

# 1DT301, Computer Technology I

Lecture #3,

Tuesday, September 10, 2019

- Subroutine call, rcall
- Use of Stack
- Push and Pop

# Reading:

- [http://www.avr-asm-tutorial.net/avr\\_en/](http://www.avr-asm-tutorial.net/avr_en/)
- <http://www.avrbeginners.net/>

AVR-Tutorial

Path: Home => AVR-Overview

Tutorial for learning assembly language for the **AVR-Single-Chip-Processors** (AT90S, ATmega and ATTiny) from ATMEL with practical examples.

(C)2017 by H.J.Wilmscher

The Single-Chip-processors of [ATMEL](#) are excellent for homebrewing every kind of processor-driven electronics. The only problem is that assembly has to be learned in order to program these devices. After having done these first steps the assembly language provides very fast, lean and effective code, by which every task can be accomodated. These pages are for beginners and help in learning the first steps.

Sitemap      New on this webpage      Error list      AVR-Webring

## Index

An overview on AVRs and their hardware	
A simulator for AVR assembler code	
<h3>Learning from applications</h3> <p>More than 60,000 downloads of assembler source code files in 2018</p> <p>The four most popular applications:</p> <ul style="list-style-type: none"><li>Reading keypad keys</li><li>Frequency counter</li><li>Signal generator</li><li>Stepper motor driver ATtiny13</li></ul> <p>The four most recently added applications:</p> <ul style="list-style-type: none"><li>ATTiny and crystals to generate accurate rectangles</li><li>RGB LCD watch</li><li>DCC77 signal analysis</li><li>Large watch ATmega48</li></ul> <p>Use of controller internal hardware components in assembler source codes on this website</p> <p>Beginner's introduction to AVR assembler language. Also available as complete PDF document for printing the whole course (<a href="#">Download, 1.1 MB</a>)</p> <p>A starter course for beginners using the simulator to examine the effect of assembler instructions, also available as a <a href="#">PDF document</a></p> <p>Four simple programming examples with extended comments as first steps of a practical introduction to assembler programming: <a href="#">Sense and requirements</a>, <a href="#">Simple programming examples</a></p> <p>All about <a href="#">timing loops</a> from microseconds over milliseconds and seconds to hours, days and months: all you need is a loop, or two, or three ...</p> <p>micro beginner course</p> <p>1010.1010 +101.0110 =???</p> <p>Software Know-How</p> <p>Tools for programming in assembler</p>	

**Beginners introduction to AVR**

Source: [http://www.avr-asm-tutorial.net/avr\\_en/](http://www.avr-asm-tutorial.net/avr_en/)



# Beginners Programming in AVR Assembler

The following pages are written for all people that try to program in assembler language for the first time and like to start with programming [ATMEL-AVRs AT90S, ATtiny, ATmega](#). Some basic programming techniques are shown that are necessary. During later learning stages these informations are useful and so try to learn this first. Tables are included for easier programming. There might be some bugs in these pages. Bug reports and hints for improving these pages are very welcome.

## Overview on the beginners programming tutorial

Theme	What is that	Instructions
<a href="#">Hardware</a>	<a href="#">ISP-Interface</a> , <a href="#">Parallel-Port-Programmer</a> , <a href="#">Experimental 2313 board</a> , <a href="#">Commercial boards</a>	-
<a href="#">Why</a>	<a href="#">Why learn assembler language?</a> , <a href="#">The concept behind assembler</a> , <a href="#">Planning an AVR project</a>	-
<a href="#">Tools</a>	<a href="#">Editor</a> , <a href="#">Assembler</a> , <a href="#">ISP</a> , <a href="#">Studio3</a> , <a href="#">Studio4</a> , <a href="#">Structure</a>	-
<a href="#">Register</a>	<a href="#">What is a Register?</a>	<a href="#">.DEF</a> , <a href="#">LDI</a> , <a href="#">MOV</a>
	<a href="#">Which different registers are available</a>	<a href="#">CLR</a> , <a href="#">AND</a> , <a href="#">OR</a> , <a href="#">CPL</a> , <a href="#">SRCL</a> , <a href="#">SRP</a> , <a href="#">SEF</a> , <a href="#">SEU</a>
	<a href="#">Register as Pointer</a>	-
	<a href="#">What can be packed into a register?</a>	-
<a href="#">Ports</a>	<a href="#">What is a Port?</a>	<a href="#">.INCLU</a>
	<a href="#">Which Ports are available?</a>	-
	<a href="#">Status register as most relevant port</a>	<a href="#">CLX</a> , <a href="#">SEX</a> , <a href="#">BCLR</a> , <a href="#">BSET</a>
<a href="#">SRAM</a>	<a href="#">What is SRAM?</a>	-
	<a href="#">For which purposes SRAM is used</a>	-
	<a href="#">How to use SRAM?</a>	<a href="#">STS</a> , <a href="#">LDS</a> , <a href="#">LD</a> , <a href="#">ST</a> , <a href="#">STD</a> , <a href="#">LDD</a>
	<a href="#">Stack in SRAM</a>	<a href="#">PUSH</a> , <a href="#">POP</a> , <a href="#">RCALL</a> , <a href="#">RET</a>
<a href="#">Program run</a>	<a href="#">What happens during a Reset?</a>	-
	<a href="#">Linear program runs and branches</a>	<a href="#">.CSEG</a> , <a href="#">.DSEG</a> , <a href="#">.ORG</a> , <a href="#">.ESEG</a> , <a href="#">.INC</a> , <a href="#">.BRNE</a> , <a href="#">.BREQ</a> , <a href="#">.BRxx</a>
	<a href="#">Timing of commands</a>	<a href="#">NOP</a> , <a href="#">DEC</a>
	<a href="#">Macros</a>	<a href="#">.MACRO</a> , <a href="#">.ENDMACRO</a>
<a href="#">Calculations</a>	<a href="#">Subroutines</a>	<a href="#">RET</a> , <a href="#">RCALL</a> , <a href="#">RJMP</a> , <a href="#">SBRC</a> , <a href="#">SBRS</a> , <a href="#">SBIC</a> , <a href="#">SBIS</a>
	<a href="#">Interrupts</a>	<a href="#">RETI</a>
	<a href="#">Numbers and characters</a>	-
	<a href="#">Bit manipulation</a>	<a href="#">OR</a> , <a href="#">OR</a> , <a href="#">AND</a> , <a href="#">AND</a> , <a href="#">CBR</a> , <a href="#">SBR</a> , <a href="#">EOR</a> , <a href="#">COM</a> , <a href="#">NEG</a> , <a href="#">BLD</a> , <a href="#">CLT</a> , <a href="#">SET</a> , <a href="#">BST</a>
<a href="#">Tables</a>	<a href="#">Shift and rotate</a>	<a href="#">LSL</a> , <a href="#">LSR</a> , <a href="#">ASR</a> , <a href="#">ROL</a> , <a href="#">ROR</a> , <a href="#">SWAP</a>
	<a href="#">Adding, Subtracting, Compares</a>	<a href="#">ADD</a> , <a href="#">ADC</a> , <a href="#">SUB</a> , <a href="#">SBC</a> , <a href="#">CP</a> , <a href="#">CPC</a> , <a href="#">CPI</a> , <a href="#">TST</a>
	<a href="#">Format changes</a>	<a href="#">.DB</a> , <a href="#">.DW</a>
	<a href="#">Commands by function</a>	-
<a href="#">AVR-page</a>	<a href="#">Commands alphabetic</a>	-
	<a href="#">Ports</a>	-
	<a href="#">Vectors</a>	-
	<a href="#">Abbreviations</a>	-
	<a href="#">Directives</a>	-
	<a href="#">Expressions</a>	<a href="#">all dot directives</a>

[To the top of this page](#)

If you need a complete description of all instructions of the AVR's with explanations you might be happy to have the document [here](#) by Anthony Peck. It also provides small projects with an 8-pin ATtiny13.

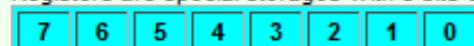
Source: [http://www.avr-asm-tutorial.net/avr\\_en/beginner/index.html](http://www.avr-asm-tutorial.net/avr_en/beginner/index.html)



# Introduction to AVR assembler programming for beginners

## What is a register?

Registers are special storages with 8 bits capacity and they look like this:



Note the numeration of these bits: the least significant bit starts with zero ( $2^0 = 1$ ).

A register can either store numbers from 0 to 255 (positive number, no negative values), or numbers from -128 to +127 (whole number with a sign bit in bit 7), or a value representing an ASCII-coded character (e.g. 'A'), or just eight single bits that do not have something to do with each other (e.g. for eight single flags used to signal eight different yes/no decisions).

The special character of registers, compared to other storage sites, is that

- they can be used directly in assembler commands,
- operations with their content require only a single command word,
- they are connected directly to the central processing unit called the accumulator,
- they are source and target for calculations.

There are 32 registers in an AVR. They are originally named R0 to R31, but you can choose to name them to more meaningful names using an assemble directive. An example:

```
.DEF MyPreferredRegister = R16
```

Note that assembler directives like this are only meaningful for the assembler but do not produce any code that is executable in the AVR target chip. Instead of using the register name R16 we can now use our own name MyPreferredRegister, if we want to use R16 within a command. So we write a little bit more text each time we use this register, but we have an association what might be the content of this register.

Using the command line

Main  
Hardware  
ISP  
ParPortProg  
Exp2313  
Boards  
**Why Asm?**  
Asm concept  
Planning Tools  
Edit  
Asm  
ISP  
Studio3  
Studio4  
Structure  
**Register**  
What?  
Which?  
Pointer  
Tables  
Select  
**Ports**  
What?  
Which?  
Status  
**SRAM**  
What?  
Why?  
How?  
Stack  
**Run**





# Beginners Programming in AVR Assembler

The following pages are written for all people that try to program in assembler language for the first time and like to start with programming [ATMEL-AVRs AT90S, ATtiny, ATmega](#). Some basic programming techniques are shown that are necessary. During later learning stages these informations are useful and so try to learn this first. Tables are included for easier programming. There might be some bugs in these pages. Bug reports and hints for improving these pages are very welcome.

## Overview on the beginners programming tutorial

Theme	What is that	Instructions
<a href="#">Hardware</a>	<a href="#">ISP</a> , <a href="#">ParPortProg</a> , <a href="#">Exp2313</a> , <a href="#">Boards</a>	-
<a href="#">Why</a>	<a href="#">Why learn assembler language?</a> , <a href="#">The concept behind assembler</a> , <a href="#">Planning an AVR project</a>	-
<a href="#">Tools</a>	<a href="#">Editor</a> , <a href="#">Assembler</a> , <a href="#">ISP</a> , <a href="#">Studio3</a> , <a href="#">Studio4</a> , <a href="#">Structure</a>	-
<a href="#">Register</a>	<a href="#">What is a Register?</a> <a href="#">Which different registers are available?</a> <a href="#">Register as Pointer</a> <a href="#">What can be packed into a register?</a>	<a href="#">.DEF</a> , <a href="#">LDI</a> , <a href="#">MOV</a> <a href="#">CLR</a> , <a href="#">AND</a> , <a href="#">BRD</a> , <a href="#">CRL</a> , <a href="#">SRCL</a> , <a href="#">SRB</a> , <a href="#">SEF</a> , <a href="#">SUFI</a> <a href="#">LD</a> , <a href="#">ST</a>
<a href="#">Ports</a>	<a href="#">What is a Port?</a> <a href="#">Which Ports are available?</a> <a href="#">Status register as most relevant port</a>	-
<a href="#">SRAM</a>	<a href="#">What is SRAM?</a> <a href="#">For which purposes SRAM is used</a> <a href="#">How to use SRAM?</a> <a href="#">Stack in SRAM</a>	<a href="#">CLX</a> , <a href="#">SEX</a> , <a href="#">BCLR</a> , <a href="#">BSET</a> <a href="#">STS</a> , <a href="#">LDS</a> , <a href="#">LD</a> , <a href="#">ST</a> , <a href="#">STD</a> , <a href="#">LDD</a> <a href="#">PUSH</a> , <a href="#">POP</a> , <a href="#">RCALL</a> , <a href="#">RET</a>
<a href="#">Program run</a>	<a href="#">What happens during a Reset?</a> <a href="#">Linear program runs and branches</a> <a href="#">Timing of commands</a> <a href="#">Macros</a> <a href="#">Subroutines</a> <a href="#">Interrupts</a>	- <a href="#">.CSEG</a> , <a href="#">.DSEG</a> , <a href="#">.ORG</a> , <a href="#">.ESEG</a> , <a href="#">.INC</a> , <a href="#">.BRNE</a> , <a href="#">.BREQ</a> , <a href="#">.BRxx</a> <a href="#">NOP</a> , <a href="#">DEC</a> <a href="#">MACRO</a> , <a href="#">ENDMACRO</a> <a href="#">RET</a> , <a href="#">RCALL</a> , <a href="#">RJMP</a> , <a href="#">SBRC</a> , <a href="#">SBRS</a> , <a href="#">SBIC</a> , <a href="#">SBIS</a> <a href="#">RETI</a>
<a href="#">Calculations</a>	<a href="#">Numbers and characters</a> <a href="#">Bit manipulation</a> <a href="#">Shift and rotate</a> <a href="#">Adding, Subtracting, Compares</a> <a href="#">Format changes</a>	- <a href="#">OR</a> , <a href="#">OR</a> , <a href="#">ANDI</a> , <a href="#">AND</a> , <a href="#">CBR</a> , <a href="#">SBR</a> , <a href="#">EOR</a> , <a href="#">COM</a> , <a href="#">NEG</a> , <a href="#">BLD</a> , <a href="#">CLT</a> , <a href="#">SET</a> , <a href="#">BST</a> <a href="#">LSL</a> , <a href="#">LSR</a> , <a href="#">ASR</a> , <a href="#">ROL</a> , <a href="#">ROR</a> , <a href="#">SWAP</a> <a href="#">ADD</a> , <a href="#">ADC</a> , <a href="#">SUB</a> , <a href="#">SBC</a> , <a href="#">CP</a> , <a href="#">CPC</a> , <a href="#">CPI</a> , <a href="#">TST</a> <a href="#">DB</a> , <a href="#">DW</a>
<a href="#">Tables</a>	<a href="#">Commands by function</a> <a href="#">Commands alphabetic</a> <a href="#">Ports</a> <a href="#">Vectors</a> <a href="#">Abbreviations</a> <a href="#">Directives</a> <a href="#">Advanced</a> <a href="#">Expressions</a> <a href="#">Shift Left</a>	- - - - - - - - all dot directives

Example: What is a Port?

[To the top of this page](#)

If you need a complete description of all instructions of the AVR with explanations you might be happy to have the document [here](#) by Anthony Peck. It also provides small projects with an 8-pin ATtiny13.

Source: [http://www.avr-asm-tutorial.net/avr\\_en/beginner/index.html](http://www.avr-asm-tutorial.net/avr_en/beginner/index.html)



## What is a Port?

Ports in the AVR are gates from the central processing unit to internal and external hard- and software components. The CPU communicates with these components, reads from them or writes to them, e.g. to the timers or the parallel ports. The most used port is the flag register, where results of previous operations are written to and branch conditions are read from.

There are 64 different ports, which are not physically available in all different AVR types. Depending on the storage space and other internal hardware the different ports are either available and accessible or not. Which of these ports can be used is listed in the data sheets for the processor type.

Ports have a fixed address, over which the CPU communicates. The address is independent from the type of AVR. So e.g. the port address of port B is always 0x18 (0x stands for hexadecimal notation). You don't have to remember these port addresses, they have convenient aliases. These names are defined in the include files (header files) for the different AVR types, that are provided from the producer. The include files have a line defining port B's address as follows:

```
.EQU PORTB, 0x18
```

So we just have to remember the name of port B, not its location in the I/O space of the chip. The include file 8515def.inc is involved by the assembler directive

```
.INCLUDE "C:\Somewhere\8515def.inc"
```

and the registers of the 8515 are all defined then and easily accessible.

Ports usually are organised as 8-bit numbers, but can also hold up to 8 single bits that don't have much to do with each other. If these single bits have a meaning they have their own name associated in the include file, e.g. to enable manipulation of a single bit. Due to that name convention you don't have to remember these bit positions. These names are defined in the data sheets and are given in the include file, too. They are provided here in the port tables.

As an example the MCU General Control Register, called MCUCR, consists of a number of single control bits that control the general property of the chip (see the description in [MCUCR in detail](#)). It is a port, fully packed with 8 control bits with their own names (ISC00, ISC01, ...). Those who want to send their AVR to a deep sleep need to know from the data sheet how to set the respective bits. Like this:

```
.DEF MyPreferredRegister = R16
    LDI MyPreferredRegister, 0b00100000
    OUT MCUCR, MyPreferredRegister
    SLEEP
```

The Out command brings the content of my preferred register, a Sleep-Enable-Bit called SE, to the port MCUCR and sets the AVR immediately to sleep, if there is a SLEEP instruction executed. As all the other bits of MCUCR are also set by the above instructions and the Sleep Mode bit SM was set to zero, a mode called half-sleep will result: no further command execution will be performed but the chip still reacts to timer and other hardware interrupts. These external events interrupt the big sleep of the CPU if they feel they should notify the CPU.

Reading a port's content is in most cases possible using the IN command. The following sequence

```
.DEF MyPreferredRegister = R16
    IN MyPreferredRegister, MCUCR
```

**Main**  
**Hardware**  
 ISP  
 ParPortProg  
 Exp2313  
 Boards  
**Why Asm?**  
 Asm concept  
**Planning Tools**  
 Edit  
 Asm  
 ISP  
 Studio3  
 Studio4  
 Structure  
**Register**  
 What?  
 Which?  
 Pointer  
 Tables  
 Select  
**Ports**  
 What?  
 Which?  
 Status  
**SRAM**  
 What?  
 Why?  
 How?  
 Stack  
**Run**  
 Reset  
 Branch  
 Timing  
 Macros  
 Subrout  
 Interrupts  
**Calc**  
 Numbers  
 Bits  
 Shift  
 Add, Sub  
 convert  
**Tables**  
 Commands  
 Functional

AVRbeginners.net
Getting Started
AVR Assembler
Converting Numbers (Also asm)
AVR Architecture
Interfacing Examples
Software
ATmega8 Dev Board

## Welcome to

### - Work In Progress -

Crash course on numbers - hex,  
binary and decimal

### -- News --

06-04-2004: [LCD 4-bit mode](#)  
explanation with asm code!

30-03-2004: [24C16 TWI EEPROM interfacing](#) page with  
C code uploaded.

27-03-2004: [LCD pages](#) and our  
[VERY FIRST C Code](#) example!

We also added an introduction  
to the [TWI](#) in the architecture  
section. Example code is on its  
way.

26-09-2003: The ATmega8  
pages are finished! [Here's the  
intro!](#)

28-08-2003: Added an  
[ATmega8 ADC example!](#)

27-08-2003: The whole site is  
now available as a [zip file!](#)

AVRbeginners.net
Getting Started
AVR Assembler
About AVR Assembler
Flow Charts
MCU Status
Math...
Simple Maths
Multiple Byte Maths
Jumps / Subroutine Calls
Jump varieties
Call varieties
More on indirect jumps
Conditional Branches/Loops
Branches Introduction
Case Structures
For Loops
While Loops
Macros
Directives/Expressions
Converting Numbers (Also asm)
Overview
Number Formats
Ascii Table
Int to...
To Int From...
AVR Architecture
Interfacing Examples
Software
ATmega8 Dev Board

## Conditional Branches

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. Loops (for, while...) make use of this.

Any add, subtract, increment, decrement or logic instruction for example leaves a status that can be used for almost any branch instruction the AVR offers. There are as well some tests which set status flags based on their arguments. Basically they are just a subtraction: Comparing two numbers to each other is done by subtracting one from the other. The result of  $a - b$  can be negative ( $b > a$ ), positive ( $b < a$ ) and zero ( $b = a$ ). This information is stored in the status register. When two numbers are added to each other, it can happen that the 8-bit result is greater than 255 and therefore "rolls over". In this case, the carry bit in SREG is set.

Some examples:

```
subi r16, 5           ; r16 = r16 - 5
breq r16_is_0         ; r16 was 5, handle that
brlo r16_is_lower0    ; r16 was lower than 5
r16_is_greater5:     ; r16 was higher than 5
```

Now some examples for conditional branches in loops:

```
ldi r16, 5             ; load desired loop count into r16
loop:
dec r16                ; decrease loop count
brne loop               ; if not equal (result=0), loop again
;
clr r16                ; clear counter
loop:
inc r16                ; increase loop count
cpi r16, 5              ; compare to desired loop count
brne loop               ; if not reached, loop again
```

Here is a list of simple tests which can also be used for branch instructions (exception: cpse - this instruction performs a compare and skips the next instruction if equal):

Valid SREG flags after test:

instruction:	arg 1:	arg 2:	"action":	I	T	H	S	V	N	Z	C
cpi	reg	const	reg - const	-	-	<>	<>	<>	<>	<>	<>
cp	reg	reg	reg - reg	-	-	<>	<>	<>	<>	<>	<>
cpc	reg	reg	reg - reg - C	-	-	<>	<>	<>	<>	<>	<>
tst	reg	---	reg AND reg	-	-	-	<>	0	<>	<>	-
cpse	reg	reg	reg - reg	-	-	-	-	-	-	-	-

Source: <http://www.avrbeginners.net/>

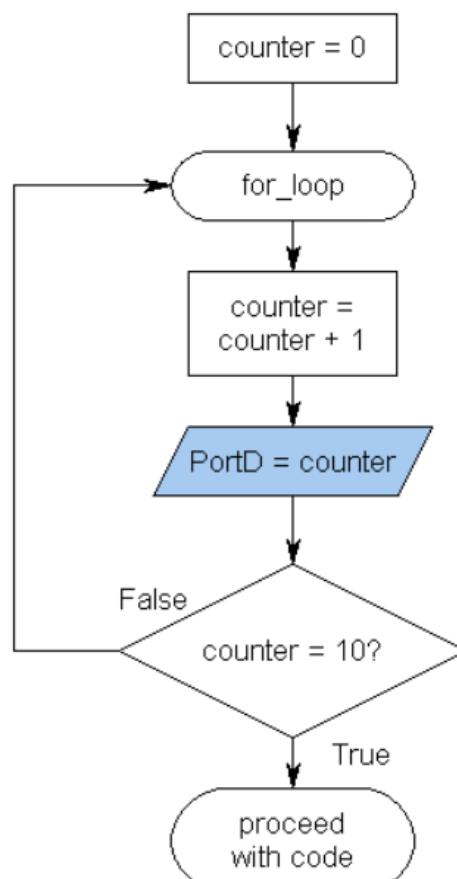
If you want more info on which results change which SREG flag, see the AVRStudio assembler help. Here is a

Flow charts are a graphical representation of code, Program states or even SRAM contents, if used in a creative way. Once you know how to use them for code you'll quickly develop your own style to create flow charts for almost anything.

When you think about implementing a special algorithm or peripheral driver it might be better to have a flow chart already done before you start hacking code. That will save lots of time. Trust me. I know. If you have code that is not sufficiently commented or just BIG, analyse it by making up a flow chart. Very often that helps, especially when you got it from the web.

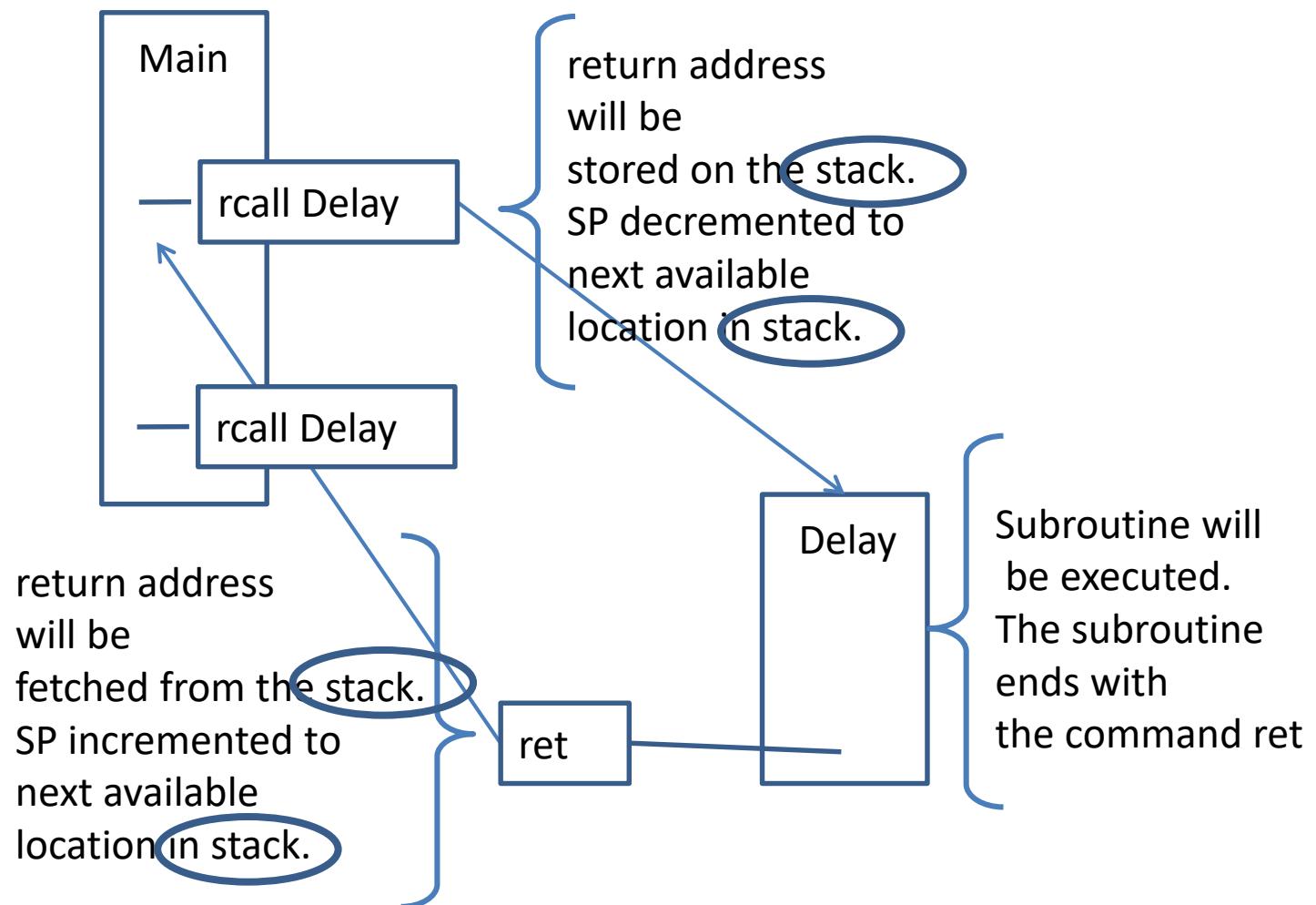
Especially when writing code in assembler they are a great help, because assembler instructions are not always self-explanatory and even well-structured code will get hard to read once it has grown to a certain size.

Here is a small example flow chart:



You can find a good program for editing flow charts at [www.rff.com](http://www.rff.com). But a piece of paper will do the job too if you need to make up one.

# Subroutine call, example.



# Stack

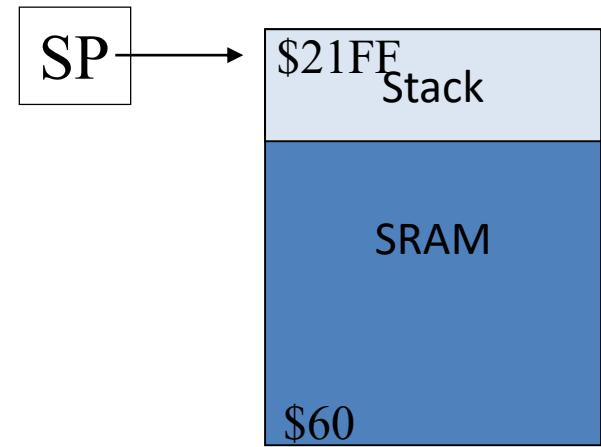


# Stack

- The stack is an area of SRAM that grows downward from a fixed reference point
  - Special instructions and I/O register support stack access
    - RET, CALL, PUSH, POP, ...
    - SP (Stack Pointer)
  - The stack requires initialization
    - Set SP to an appropriate address (empty stack)
  - The stack is managed by application and is used by the processor interrupt system

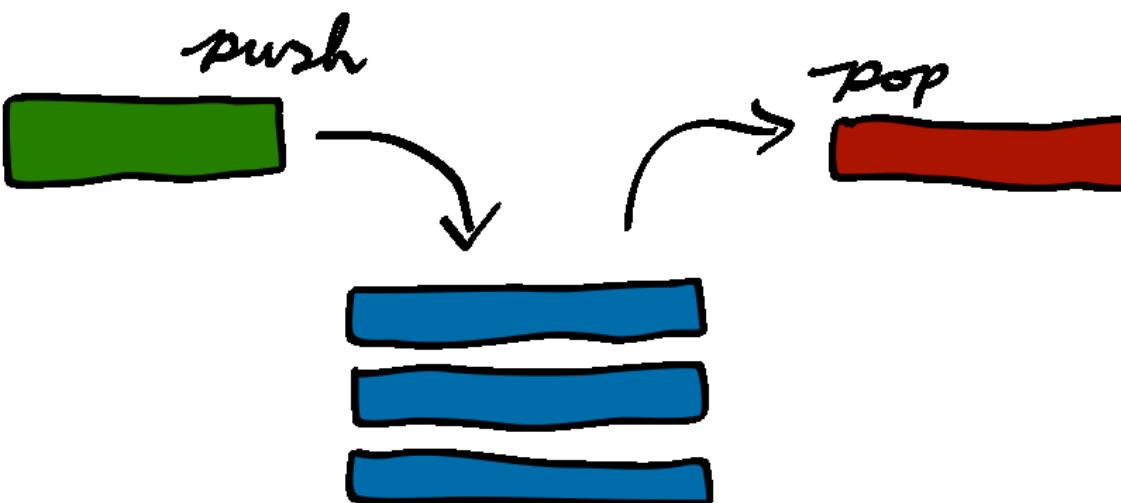
# Initiate the Stack Pointer, SP

- Typically, the stack starts at the highest SRAM address and grows downward through memory
- SP points to the next available byte of stack storage (top of stack after a push)
- Initialization:
  - ldi R16, high(RAMEND)
  - out SPH, R16
  - ldi R16, low(RAMEND)
  - out SPL, R16
- RAMEND = highest address in RAM

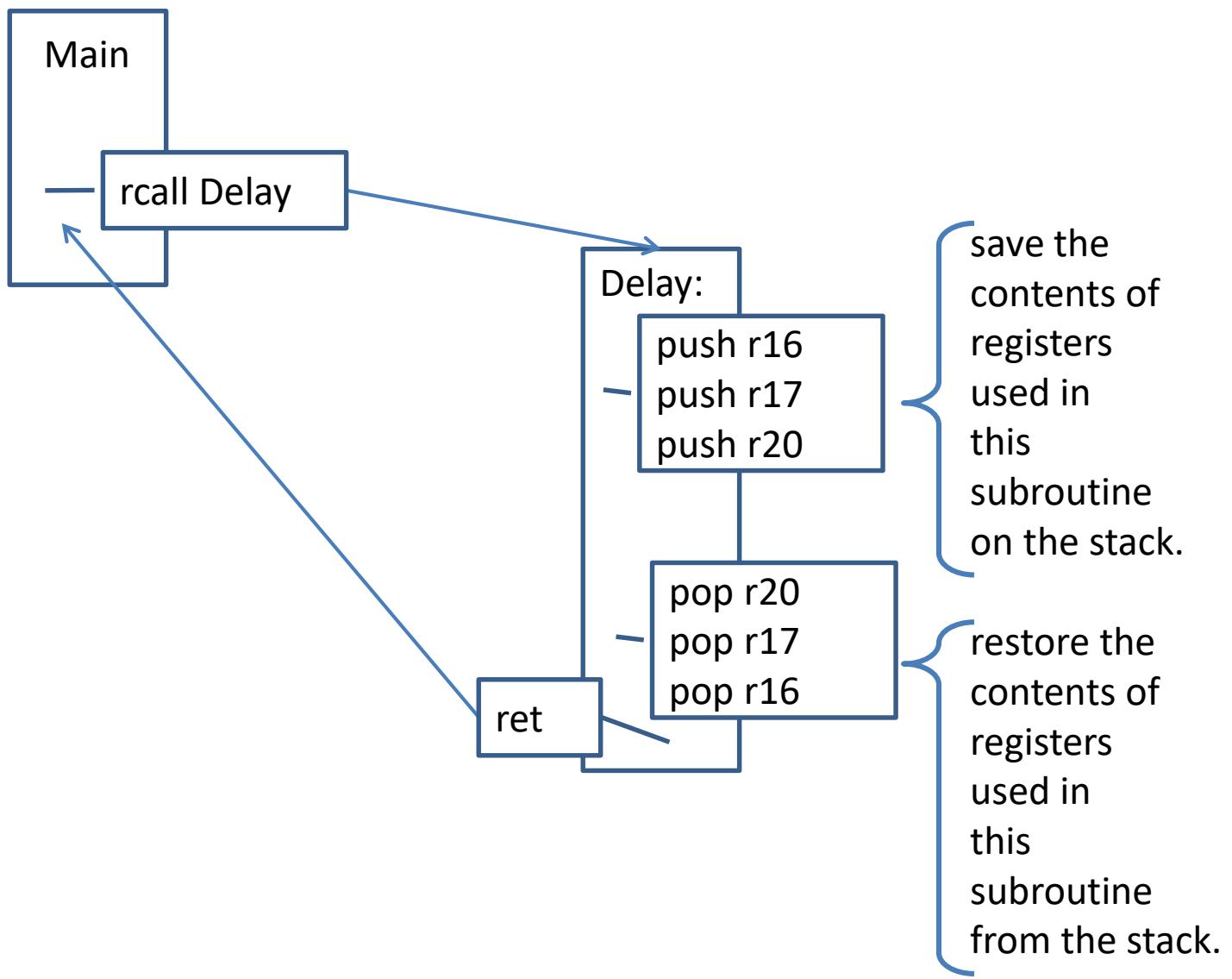


push and pop,

- for store and restore register contents



# Subroutine call, example.



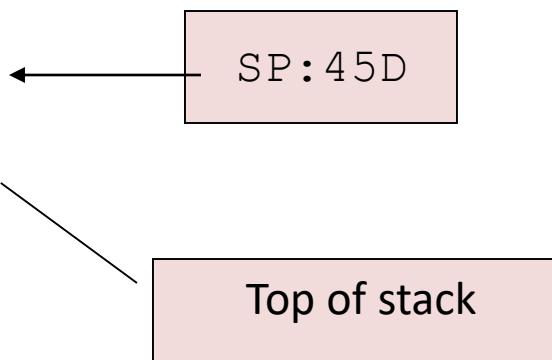
# PUSH and POP

- push Rr
- Register data copied to byte at address in SP
- SP is decremented
- pop Rd
- SP is incremented
- Byte at address in SP is copied to Rd

# PUSH

push R5

Address:	Content:
45A	2C
45B	34
45C	36
45D	11
45E	F2
45F	5C

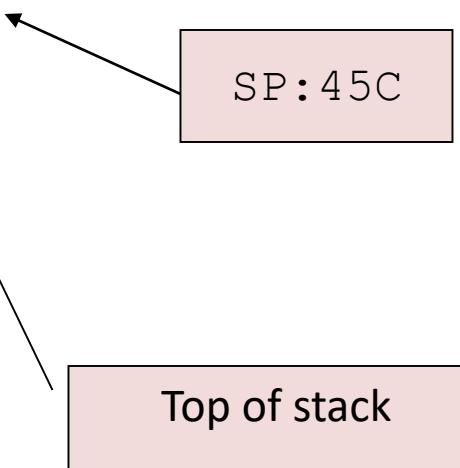


Register:	Content:
R3	7F
R4	00
R5	A2
R6	77

# POP

pop R4

Address:	Content:
45A	2C
45B	34
45C	36
45D	A2
45E	F2
45F	5C



Register:	Content:
R3	7F
R4	00
R5	A2
R6	77

# Stack Usage

- Temporary storage of information
- Storage of return addresses for procedure calls and interrupts
- Storage for local variables during execution of a procedure

# Temporary Storage

- push R16 ;save register
- ldi R16, \$FF ;use register
- out PORTB, R16
- pop R16 ;restore register

# Program example

- Simple program with subroutine call
- Saving register contents on stack in subroutine

C:\Document\\_\Kurser\Datorteknik\HT\_2019\Program\_examples\Lecture\_examples\Ex...

SP, Stack Pointer  
initializing

Data direction  
register initializing

Subroutine

The screenshot shows a development environment for a microcontroller. On the left is a table of processor registers and their values. In the center is an assembly code editor with comments explaining the operations. On the right are two windows showing port configurations.

**Registers:**

Name	Value
Program Counter	0x000000
Stack Pointer	0x0000
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	0
Frequency	4.0000 MHz
Stop Watch	0.00 us
SREG	[Binary]
Registers	[Yellow Box]
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0x00
R17	0x00
R18	0x00
R19	0x00

**Assembly Code:**

```
.include "m2560def.inc"
ldi r16, high(RAMEND)           ; high part of highest RAM address to r16
out SPH, r16                    ; write a Stack Pointer, SPH
ldi r16, low(RAMEND)            ; high part of lowest RAMM-address to r16
out SPL, r16                    ; write a Stack Pointer, SPL

ldi r16, 0xFF                   ; Set Data Direction Registers
out DDRB, r16                   ; port B as outputs
ldi r16, 0x00                   ; port A as inputs
out DDRA, r16

ldi r16, 170                     ; set register r16

main_loop:
out PORTB, r16
rcall subr_01                    ; call subroutine
rjmp main_loop                  ; infinite loop

subr_01:
push r16                        ; save r16 on stack
push r17                        ; save r17 on stack
com r16                         ; invert content in r16
com r17                         ; invert content in r17
pop r17                         ; restore r17 from stack
pop r16                         ; restore r16 from stack
ret                            ; return to main program
```

**Port Configuration (PORTB):**

Name	Value
PORTB	0x00
Port B Data Register	0x00
Port B Data Direction Register	0x00
Port B Input Pins	0x00

**Port Configuration (PORTC):**

Name	Address	Value	Bits
PORTC	0x04 (0x24)	0x00	[Binary]

**Port Configuration (PORTF):**

Name	Address	Value	Bits
PORTF	0x03 (0x23)	0x00	[Binary]

**Port Configuration (PORTB):**

Name	Address	Value	Bits
PORTB	0x05 (0x25)	0x00	[Binary]

**Main loop**

File Project Build Edit View Tools Debug Window Help

Build F7

Build and Run Ctrl+F7

Trace Disabled

Project Example\_2019\_09\_10\_Stack\_Pointer

- Source Files
  - Example\_2019\_09\_10\_Stack\_Pointer
- Included Files
- Labels
- Output
- Object File

.include "m2560def.inc"

```

ldi r16, high(RAMEND) ; high part of highest RAM-address to r16
out SPH, r16            ; write o Stack Pointer, SPH
ldi r16, low(RAMEND)   ; high part of highest RAM-address to r16
out SPL, r16            ; write o Stack Pointer, SPL

ldi r16, 0xFF           ; Set Data Direction Registers
out DDRB, r16            ; port B as outputs
ldi r16, 0x00           ;
out DDRA, r16            ; port A as inputs

ldi r16, 170             ; set register r16

main_loop:
    ldi PORTB, r16

    rcall subr_01          ; call subroutine

    rjmp main_loop         ; infinite loop

subr_01:
    push r16              ; save r16 on stack
    push r17              ; save r17 on stack

    com r16                ; invert content in r16
    com r17

    pop r17               ; restore r17 from stack
    pop r16               ; restore r16 from stack

    ret                   ; return to main program

```

Name

- AD\_CONVERTER
- ANALOG
- BOOT
- CPU
- EEPROM
- EXTERNAL\_INTERRUPT
- JTAG
- PORTA
- PORTB
- PORTC
- PORTD
- PORTE
- PORTF
- PORTG
- PORTH
- PORTI
- PORTJ
- PORTK
- PORTL
- PORTM
- PORTN
- PORTO
- PORTP
- PORTQ
- PORTR
- PORTS
- PORTT
- PORTU
- PORTV
- PORTW
- PORTX
- PORTY

Name

- DDRB
- PINB
- PORTB

Build

Segment	Begin	End	Code	Data	Used	Size	Use%

Build Message Find in Files Breakpoints and Tracepoints

Compile and start simulator

The screenshot shows a microcontroller development environment with several windows:

- Registers View:** Shows memory locations R00 to R25 with their corresponding hex values.
- Code View:** Displays assembly code for a program named "Example\_2019\_09\_10\_Stack". The code includes comments and instructions like LDI, OUT, and RCALL. A yellow arrow points to the first instruction, **ldi r16, high(RAMEND)**, which is highlighted with a blue oval.
- I/O View:** Shows the state of various pins: DDRB (0x04), PINB (0x03), and PORTB (0x05). A callout bubble with a yellow arrow points to the "Value" column for PORTB, indicating it represents the Program Counter (PC).
- Memory Dump View:** Shows the memory dump for the current project.

**Yellow arrow indicates PC, Program Counter**

Name	Address	Value	Bits
DDRB	0x04 (0x24)	0x00	□□□□□
PINB	0x03 (0x23)	0x00	□□□□□
PORTB	0x05 (0x25)	0x00	□□□□□

PC, Program Counter  
also here

Yellow arrow  
indicates PC, Pro-  
Counter

The screenshot shows a development environment for a microcontroller. On the left, the 'Processor' window displays various registers and their values. A blue oval highlights the 'Program Counter' register, which has a value of 0x000000. A yellow arrow points from this highlighted PC to the assembly code in the center, specifically pointing to the instruction 'ldi r16, high(RAMEND)'. Another blue oval surrounds this same instruction in the code editor. The central area contains the assembly code for the program, including comments explaining the stack setup. To the right, the 'I/O View' window lists various pins and their configurations.

Name	Value
Program Counter	0x000000
Stack Pointer	0x21FF
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	0
Frequency	4.0000 MHz
Stop Watch	0.00 us
SREG	0000 0100 0000 0000

Registers

R00 R01 R02 R03 R04 R05 R06 R07 R08 R09 R10 R11 R12 R13 R14 R15

```
File: C:\Document\Kurser\Datorteknik\HT_2019\Program_examples\Lecture_examples\Example_2019_09_10_Stack.s
ID: 1
Date: 2019-09-09
Author: Anders Haggren
Function:

Lecture examples 2019-09-10, lecture #3

Use of Stack and Stack Pointer setup

;-----<
.include "m2560def.inc"

ldi r16, high(RAMEND) ; write high part of highest RAM-address to r16
out SPH, r16            ; write high part of highest RAM-address to r16
ldi r16, low(RAMEND)   ; write low part of highest RAM-address to r16
out SPL, r16             ; write low part of highest RAM-address to r16

ldi r16, 0xFF            ; Set Data Direction Registers
out DDRB, r16             ; port B as outputs
ldi r16, 0x00            ; port A as inputs
out DDRA, r16

ldi r16, 170              ; set register r16

main_loop:
out PORTB, r16           ; call subroutine
rcall subr_01
njmp main_loop            ; infinite loop
```

I/O View

- + AD\_CONV
- + ANALOG
- + BOOT\_LC
- + CPU
- + EEPROM
- + EXTERN
- + JTAG
- + PORTA

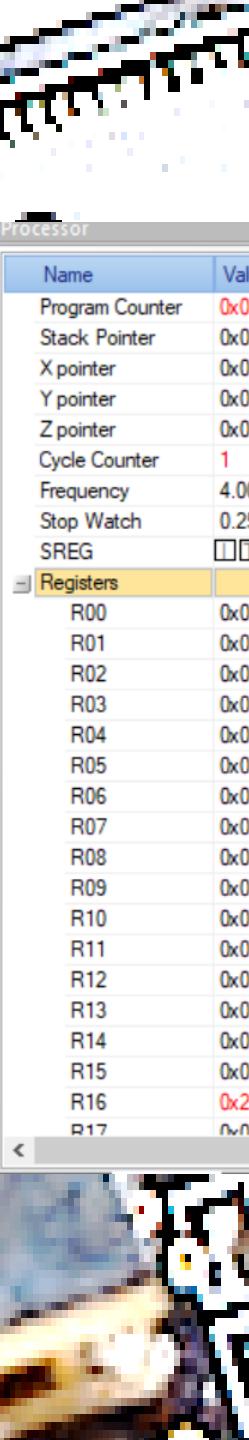
Project Processor

C:\Document\Kurser\Datorteknik\HT\_2019\Program\_examples\Lecture\_examples\Example\_2019\_09\_10\_Stack.s

Name	Value
Program Counter	0x000001
Stack Pointer	0x21FF
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	1
Frequency	4.0000 MHz
Stop Watch	0.25 us
SREG	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0x21
17	0x00

PC has moved to next line.  
Next instruction,  
out SPH, r16

R16 is loaded with  
the value 0x21



Processor

Name	Value
Program Counter	0x000001
Stack Pointer	0x0000
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	1
Frequency	4.0000 MHz
Stop Watch	0.25 us
SREG	
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0x21
R17	n.nn

C:\Document\Kurser\Datorteknik\HT\_2019\Program\_examples\Lecture\_exa...

```
ldi r16, high(RAMEND) ; high part of highest RAMN-address to r16
but SPH, r16 ; write o Stack Pointer, SPH
ldi r16, low(RAMEND) ; high part of highest RAMN-address to r16
out SPL, r16 ; write o Stack Pointer, SPL

ldi r16, 0xFF ; Set Data Direction Registers
out DDRB, r16 ; port B as outputs
ldi r16, 0x00
out DDRA, r16 ; port A as inputs

ldi r16, 170 ; set register r16

main_loop:
out PORTB, r16

rcall subr_01 ; call subroutine

rjmp main_loop ; infinite loop

subr_01:
push r16 ; save r16 on stack
push r17 ; save r17 on stack

com r16 ; invert content in r16
com r17

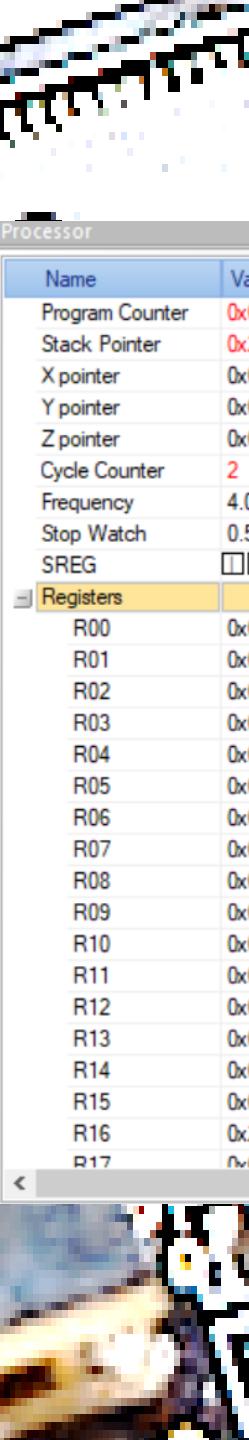
pop r17 ; restore r17 from stack
pop r16 ; restore r16 from stack

ret ; return to main program
```

I/O View

Name	Value
AD_CONVERTER	
ANALOG_COMPARA...	
BOOT_LOAD	
CPU	
EEPROM	
EXTERNAL_INTERRUPT	
JTAG	
PORTA	
PORTB	
PORTC	
PORTD	
PORTE	
PORTF	
PORTG	
PORTU	

Name	Address	Value
------	---------	-------



Processor

Name	Value
Program Counter	0x000002
Stack Pointer	0x2100
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	2
Frequency	4.0000 MHz
Stop Watch	0.50 us
SREG	
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0x21
R17	0x00

C:\Document\\_\Kurser\Datorteknik\HT\_2019\Program\_examples\Lecture\_exa...

```
ldi r16, high(RAMEND) ; high part of highest RAM-address to r16
out SPH, r16 ; write o Stack Pointer, SPH
→ ldi r16, low(RAMEND) ; high part of highest RAM-address to r16
out SPL, r16 ; write o Stack Pointer, SPL

ldi r16, 0xFF ; Set Data Direction Registers
out DDRB, r16 ; port B as outputs
ldi r16, 0x00 ;
out DDRA, r16 ; port A as inputs

ldi r16, 170 ; set register r16

main_loop:
out PORTB, r16

rcall subr_01 ; call subroutine

rjmp main_loop ; infinite loop

subr_01:
push r16 ; save r16 on stack
push r17 ; save r17 on stack

com r16 ; invert content in r16
com r17

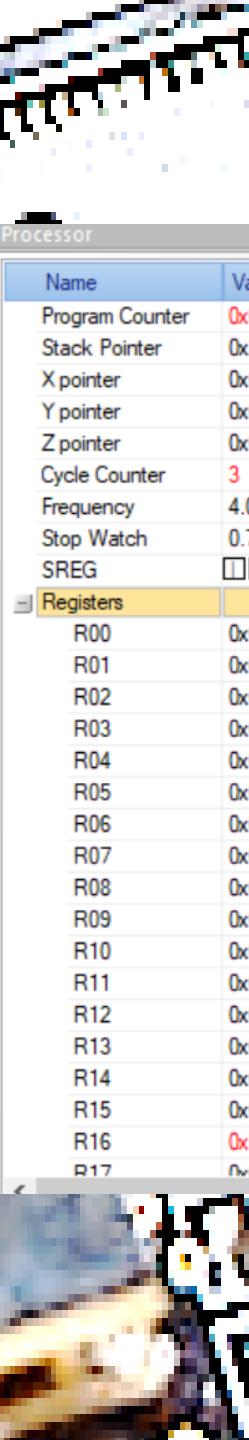
pop r17 ; restore r17 from stack
pop r16 ; restore r16 from stack

ret ; return to main program
```

I/O View

ANALOG\_CO

Name	Address
AD_CONVERTER	
ANALOG_COMPARA...	
BOOT_LOAD	
CPU	
EEPROM	
EXTERNAL_INTERRUPT	
JTAG	
PORTA	
PORTB	
PORTC	
PORTD	
PORTE	
PORTF	
PORTG	
PORTU	



## Processor

Name	Value
Program Counter	0x000003
Stack Pointer	0x2100
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	3
Frequency	4.0000 MHz
Stop Watch	0.75 us
SREG	
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0xFF
R17	0x00

C:\Document\\_\Kurser\Datorteknik\HT\_2019\Program\_examples\Lecture\_exa...

```
ldi r16, high(RAMEND) ; high part of highest RAMN-address to r16
out SPH, r16           ; write o Stack Pointer, SPH
ldi r16, low(RAMEND)  ; high part of highest RAMN-address to r16
put SPL, r16           ; write o Stack Pointer, SPL

ldi r16, 0xFF           ; Set Data Direction Registers
out DDRB, r16           ; port B as outputs
ldi r16, 0x00           ;
out DDRA, r16           ; port A as inputs

ldi r16, 170             ; set register r16

main_loop:
out PORTB, r16

rcall subr_01            ; call subroutine

rjmp main_loop           ; infinite loop

subr_01:
push r16                ; save r16 on stack
push r17                ; save r17 on stack

com r16                 ; invert content in r16
com r17

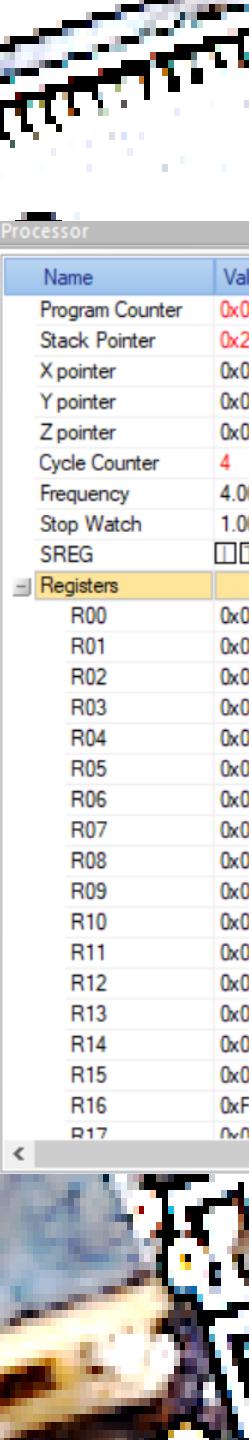
pop r17                 ; restore r17 from stack
pop r16                 ; restore r16 from stack

ret                     ; return to main program
```

## I/O View

Name
+ AD_CONVERTER
+ ANALOG_COMPARA...
+ BOOT_LOAD
+ CPU
+ EEPROM
+ EXTERNAL_INTERRUPT
+ JTAG
+ PORTA
+ PORTB
+ PORTC
+ PORTD
+ PORTE
+ PORTF
+ PORTG
+ PORTH

Name Address



Processor

Name	Value
Program Counter	0x000004
Stack Pointer	0x21FF
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	4
Frequency	4.0000 MHz
Stop Watch	1.00 us
SREG	
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0xFF
R17	0x00

C:\Document\\_\Kurser\Datorteknik\HT\_2019\Program\_examples\Lecture\_exa...

```
ldi r16, high(RAMEND) ; high part of highest RAMN-address to r16
out SPH, r16 ; write o Stack Pointer, SPH
ldi r16, low(RAMEND) ; high part of highest RAMN-address to r16
out SPL, r16 ; write o Stack Pointer, SPL

ldi r16, 0xFF ; Set Data Direction Registers
out DDRB, r16 ; port B as outputs
ldi r16, 0x00 ; port A as inputs
out DDRA, r16

ldi r16, 170 ; set register r16

main_loop:
out PORTB, r16

rcall subr_01 ; call subroutine

rjmp main_loop ; infinite loop

subr_01:
push r16 ; save r16 on stack
push r17 ; save r17 on stack

com r16 ; invert content in r16
com r17

pop r17 ; restore r17 from stack
pop r16 ; restore r16 from stack

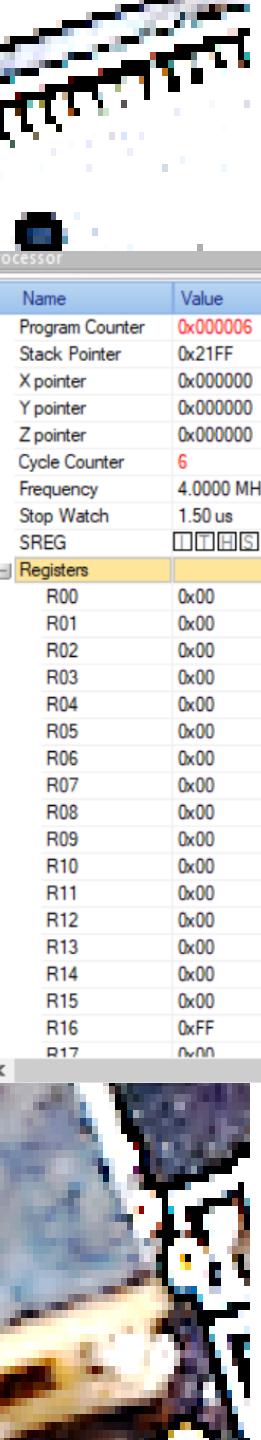
ret ; return to main program
```

I/O View

ANALOG\_COMPARA...

Name	Value
AD_CONVERTER	0
ANALOG_COMPARA...	0
BOOT_LOAD	0
CPU	0
EEPROM	0
EXTERNAL_INTERRUPT	0
JTAG	0
PORTA	0
PORTB	0
PORTC	0
PORTD	0
PORTE	0
PORTF	0
PORTG	0
PORTH	0

Name	Address	Value
------	---------	-------



Processor

Name	Value
Program Counter	0x000006
Stack Pointer	0x21FF
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	6
Frequency	4.0000 MHz
Stop Watch	1.50 us
SREG	00000000

Registers

R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0xFF
R17	0x00

C:\Document\Kurser\Datorteknik\HT\_2019\Program\_examples\Lecture\_exa... □ X

>>>>>>>>>>>>>>>>>>>>>>

1DT301, Computer Technology I  
Date: 2019-09-09  
Author:  
Anders Haggren  
Function:

Lecture examples 2019-09-10, lecture #3  
Use of Stack and Stack Pointer setup

<<<<<<<<<<<<<<<<<<<<<<<<

```
.include "m2560def.inc"

ldi r16, high(RAMEND)           ; high part of highest RAMN-address to r16
out SPH, r16                    ; write o Stack Pointer, SPH
ldi r16, low(RAMEND)            ; high part of highest RAMN-address to r16
out SPL, r16                    ; write o Stack Pointer, SPL

ldi r16, 0xFF                   ; Set Data Direction Registers
out DDRB, r16                  ; port B as outputs
ldi r16, 0x00                   ; port A as inputs
out DDRA, r16

ldi r16, 170                     ; set register r16

main_loop:
out PORTB, r16

rcall subr_01                     ; call subroutine
```

I/O View

PORTR

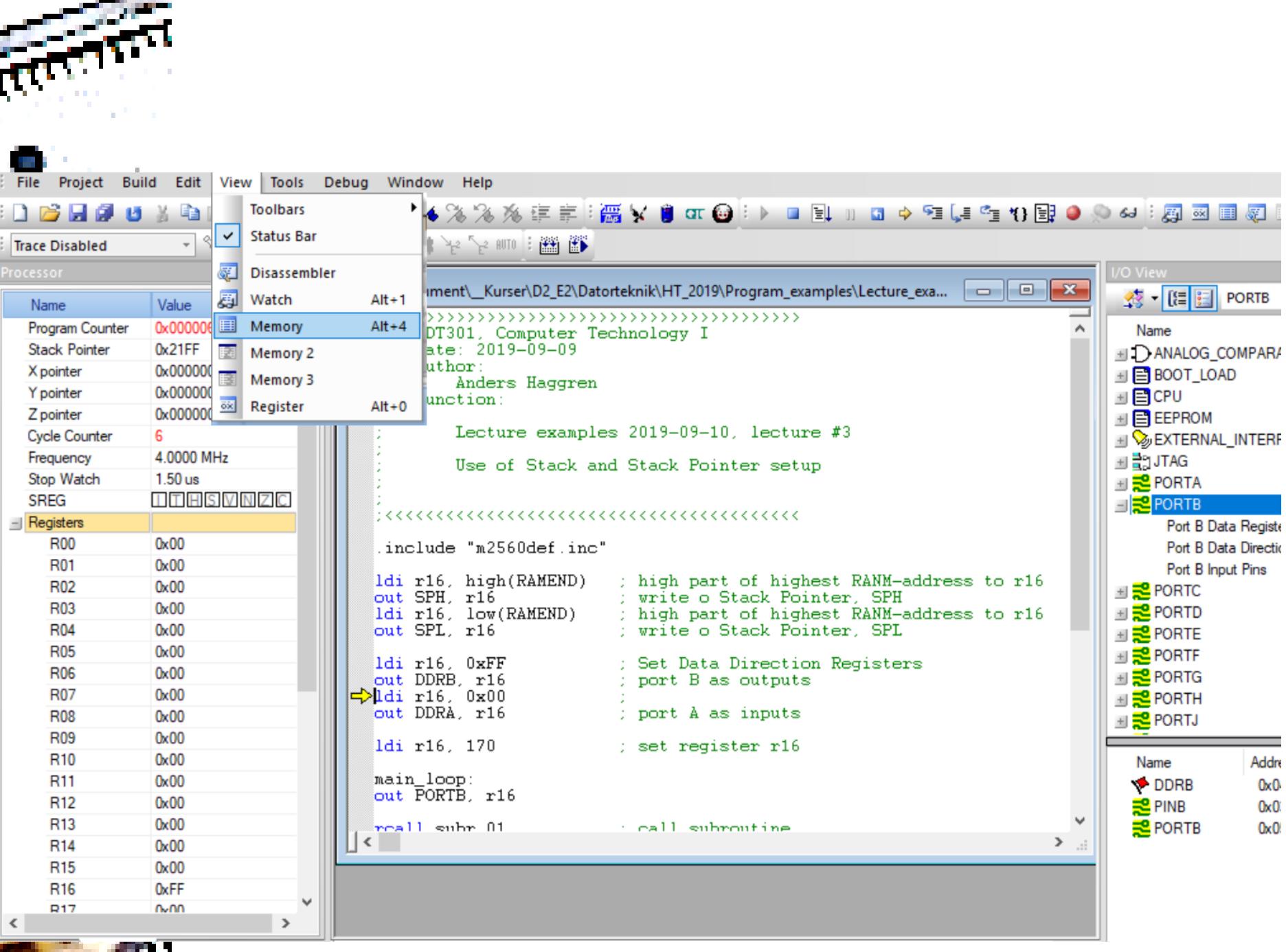
Name	Value
ANALOG_COMPARA...	
BOOT_LOAD	
CPU	
EEPROM	
EXTERNAL_INTERRUPT	
JTAG	
PORTR	
PORTR	
PORTB	
Port B Data Register	0x00
Port B Data Directio...	0xFF
Port B Input Pins	0x00

PORTR

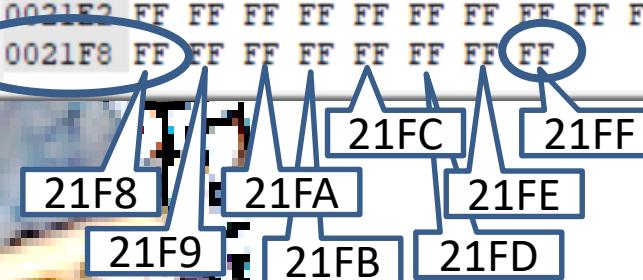
PORTC
PORTD
PORTE
PORTF
PORTG
PORTR
PORTJ

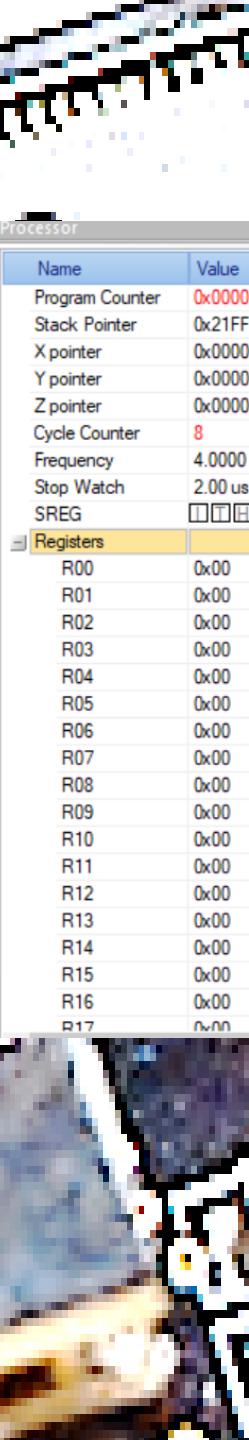
PORTB

Name	Address	Value	Bits
DDRB	0x04 (0x24)	0xFF	██████████
PINB	0x03 (0x23)	0x00	□□□□□□
PORTB	0x05 (0x25)	0x00	□□□□□□



## Memory





Processor

Name	Value
Program Counter	0x000008
Stack Pointer	0x21FF
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	8
Frequency	4.0000 MHz
Stop Watch	2.00 us
SREG	00HVVVNNNN
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0x00
R17	0x00

.include "m2560def.inc"

```
ldi r16, high(RAMEND) ; high part of highest RAMN-address to r16
out SPH, r16 ; write o Stack Pointer, SPH
ldi r16, low(RAMEND) ; high part of highest RAMN-address to r16
out SPL, r16 ; write o Stack Pointer, SPL

ldi r16, 0xFF ; Set Data Direction Registers
out DDRB, r16 ; port B as outputs
ldi r16, 0x00 ; port A as inputs
out DDRA, r16

ldi r16, 170 ; set register r16

main_loop:
out PORTB, r16

rcall subr_01 ; call subroutine

rjmp main_loop ; infinite loop

subr_01:
push r16 ; save r16 on stack
push r17 ; save r17 on stack

com r16 ; invert content in r16
com r17 ; invert content in r17

pop r17 ; restore r17 from stack
```

I/O View

ANALOG\_COMPARATOR

Name	Value
D	ANALOG_COMPARE...
BOOT_LOAD	
CPU	
EEPROM	
EXTERNAL_INTERRUPT	
JTAG	
PORTA	
PORTB	Port B Data Register 0x00 Port B Data Directio... 0xFF Port B Input Pins 0x00
PORTC	
PORTD	
PORTE	
PORTF	

Name	Address	Value	Bits
DDR B	0x04 (0x24)	0xFF	1111 1111 1111 1111
PIN B	0x03 (0x23)	0x00	0000 0000 0000 0000
PORT B	0x05 (0x25)	0x00	0000 0000 0000 0000

R16 will be loaded with decimal value 170. What is that in binary?

Content of R16 will be copied to PORTB

The screenshot shows a development environment with three main windows:

- Processor View:** Shows the Program Counter at 0x000009, Stack Pointer at 0x21FF, and various pointers (X, Y, Z) set to 0x000000. The Registers window shows R00-R15 all at 0x00, and R16 at 0xAA.
- Code Editor:** Displays assembly code for a main loop. The instruction `out PORTB, r16` is highlighted with a blue oval and an arrow pointing to it from the text above. The code also includes initializations for stack pointers, port directions, and a subroutine call.
- I/O View:** Shows the state of various pins. The **PORTB** row is selected, showing its Data Register at 0x00, Data Direction Register at 0xFF, and Input Pins at 0x00. Below it, other ports (PORTC, PORTD, PORTE) are listed.

PORTB

R16 has got  
hexadecimal value AA  
which is equal to  
decimal 170.

The screenshot shows a development environment for a PIC16F877A microcontroller. The interface includes:

- Processor View:** Displays system status and registers. The Program Counter is at 0x00000A, Stack Pointer is at 0x21FF, and X, Y, Z pointers are at 0x000000. The SREG register is shown with all bits set to 0.
- Registers View:** Shows the state of all 16-bit registers (R00-R16). R16 contains the value 0xAA.
- Code View:** Displays the assembly code for the program. A yellow arrow points to the `rcall subr_01` instruction. The code also includes initializations for RAMEND, stack pointers, and port configurations.
- I/O View:** Shows the state of various pins and ports. The PORTB register is highlighted, containing the value 0xAA. Other pins like DDRB, PINB, and PORTC are also listed.

A callout box highlights the PORTB entry in the I/O View, stating: "PORTB got the hexadecimal value AA which is binary 1010 1010".

Name	Value
Program Counter	0x00000A
Stack Pointer	0x21FF
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	10
Frequency	4.0000 MHz
Stop Watch	2.50 us
SREG	0000 0000 0000 0000
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0xAA

```
.include "m2560def.inc"

ldi r16, high(RAMEND)           ; high part of highest RAMN-address to r16
out SPH, r16                    ; write o Stack Pointer, SPH
ldi r16, low(RAMEND)            ; high part of highest RAMN-address to r16
out SPL, r16                    ; write o Stack Pointer, SPL

ldi r16, 0xFF                   ; Set Data Direction Registers
out DDRB, r16                  ; port B as outputs
ldi r16, 0x00                   ; port A as inputs
out DDRA, r16

ldi r16, 170                   ; set register r16

main_loop:
    out PORTB, r16              ; call subroutine
    rjmp main_loop              ; infinite loop

subr_01:
    push r16                   ; save r16 on stack
    push r17                   ; save r17 on stack
    com r16                     ; invert content in r16
    com r17
    pop r17                   ; restore r17 from stack
```

Name	Value
ANALOG_COMPARA...	
BOOT_LOAD	
CPU	
EEPROM	
EXTERNAL_INTERRUPTS	
JTAG	
PORTR	
PORTB	0xAA
Port B Data Register	0xAA
Port B Data Directio...	0xFF
Port B Input Pins	0x00
PORTR	
PORTRD	
PORTE	
PORTE	

**PORTB got the hexadecimal value AA which is binary 1010 1010**

The screenshot shows a development environment for a microcontroller, likely an Atmel AVR, with the following windows:

- Processor**: Shows system status and registers. The Program Counter is at 0x00000A. Registers R00-R17 are all 0x00, except R16 which is 0xAA.
- Code View**: Displays assembly code. A blue oval highlights the instruction `rcall subr_01`. Another blue oval highlights the label `subr_01`. A callout box labeled "Subroutine subr\_01" points to the label. A callout box labeled "Subroutine call rcall subr\_01" points to the `rcall` instruction.
- I/O View**: Shows port pins. The PORTB register is highlighted, showing its value is 0xAA. Callouts point to the `PORTB` entry in the tree and the `PORTB` row in the table.

**Assembly Code (Code View)**

```
;-----  
.include "m2560def.inc"  
  
ldi r16, high(RAMEND) ; high part of highest RAM-address to r16  
out SPH, r16 ; write o Stack Pointer, SPH  
ldi r16, low(RAMEND) ; high part of highest RAM-address to r16  
out SPL, r16 ; write o Stack Pointer, SPL  
  
ldi r16, 0xFF ; Set Data Direction Registers  
out DDRB, r16 ; port B as outputs  
ldi r16, 0x00 ;  
out DDRA, r16 ; port A as inputs  
  
ldi r16, 170 ; set register r16  
  
main_loop:  
    out PORTB, r16 ; call subroutine  
    rjmp main_loop ; infinite loop  
  
subr_01:  
    push r16 ; save r16 on stack  
    push r17 ; save r17 on stack  
  
    com r16 ; invert content in r16  
    com r17 ;  
  
    pop r17 ; restore from stack
```

PC, Program Counter =  
0x00000A

The screenshot shows a development environment for a microcontroller. On the left, the 'Processor' window displays various寄存器 (Registers) and their values. The 'Program Counter' is highlighted with a red oval and has a value of 0x00000A. The 'Stack Pointer' is also highlighted with a red oval and has a value of 0x21FF. In the center, the assembly code window shows the main loop and a subroutine. A yellow arrow points from the 'rjmp main\_loop' instruction to the 'subr\_01' label. On the right, the 'I/O View' window shows the state of various pins and ports. The PORTB register is selected, with its value set to 0xAA. Below it, the DDRB, PINB, and PORTD registers are listed with their addresses and current values.

Name	Address	Value	Bits
DDRB	0x04 (0x24)	0xFF	████████████████
PINB	0x03 (0x23)	0x00	□□□□□□□□
PORTB	0x05 (0x25)	0xAA	██████████████

SP, Stack Pointer  
= 0x21FF

PC, Program Counter =  
0x 00 00 0C

The screenshot shows a development environment for a microcontroller. On the left is a 'Registers' window displaying various processor states. The 'Program Counter' is highlighted with a red circle and has a callout pointing to the text 'PC, Program Counter = 0x 00 00 0C'. The 'Stack Pointer' (SP) is also circled in red and has a callout pointing to the text 'SP, Stack Pointer = 0x21FC'. In the center is the assembly code editor showing the following sequence:

```
.include "m2560def.inc"
ldi r16, high(RAMEND) ; high part of highest RAMN-address to r16
out SPH, r16            ; write o Stack Pointer, SPH
ldi r16, low(RAMEND)   ; high part of highest RAMN-address to r16
out SPL, r16            ; write o Stack Pointer, SPL
ldi r16, 0xFF           ; Set Data Direction Registers
out DDRB, r16            ; port B as outputs
ldi r16, 0x00           ; port A as inputs
out DDRA, r16
ldi r16, 170             ; set register r16
main_loop:
out PORTB, r16
rcall subr_01             ; call subroutine
rjmp main_loop           ; infinite loop
subr_01:
push r16                ; save r16 on stack
push r17                ; save r17 on stack
com r16                 ; invert content in r16
com r17
pop r17                 ; restore r17 from stack
```

On the right is an 'I/O View' window showing the state of various pins. The 'PORTB' section is selected, and its value is 0xAA. Below it, the memory map for PORTB is shown:

Name	Address	Value	Bits
DDR B	0x04 (0x24)	0xFF	████████████████
PIN B	0x03 (0x23)	0xAA	███□□□□□□
PORT B	0x05 (0x25)	0xAA	███□□□□□□

SP, Stack Pointer  
= 0x21FC



## Memory

Data

8/16 abc

Address: 0x2040

Cols: Auto

002040	FF	YYYYYYYYYYYYYYYYYYYYYYYY
002056	FF	YYYYYYYYYYYYYYYYYYYYYYYY
00206C	FF	YYYYYYYYYYYYYYYYYYYYYYYY
002082	FF	YYYYYYYYYYYYYYYYYYYYYYYY
002098	FF	YYYYYYYYYYYYYYYYYYYYYYYY
0020AE	FF	YYYYYYYYYYYYYYYYYYYYYYYY
0020C4	FF	YYYYYYYYYYYYYYYYYYYYYYYY
002132	FF	YYYYYYYYYYYYYYYYYYYYYYYY
002148	FF	YYYYYYYYYYYYYYYYYYYYYYYY
00215E	FF	YYYYYYYYYYYYYYYYYYYYYYYY
002174	FF	YYYYYYYYYYYYYYYYYYYYYYYY
00218A	FF	YYYYYYYYYYYYYYYYYYYYYYYY
0021A0	FF	YYYYYYYYYYYYYYYYYYYYYYYY
0021B6	FF	YYYYYYYYYYYYYYYYYYYYYYYY
0021CC	FF	YYYYYYYYYYYYYYYYYYYYYYYY
0021E2	FF	YYYYYYYYYYYYYYYYYYYYYYYY
0021F8	FF	YYYY... 00 00 0B

Next available position,  
21FC

3 bytes occupied,  
21FF, 21FE, 21FD

21F8  
21FA  
21FB  
21F9  
21FD

Value here is the hexadecimal value  
00 00 0B  
which is the return address from subroutine

PC, Program Counter =  
0x 00 00 0C

Processor

Name	Value
Program Counter	0x00000C
Stack Pointer	0x21FC
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	14
Frequency	4.0000 MHz
Stop Watch	3.50 us
SREG	00H S V N Z C
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0xAA
R17	0x00

C:\Users\...\_Kurser\...\Program\_examples\Lecture\_exa...

```
.include "m2560def.inc"

ldi r16, high(RAMEND) ; high part of highest RAM-address to r16
out SPH, r16           ; write o Stack Pointer, SPH
ldi r16, low(RAMEND)  ; high part of highest RAM-address to r16
out SPL, r16           ; write o Stack Pointer, SPL

ldi r16, 0xFF           ; Set Data Direction Registers
out DDRB, r16           ; port B as outputs
ldi r16, 0x00           ; port A as inputs
out DDRA, r16

ldi r16, 170            ; set register r16

main_loop:
out PORTB, r16          ; infinite loop

rcall subr_01             ; call subroutine

rjmp main_loop

subr_01:
push r16                ; save r16 on stack
push r17                ; save r17 on stack

com r16                 ; invert content in r16
com r17

pop r17                 ; restore r17 from stack
```

I/O View

ANALOG\_COMPARATOR

Name	Value
D ANALOG_COMPARE...	
BOOT_LOAD	
CPU	
E EEPROM	
EXTERNAL_INTERRUPT...	
JTAG	
PORTA	
PORTB	
Port B Data Register	0xAA
Port B Data Directio...	0xFF
Port B Input Pins	0xAA
PORTC	
PORTD	
PORTE	
PORTF	

Name	Address	Value	Bits
DDR B	0x04 (0x24)	0xFF	0000000000000000
PIN B	0x03 (0x23)	0xAA	0000000000000000
PORT B	0x05 (0x25)	0xAA	0000000000000000

r16 = 0xAA  
r17 = 0x00

push r16 will store the  
content of r16 on the stack

SP, Stack pointer = 0x21FB  
SP always point to next free position

Processor

Name	Value
Program Counter	0x00000D
Stack Pointer	0x21FB
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	16
Frequency	4.0000 MHz
Stop Watch	4.00 us
SREG	00000000
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0xA
R17	0x00

Memory

Address	Value
002174	FF FF FF
00218A	FF FF F
0021A0	FF FF FF
0021B6	FF FF FF
0021CC	FF FF FF
0021E2	FF FF FF
0021F8	FF FF FF FF AA 00 00 0B

Code View

```
;-----  
.include "m2560def.inc"  
  
ldi r16, high(RAMEND) ; high part of highest RAMN-address to r16  
out SPH, r16 ; write o Stack Pointer, SPH  
ldi r16, low(RAMEND) ; high part of highest RAMN-address to r16  
out SPL, r16 ; write o Stack Pointer, SPL  
  
ldi r16, 0xFF ; Set Data Direction Registers  
out DDRB, r16 ; port B as outputs  
ldi r16, 0x00 ;  
out DDRA, r16 ; port A as inputs  
  
ldi r16, 170 ; set register r16  
  
main_loop:  
out PORTB, r16 ;  
  
rcall subr_01 ; call subroutine  
  
rjmp main_loop ; infinite loop  
  
subr_01:  
push r16 ; save r16 on stack  
push r17 ; save r17 on stack  
  
com r16 ; invert content in r16
```

I/O View

Name	Value
ANALOG_COMPARA...	
BOOT_LOAD	
CPU	
EEPROM	
EXTERNAL_INTERRUPT	
JTAG	
PORTA	
PORTB	
Port B Data Register	0xAA
Port B Data Directio...	0xFF
Port B Input Pipe	0-AA
PINB	0x03 (0x23)
PORTB	0x05 (0x25)

Next free position on stack is 0x21FB

0xAA stored in position 0x21FC on stack

Processor

Name	Value
Program Counter	0x00000F
Stack Pointer	0x21FA
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	19
Frequency	4.0000 MHz
Stop Watch	4.75 us
SREG	0000000000000000
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0x55
R17	0x00

C:\Document\Kurser\Kursdator\Program\_examples\Lecture\_exa...

```

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
.include "m2560def.inc"

ldi r16, high(RAMEND) ; high part of highest RAMN-address to r16
out SPH, r16 ; write o Stack Pointer, SPH
ldi r16, low(RAMEND) ; high part of highest RAMN-address to r16
out SPL, r16 ; write o Stack Pointer, SPL

ldi r16, 0xFF ; Set Data Direction Registers
out DDRB, r16 ; port B as outputs
ldi r16, 0x00 ; port A as inputs
out DDRA, r16

ldi r16, 170 ; set register r16

main_loop:
out PORTB, r16

rcall subr_01 ; call subroutine

rjmp main_loop ; infinite loop

subr_01:
push r16 ; save r16 on stack
push r17 ; save r17 on stack

com r16 ; invert content in r16
com r17 ; restore r17 from stack

```

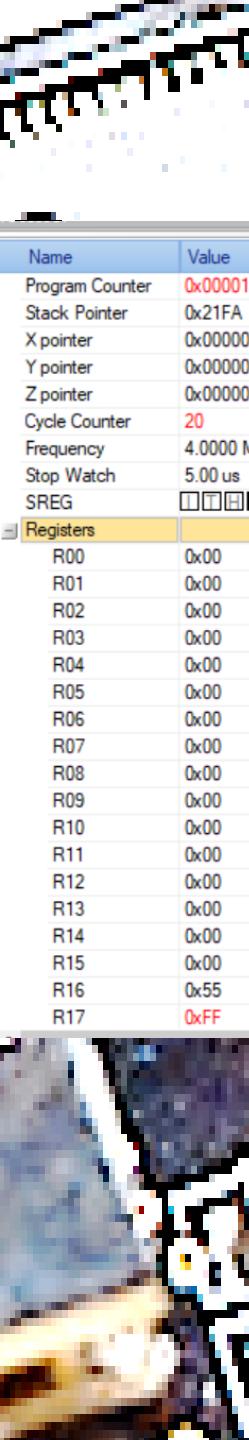
I/O View

ANALOG\_COMPARATOR

Name	Value
ANALOG_COMPARA...	
BOOT_LOAD	
CPU	
EEPROM	
EXTERNAL_INTERRUPT...	
JTAG	
PORTE	
PORTB	
Port B Data Register	0xAA
Port B Data Directio...	0xFF
Port B Input Pins	0xAA
PORTE	

Name	Address	Value	Bits
DDR	0x04 (0x24)	0xFF	11111111
PINB	0x03 (0x23)	0xAA	10101010
PORTB	0x05 (0x25)	0xAA	10101010

r16 is inverted bit by bit,  
0b1010 1010 is changed to 0b0101 0101 = 0x55



Name	Value
Program Counter	0x000010
Stack Pointer	0x21FA
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	20
Frequency	4.0000 MHz
Stop Watch	5.00 us
SREG	00000S0V0N0C0
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0x55
R17	0xFF

```
;-----  
.include "m2560def.inc"  
  
ldi r16, high(RAMEND) ; high part of highest RAMN-address to r16  
out SPH, r16 ; write o Stack Pointer, SPH  
ldi r16, low(RAMEND) ; high part of highest RAMN-address to r16  
out SPL, r16 ; write o Stack Pointer, SPL  
  
ldi r16, 0xFF ; Set Data Direction Registers  
out DDRB, r16 ; port B as outputs  
ldi r16, 0x00 ;  
out DDRA, r16 ; port A as inputs  
  
ldi r16, 170 ; set register r16  
  
main_loop:  
out PORTB, r16  
  
rcall subr_01 ; call subroutine  
  
rjmp main_loop ; infinite loop  
  
subr_01:  
push r16 ; save r16 on stack  
push r17 ; save r17 on stack  
  
com r16 ; invert content in r16  
com r17 ;  
  
pop r17 ; restore r17 from stack  
pop r16 ; restore r16 from stack  
  
ret ; return to main program
```

ANALOG\_COMPATOR

Name	Value
ANALOG_COMPARE...	
BOOT_LOAD	
CPU	
EEPROM	
EXTERNAL_INTERRUPT...	
JTAG	
PORTA	
PORTB	
PORTC	
PORTD	
PORTE	
PORTF	

Name	Address	Value	Bits
DDRB	0x04 (0x24)	0xFF	████████████████
PINB	0x03 (0x23)	0xAA	███□□□□□□
PORTB	0x05 (0x25)	0xAA	███□□□□□□

SP, Stack pointer = 0x21FC  
 which is now available for storage,  
 but not cleared

The screenshot shows the processor tab with the Stack Pointer (SP) highlighted at 0x21FC. The Registers tab lists R00-R17, with R16 set to 0xAA. The assembly code in the center pane shows the execution flow from main\_loop to subr\_01, then back to main\_loop, and finally to ret. The I/O View tab on the right shows PortB settings.

```

ldi r16, low(RAMEND)
out SPL, r16 ; write o Stack Pointer, SPL

ldi r16, 0xFF
out DDRB, r16 ; Set Data Direction Registers
ldi r16, 0x00
out DDRA, r16 ; port B as outputs
; port A as inputs

ldi r16, 170 ; set register r16

main_loop:
out PORTB, r16

rcall subr_01 ; call subroutine

rjmp main_loop ; infinite loop

subr_01:
push r16 ; save r16 on stack
push r17 ; save r17 on stack

com r16 ; invert content in r16
com r17

pop r17 ; restore r17 from stack
pop r16 ; restore r16 from stack

ret ; return to main program
  
```

Memory

Next free position on stack is  
 0x21FC

The Memory window displays memory starting at address 0x2132. The stack grows downwards, with the next free position at 0x21FC. The value at 0x21FC is AA, which is circled. The memory dump shows FFs above 0x2132 and 00s below 0x21FC.

Data	Address	Value
FF FF FF FF	002132	FF
FF FF FF FF	002148	FF
FF FF FF FF	00215E	FF
FF FF FF FF	002174	FF
FF FF FF FF	00218A	FF
FF FF FF FF	0021A0	FF
FF FF FF FF	0021B6	FF
FF FF FF FF	0021CC	FF
FF FF FF FF	0021E2	FF
FF FF FF 00	0021F8	AA
00 00 0B		

## PC, Program Counter

0x 00 00 12

The screenshot shows a debugger interface with several windows:

- Registers**: Shows various registers (R00-R17) with their current values.
- Memory**: Shows memory starting at address 0x2132, mostly filled with FF (hexadecimal).
- I/O View**: Shows port settings for PORTB, PORTC, PORTD, and PORTE.
- Code Window**: Displays assembly code:

```
ldi r16, low(RAMEND)
out SPL, r16
ldi r16, high(RAMEND)
out SPH, r16
ldi r16, 0xFF
out DDRB, r16
ldi r16, 0x00
out DDRA, r16
ldi r16, 170
main_loop:
    out PORTB, r16
    rcall subr_01
    rjmp main_loop
subr_01:
    push r16
    push r17
    com r16
    com r17
    pop r17
    pop r16
    ret
    ; restore r17 from stack
    ; restore r16 from stack
    ; return to main program
```

A blue oval highlights the **Program Counter** and **Stack Pointer** in the Registers window. A callout box points to the PC value (0x000012). Another callout box points to the **ret** instruction in the assembly code, with the text: "ret, return from subroutine is next instruction to be executed".

The screenshot shows a debugger interface with several windows:

- Registers**: Shows various registers (R00-R17) with their current values.
- Memory**: Shows memory starting at address 0x2132, mostly filled with FF (hexadecimal). A blue oval highlights the value 00 00 0B at address 0x21F8, which is the return address.
- I/O View**: Shows port settings for PORTB, PORTC, PORTD, and PORTE.
- Code Window**: Displays assembly code:

```
ldi r16, low(RAMEND)
out SPL, r16
ldi r16, high(RAMEND)
out SPH, r16
ldi r16, 0xFF
out DDRB, r16
ldi r16, 0x00
out DDRA, r16
ldi r16, 170
main_loop:
    out PORTB, r16
    rcall subr_01
    rjmp main_loop
subr_01:
    push r16
    push r17
    com r16
    com r17
    pop r17
    pop r16
    ret
    ; restore r17 from stack
    ; restore r16 from stack
    ; return to main program
```

A callout box points to the value 00 00 0B in memory, with the text: "Return address will be fetched from stack. Value is 0x 00 00 0B".

PC, Program Counter, 0x 00 00 0B  
SP, Stack Pointer, 0x21FF

Name	Value
Program Counter	0x00000B
Stack Pointer	0x21FF
X pointer	0x000000
Y pointer	0x000000
Z pointer	0x000000
Cycle Counter	29
Frequency	4.0000 MHz
Stop Watch	7.25 us
SREG	
Registers	
R00	0x00
R01	0x00
R02	0x00
R03	0x00
R04	0x00
R05	0x00
R06	0x00
R07	0x00
R08	0x00
R09	0x00
R10	0x00
R11	0x00
R12	0x00
R13	0x00
R14	0x00
R15	0x00
R16	0xAA
R17	0x00

```
include "m2560def.inc"
ldi r16, high(RAMEND)           ; high part of highest RAMN-address to r16
out SPH, r16                     ; write o Stack Pointer, SPH
ldi r16, low(RAMEND)            ; high part of highest RAMN-address to r16
out SPL, r16                     ; write o Stack Pointer, SPL

ldi r16, 0xFF                     ; Set Data Direction Registers
out DDRB, r16                    ; port B as outputs
ldi r16, 0x00                     ; port A as inputs
out DDRA, r16

ldi r16, 170                      ; set reg

main_loop:
out PORTB, r16

rcall subr_01                     ; call subr_01
rjmp main_loop                   ; infinite loop

subr_01:
push r16                          ; save r16 on stack
push r17                          ; save r17 on stack
com r16                           ; invert content in r16
com r17

pop r17                           ; restore r17 from stack
pop r16                           ; restore r16 from stack

ret                             ; return to main program
```

Next instruction to be executed.

r16 and r17 are restored,  
r16 = 0xAA and r17 = 0x00

The stack is not cleared, but SP points to highest position, 0x21FF which is free.

Memory2						
Data	8/16	abc.	Address:	0x2180	Cols:	Auto
002180 FF						
00219C FF						
0021B8 FF						
0021D4 FF						
0021F0 FF FF FF FF FF FF FF FF 00 AA 00 00 0B						

# 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). In the assembler, labels are used instead of relative operands. 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)  $PC \leftarrow PC + k + 1$  Devices with 16 bits PC, 128K bytes Program memory maximum.
- (ii)  $PC \leftarrow PC + k + 1$  Devices with 22 bits PC, 8M bytes Program memory maximum.

### Syntax:

(i) `RCALL k`

### Operands:

$-2K \leq k < 2K$

### Program Counter:

$PC \leftarrow PC + k + 1$

### Stack:

$STACK \leftarrow PC + 1$   
 $SP \leftarrow SP - 2$  (2 bytes, 16 bits)

(ii) `RCALL k`

$-2K \leq k < 2K$

$PC \leftarrow PC + k + 1$

$STACK \leftarrow PC + 1$   
 $SP \leftarrow SP - 3$  (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
...
```

# RET – Return from Subroutine

## Description:

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

### Operation:

- (i)  $PC(15:0) \leftarrow \text{STACK}$  Devices with 16 bits PC, 128K bytes Program memory maximum.  
(ii)  $PC(21:0) \leftarrow \text{STACK}$  Devices with 22 bits PC, 8M bytes Program memory maximum.

(i) **Syntax:**  
RET

**Operands:**  
None

**Program Counter:**  
See Operation

**Stack:**  
 $SP \leftarrow SP + 2$ , (2bytes, 16 bits)

(ii) RET

None

See Operation

$SP \leftarrow SP + 3$ , (3bytes, 22 bits)

### 16-bit Opcode:

1001	0101	0000	1000
------	------	------	------

## Status Register (SREG) and Boolean Formula:

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

## Example:

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

# PUSH – Push Register on Stack

## Description:

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

This instruction is not available in all devices. Refer to the device specific instruction set summary.

### Operation:

$\text{STACK} \leftarrow \text{Rr}$

### Syntax:

(i)  $\text{PUSH Rr}$

### Operands:

$0 \leq r \leq 31$

### Program Counter:

$\text{PC} \leftarrow \text{PC} + 1$

### Stack:

$\text{SP} \leftarrow \text{SP} - 1$

### 16-bit Opcode:

1001	001d	dddd	1111
------	------	------	------

## Status Register (SREG) and Boolean Formula:

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

## 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
```

# POP – Pop Register from Stack

## Description:

This 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.

### Operation:

(i)  $Rd \leftarrow \text{STACK}$

### Syntax:

(i)  $\text{POP } Rd$

### Operands:

$0 \leq d \leq 31$

### Program Counter:

$PC \leftarrow PC + 1$

### Stack:

$SP \leftarrow SP + 1$

### 16-bit Opcode:

1001	000d	dddd	1111
------	------	------	------

## Status Register (SREG) and Boolean Formula:

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

## 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
```

Words: 1 (2 bytes)