (do nothing)
```

Branch Instructions

RCALL – Relative Call to Subroutine

Relative call to an address within **PC** - 2K + 1 and **PC** + 2K (words). The return address (the instruction after the **RCALL**) is stored onto the Stack. See also **CALL**. For some AVR microcontrollers, this instruction can address the entire memory from every address location. The Stack Pointer uses a post-decrement scheme during **RCALL**.

Example:

```
            rcall   routine     ; Call subroutine
            ...
routine:    push    r14         ; Save r14 on the Stack
            ...
            pop     r14         ; Restore r14
            ret                 ; Return from subroutine
```

Avoid side effects!

Branch Instructions

RET – Return from Subroutine

Returns from subroutine. The return address is loaded from the STACK. The Stack Pointer uses a pre-increment scheme during **RET**.

Example:

```
            call    routine     ; Call subroutine
            ...
routine:    push    r14         ; Save r14 on the Stack
            push    r13         ; Save r13 on the Stack
            ...
            pop     r13         ; Restore r13
            pop     r14         ; Restore r14
            ret                 ; Return from subroutine
```

Branch Instructions

RETI – Return from Interrupt

Returns from interrupt. The return address is loaded from the STACK and the Global Interrupt Flag is set.

Note that the **Status Register is not automatically stored** when entering an interrupt routine, and it is not restored when returning from an interrupt routine. This must be handled by the application program. The Stack Pointer uses a pre-increment scheme during **RETI**.

Example:

```
            ...
extint:     push    r0      ; Save r0 on the Stack
            ...
            pop     r0      ; Restore r0
            reti            ; Return and enable interrupts
```


RCALL vs. BRBS

BRBS – Branch if Bit in SREG is Set

Description:

Conditional relative branch. Tests a single bit in SREG and branches relatively to PC if the bit is set. This instruction branches relatively to PC in either direction (PC - 63 ≤ destination ≤ PC + 64). The parameter k is the offset from PC and is represented in two's complement form.

Operation:

(i) If SREG(s) = 1 then PC ← PC + k + 1, else PC ← PC + 1

Syntax: (i) BRBS s,k

Operands: 0 ≤ s ≤ 7, -64 ≤ k ≤ +63

Program Counter: PC ← PC + k + 1; PC ← PC + 1, if condition is false

16-bit Opcode:

1111	00kk	kkkk	kkkk

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

Example:

```
        bst   r0,3      ; Load T bit with bit 3 of r0
        brbs  6,bitset  ; Branch T bit was set
        ...
bitset: nop             ; Branch destination (do nothing)
```

Words: 1 (2 bytes)
Cycles: 1 if condition is false
2 if condition is true

RCALL uses the Stack, but BRBS does not!

RCALL – Relative Call to Subroutine

Description:

Relative call to an address within PC - 2K + 1 and PC + 2K (words). The return address (the instruction after the RCALL) is stored onto the Stack. See also CALL. For AVR microcontrollers with Program memory not exceeding 4K words (8K bytes) this instruction can address the entire memory from every address location. The Stack Pointer uses a post-decrement scheme during RCALL.

Operation:

(i) PC ← PC + k + 1 Devices with 16 bits PC, 128K bytes Program memory maximum.
(ii) PC ← PC + k + 1 Devices with 22 bits PC, 8M bytes Program memory maximum.

Syntax: (i) RCALL k; (ii) RCALL k

Operands: -2K ≤ k < 2K

Program Counter: (i) PC ← PC + k + 1; (ii) PC ← PC + k + 1

Stack: (i) STACK ← PC + 1, SP ← SP - 2 (2 bytes, 16 bits); (ii) STACK ← PC + 1, SP ← SP - 3 (3 bytes, 22 bits)

16-bit Opcode:

1101	kkkk	kkkk	kkkk

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

Example:

```
            rcall   routine ; Call subroutine
            ...
routine:    push    r14     ; Save r14 on the Stack
            ...
            pop     r14     ; Restore r14
            ret             ; Return from subroutine
```

Words : 1 (2 bytes)
Cycles : 3, devices with 16 bit PC; 4, devices with 22 bit PC
Cycles XMEGA: 2, devices with 16 bit PC; 3, devices with 22 bit PC
Cycles Reduced Core tinyAVR: 4

Arithmetic & Logic Instructions

Arithmetic & Logic Instructions

Syntax	Description	Example
inc Rd	**Increment-** Adds one (1) to the contents of register **Rd** and places the result in the destination register Rd. The **C** Flag in **SREG** is not affected by the operation, thus allowing the **INC** instruction to be used on a loop counter in multiple-precision computations. ****	inc R28
dec Rd	**Decrement -** Subtracts 1 from the contents of register **Rd** and places the result in the destination register **Rd**. The **C** Flag in **SREG** is not affected by the operation, thus allowing the **DEC** instruction to be used on a loop counter in multiple-precision computations. ****	dec R18
neg Rd	**Two's Complement** – Replaces the contents of register Rd with its two's complement; the value 0x80 is left unchanged	neg R17

**** Implications for conditional branch instructions → when operating on unsigned values, only **BREQ** and **BRNE** branches (Z flag) can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

Arithmetic & Logic Instructions

BIN	HEX	Unsigned DEC	Signed DEC
0b00000000	0x00	0	0
0b00000001	0x01	1	1
…	…	…	…
0b01111111	0x7F	127	127
0b10000000	0x80	128	-128
0b10000001	0x81	129	-127
0b10000010	0x82	130	-126
…	…	…	…
0b11111110	0xFE	254	-2
0b11111111	0xFF	255	-1

Examples

Example:

```
        clr     r22         ; clear r22
loop:   inc     r22         ; increment r22
        ...
        cpi     r22,$4F     ; Compare r22 to $4f
        brne    loop        ; Branch if not equal
        nop                 ; Continue (do nothing)
```

Example:

```
        ldi   r17,$10   ; Load constant in r17
loop:   add   r1,r2     ; Add r2 to r1
        dec   r17       ; Decrement r17
        brne  loop      ; Branch if r17<>0
        nop             ; Continue (do nothing)
```

Example

Example: Assume **R18** holds the value 0x71. Then, after the instruction

```
neg r18
```

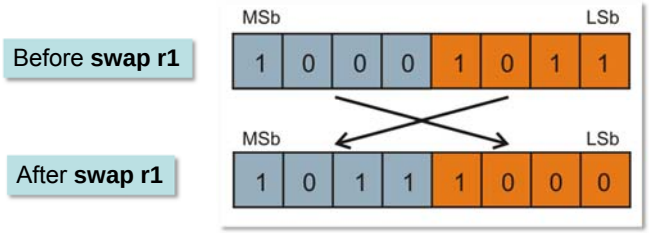
R18 will hold 0x8F. How? – Recall to get 2s complement, invert bits and add 1

0x71	`0b0111 0001`	
Bits inverted	`0b1000 1110`	
Add 1	`+0b0000 0001`	
2s Complement	`0b1000 1111`	=0x8F

Arithmetic & Logic Instructions

SWAP – Swap Nibbles

Swaps high and low nibbles in a register.



Example:

```
inc     r1    ; Increment r1
swap    r1    ; Swap high and low nibble of r1
inc     r1    ; Increment high nibble of r1
swap    r1    ; Swap back
```

We will use swap instruction when working with LCDs later in this class.

Arithmetic & Logic Instructions

Syntax	Description	Example
and Rd, Rr	**Logical AND -** Performs the bitwise logical AND between the contents of register **Rd** and register **Rr** and places the result in the destination register **Rd.** See also: **ANDI** - Logical AND with Immediate	and R1, R28
or Rd, Rr	**Logical OR -** Performs the bitwise logical OR between the contents of register **Rd** and register **Rr** and places the result in the destination register **Rd**. See also: **ORI** - Logical OR with Immediate	or R0, R18
eor Rd, Rr	**Exclusive OR –** Performs the bitwise logical EOR between the contents of register **Rd** and register **Rr** and places the result in the destination register Rd.	eor R17, R16

Examples

Example: Assume **R18** holds 0xB8 and **R19** holds 0x58. Then, after the instruction **EOR R18, R19**, register **R18** will hold the value 0xE0:

0xB8	`0b1011 1000`	
0x58	`0b0101 1000`	
Bitwise Exclusive OR	`0b1110 0000`	= 0xE0

Example: Assume **R18** holds 0xB8 and **R19** holds 0x58. Then, after the instruction **OR R18, R19**, register **R18** will hold the value 0xF8:

0xB8	`0b1011 1000`	
0x58	`0b0101 1000`	
Bitwise OR	`0b1111 1000`	= 0xF8

Example

Example:

```
and   r2,r3     ; Bitwise and r2 and r3, result in r2
ldi   r16,1     ; Set bitmask 0000 0001 in r16
and   r2,r16    ; Isolate bit 0 in r2
```

Example: Assume **R18** holds 0xB8 and **R19** holds 0x58. Then, after the instruction **AND R18, R19**, register **R18** will hold the value 0x18:

0xB8	**0b1011 1000**	
0x58	**0b0101 1000**	
Bitwise AND	**0b0001 1000**	= 0x18

Arithmetic & Logic Instructions

Syntax	Description	Example
add Rd, Rr	**Add without Carry -** Adds the contents of two registers without the **C** Flag and places the result in the destination register **Rd.** (Rd ← Rd + Rr)	add R1, R28
adc Rd, Rr	**Add with Carry -** Adds the contents of two registers and the contents of the **C** Flag and places the result in the destination register **Rd.** (Rd ← Rd + Rr + C) **	adc R0, R18
sub Rd, Rr	**Subtract without Carry –** Subtracts the contents of two registers and places the result in the destination register. Rd (Rd ← Rd - Rr)	sub R17, R16
sbc Rd, Rr	**Subtract with Carry -** Subtracts the contents of two registers and subtracts the contents of the **C** Flag and places the result in the destination register **Rd.** (Rd ← Rd - Rr - C) **	sbc R14, R1

** Instruction enables multi-byte arithmetic on AVR 8-bit architecture

Example

Example: Assume **R18** holds 0xB8 and **R19** holds 0x58. Then, after the instruction **SUB R18, R19**, register **R18** will hold the value 0x60:

0xB8	**0b1011 1000**	
0x58	**0b0101 1000**	
0xB8-0x58	**0b0110 0000**	= 0x60

Example

Example: add two 16-bit numbers

```
                ; Add R1:R0 to R3:R2
add   r2,r0     ; Add low byte
adc   r3,r1     ; Add with carry high byte
```

May set **C** flag in **SREG**

Example:

```
                ; Subtract r1:r0 from r3:r2
sub     r2,r0   ; Subtract low byte
sbc     r3,r1   ; Subtract with carry high byte
```

Arithmetic & Logic Instructions

ADIW – Add Immediate to Word | Not available on all AVRs, but on the ATmega88PA

Adds an immediate value (0 - 63) to a register pair and places the result in the register pair. This instruction operates on the upper four register pairs, and is well suited for operations on the pointer registers. This instruction is not available in all devices. Refer to the device specific instruction set summary

Example:

```
adiw  r25:24,1  ; Add 1 to r25:r24
adiw  ZH:ZL,63  ; Add 63 to the Z-pointer(r31:r30)
```

Example

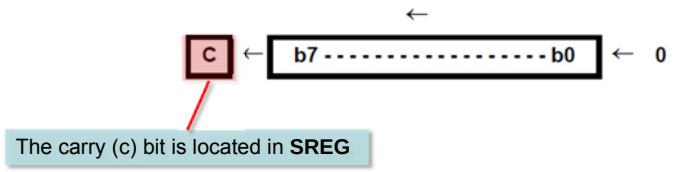
```
msg: .DB "Hello "
   …
   ldi   r30,LOW(msg<<1)    ; Load Z register low
   ldi   r31,HIGH(msg<<1)   ; Load Z register high
   lpm                      ; r0 <-- load byte
   …
   adiw  zh:zl,1            ; Increment Z pointer
```


Shift Instructions

Shift Instructions

LSL – Logical Shift Left

Shifts all bits in **Rd** one place to the left. Bit 0 is cleared. Bit 7 is loaded into the **C** Flag of the **SREG**. This operation effectively multiplies a value by two.



The carry (c) bit is located in **SREG**

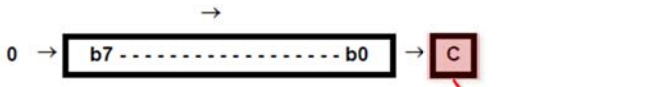
Example:

```
add     r0,r4     ; Add r4 to r0
lsl     r0        ; Multiply r0 by 2
```


Shift 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 of the **SREG**. This operation effectively divides an unsigned value by two. The **C** Flag can be used to round the result.



The carry (c) bit is located in **SREG**

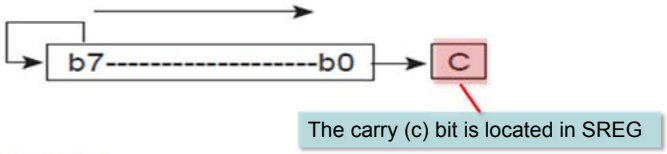
Example:

```
add     r0,r4     ; Add r4 to r0
lsr     r0        ; Divide r0 by 2
```

Shift Instructions

ASR – Arithmetic Shift Right

Shifts all bits in **Rd** one place to the right. Bit 7 is held constant. Bit 0 is loaded into the **C** Flag of the **SREG**. This operation effectively divides a signed value by two without changing its sign. The Carry Flag can be used to round the result.



Example:

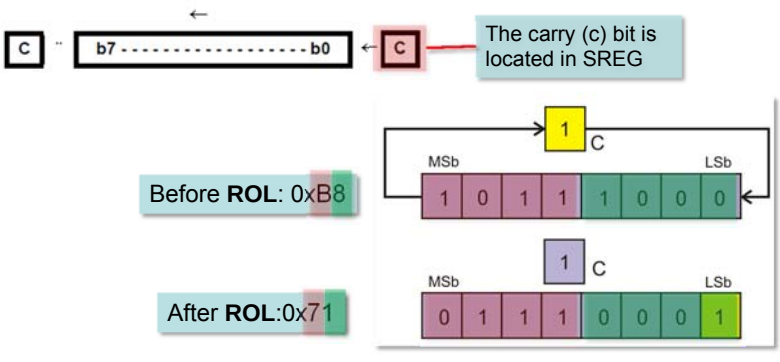
```
ldi   r16,$10   ; Load decimal 16 into r16
asr   r16       ; r16=r16 / 2
ldi   r17,$FC   ; Load -4 in r17
asr   r17       ; r17=r17/2
```


Shift Instructions

ROL – Rotate Left trough Carry

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

This operation, combined with **LSL**, effectively multiplies multi-byte values by two.

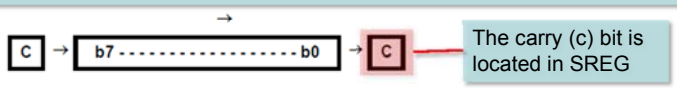


Shift Instructions

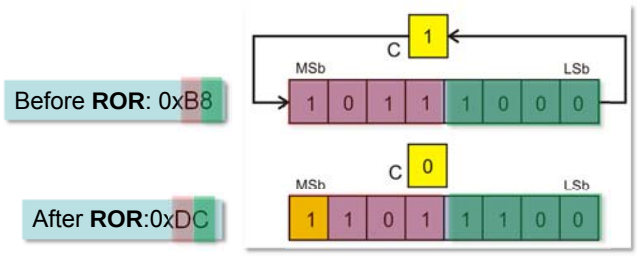
ROR – Rotate Right through Carry

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.

This operation, combined with **ASR**, effectively divides multi-byte signed values by two. Combined with **LSR** it effectively divides multi-byte unsigned values by two. The Carry Flag can be used to round the result.



The carry (c) bit is located in SREG



Stack Instructions

Stack Instructions

PUSH and POP – Save and Restore on Stack

The **PUSH** instruction stores the contents of register **Rr** on the STACK. The Stack Pointer is post-decremented by 1 after the **PUSH**.

The POP instruction loads register **Rd** with a byte from the STACK. The Stack Pointer is pre-incremented by 1 before the **POP**. This instruction is not available in all devices. Refer to the device specific instruction set summary.

A **PUSH** should have a matching **POP** or the stack will be corrupted

Example:

```
        call    routine   ; Call subroutine
        ...
routine: push   r14       ; Save r14 on the Stack
        push    r13       ; Save r13 on the Stack
        ...
        pop     r13       ; Restore r13
        pop     r14       ; Restore r14
        ret               ; Return from subroutine
```


Stack Pointer (SP)

Bit	15	14	13	12	11	10	9	8	
0x3E	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
0x3D	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	
Initial Value	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	

SP on ATtiny45

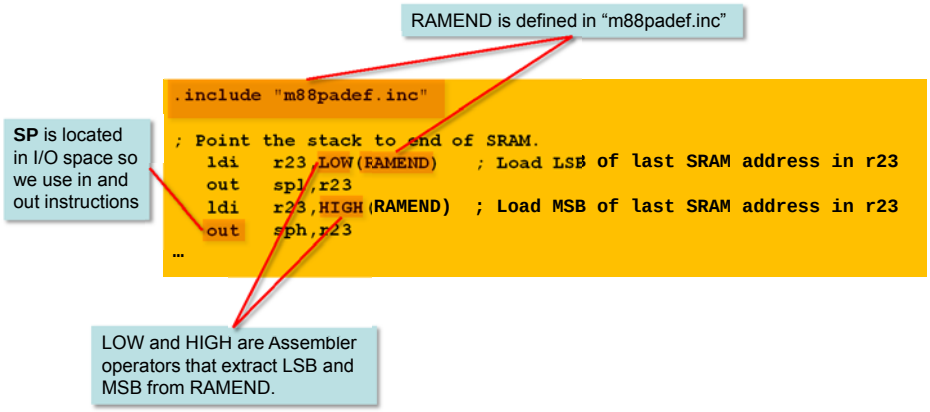
The stack is implemented in SRAM, and **SP** always points to the top of the stack and decrements with a push instruction.

Thus, to utilize the stack using the push and pop instructions, one must initialize **SP** properly.

On AVRs, **SP** is initialized to the end of SRAM (see diagram above)

Other AVR documentation suggest that at reset, **SP** is initialized to 0x00, which is where the 32 general purpose registers start—see the AVR memory map, so programmer has to initialize **SP.**

Initializing Stack Pointer (SP)



More Instructions …

SLEEP

This instruction sets the circuit in sleep mode defined by the MCU Control Register.

Example:

```
mov     r0,r11          ; Copy r11 to r0
ldi     r16,(1<<SE)     ; Enable sleep mode
out     MCUCR, r16
sleep                   ; Put MCU in sleep mode
```

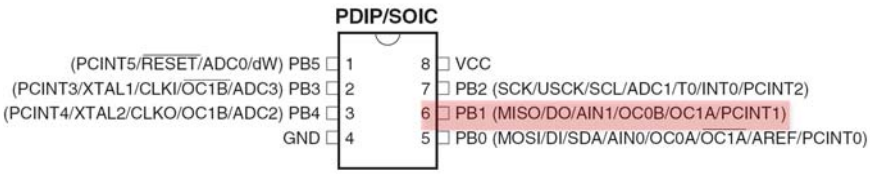
Question: Why sleep?

Answer: In sleep modes, the clocks (are) running slower/stopped, other peripherals may be powered down → save power

Question: How will MCU wake up?

Answer: One can configure the controller so that Timers, WDT, and external interrupts and pin changes wake it up from a sleep.

Waking up from Sleep



One can configure the microcontroller so that it wakes up when there is a high-low or low-high (i.e., pin change) on some of the pins.

One can configure the microcontroller so that it wakes up when WDT times out

…

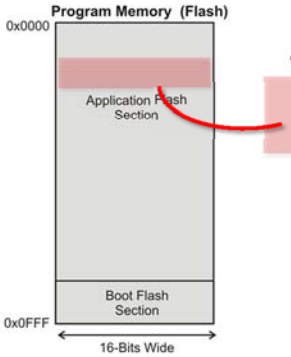
More Instructions …

LPM – Load Program Memory

Loads one byte pointed to by the **Z**-register into the destination register **Rd**. If **Rd** is not specified, it is implied to be **R0**.

Example:

```
ldi   ZH, high(Table_1<<1); Initialize Z-pointer
ldi   ZL, low(Table_1<<1)
lpm   r16, Z              ; Load constant from Program
                          ; Memory pointed to by Z (r31:r30)
...
Table_1:
.dw 0x5876                ; 0x76 is addresses when Z_LSB = 0
                          ; 0x58 is addresses when Z_LSB = 1
...
```



These constant are located in Program (Flash) memory and not SRAM…

Example: In the snippet below, for **LPM** a destination register is not supplied, so **R0** is implied

```
msg: .DB "Hello "
   …
   ldi   r30,LOW(msg<<1)    ; Load Z register low
   ldi   r31,HIGH(msg<<1)   ; Load Z register high
   lpm                      ; r0 <-- load byte
   …
   adiw  zh:zl,1              ; Increment Z pointer
```

Example: In the snippet below, the **LPM Rd,Z+** variant is used

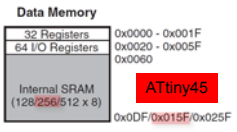
```
displayC:
   ldi   r30,LOW(msg<<1)   ; Load Z register low
   ldi   r31,HIGH(msg<<1)  ; Load Z register high
L20:
   lpm   r2,Z+             ; r2 <-- first byte
   tst   r2                ; Reached end of message?
   breq  done              ; Yes => quit
   …
   rjmp  L20
done:
   ret
```

For Further information on the LPM instruction, please see the Atmel Application Note AVR108: Setup and Use of the LPM Instruction on the company's website.

Quiz

- **Will the program given below work as "expected"? (TCCR0A, TCCR0B: registers of timer/counter 0)**

```
.include "tn45def.inc"
.cseg
.org 0

ldi r16, (1<<COM0A1)|(1<<COM0B1)|(1<<WGM00)
sts TCCR0A, r16
ldi r16, (1<<CS01)
sts TCCR0B, r16

sbi DDRB, 0
sbi DDRB, 1
…
```



Quiz

```
; ***** I/O REGISTER DEFINITIONS ********************************
; NOTE:
; Definitions marked "MEMORY MAPPED"are extended I/O ports
; and cannot be used with IN/OUT instructions
.equ    SREG    = 0x3f
.equ    SPL     = 0x3d
.equ    SPH     = 0x3e
.equ    GIMSK   = 0x3b
.equ    GIFR    = 0x3a
.equ    TIMSK   = 0x39
.equ    TIFR    = 0x38
.equ    SPMCSR  = 0x37
.equ    MCUCR   = 0x35
.equ    MCUSR   = 0x34
.equ    TCCR0B  = 0x33
.equ    TCNT0   = 0x32
.equ    OSCCAL  = 0x31
.equ    TCCR1   = 0x30
.equ    TCNT1   = 0x2f
.equ    OCR1A   = 0x2e
.equ    OCR1C   = 0x2d
.equ    GTCCR   = 0x2c
.equ    OCR1B   = 0x2b
.equ    TCCR0A  = 0x2a
.equ    OCR0A   = 0x29
.equ    OCR0B   = 0x28
.equ    PLLCSR  = 0x27
.equ    CLKPR   = 0x26
.equ    DT1A    = 0x25
.equ    DT1B    = 0x24
.equ    DTPS    = 0x23
.equ    DWDR    = 0x22
.equ    WDTCR   = 0x21
.equ    PRR     = 0x20
.equ    EEARH   = 0x1f
.equ    EEARL   = 0x1e
.equ    EEDR    = 0x1d
.equ    EECR    = 0x1c
.equ    PORTB   = 0x18
.equ    DDRB    = 0x17
.equ    PINB    = 0x16
.equ    PCMSK   = 0x15
.equ    DIDR0   = 0x14
.equ    GPIOR2  = 0x13
.equ    GPIOR1  = 0x12
.equ    GPIOR0  = 0x11
.equ    USIBR   = 0x10
.equ    USIDR   = 0x0f
.equ    USISR   = 0x0e
.equ    USICR   = 0x0d
.equ    ACSR    = 0x08
.equ    ADMUX   = 0x07
.equ    ADCSRA  = 0x06
.equ    ADCH    = 0x05
.equ    ADCL    = 0x04
.equ    ADCSRB  = 0x03
```

→I/O address! ≠ data address

4. Register Summary

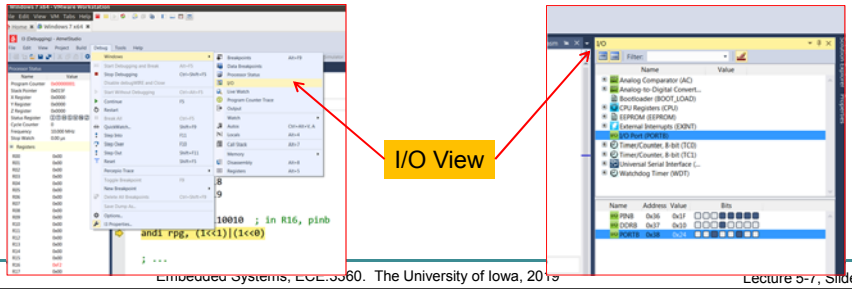
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x3F	SREG	I	T	H	S	V	N	Z	C	page 8
0x3E	SPH	–	–	–	–	–	–	SP9	SP8	page 11
0x3D	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	page 11
0x3C	Reserved									
0x3B	GIMSK	–	INT0	PCIE	–	–	–	–	–	page 51
0x3A	GIFR	–	INTF0	PCIF	–	–	–	–	–	page 52
0x39	TIMSK	–	OCIE1A	OCIE1B	OCIE0A	OCIE0B	TOIE1	TOIE0	–	pages 81, 103
0x38	TIFR	–	OCF1A	OCF1B	OCF0A	OCF0B	TOV1	TOV0	–	page 81
0x37	SPMCSR	–	–	RSIG	CTPB	RFLB	PGWRT	PGERS	SPMEN	page 140
0x36	Reserved									
0x35	MCUCR	BODS	PUD	SE	SM1	SM0	BODSE	ISC01	ISC00	pages 37, 51, 64
0x34	MCUSR	–	–	–	–	WDRF	BORF	EXTRF	PORF	page 44
0x33	TCCR0B	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	page 75
0x32	TCNT0	Timer/Counter0								page 80
0x31	OSCCAL	Oscillator Calibration Register								page 31
0x30	TCCR1	CTC1	PWM1A	COM1A1	COM1A0	CS13	CS12	CS11	CS10	pages 88, 100
0x2F	TCNT1	Timer/Counter1								pages 91, 102
0x2E	OCR1A	Timer/Counter1 Output Compare Register A								pages 91, 102
0x2D	OCR1C	Timer/Counter1 Output Compare Register C								pages 91, 102
0x2C	GTCCR	TSM	PWM1B	COM1B1	COM1B0	FOC1B	FOC1A	PSR1	PSR0	pages 77, 90, 101
0x2B	OCR1B	Timer/Counter1 Output Compare Register B								page 92
0x2A	TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	page 77
0x29	OCR0A	Timer/Counter0 – Output Compare Register A								page 80
0x28	OCR0B	Timer/Counter0 – Output Compare Register B								page 81
0x27	PLLCSR	LSM	–	–	–	–	PCKE	PLLE	PLOCK	pages 94, 103
0x26	CLKPR	CLKPCE	–	–	–	CLKPS3	CLKPS2	CLKPS1	CLKPS0	page 33
0x25	DT1A	DT1AH3	DT1AH2	DT1AH1	DT1AH0	DT1AL3	DT1AL2	DT1AL1	DT1AL0	page 107
0x24	DT1B	DT1BH3	DT1BH2	DT1BH1	DT1BH0	DT1BL3	DT1BL2	DT1BL1	DT1BL0	page 107
0x23	DTPS1	–	–	–	–	–	–	DTPS11	DTPS10	page 106
0x22	DWDR	DWDR[7:0]								page 140
0x21	WDTCR	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	page 45
0x20	PRR	–	–	–	–	PRTIM1	PRTIM0	PRUSI	PRADC	page 36
0x1F	EEARH	–	–	–	–	–	EEAR2	EEAR1	EEAR0	page 20
0x1E	EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	page 21
0x1D	EEDR	EEPROM Data Register								page 21
0x1C	EECR	–	–	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	page 21
0x1B	Reserved									
0x1A	Reserved									
0x19	Reserved									
0x18	PORTB	–	–	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	page 64
0x17	DDRB	–	–	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	page 64
0x16	PINB	–	–	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	page 64
0x15	PCMSK	–	–	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	page 52
0x14	DIDR0	–	–	ADC0D	ADC2D	ADC3D	ADC1D	AIN1D	AIN0D	pages 121, 138
0x13	GPIOR2	General Purpose I/O Register 2								page 10
0x12	GPIOR1	General Purpose I/O Register 1								page 10
0x11	GPIOR0	General Purpose I/O Register 0								page 10
0x10	USIBR	USI Buffer Register								page 135
0x0F	USIDR	USI Data Register								page 135
0x0E	USISR	USISIF	USIOIF	USIPF	USIDC	USICNT3	USICNT2	USICNT1	USICNT0	page 135
0x0D	USICR	USISIE	USIOIE	USIWM1	USIWM0	USICS1	USICS0	USICLK	USITC	page 136
0x0C	Reserved									
0x0B	Reserved									
0x0A	Reserved									
0x09	Reserved									
0x08	ACSR	ACD	ACBG	ACO	ACI	ACIE	–	ACIS1	ACIS0	page 130
0x07	ADMUX	REFS1	REFS0	ADLAR	REFS2	MUX3	MUX2	MUX1	MUX0	page 134
0x06	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	page 136
0x05	ADCH	ADC Data Register High Byte								page 137
0x04	ADCL	ADC Data Register Low Byte								page 137
0x03	ADCSRB	BIN	ACME	IPR	–	–	ADTS2	ADTS1	ADTS0	pages 120, 137
0x02	Reserved									
0x01	Reserved									
0x00	Reserved									

Note: 1. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses

Debugging in Atmel Studio 7

- **Use I/O View, Memory View, and Watch!**
- **For example:**
  - **Use I/O View to see which bits are set in DDRB or PORTB!**
  - **Can use the mouse to set/clear bits in registers!**
- **See online documentation at:**

https://www.microchip.com/webdoc/GUID-ECD8A826-B1DA-44FC-BE0B-5A53418A47BD/index.html?GUID-8B95323F-DC46-41D1-B28A-0CD545A35B8B



… see AVR instruction set for more information!

… EOL