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

Assembler Source

• 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:

- 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

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, 3	30 ; Load the nu	umber 30 into R16	
MOV R0, R1	.6 ; Copy R16 in	nto R0, R0 <- R16	
INC R0	; Increment F	R0, R0 <- R0+1	
ADD RO, RI	.6 ; R0 <- R0 +	R16, value in R0	should now be 61

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:

LDI R16, \$A4 ; Load the immediate hex value into R16 LD R17, X ; Load value from memory address in X-Pointer ADC R16, R17 ; Add with carry, R16 <- R16 + R17 + Carry Bit ST Y, R16 ; Store value in R16 to address in Y-Pointer

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:

Name	Byte	Assignment
V Pagistar	Low	R26
A-Register	High	R27
V Pagistar	Low	R28
Y-Register	High	R29
7 Pagistar	Low	R30
Z-Kegister	High	R31

Table 1: Address Register Assignments

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 LDI R27, \$02 ; Load 0x02 into the hig LD R16, X+ ; Load value from SRAM, INC R16 ; Manipulate value ST X, R16 ; Store value to SRAM	Byte of X h Byte of X increment X
---	---

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.

Bit	Name	Description
7	Ι	Global Interrupt Enable
6	Т	Bit Copy Storage
5	Н	Half Carry Flag
4	S	Sign Bit
3	V	Twos Compliment Overflow Flag
2	N	Negative Flag
1	Ζ	Zero Flag
0	С	Carry Flag

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.

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.

Bit	15	14	13	12	11	10	9	8	
	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	-

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







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

I/O Ports

		PUD			
DDxn	PORTxn	(in SFIOR)	I/O	Pull-up	Comment
0	0	Х	Input	No	Tri-state (Hi-Z)
					Pxn will source current if ext.
0	1	0	Input	Yes	pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	Х	Output	No	Output Low (Sink)
1	1	Х	Output	No	Output High (Source)

LDI OUT LDI OUT	R16, \$FF DDRB, R16 R16, \$FF PORTB, R16	<pre>; Select Direction as Output on all pins ; Set value in DDRB ; Set Initial value to high on all pins ; Set PORTB value, Port B pins should be high</pre>
LDI OUT LDI OUT	R16, \$00 DDRD, R16 R16, \$00 PORTD, R16	<pre>; Select Direction as Input on all pins ; Set value in DDRD ; Use normal Tri-state with no Pull-up resister ; Port D is now ready as input</pre>

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

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

- 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	on	
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	rl,Z	;	Load VAR1 into register 1

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 ; Start Data Segment vartab: .BYTE 4 ; Reserve 4 bytes in SRAM .CSEG const: .DW 2 ; Write 0x0002 in program memory mov r1, r0 ; Do something

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.

Syntax:		LABEL	: .DB	expres	ssionl:	ist
Example: .CSEG				L		
consts:						
. DI	з О,	255,	0b0101	10101,	-128,	\$AA
text:						
. DI	з " Н	ello	World"			

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, <mark>0</mark> x3f	;	Read SREG	into	ior 1	register
eor	temp,ior	;	Exclusive	or te	emp ar	^{nd ior} 21

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			;	Start data segment
	var1:.BY	FE 1	;	reserve 1 byte to var1
	table:.B	YTE 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

- 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 can not be changed or redefined.

```
Syntax:

.EQU label = expression

Example:

.EQU io_offset = 0x23

.EQU porta = io_offset + 2

.CSEG ; Start code segment

clr 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:

.EQU

.EQU

.EQU

- ; iodefs.asm:
- ; Status register
- ; Stack pointer high
- ; Stack pointer low
- ; incdemo.asm
- ; Include I/O definitions
- ; Read status register

.INCLUDE "iodefs.asm"

sreq=0x3f

sphiqh=0x3e

splow=0x3d

in r0,sreg

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		;	Start data segment
.ORG 0x67		;	Set SRAM address to hex 67
variable	:.BYTE 1	;	Reserve a byte at SRAM
		;	adr.67H
.ESEG		;	Start EEPROM Segment
.ORG 0x20		;	Set EEPROM location
		;	counter
eevar:	.DW 0xfeff	;	Initialize one word
.CSEG			
.ORG 0x10		;	Set Program Counter to hex
		;	10
mov	r0,r1	;	Do something

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 ; Start code segment

clr r2 ; Clear register 2

out porta,r2 ; Write to Port A

28
```

Expressions

 The Assembler incorporates expressions. Expressions can consist of operands, operators and functions. All expressions are internally 32 bits.

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, 0b1111111

 PC - the current value of the Program memory location counter
 3

Logical Not	Symbol:	ļ.					
	Description:	Unary operator which returns 1 if the expression was zero, and returns (if the expression was nonzero					
	Precedence:	14					
	Example:	ldi	r16,!0xf0	; Load r16 with 0x00)		
Bitwise Not	Symbol:	~					
	Description:	Unary o	perator which return	s the input expression with	all bits inverted		
	Precedence:	14					
	Example:	ldi	r16,~0xf0	; Load r16 with 0x0f	E		
Unary Minus	Symbol:	-					
	Description:	Unary o	perator which return	s the arithmetic negation of	an expression		
	Precedence:	14					
	Example:	ldi	r16,-2	; Load -2(0xfe) in r	c16		
Multiplication	Symbol:	*					
	Description:	Binary o	perator which return	is the product of two expres	sions		
	Precedence:	13					
	Example:	ldi	r30,label*2	; Load r30 with labe	el*2		

Division	Symbol:	1						
	Description:	Binary divided	operator which return by the right expressi	ns the integr on	er quotient o	of the left expres	sion	
	Precedence:	13						
	Example:	ldi	r30,label/2	; Load r	30 with	label/2		
Addition	Symbol:	+						
	Description:	Binary operator which returns the sum of two expressions						
	Precedence:	12						
	Example:	ldi	r30,c1+c2	; Load r	30 with	c1+c2		
Subtraction	Symbol:	-						
	Description:	Binary operator which returns the left expression minus the right expression						
	Precedence:	12						
	Example:	ldi	r17,c1-c2	;Load r1	17 with c	1-c2		
Shift left	Symbol:	<<						
	Description:	Binary times g	operator which return iven by the right expr	ns the <mark>l</mark> eft e ression	xpression s	hifted left a num	ber of	
	Precedence:	11						
	Example:	ldi	r17,1< <bitmask< th=""><th>;Load r1</th><th>17 with 1</th><th>shifted</th><th>~~</th></bitmask<>	;Load r1	17 with 1	shifted	~~	
				;left bi	ltmask ti	mes	32	

Shift right	Symbol:	>>						
	Description:	Binary operator which returns the left expression shifted right a number of times given by the right expression.						
	Precedence:	11						
	Example:	ldi	r17,c1>>c2	;Load r17 w	ith (c1 shifted		
				;right c2 t	imes			
Less than	Symbol:	<						
	Description:	Binary operator which returns 1 if the signed expression to the left is Less than the signed expression to the right, 0 otherwise						
	Precedence:	10						
	Example:	ori	r18,bitmask*(c1	. <c2)+1< th=""><th>;Or</th><th>r18 with</th></c2)+1<>	;Or	r18 with		
					; an	expression		
Less or Equal	Symbol:	<=						
	Description:	Binary of than or	operator which return Equal to the signed e	s 1 if the signed expression to the	l expr e righ	ession to the left is Less t, 0 otherwise		
	Precedence:	10						
	Example:	ori	r18,bitmask*(c1	<=c2)+1	;Or	r18 with		
					; an	expression		

Greater than	Symbol:	>						
	Description:	Binary operator which returns 1 if the signed expression to the left is Greater than the signed expression to the right, 0 otherwise						
	Precedence:	10						
	Example:	ori	r18,bitmask*(c1>c2)+1	;Or r18 with				
				;an expression				
Greater or Equal	Symbol:	>=						
	Description:	Binary operator which returns 1 if the signed expression to the left is Greater than or Equal to the signed expression to the right, 0 otherwise						
	Precedence:	10						
	Example:	ori	r18,bitmask*(c1>=c2)+1	;Or r18 with				
				;an expression				
Equal	Symbol:	==						
	Description:	Binary o Equal to	operator which returns 1 if the signed the signed expression to the right, 0	expression to the left is otherwise				
	Precedence:	9						
	Example:	andi	r19,bitmask*(c1==c2)+1	;And r19 with				
				;an expression				

Not Equal	Symbol:	!=					
	Description:	Binary operator which returns 1 if the signed expression to the left is Ne Equal to the signed expression to the right, 0 otherwise					
	Precedence:	9					
	Example:	.SET flag=(c1!=c2) ;Set flag to 1 or 0					
Bitwise And	Symbol:	&					
	Description:	Binary operator which returns the bitwise And between two expressions					
	Precedence:	8					
	Example:	ldi r18,High(c1&c2) ;Load r18 with an expression					
Bitwise Xor	Symbol:	٨					
	Description:	Binary operator which returns the bitwise Exclusive Or between two expressions					
	Precedence:	7					
	Example:	ldi r18,Low(c1 ^c 2) ;Load r18 with an expression					

Bitwise Or	Symbol:						
	Description:	Binary operator which returns the bitwise Or between two expressions					
	Precedence:	6					
	Example:	ldi	r18,Low(c1 c2)	;Load r18	with an	expression	
Logical And	Symbol:	&&					
	Description:	Binary operator which returns 1 if the expressions are both nonzero, 0 otherwise					
	Precedence:	5					
	Example:	ldi	r18,Low(c1&&c2)	:Load r18	with an	expression	
				, 1000 II0	WICH GH	011910221011	
Logical Or	Symbol:	II		,10000 110	witchi dii	0	
Logical Or	Symbol: Description:	 Binary o nonzero	operator which returns 1 i o, 0 otherwise	f one or both	of the expr	essions are	
Logical Or	Symbol: Description: Precedence:	 Binary o nonzero 4	operator which returns 1 i o, 0 otherwise	f one or both	of the expr	essions are	
Functions

The following functions are defined:

- LOW(expression) returns the low byte of an expression
- HIGH(expression) returns the second byte of an expression
- BYTE2(expression) is the same function as HIGH
- BYTE3(expression) returns the third byte of an expression
- BYTE4(expression) returns the fourth byte of an expression
- LWRD(expression) returns bits 0-15 of an expression
- HWRD(expression) returns bits 16-31 of an expression
- PAGE(expression) returns bits 16-21 of an expression
- EXP2(expression) returns 2^expression
- LOG2(expression) returns the integer part of log2(expression)

Instructions

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

CMDARG1, ARG2Instruction NameArgument1Argument1Argument2ARG1 ← 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.

Ending Letter	Meaning	Description
С	Carry	Operation will involve the carry bit
Ι	Immediate	Operation involves an immediate value that is passed as the
		second argument.
W	Word	The operation is a 16-bit operation.
S	Signed	The operation handles signed numbers
SU	Signed/Unsigned	The operation handles both signed and unsigned.

 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.

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.

Conditional branches

Conditional branches will only modify the PC if the corresponding condition is met. 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.

Conditional branches

Test	Boolean	Mnemonic	Complementary	Boolean	Mnemonic	Comment
Rd > Rr	$Z \bullet (N \oplus V) = 0$	BRLT ⁽¹⁾	$Rd \leq Rr$	Z+(N ⊕ V) = 1	BRGE*	Signed
$Rd \ge Rr$	(N ⊕ V) = 0	BRGE	Rd < Rr	(N ⊕ V) = 1	BRLT	Signed
Rd = Rr	Z = 1	BREQ	Rd ≠ Rr	Z = 0	BRNE	Signed
$Rd \leq Rr$	$Z+(N \oplus V) = 1$	BRGE ⁽¹⁾	Rd > Rr	$Z \bullet (N \oplus V) = 0$	BRLT*	Signed
Rd < Rr	(N ⊕ V) = 1	BRLT	$Rd \geq Rr$	$(N \oplus V) = 0$	BRGE	Signed
Rd > Rr	C + Z = 0	BRLO ⁽¹⁾	$Rd \leq Rr$	C + Z = 1	BRSH*	Unsigned
$Rd \ge Rr$	C = 0	BRSH/BRCC	Rd < Rr	C = 1	BRLO/BRCS	Unsigned
Rd = Rr	Z = 1	BREQ	Rd ≠ Rr	Z = 0	BRNE	Unsigned
Rd ≤ Rr	C + Z = 1	BRSH ⁽¹⁾	Rd > Rr	C + Z = 0	BRLO*	Unsigned
Rd < Rr	C = 1	BRLO/BRCS	$Rd \geq Rr$	C = 0	BRSH/BRCC	Unsigned
Carry	C = 1	BRCS	No carry	C = 0	BRCC	Simple
Negative	N = 1	BRMI	Positive	N = 0	BRPL	Simple
Overflow	V = 1	BRVS	No overflow	∨ = 0	BRVC	Simple
Zero	Z = 1	BREQ	Not zero	Z = 0	BRNE	Simple

Note: 1. Interchange Rd and Rr in the operation before the test, i.e., CP Rd, $Rr \rightarrow CP Rr$, Rd.

Conditional branches

Instruction	Abbreviation of	Comment
BREQ <i>Ibl</i>	Branch if Equal	Jump to location <i>lbl</i> if $Z = 1$,
BRNE <i>Ibl</i>	Branch if Not Equal	Jump if Z = 0, to location <i>lbl</i>
BRCS Ibl	Branch if Carry Set	Jump to location <i>lbl</i> , if $C = 1$
BRLO <i>Ibl</i>	Branch if Lower	
BRCC <i>Ibl</i>	Branch if Carry Cleared	Jump to location <i>lbl</i> , if $C = 0$
BRSH <i>lbl</i>	Branch if Same or Higher	
BRMI <i>Ibl</i>	Branch if Minus	Jump to location lbl, if $N = 1$
BRPL <i>Ibl</i>	Branch if Plus	Jump if $N = 0$
BRGE <i>Ibl</i>	Branch if Greater or Equal	Jump if S = 0
BRLT <i>Ibl</i>	Branch if Less Than	Jump if S = 1
BRHS <i>Ibl</i>	Branch if Half Carry Set	If $H = 1$ then jump to <i>lbl</i>
BRHC Ibl	Branch if Half Carry Cleared	if $H = 0$ then jump to lbl
BRTS	Branch if T flag Set	If $T = 1$ then jump to lbl
BRTC	Branch if T flag Cleared	If $T = 0$ then jump to lbl
BRIS	Branch if I flag set	If $I = 1$ then jump to lbl
BRIC	Branch if I flag cleared	If I = 0 then jump to lbl 47

SREG Definition

비트명	읽기/쓰기	초깃값	설명	
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 서비 스 루틴이 종료되면 자동으로 1이 된다. • SEI 명령으로 1, CLI 명령으로 0을 만들 수 있다.	
Т	R/W	0	비트 복사/저장 비트 • BST 명령으로 레지스터의 한 비트를 T 비트에 복사한다. • BLD 명령으로 T 비트를 레지스터의 한 비트에 복사한다.	
Н	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⊕b6, 비트 7과 비트 6의 배타적 OR	
Ν	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.	
С	R/W	0	캐리 플래그 • 연산 결과 MSB에서 캐리가 발생하면 1이 된다.	4

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 **Idi 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

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.

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.

```
.equ addr = $14D0
Loop: lds r0, addr
inc r0
sts addr, r0
rjmp Loop
```

- ; Address of data to be manipulated
- ; Load data to R0 from memory
- ; Increment R0
- ; Store data back to memory
 - ; Jump back to loop

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.

Bit	Bit Name	Set Bit	Clear Bit
С	Carry Bit	SEC	CLC
Ν	Negative Flag	SEN	CLN
Ζ	Zero Flag	SEZ	CLZ
Ι	Global Interrupt Flag	SEI	CLI
S	Signed Test Flag	SES	CLS
V	Two's Complement OVF Flag	SEV	CLV
Т	T Flag	SET	CLT
Η	Half Carry Flag	SEH	CLH

Register Direct(Single Register)

The operand is contained in register d (Rd).

INC R0, DEC R5, LSL R9



Register Direct(Two Registers)

Operands are contained in register r (Rr) and d (Rd). The result is stored in register d (Rd).



Immediate Mode

Operates on register and immediate, stores value in register.

- SUBI R16,8
- ADIW R16,5
- LDI R16,3

Data Direct

A 16-bit Data Address is contained in the 16 LSBs of a twoword instruction. Rd/Rr specify the destination or source register.



Data Indirect

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

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



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



I/O Ports using Indirect

- Ports can be accessed using SRAM access commands
 - Add ox20 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)

 Direct Program Addressing, JMP, and CALL

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



 Indirect Program Addressing, IJMP, and ICALL

Program execution continues at address contained by the Zregister (i.e., the PC is loaded with the contents of the Zregister).



 Relative Program Addressing, RJMP, and RCALL

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



IF Statement

if (n >= 3){
 variable++;
 n = variable;
}

IF Statement

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

	срі	n, cmp	; Compare value
IF:	brsh	EXEC	; If $n \ge 3$ 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

IF Statement

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

	срі	n, cmp	; Compare value
IF:	brlo	NEXT	; If $n \ge 3$ is false then skip code
	inc	variable	; increment variable
	mov	n, variable	; Set n = variable
NEXT:			; Continue on with code

IF-ELSE Statement

```
if (n == 5) {
     expr++;
}
else {
     n = expr;
}
```

IF-ELSE Statement

- .def n = r16
- .def variable = r1
- .equ cmp = 5

	срі	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

IF-ELSE Statement

- .def n = r16
- .def variable = r1
- .equ cmp = 5

	срі	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

WHILE Statement

```
n = 0;
while (n < 10) {
sum += n;
n++;
}
```
WHILE Statement

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

	ldi	n, 0	; n=0
WHIL:	срі	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

WHILE Statement

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

	ldi	n,0	; n=0
WHIL:	срі	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

DO Statement

n=0; do {

sum += n; n++; } while(n < 10);

DO Statement

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

	ldi	n,0	; n=0
DO:	add	sum, n	; sum += n
	inc	n	; n++
	срі	n, limit	; compare n to limit
	brlo	DO	; if n < 10, goto DO
NEXT:	nop		

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

```
expr1;
while (expr2) {
statement
expr3;
}
```

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

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

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

	ldi	n, 0	; Initialize n to 0
FOR:	срі	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:			

- .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:			

SWITCH Statement switch (val) { case 1: a_cnt++; break; case 2: case 3: b_cnt++ break; default: c_cnt++; }

.defval = 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
	срі	val, 1	; Compare val to 1
	breq	S_1	; Branch to S_1 if val == 1
	срі	val, 2	; Compare val to 2
	breq	S_3	; Branch to S_3 if val == 2
	срі	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:	inc	a_cnt	; Execute case 1
	rjmp	NEXT	; Break out of switch
S_ 3:	inc	b_cnt	; Execute case 2
NEXT:	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 implement, 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.

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.

.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

OIGITS

DONE: rjmp DONE

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
	рор	temp
	ret	

🐞 AVR Studio – [C:\work\avr\divide\divide,asm]						
Eile <u>P</u> roject	<u>B</u> uild <u>E</u> dit	<u>∨</u> iew	<u>T</u> ools	<u>D</u> ebug	<u>W</u> indow	/ <u>H</u> elp
፡ 🗋 💕 🔙 🕼 🙂) 🗈 🛍 é	<u>3</u> 19 ((* F2 6	а:м,	6 % %	🗞 🌾 🗐 🔚 🙀 🖡 🕨 💷 💷 💷 🛥 🔶 🕾 🚂 🌚 🧶
Trace Disabled	- 2 %	< 8.2	L 木 i	Con AVR	* * *	2 AUTO 🗄 🎬 🎬
Processor		-	×	.includ	e "m128d	def.inc"
Program Cou Stack Pointer X pointer Y pointer	0×000006 0×10FF 0×0000 0×0000			.def .def .def INIT:	ones_di tens_di temp ldi	ligit = r16 ligit = r17 = r18 r16. high(BAMEND) ; Initialize the stack pointer high byte
Z pointer Cycle Counter Frequency Ston Watch	0x0000 200497 1,0000 MHz 200497 00 us				out Idi out	SPH, r16 r16, low(RAMEND) ; Initialize the stack pointer low byte SPL, r16
SREG Begisters R00		NZC	4	>DONE:	ldi rcall rjmp	ones_digit,34 DIGITS DONE
R01 R02 R03	0x00 0x00 0x00			DIGITS:	push	temp
R04 R05 R06	0×00 0×00 0×00			REPEAT :	ldi sub brmi	tens_digit,0 temp,10 ones_digit, temp MINUS
R07 R08 R09	0×00 0×00 0×00			MINUS:	inc rjmp add pop	tens_digit REPEAT ones_digit,temp temp
R10 R11 R12	0x00 0x00 0x00				ret	
R13 R14 B15	0x00 0x00 0x00					
R16 R17 R18	0x04 0x03 0x00					88