AVR Assembly Language Programming
Assembler Source

- The Assembler works on source files containing instruction mnemonics, labels and directives. The instruction mnemonics and the directives often take operands.
- Every input line can be preceded by a label, which is an alphanumeric string terminated by a colon. Labels are used as targets for jump and branch instructions and as variable names in Program memory and RAM.
- An input line may take one of the four following forms:
  1. [label:] directive [operands] [Comment]
  2. [label:] instruction [operands] [Comment]
  3. Comment
  4. Empty line
A comment has the following form:

; [Text]

Items placed in braces are optional. The text between the comment-delimiter (;) and the end of line (EOL) is ignored by the Assembler.

**Examples:**

```asm
label: .EQU var1=100 ; Set var1 to 100 (Directive)
 .EQU var2=200 ; Set var2 to 200

test:  rjmp test ; Infinite loop (Instruction)
 ; Pure comment line
 ; Another comment line
```
General Purpose Registers

- Lower Registers

The lower 16 registers, R0 – R15, work just the rest of the registers with the exception of loading immediate data. These registers have access to the full range of the Data Memory, ALU, and additional peripherals. Here is an example of using the loading immediate data into the lower registers:

```
LDI   R16, 30    ; Load the number 30 into R16
MOV   R0, R16   ; Copy R16 into R0, R0 <- R16
INC   R0        ; Increment R0, R0 <- R0+1
ADD   R0, R16   ; R0 <- R0 + R16, value in R0 should now be 61
```
General Purpose Registers

- **Upper Registers**

The upper 16 registers, R16 – R31, have additional capabilities. They have access to immediate data using the **LDI** instruction. These registers will be the ones that get the most use throughout your program. To move data into or out of these registers, the various different Load and Store instructions are needed. All arithmetic instructions work on these registers. Here is an example of using the upper registers:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Source</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDI</td>
<td>R16</td>
<td>$A4</td>
<td>Load the immediate hex value into R16</td>
</tr>
<tr>
<td>LD</td>
<td>R17</td>
<td>X</td>
<td>Load value from memory address in X-Pointer</td>
</tr>
<tr>
<td>ADC</td>
<td>R16, R17</td>
<td></td>
<td>Add with carry, R16 ← R16 + R17 + Carry Bit</td>
</tr>
<tr>
<td>ST</td>
<td>Y, R16</td>
<td></td>
<td>Store value in R16 to address in Y-Pointer</td>
</tr>
</tbody>
</table>
General Purpose Registers

- X-, Y-, Z-Registers

The last six of the General Purpose Registers have additional functionality. They serve as the pointers for indirect addressing. The ATmega128 has a 16-bit addressing scheme that requires two registers for the address alone. The AVR RISC structure supports this scheme with the X, Y, and Z-Registers. These registers are the last six General Purpose Registers (R26-R31). The following table details the register assignments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Byte</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Register</td>
<td>Low</td>
<td>R26</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>R27</td>
</tr>
<tr>
<td>Y-Register</td>
<td>Low</td>
<td>R28</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>R29</td>
</tr>
<tr>
<td>Z-Register</td>
<td>Low</td>
<td>R30</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>R31</td>
</tr>
</tbody>
</table>
General Purpose Registers

- **X-, Y-, Z-Registers**

The following code is an example of how to use these special registers. The code will read a value from SRAM, manipulate it, and then store it back at the next address in SRAM.

```
LDI R26, $5A ; Load 0x5A into the low Byte of X
LDI R27, $02 ; Load 0x02 into the high Byte of X
LD R16, X+ ; Load value from SRAM, increment X
INC R16 ; Manipulate value
ST X, R16 ; Store value to SRAM
```
Special Function Registers

- Status Register (SREG)

The Status Register or SREG contains the important information about the ALU such as the Carry Bit, Overflow Bit, and Zero Bit. These bits are set and cleared during ALU instructions. This register becomes extremely useful during branching operations. The following table details the bit assignments within the SREG.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>I</td>
<td>Global Interrupt Enable</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>Bit Copy Storage</td>
</tr>
<tr>
<td>5</td>
<td>H</td>
<td>Half Carry Flag</td>
</tr>
<tr>
<td>4</td>
<td>S</td>
<td>Sign Bit</td>
</tr>
<tr>
<td>3</td>
<td>V</td>
<td>Twos Compliment Overflow Flag</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>Negative Flag</td>
</tr>
<tr>
<td>1</td>
<td>Z</td>
<td>Zero Flag</td>
</tr>
<tr>
<td>0</td>
<td>C</td>
<td>Carry Flag</td>
</tr>
</tbody>
</table>
Special Function Registers

- Stack Pointer

The Stack is mainly used for storing temporary data, for storing local variables and for storing return addresses after interrupts and subroutine calls. The Stack Pointer Register always points to the top of the Stack. Note that the Stack is implemented as growing from higher memory locations to lower memory locations. This implies that a Stack **PUSH** command decreases the Stack Pointer.
Special Function Registers

- Stack Pointer

The AVR Stack Pointer is implemented as two 8-bit Special Function Registers, Stack Pointer High Register (SPH) and Stack Pointer Low Register (SPL). The following diagram is a representation of the Stack Pointer.
Special Function Registers

- Stack Pointer

The following example demonstrates how to initialize the Stack Pointer. Remember to include the definition file for the ATmega128 at the beginning of the program to utilize Register naming schemes.

```
.include “m128def.inc” ; Include definition file in program

LDI R16, LOW(RAMEND) ; Low Byte of End SRAM Address
OUT SPL, R16 ; Write byte to SPL
LDI R16, HIGH(RAMEND) ; High Byte of End SRAM Address
OUT SPH, R16 ; Write byte to SPH
```
Stack

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

SP

0000

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LDI R16, HIGH(RAMEND)</td>
</tr>
<tr>
<td>0001</td>
<td>OUT SPH, R16</td>
</tr>
<tr>
<td>0002</td>
<td>LDI R16, LOW(RAMEND)</td>
</tr>
<tr>
<td>0003</td>
<td>OUT SPL, R16</td>
</tr>
<tr>
<td>0004</td>
<td>LDI R20, 0x10</td>
</tr>
<tr>
<td>0005</td>
<td>LDI R21, 0x20</td>
</tr>
<tr>
<td>0006</td>
<td>LDI R22, 0x30</td>
</tr>
<tr>
<td>0007</td>
<td>PUSH $10</td>
</tr>
<tr>
<td>0008</td>
<td>PUSH $20</td>
</tr>
<tr>
<td>0009</td>
<td>PUSH $30</td>
</tr>
<tr>
<td>000A</td>
<td>POP R21</td>
</tr>
<tr>
<td>000B</td>
<td>POP R0</td>
</tr>
<tr>
<td>000C</td>
<td>POP R20</td>
</tr>
<tr>
<td>000D</td>
<td>L1: RJMP L1</td>
</tr>
</tbody>
</table>
## Special Function Registers

### I/O Ports

<table>
<thead>
<tr>
<th>DDxn</th>
<th>PORTxn</th>
<th>PUD (in SFIOR)</th>
<th>I/O</th>
<th>Pull-up</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Input</td>
<td>No</td>
<td>Tri-state (Hi-Z)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Input</td>
<td>Yes</td>
<td>Px_n will source current if ext. pulled low.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Input</td>
<td>No</td>
<td>Tri-state (Hi-Z)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Output</td>
<td>No</td>
<td>Output Low (Sink)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>Output</td>
<td>No</td>
<td>Output High (Source)</td>
</tr>
</tbody>
</table>

LDI R16, $FF ; Select Direction as Output on all pins
OUT DDRB, R16 ; Set value in DDRB
LDI R16, $FF ; Set Initial value to high on all pins
OUT PORTB, R16 ; Set PORTB value, Port B pins should be high

LDI R16, $00 ; Select Direction as Input on all pins
OUT DDRD, R16 ; Set value in DDRD
LDI R16, $00 ; Use normal Tri-state with no Pull-up resister
OUT PORTD, R16 ; Port D is now ready as input
Pre-compiler Directives

- Pre-compiler directives are special instructions that are executed before the code is compiled and directs the compiler. These instructions are denoted by the preceding dot, i.e. .EQU.

- The directives are not translated directly into opcodes. Instead, they are used to adjust the location of the program in memory, define macros, initialize memory, and so on.
# Pre-compiler Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.BYTE</td>
<td>Reserve byte to a variable</td>
</tr>
<tr>
<td>.CSEG</td>
<td>Code Segment</td>
</tr>
<tr>
<td>.DB</td>
<td>Define constant byte(s)</td>
</tr>
<tr>
<td>.DEF</td>
<td>Define a symbolic name on a register</td>
</tr>
<tr>
<td>.DEVICE</td>
<td>Define which device to assemble for</td>
</tr>
<tr>
<td>.DSEG</td>
<td>Data Segment</td>
</tr>
<tr>
<td>.DW</td>
<td>Define constant words</td>
</tr>
<tr>
<td>.ENDMACRO</td>
<td>End macro</td>
</tr>
<tr>
<td>.EQU</td>
<td>Set a symbol equal to an expression</td>
</tr>
<tr>
<td>.ESEG</td>
<td>EEPROM segment</td>
</tr>
<tr>
<td>.EXIT</td>
<td>Exit from a file</td>
</tr>
<tr>
<td>.INCLUDE</td>
<td>Read source from another file</td>
</tr>
<tr>
<td>.LIST</td>
<td>Turn listfile generation on</td>
</tr>
<tr>
<td>.LISTMAC</td>
<td>Turn macro expression on</td>
</tr>
<tr>
<td>.MACRO</td>
<td>Begin Macro</td>
</tr>
<tr>
<td>.NOLIST</td>
<td>Turn listfile generation off</td>
</tr>
<tr>
<td>.ORG</td>
<td>Set program origin</td>
</tr>
<tr>
<td>.SET</td>
<td>Set a symbol to an expression</td>
</tr>
</tbody>
</table>
Pre-compiler Directives: BYTE

- Reserve bytes to a variable
  - The BYTE directive reserves memory resources in the SRAM.
  - In order to be able to refer to the reserved location, the BYTE directive should be preceded by a label.
  - The directive takes one parameter, which is the number of bytes to reserve.
  - The directive can only be used within a Data Segment (see directives CSEG, DSEG and ESEG).
- Note that a parameter must be given. The allocated bytes are not initialized.
Pre-compiler Directives: **BYTE**

Syntax:

```
LABEL: .BYTE expression
```

Example:

```
.DSEG

var1: .BYTE 1 ; reserve 1 byte to var1
table: .BYTE tab_size ; reserve tab_size bytes

.CSEG

ldi r30, low(var1) ; Load Z register low
ldi r31, high(var1) ; Load Z register high
ld r1, Z ; Load VAR1 into register 1
```
Pre-compiler Directives: CSEG

- **Code Segment**
- The CSEG directive defines the start of a Code Segment. An assembler file can contain multiple Code Segments, which are concatenated into one Code Segment when assembled. The directive does not take any parameters.

**Syntax:**

```
.CSEG
```

**Example:**

```
.DSEG
  vartab: .BYTE 4 ; Reserve 4 bytes in SRAM
.DSEG
  .CSEG
  const: .DW 2 ; Write 0x0002 in program memory
  mov r1, r0 ; Do something
```
Pre-compiler Directives: DB

- **Define constant byte(s)**
- The DB directive reserves memory resources in the program memory or the EEPROM memory. In order to be able to refer to the reserved locations, a label should precede the DB directive.
- The DB directive takes a list of expressions, and must contain at least one expression. The list of expressions is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between –128 and 255 since each expression is represented by 8-bits. A negative number will be represented by the 8-bits two’s complement of the number.
Pre-compiler Directives: DB

Syntax:

```
  LABEL: .DB expressionlist
```

Example:

```
.CSEG
consts:
  .DB 0, 255, 0b01010101, -128, $AA
text:
  .DB "Hello World"
```
Pre-compiler Directives: DEF

- Set a symbolic name on a register
- The DEF directive allows the registers to be referred to through symbols. A defined symbol can be used to the rest of the program to refer to the registers it is assigned to. A register can have several symbolic names attached to it. A symbol can be redefined later in the program.

Syntax:

```
.DEF Symbol=Register
```

Example:

```
.DEF temp=R16
.DEF ior=R0
.CSEG

ldi    temp,0xf0 ; Load 0xf0 into temp register
in     ior,0x3f ; Read SREG into ior register
eor    temp,ior ; Exclusive or temp and ior
```
Pre-compiler Directives: DSEG

- **Data Segment**
  - The DSEG directive defines the start of a Data Segment. An Assembler file can consist of several Data Segments, which are concatenated into one Data Segment when assembled.
  - A Data Segment will normally only consist of BYTE directives (and labels). The Data Segments have their own location counter which is a byte counter. The ORG directive (see description later in this document) can be used to place the variables at specific locations in the SRAM. The directive does not take any parameters.
Pre-compiler Directives: **DSEG**

**Syntax:**

```
.DSEG
```

**Example:**

```
.DSEG
    var1: .BYTE 1 ; reserve 1 byte to var1
    table:.BYTE tab_size ; reserve tab_size bytes.
    .CSEG
        ldi r30, low(var1) ; Load Z register low
        ldi r31, high(var1) ; Load Z register high
        ld r1, Z ; Load var1 into register 1
```
Pre-compiler Directives: **EQU**

- **Set a symbol equal to an expression**
- **The EQU directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the EQU directive is a constant and cannot be changed or redefined.**

Syntax:

```plaintext
.EQU label = expression
```

Example:

```plaintext
.EQU io_offset = 0x23
.EQU porta = io_offset + 2
.CSEG
    clr r2 ; Start code segment
    out porta,r2 ; Clear register 2
    out porta,r2 ; Write to Port A
```
Pre-compiler Directives: **INCLUDE**

- The **INCLUDE** directive tells the Assembler to start reading from a specified file. The Assembler then assembles the specified file until end of file (EOF) or an EXIT directive is encountered. An included file may itself contain INCLUDE directives.

  **Syntax:**
  ```
  .INCLUDE "filename"
  ```

  **Example:**
  ```
  ; iodefs.asm:
  .EQU sreg=0x3f       ; Status register
  .EQU sphigh=0x3e     ; Stack pointer high
  .EQU splow=0x3d      ; Stack pointer low
  
  .INCLUDE "iodefs.asm" ; Include I/O definitions
  in r0,sreg           ; Read status register
  ```
Set program origin

The ORG directive sets the location counter to an absolute value. The value to set is given as a parameter. If an ORG directive is given within a Data Segment, then it is the SRAM location counter which is set, if the directive is given within a Code Segment, then it is the Program memory counter which is set and if the directive is given within an EEPROM Segment, then it is the EEPROM location counter which is set.
Pre-compiler Directives: `ORG`

Syntax:

```
.ORG expression
```

Example:

```
.DSEG
.ORG 0x67
    variable: .BYTE 1
    ; Start data segment
    ; Set SRAM address to hex 67
    ; Reserve a byte at SRAM
    ; adr.67H

.ESEG
.ORG 0x20
    ; Start EEPROM Segment
    ; Set EEPROM location
    ; counter

.eevar:  .DW 0xfeff
    ; Initialize one word

.CSEG
.ORG 0x10
    ; Set Program Counter to hex 10
    ; Do something
```

Pre-compiler Directives: SET

- Set a symbol equal to an expression
- The SET directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the SET directive can be changed later in the program.

Syntax:

```
.SET label = expression
```

Example:

```
.SET io_offset = 0x23
.SET porta = io_offset + 2
.CSEG
   clr r2          ; Start code segment
   out porta,r2    ; Clear register 2
   ; Write to Port A
```
The Assembler incorporates expressions. Expressions can consist of operands, operators and functions. All expressions are internally 32 bits.
Expressions: Operands

Operands

The following operands can be used:

- User defined labels which are given the value of the location counter at the place they appear.
- User defined variables defined by the SET directive
- User defined constants defined by the EQU directive
- Integer constants: constants can be given in several formats, including
  a) Decimal (default): 10, 255
  b) Hexadecimal (two notations): 0x0a, $0a, 0xff, $ff
  c) Binary: 0b00001010, 0b11111111
- PC - the current value of the Program memory location counter
### Expressions: Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Description</th>
<th>Precedence</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Not</td>
<td>!</td>
<td>Unary operator which returns 1 if the expression was zero, and returns 0 if the expression was nonzero</td>
<td>14</td>
<td>ldi r16, !0xf0 ; Load r16 with 0x00</td>
</tr>
<tr>
<td>Bitwise Not</td>
<td>~</td>
<td>Unary operator which returns the input expression with all bits inverted</td>
<td>14</td>
<td>ldi r16, ~0xf0 ; Load r16 with 0xf0</td>
</tr>
<tr>
<td>Unary Minus</td>
<td>-</td>
<td>Unary operator which returns the arithmetic negation of an expression</td>
<td>14</td>
<td>ldi r16, -2 ; Load -2(0xfe) in r16</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>Binary operator which returns the product of two expressions</td>
<td>13</td>
<td>ldi r30, label<em>2 ; Load r30 with label</em>2</td>
</tr>
</tbody>
</table>
## Expressions: Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Description</th>
<th>Precedence</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division</td>
<td>/</td>
<td>Binary operator which returns the integer quotient of the left expression divided by the right expression</td>
<td>13</td>
<td>ldi r30, label/2 ; Load r30 with label/2</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>Binary operator which returns the sum of two expressions</td>
<td>12</td>
<td>ldi r30, c1+c2 ; Load r30 with c1+c2</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>Binary operator which returns the left expression minus the right expression</td>
<td>12</td>
<td>ldi r17, c1-c2 ; Load r17 with c1-c2</td>
</tr>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>Binary operator which returns the left expression shifted left a number of times given by the right expression</td>
<td>11</td>
<td>ldi r17, 1&lt;&lt;bitmask ; Load r17 with 1 shifted ; left bitmask times</td>
</tr>
</tbody>
</table>
# Expressions: Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Description</th>
<th>Precedence</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>Binary operator which returns the left expression shifted right a number of</td>
<td>11</td>
<td>ldi r17, c1 &gt;&gt; c2 ; Load r17 with c1 shifted right c2 times</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
<td>Binary operator which returns 1 if the signed expression to the left is Less</td>
<td>10</td>
<td>ori r18, bitmask*(c1 &lt; c2) + 1 ; Or r18 with an expression</td>
</tr>
<tr>
<td>Less or Equal</td>
<td>&lt;=</td>
<td>Binary operator which returns 1 if the signed expression to the left is Less</td>
<td>10</td>
<td>ori r18, bitmask*(c1 &lt;= c2) + 1 ; Or r18 with an expression</td>
</tr>
</tbody>
</table>
## Expressions: Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Description</th>
<th>Precedence</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater than</td>
<td></td>
<td>Binary operator which returns 1 if the signed expression to the left is</td>
<td>10</td>
<td>ori r18, bitmask*(c1&gt;c2)+1 ; Or r18 with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Greater than the signed expression to the right, 0 otherwise</td>
<td></td>
<td>; an expression</td>
</tr>
<tr>
<td>Greater or Equal</td>
<td></td>
<td>Binary operator which returns 1 if the signed expression to the left is</td>
<td>10</td>
<td>ori r18, bitmask*(c1&gt;=c2)+1 ; Or r18 with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Greater than or Equal to the signed expression to the right, 0 otherwise</td>
<td></td>
<td>; an expression</td>
</tr>
<tr>
<td>Equal</td>
<td></td>
<td>Binary operator which returns 1 if the signed expression to the left is</td>
<td>9</td>
<td>andi r19, bitmask*(c1==c2)+1 ; And r19 with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Equal to the signed expression to the right, 0 otherwise</td>
<td></td>
<td>; an expression</td>
</tr>
</tbody>
</table>
## Expressions: Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Description</th>
<th>Precedence</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Equal</td>
<td><code>!=</code></td>
<td>Binary operator which returns 1 if the signed expression to the left is Not Equal to the signed expression to the right, 0 otherwise</td>
<td>9</td>
<td><code>.SET flag=(c1!=c2) ;Set flag to 1 or 0</code></td>
</tr>
<tr>
<td>Bitwise And</td>
<td><code>&amp;</code></td>
<td>Binary operator which returns the bitwise And between two expressions</td>
<td>8</td>
<td><code>ldi r18,High(c1&amp;c2) ;Load r18 with an expression</code></td>
</tr>
<tr>
<td>Bitwise Xor</td>
<td><code>^</code></td>
<td>Binary operator which returns the bitwise Exclusive Or between two expressions</td>
<td>7</td>
<td><code>ldi r18,Low(c1^c2) ;Load r18 with an expression</code></td>
</tr>
<tr>
<td>Operators</td>
<td>Symbol</td>
<td>Description</td>
<td>Precedence</td>
<td>Example</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Bitwise Or</td>
<td></td>
<td></td>
<td>Binary operator which returns the bitwise Or between two expressions</td>
<td>6</td>
</tr>
<tr>
<td>Logical And</td>
<td>&amp; &amp;</td>
<td>Binary operator which returns 1 if the expressions are both nonzero, 0 otherwise</td>
<td>5</td>
<td>ldi r18,Low(c1 &amp; &amp; c2) ;Load r18 with an expression</td>
</tr>
<tr>
<td>Logical Or</td>
<td></td>
<td></td>
<td></td>
<td>Binary operator which returns 1 if one or both of the expressions are nonzero, 0 otherwise</td>
</tr>
</tbody>
</table>
Functions

The following functions are defined:

- \( \text{LOW}(\text{expression}) \) returns the low byte of an expression
- \( \text{HIGH}(\text{expression}) \) returns the second byte of an expression
- \( \text{BYTE2}(\text{expression}) \) is the same function as \( \text{HIGH} \)
- \( \text{BYTE3}(\text{expression}) \) returns the third byte of an expression
- \( \text{BYTE4}(\text{expression}) \) returns the fourth byte of an expression
- \( \text{LWRD}(\text{expression}) \) returns bits 0-15 of an expression
- \( \text{HWRD}(\text{expression}) \) returns bits 16-31 of an expression
- \( \text{PAGE}(\text{expression}) \) returns bits 16-21 of an expression
- \( \text{EXP2}(\text{expression}) \) returns \( 2^{\text{expression}} \)
- \( \text{LOG2}(\text{expression}) \) returns the integer part of \( \log_2(\text{expression}) \)
Instructions

- Arithmetic and logic instructions
- Branch instructions
- Data transfer instructions

CMD
Instruction Name

ARG1, ARG2
Arguments

ARG1 ← CMD(ARG1, ARG2)
Result of operation
Arithmetic and Logic Instructions

Almost all of the arithmetic and logic instructions consist of a two arguments and can modify all of the status bits in the SREG. All of the arithmetic and logic instructions are 8-bit only.

- Addition: ADD, ADC, ADIW
- Subtraction: SUB, SUBI, SBC, SBCI, SBIW
- Logic: AND, ANDI, OR, ORI, EOR
- Compliments: COM, NEG
- Register Bit Manipulation: SBR, CBR
- Register Manipulation: INC, DEC, TST, CLR, SER
- Multiplication: MUL, MULS, MULSU
- Fractional Multiplication: FMUL, FMULS, FMULSU
Arithmetic and Logic Instructions

- There is a common nomenclature to the naming of the instructions. The following table explains the nomenclature.

<table>
<thead>
<tr>
<th>Ending Letter</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Carry</td>
<td>Operation will involve the carry bit</td>
</tr>
<tr>
<td>I</td>
<td>Immediate</td>
<td>Operation involves an immediate value that is passed as the second argument.</td>
</tr>
<tr>
<td>W</td>
<td>Word</td>
<td>The operation is a 16-bit operation.</td>
</tr>
<tr>
<td>S</td>
<td>Signed</td>
<td>The operation handles signed numbers</td>
</tr>
<tr>
<td>SU</td>
<td>Signed/Unsigned</td>
<td>The operation handles both signed and unsigned.</td>
</tr>
</tbody>
</table>
Branch Instructions

- Branch Instructions are used to introduce logical decisions and flow of control within a program. About 20% of any program consists of branches. A branch instruction is basically an instruction that can modify the Program Counter (PC) and redirect where the next instruction is fetched. There are two types of branch instructions, unconditional branches and conditional branches.
Unconditional branches

Unconditional branches modify the PC directly. These instructions are known as jumps because they cause the program to “jump” to another location in program memory. There are several types of jump instructions (RJMP, IJMP, EIJMP, JMP), but the most common one is the relative jump, RJMP, because it takes the least amount of cycles to perform and can access the entire memory array.
Branch Instructions

- **Unconditional branches**

There are also special unconditional branch instructions known as function calls, or calls (RCALL, ICALL, EICALL, CALL). The function calls work just like the jump instructions, except they also push the next address of the PC on to the stack before making the jump. There is also a corresponding return instruction, RET, that pops the address from the stack and loads it into the PC. These instructions are used to create functions in AVR assembly.
Branch Instructions

- **Conditional branches**

  Conditional branches will only modify the PC if the corresponding condition is meant. In AVR, the condition is determined by looking at the Status Register (SREG) bits. For example, the Branch Not Equal, BRNE, instruction will look at the Zero Flag (Z) of the SREG. If Z = 0, then the branch is taken, else the branch is not taken. At first this might not seem very intuitive, but in AVR, all the comparisons take place before the branch.
Conditional branches

There are several things that can modify the SREG bits. Most arithmetic and logic instructions can modify all of the SREG bits. But what are more commonly used is the compare instructions, (CP, CPC, CPI, CPSE). The compare instructions will subtract the two corresponding registers in order to modify the SREG. The result of this subtraction is not stored back to the first argument.

With this in mind, take a look at BRNE again. If the values in two register are equal when they are subtracted, then the resulting value would be zero and then Z = 1. If they were not equal then Z would be 0. Now when BRNE is called, the Z bit can determine the condition.
# Branch Instructions

## Conditional branches

<table>
<thead>
<tr>
<th>Test</th>
<th>Boolean</th>
<th>Mnemonic</th>
<th>Complementary</th>
<th>Boolean</th>
<th>Mnemonic</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd &gt; Rr</td>
<td>(Z \cdot (N \oplus V) = 0)</td>
<td>BRLT(^{(1)})</td>
<td>Rd (\leq) Rr</td>
<td>(Z \cdot (N \oplus V) = 1)</td>
<td>BRGE(^*)</td>
<td>Signed</td>
</tr>
<tr>
<td>Rd (\geq) Rr</td>
<td>((N \oplus V) = 0)</td>
<td>BRGE</td>
<td>Rd &lt; Rr</td>
<td>((N \oplus V) = 1)</td>
<td>BRLT</td>
<td>Signed</td>
</tr>
<tr>
<td>Rd = Rr</td>
<td>(Z = 1)</td>
<td>BREQ</td>
<td>Rd (\neq) Rr</td>
<td>(Z = 0)</td>
<td>BRNE</td>
<td>Signed</td>
</tr>
<tr>
<td>Rd (\leq) Rr</td>
<td>((N \oplus V) = 1)</td>
<td>BRGE(^{(1)})</td>
<td>Rd &gt; Rr</td>
<td>((N \oplus V) = 0)</td>
<td>BRLT(^*)</td>
<td>Signed</td>
</tr>
<tr>
<td>Rd &lt; Rr</td>
<td>((N \oplus V) = 1)</td>
<td>BRLT</td>
<td>Rd (\geq) Rr</td>
<td>((N \oplus V) = 0)</td>
<td>BRGE</td>
<td>Signed</td>
</tr>
<tr>
<td>Rd &gt; Rr</td>
<td>(C + Z = 0)</td>
<td>BRLO(^{(1)})</td>
<td>Rd (\leq) Rr</td>
<td>(C + Z = 1)</td>
<td>BRSH(^*)</td>
<td>Unsigned</td>
</tr>
<tr>
<td>Rd (\geq) Rr</td>
<td>(C = 0)</td>
<td>BRSH/BRCC</td>
<td>Rd &lt; Rr</td>
<td>(C = 1)</td>
<td>BRLO/BRCS</td>
<td>Unsigned</td>
</tr>
<tr>
<td>Rd = Rr</td>
<td>(Z = 1)</td>
<td>BREQ</td>
<td>Rd (\neq) Rr</td>
<td>(Z = 0)</td>
<td>BRNE</td>
<td>Unsigned</td>
</tr>
<tr>
<td>Rd (\leq) Rr</td>
<td>(C + Z = 1)</td>
<td>BRSH(^{(1)})</td>
<td>Rd &gt; Rr</td>
<td>(C + Z = 0)</td>
<td>BRLO(^*)</td>
<td>Unsigned</td>
</tr>
<tr>
<td>Rd &lt; Rr</td>
<td>(C = 1)</td>
<td>BRLO/BRCS</td>
<td>Rd (\geq) Rr</td>
<td>(C = 0)</td>
<td>BRSH/BRCC</td>
<td>Unsigned</td>
</tr>
<tr>
<td>Carry</td>
<td>(C = 1)</td>
<td>BRCS</td>
<td>No carry</td>
<td>(C = 0)</td>
<td>BRCC</td>
<td>Simple</td>
</tr>
<tr>
<td>Negative</td>
<td>(N = 1)</td>
<td>BRMI</td>
<td>Positive</td>
<td>(N = 0)</td>
<td>BRPL</td>
<td>Simple</td>
</tr>
<tr>
<td>Overflow</td>
<td>(V = 1)</td>
<td>BRVS</td>
<td>No overflow</td>
<td>(V = 0)</td>
<td>BRVC</td>
<td>Simple</td>
</tr>
<tr>
<td>Zero</td>
<td>(Z = 1)</td>
<td>BREQ</td>
<td>Not zero</td>
<td>(Z = 0)</td>
<td>BRNE</td>
<td>Simple</td>
</tr>
</tbody>
</table>

Note: 1. Interchange Rd and Rr in the operation before the test, i.e., CP Rd,Rr \(\rightarrow\) CP Rr,Rd.
## Branch Instructions

### Conditional branches

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

#### SREG Definition

<table>
<thead>
<tr>
<th>비트명</th>
<th>읽기/쓰기</th>
<th>초깃값</th>
<th>설명</th>
</tr>
</thead>
</table>
| I     | R/W       | 0      | 전역 인터럽트 활성화 비트  
  • 이 비트가 1이고, 개별적인 인터럽트 활성화 비트가 1이 되어야 인터럽트가 작동한다.  
  • 인터럽트 서비스 루틴이 실행되면 자동으로 0이 되고, IRET 명령으로 인터럽트 서비스 루틴이 종료되면 자동으로 1이 된다.  
  • SIE 명령으로 1, CLI 명령으로 0을 만들 수 있다. |
| T     | R/W       | 0      | 비트 복사/저장 비트  
  • BST 명령으로 레지스터의 한 비트를 T 비트에 복사한다.  
  • BLD 명령으로 T 비트를 레지스터의 한 비트에 복사한다. |
| H     | R/W       | 0      | 반캐리(Half Carry) 비트  
  • 연산 결과 비트 3에서 비트 4로 캐리가 발생하면 1이 된다.  
  • BCD(Binary-Coded Decimal) 연산에 유용하다. |
| S     | R/W       | 0      | 부호 비트  
  • N=V, 음수 플래그와 2의 보수 오버플로 플래그의 배타적 OR |
| V     | R/W       | 0      | 2의 보수 오버플로 플래그  
  • 연산 결과 b7+6, 비트 7과 비트 6의 배타적 OR |
| N     | R/W       | 0      | 음수 플래그  
  • 연산 결과 MSB가 1이면 음수 플래그가 1이 된다. |
| Z     | R/W       | 0      | 제로 플래그  
  • 연산 결과 모든 비트가 0이면 제로 플래그가 1이 된다. |
| C     | R/W       | 0      | 캐리 플래그  
  • 연산 결과 MSB에서 캐리가 발생하면 1이 된다. |
Data Transfer Instructions

- Immediate addressing

Immediate addressing is simply a way to move a constant value into a register. Only one instruction supports immediate addressing, LDI. Also note that this instruction will only work on the upper 16 General Purpose Registers, R16 – R31. The following is an example of when LDI would be used.
Suppose there was a loop that needed to be looped 16 times. Well, a counter register could be loaded with the value 16 and then decremented after each loop. When the register reached zero, then the program will exit from the loop. Since the value 16 is a constant, we can load into the counter register by immediate addressing. The following code demonstrates this example.

```
.def counter = r22
ldi counter, 16
Loop: breq Exit
    adc r0, r1
    dec counter
    rjmp Loop
Exit: inc r0
```

; Create a register variable
; Load the immediate value 16 in counter
; If zero, exit loop
; Do something
; Decrement the counter
; Redo the loop
; Continue on with program
Data Transfer Instructions

- Direct addressing

Direct addressing is the simplest way of moving data from one area of memory to another. Direct addressing requires only the address to access the data. But it is limited to the use of the register file. For example, if you wanted to move a byte of data from one area in Data Memory to another area in Data Memory, you must first Load the data a register and then Store the data into the other area of memory. In general, every data manipulation instruction, except LDI, comes in a Load and Store pair. For Direct Addressing modes, the instruction pairs are LDS/STS and IN/OUT.
Data Transfer Instructions

The point of having multiple instruction pairs is to access different areas of memory.

- **LDS/STS** – Move data in and out of the entire range of the SRAM Data Memory
- **IN/OUT** – Move data in and out of the IO Memory or $0020 - $005F of the SRAM Data Memory. IN/OUT takes less instruction cycles than LDS/STS does.

The following is an example loop that continually increments the data value at a particular address.

```assembly
.equ   addr = $14D0
Loop: lds   r0, addr ; Address of data to be manipulated
       inc   r0 ; Load data to R0 from memory
       sts   addr, r0 ; Increment R0
       rjmp  Loop ; Store data back to memory
```
Bit and Bit-test Instructions

- **Shift and Rotate**

The AVR Instruction set specifies register shifts as two types of instructions, shifts and rotates. Shifting will just shift the last bit out to carry bit and shift in a 0 to the first bit. Rotating will shift out the last bit to the carry bit and shift in the carry bit to the first bit. Therefore rotating a register will not loose any bit data while shifting a register will loose the last bit. The instruction mnemonics are LSL, LSR, ROL, and ROR for Logical Shift Left, Logical Shift Right, Rotate Left Through Carry, and Rotate Right Through Carry respectively.
Bit and Bit-test Instructions

- Bit Manipulation

Bit Manipulation Instructions allow the programmer to manipulate individual bits within a register by setting, or making the value 1, and clearing, or making the value 0, the individual bits. There are three instruction pairs to manipulate the SREG, an I/O Register, or a General Purpose Register through the T flag in the SREG. BSET and BCLR will set and clear respectively any bit within the SREG register. SBI and CBI will set and clear any bit in any I/O register. BST will store any bit in any General Purpose Register to the T flag in the SREG and BLD will load the value of the T flag in the SREG to any bit in any General Purpose Register.
Bit and Bit-test Instructions

- SREG Manipulation

Although the instructions SBI and CBI will allow a programmer to set and clear any bit in the SREG, there are additional instructions that will set and clear specific bits within the SREG. This is useful for when the programmer does not want to keep track of which bit in the SREG is for what. The following table shows the mnemonics for each set and clear instruction pair are in the table below.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Bit Name</th>
<th>Set Bit</th>
<th>Clear Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Carry Bit</td>
<td>SEC</td>
<td>CLC</td>
</tr>
<tr>
<td>N</td>
<td>Negative Flag</td>
<td>SEN</td>
<td>CLN</td>
</tr>
<tr>
<td>Z</td>
<td>Zero Flag</td>
<td>SEZ</td>
<td>CLZ</td>
</tr>
<tr>
<td>I</td>
<td>Global Interrupt Flag</td>
<td>SEI</td>
<td>CLI</td>
</tr>
<tr>
<td>S</td>
<td>Signed Test Flag</td>
<td>SES</td>
<td>CLS</td>
</tr>
<tr>
<td>V</td>
<td>Two’s Complement OVF Flag</td>
<td>SEV</td>
<td>CLV</td>
</tr>
<tr>
<td>T</td>
<td>T Flag</td>
<td>SET</td>
<td>CLT</td>
</tr>
<tr>
<td>H</td>
<td>Half Carry Flag</td>
<td>SEH</td>
<td>CLH</td>
</tr>
</tbody>
</table>
Addressing Modes

- **Register Direct (Single Register)**
  The operand is contained in register d (Rd).

- INC R0, DEC R5, LSL R9
Addressing Modes

- Register Direct (Two Registers)
  Operands are contained in register \( r \) (Rr) and \( d \) (Rd). The result is stored in register \( d \) (Rd).

- ADD R1,R3
- SUB R5,R7

![Diagram of register file showing OP, Rr, Rd, d, and r connections]
Addressing Modes

- Immediate Mode
  Operates on register and immediate, stores value in register.
  - SUBI R16,8
  - ADIW R16,5
  - LDI R16,3
Addressing Modes

- **Data Direct**
  
  A 16-bit Data Address is contained in the 16 LSBs of a two-word instruction. Rd/Rr specify the destination or source register.
  - STS K,Rs
  - LDS Rd, K
Addressing Modes

- **Data Indirect**

  Operand address is the contents of the X-, Y-, or the Z-register.

  - LD Rd,X
  - LD Rd,X+ ; indirect with post decrement
  - ST –Y,Rs ; indirect with pre-decrement
Addressing Modes

- **I/O Direct**

  Operand address is contained in six bits of the instruction word. \( n \) is the destination or source register address.

  - `IN R10,PINB`
  - `OUT PORTB,R1`
Addressing Modes

- I/O Ports using Indirect
  - Ports can be accessed using SRAM access commands
    - Add $0x20$ to the port number
      - First 32 numbers are the registers
  - Example
    - .DEF register = R16
    - LDI ZH, HIGH(PORTB+32)
    - LDI ZL, LOW(PORTB+32)
    - LD register, Z
  - For I/O Registers located in extended I/O:
    - Commands like “In/Out” cannot be used
    - Instead replaced with direct and indirect memory instructions
      - LDS and STS (Load and Store from SRAM)
Addressing Modes

- Direct Program Addressing, JMP, and CALL

Program execution continues at the address immediate in the instruction word.
Addressing Modes

- Indirect Program Addressing, IJMP, and ICALL

Program execution continues at address contained by the Z-register (i.e., the PC is loaded with the contents of the Z-register).
Addressing Modes

- Relative Program Addressing, RJMP, and RCALL

Program execution continues at address PC + k + 1. The relative address k is from -2048 to 2047.
Flow of Control

- IF Statement

```python
if (n >= 3){
    variable++;
    n = variable;
}
```
Flow of Control

- **IF Statement**

  .def  n = r16
  .def  variable = r1
  .equ  cmp = 3

  cpi n, cmp ; Compare value
  IF:  brsh EXEC ; If n >= 3 then branch to EXEC
  rjmp NEXT ; Jump to NEXT if n >= 3 is false
  EXEC: inc variable ; increment variable
        mov n, variable ; Set n = variable
  NEXT: ; continue on with code
Flow of Control

**IF Statement**

```assembly
.def n = r16
.def variable = r1
.equ cmp = 3

cpi n, cmp ; Compare value
IF: brlo NEXT ; If n >= 3 is false then skip code
inc variable ; increment variable
mov n, variable ; Set n = variable
NEXT: ; Continue on with code
```
Flow of Control

- IF-ELSE Statement

```java
if (n == 5) {
    expr++;
} else {
    n = expr;
}
```
Flow of Control

- **IF-ELSE Statement**
  
  ```
  .def n = r16
  .def variable = r1
  .equ cmp = 5
  
  cpi n, cmp ; Compare value
  breq IF ; Branch to IF if n == 3
  rjmp ELSE ; Branch to ELSE if n == 3 is false
  
  IF:
  inc variable ; Increment variable
  rjmp NEXT ; Goto NEXT
  
  ELSE:
  mov n, variable ; Set n = variable
  
  NEXT: ; Continue
  ```
Flow of Control

- **IF-ELSE Statement**

  ```
  .def  n = r16
  .def  variable = r1
  .equ  cmp = 5
  
  cpi  n, cmp ; Compare value
  IF:  brne  ELSE ; Goto ELSE since n == 3 is false
  inc  variable ; Execute the IF statement
  rjmp  NEXT ; Continue on with code
  ELSE:  mov  n, variable ; Execute the ELSE statement
  NEXT:      ; Continue on with code
  ```
Flow of Control

- WHILE Statement

n = 0;
while (n < 10) {
    sum += n;
    n++;
}
Flow of Control

- **WHILE Statement**

  .def n = r16
  .def sum = r3
  .equ limit = 10

  ldi n, 0 ; n=0
  WHIL: cpi n, limit ; Compare n with limit
        brlo WHEX ; When n < limit, goto WHEX
        rjmp NEXT ; Condition is not met, continue with program
  WHEX: add sum, n ; sum += n
         inc n ; n++
         rjmp WHIL ; Go back to beginning of WHILE loop
  NEXT: ; Continue on with code
Flow of Control

- **WHILE Statement**

  ```
  .def n = r16
  .def sum = r3
  .equ limit = 10

  ldi n, 0 ; n=0
  WHIL: cpi n, limit ; Compare n with limit
         brsh NEXT ; When not n < limit, goto NEXT
         add sum, n ; sum += n
         inc n ; n++
         rjmp WHIL ; Go back to beginning of WHILE loop
  NEXT: nop ; Continue on with code
  ```
Flow of Control

- DO Statement

n=0;
do {
    sum += n;
n++;
} while( n < 10 );
Flow of Control

- **DO Statement**

```assembly
.def n = r16
.def sum = r3
equ limit = 10

ldi n,0 ; n=0
DO:
  add sum, n ; sum += n
  inc n ; n++
  cpi n, limit ; compare n to limit
  brlo DO ; if n < 10, goto DO
NEXT: nop
```
### FOR Statement

```java
for (expr1; expr2; expr3) {
    statement
}
```

```java
expr1;
while (expr2) {
    statement
    expr3;
}
```
FOR Statement

```cpp
for (n=0; n<10; n++) {
    sum += n;
}
```

```cpp
n=0;
while (n<10) {
    sum += n;
    n++;  
}
```
Flow of Control

- FOR Statement

```
.def    n = r16
.def    sum = r3
.equ    max = 10

ldi n, 0 ; Initialize n to 0
FOR:    cpi n, max ; Compare n to max value
        brlo EXEC ; If n < max, the goto EXEC
        rjmp NEXT ; If n < max is false, break out of FOR loop
EXEC:   add sum, n ; sum += n
        inc n ; n++
        rjmp FOR ; goto the start of FOR loop
NEXT:
```
Flow of Control

- **FOR Statement**

  .def  n = r16
  .def  sum = r3
  .equ  max = 10

  ldi  n, max  ; Initialize n to max
  FOR:  add  sum, n  ; sum += n
         dec  n       ; decrement n
         brne  FOR   ; repeat loop if n is not equal to 0

  NEXT:
Flow of Control

- **SWITCH Statement**

```java
switch (val) {
  case 1:
    a_cnt++;
    break;
  case 2:
    case 3:
    b_cnt++
    break;
    break;
  default:
    c_cnt++;
}
```
Flow of Control

.def  val = r16
.def  a_cnt = r5
.def  b_cnt = r6
.def  c_cnt = r7

ldi    val, 3                      : initialize val with an arbitrary number;

SWITCH:                                  ; The beginning of the SWITCH statement
  cpi    val, 1                      ; Compare val to 1
  breq   S_1                         ; Branch to S_1 if val == 1
  cpi    val, 2                      ; Compare val to 2
  breq   S_3                         ; Branch to S_3 if val == 2
  cpi    val, 3                      ; Compare val to 3
  breq   S_3                         ; Branch to S_3 if val == 3
  inc    c_cnt                       ; Execute Default
  rjmp   NEXT                        ; Break out of switch

S_1:                                ; Execute case 1
  inc    a_cnt
  rjmp   NEXT                        ; Break out of switch

S_3:                                ; Execute case 2
  inc    b_cnt

NEXT:                                ; Execute case 3
  nop
A subroutine or function is called via the CALL, RCALL, ICALL, or EICALL instructions and is matched with an RET instruction to return to the instruction address after the call. The function or subroutine is preceded by a label that signifies the name of function or subroutine. When a CALL instruction is implemented, the processor first pushes the address of the next instruction after the CALL instruction onto the stack. This is important to realize since it means that the stack **must** be initialized before functions or subroutines can be used.
The CALL instruction will then jump to the address specified by label used as the parameter. The next instruction to be executed will then be the first instruction with the subroutine or function. Upon exiting the subroutine or function, the return instruction, RET, must be called. The RET instruction will then pop the address of the next instruction after the CALL instruction from the stack and load into the PC. Thus the next instruction to be executed is the instruction after the CALL instruction.
Functions and Subroutines

It is important to keep track of what is pushed and popped on the stack. If within a subroutine or function, data is not popped correctly, the RET instruction can pop the wrong data values for the address and thus the program will not function correctly. Additionally, never exit a subroutine or function via another jump instruction other than RET. Doing so will cause the data in the stack to never be popped and thus the stack will become out of sink.
Functions and Subroutines

.include "m128def.inc"

.def ones_digit = r16
.def tens_digit = r17
.def temp = r18

INIT: ldi r16, high(RAMEND) ; Initialize the stack pointer high byte
       out SPH, r16
       ldi r16, low(RAMEND) ; Initialize the stack pointer low byte
       out SPL, r16

       ldi ones_digit, 34
       rcall DIGITS

DONE:  rjmp DONE
Functions and Subroutines

DIGITS:

push  temp
ldi tens_digit,0
ldi temp,10

REPEAT:
sub  ones_digit, temp
brmi MINUS
inc  tens_digit
rjmp REPEAT

MINUS:
add  ones_digit,temp
pop  temp
ret
Functions and Subroutines

```asm
.include "m128def.inc"

.def ones_digit = r16
.def tens_digit = r17
.def temp = r18

INIT:  ld i r16, high(RAMEND); Initialize the stack pointer high byte
       out SPH, r16
       ld i r16, low(RAMEND); Initialize the stack pointer low byte
       out SPL, r16

       ld i ones_digit, 34
       rcall DIGITS

DONE:  rjmp DONE

DIGITS:
       push temp
       ld i tens_digit, 0
       ld i temp, 10
       REPEAT: sub ones_digit, temp
               brmi MINUS
               inc tens_digit
               rjmp REPEAT
       MINUS:  add ones_digit, temp
               pop temp
               ret
```