APPENDIX A

AVR INSTRUCTIONS EXPLAINED

OVERVIEW

In this appendix, we describe each intruction of the ATmega32. In many cases, a simple code example is given to clarify the instruction.

At the end there is a table that shows all the registers and their bits.

Instructions are Copyright of Atmel Semiconductor, Inc. 2009, Used by Permission

SECTION A.1: INSTRUCTION SUMMARY

DATA TRANSFER INSTRUCTIONS

Mnemonics	Operands	Description	Operation	Flags
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None
MOVW	Rd, Rr	Copy Register Word	Rd + 1:Rd ← Rr + 1:Rr	None
LDI	Rd, K	Load Immediate	Rd ← K	None
LD	Rd, X	Load Indirect	Rd ← (X)	None
LD	Rd, X+	Load Indirect and Post-Inc.	Rd ← (X), X ← X + 1	None
LD	Rd, –X	Load Indirect and Pre-Dec.	X ← X – 1, Rd ← (X)	None
LD	Rd, Y	Load Indirect	Rd ← (Y)	None
LD	Rd, Y+	Load Indirect and Post-Inc.	Rd ← (Y), Y ← Y + 1	None
LD	Rd, –Y	Load Indirect and Pre-Dec.	Y ← Y – 1, Rd ← (Y)	None
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None
LD	Rd, Z	Load Indirect	Rd ← (Z)	None
LD	Rd, Z+	Load Indirect and Post-Inc.	Rd ← (Z), Z ← Z+1	None
LD	Rd, –Z	Load Indirect and Pre-Dec.	Z ← Z – 1, Rd ← (Z)	None
LDD	Rd, Z + q	Load Indirect with Displacement	Rd ← (Z + q)	None
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None
ST	X, Rr	Store Indirect	(X) ← Rr	None
ST	X+, Rr	Store Indirect and Post-Inc.	(X) ← Rr, X ← X + 1	None
ST	–X, Rr	Store Indirect and Pre-Dec.	X ← X – 1, (X) ← Rr	None
ST	Y, Rr	Store Indirect	(Y) ← Rr	None
ST	Y+, Rr	Store Indirect and Post-Inc.	(Y) ← Rr, Y ← Y + 1	None
ST	–Y, Rr	Store Indirect and Pre-Dec.	Y ← Y – 1, (Y) ← Rr	None
STD	Y + q, Rr	Store Indirect with Displacement	(Y + q) ← Rr	None
ST	Z, Rr	Store Indirect	(Z) ← Rr	None
ST	Z+, Rr	Store Indirect and Post-Inc.	(Z) ← Rr, Z ← Z + 1	None
ST	–Z, Rr	Store Indirect and Pre-Dec.	Z ← Z – 1, (Z) ← Rr	None
STD	Z + q, Rr	Store Indirect with Displacement	(Z + q) ← Rr	None
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None
LPM		Load Program Memory	R0 ← (Z)	None
LPM	Rd, Z	Load Program Memory	Rd ← (Z)	None
LPM	Rd, Z+	Load Program Memory and Post-Inc.	Rd ← (Z), Z ← Z+1	None
SPM		Store Program Memory	(Z) ← R1:R0	None
IN	Rd, P	In Port	Rd ← P	None
OUT	P, Rr	Out Port	P ← Rr	None
PUSH	Rr	Push Register on Stack	Stack ← Rr	None
POP	Rd	Pop Register from Stack	Rd ← Stack	None

BRANCH INSTRUCTIONS

Mnem.	Oper.	Description	Operation	Flags
RJMP	k	Relative Jump	PC ← PC + k + 1	None
IJMP		Indirect Jump to (Z)	PC ← Z	None
JMP	k	Direct Jump	PC ← k	None
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None
ICALL		Indirect Call to (Z)	PC ← Z	None
CALL	k	Direct Subroutine Call	PC ← k	None
RET		Subroutine Return	PC ← Stack	None
RETI		Interrupt Return	PC ← Stack	I
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC ← PC + 2 or 3	None
CP	Rd,Rr	Compare	Rd − Rr	Z,N,V,C,H
CPC	Rd,Rr	Compare with Carry	Rd − Rr − C	Z,N,V,C,H
CPI	Rd,K	Compare Register with Immediate	Rd − K	Z,N,V,C,H
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None
SBRS	Rr, b	Skip if Bit in Register is Set	if (Rr(b)=1) PC ← PC + 2 or 3	None
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b)=0) PC ← PC + 2 or 3	None
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b)=1) PC ← PC + 2 or 3	None
BRBS	s, k	Branch if Status Flag Set	if (SREG(s)=1) then PC←PC+k+1	None
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s)=0) then PC←PC+k+1	None
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None
BRGE	k	Branch if Greater or Equal,Signed	if (N and V= 0) then PC←PC + k +1	None
BRLT	k	Branch if Less Than Zero, Signed	if (N and V= 1) then PC←PC + k +1	None
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None

BIT AND BIT-TEST INSTRUCTIONS

Mnem.	Operan.	Description	Operation	Flags
SBI	P, b	Set Bit in I/O Register	I/O(P, b) ← 1	None
CBI	P, b	Clear Bit in I/O Register	I/O(P, b) ← 0	None
LSL	Rd	Logical Shift Left	Rd(n + 1) ← Rd(n), Rd(0) ← 0	Z,C,N,V
LSR	Rd	Logical Shift Right	Rd(n)←Rd(n+1), Rd(7)←0	Z,C,N,V
ROL	Rd	Rotate Left Through Carry	Rd(0)←C, Rd(n+1)←Rd(n), C←Rd(7)	Z,C,N,V
ROR	Rd	Rotate Right Through Carry	Rd(7) ← C, Rd(n) ← Rd(n + 1), C ← Rd(0)	Z,C,N,V
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n + 1), n = 0..6	Z,C,N,V
SWAP	Rd	Swap Nibbles	Rd(3..0) ← Rd(7..4), Rd(7..4) ← Rd(3..0)	None
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None
SEC		Set Carry	C ← 1	C
CLC		Clear Carry	C ← 0	C
SEN		Set Negative Flag	N ←1	N
CLN		Clear Negative Flag	N ← 0	N
SEZ		Set Zero Flag	Z ←1	Z
CLZ		Clear Zero Flag	Z ← 0	Z
SEI		Global Interrupt Enable	I ← 1	I
CLI		Global Interrupt Disable	I ← 0	I
SES		Set Signed Test Flag	S ← 1	S
CLS		Clear Signed Test Flag	S ← 0	S
SEV		Set Two's Complement Overflow	V ← 1	V
CLV		Clear Two's Complement Overflow	V ← 0	V
SET		Set T in SREG	T ← 1	T
CLT		Clear T in SREG	T ← 0	T
SEH		Set Half Carry Flag in SREG	H ←1	H
CLH		Clear Half Carry Flag in SREG	H ← 0	H

ARITHMETIC AND LOGIC INSTRUCTIONS

Mnem.	Operands	Description	Operation	Flags
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H
ADIW	Rdl, K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd – Rr	Z,C,N,V,H
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd – K	Z,C,N,V,H
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd – Rr – C	Z,C,N,V,H
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd – K – C	Z,C,N,V,H
SBIW	Rdl, K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl – K	Z,C,N,V,S
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd Rr	Z,N,V
COM	Rd	One's Complement	Rd ← $FF − Rd	Z,C,N,V
NEG	Rd	Two's Complement	Rd ← $00 − Rd	Z,C,N,V,H
SBR	Rd, K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V
CBR	Rd, K	Clear Bit(s) in Register	Rd ← Rd • ($FF – K)	Z,N,V
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V
DEC	Rd	Decrement	Rd ← Rd − 1	Z,N,V
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V
CLR	Rd	Clear Register	Rd ← $00	Z,N,V
SER	Rd	Set Register	Rd ← $FF	None
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd x Rr	Z,C
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd x Rr	Z,C
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd x Rr	Z,C
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← (Rd x Rr)<< 1	Z,C
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← (Rd x Rr)<< 1	Z,C
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← (Rd x Rr)<< 1	Z,C

MCU CONTROL INSTRUCTIONS

Mnemonics	Operands	Description	Operation	Flags
NOP		No Operation		None
SLEEP		Sleep	(see specific descr. for Sleep function)	None
WDR		Watchdog Reset	(see specific descr. for WDR/timer)	None
BREAK		Break	For On-Chip Debug Only	None

SECTION A.2: AVR INSTRUCTIONS FORMAT

ADC Rd, Rr ; **Add with carry**
$0 \le d \le 31, 0 \le r \le 31$; **Rd ← Rd + Rr + C**

Adds two registers and the contents of the C flag and places the result in the destination register Rd.

Flags: H, S, V, N, Z, C Cycles: 1

Example:

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

ADD Rd, Rr ; **Add without carry**
$0 \le d \le 31, 0 \le r \le 31$; **Rd ← Rd + Rr**

Adds two registers without the C flag and places the result in the destination register Rd.

Flags: H, S, V, N, Z, C Cycles: 1

Example:

```
add r1,r2        ;Add r2 to r1 (r1=r1+r2)
add r28,r28      ;Add r28 to itself (r28=r28+r28)
```

ADIW Rd+1:Rd, K ; **Add Immediate to Word**
$d \in \{24,26,28,30\}, 0 \le K \le 63$; **Rd + 1:Rd ← Rd + 1:Rd + K**

Adds an immediate value (0–63) to a register pair and places the result in the register pair. This instruction operates on the upper four register pairs, and is well suited for operations on the pointer registers.

Flags: S, V, N, Z, C Cycles: 2

Example:

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

AND Rd, Rr ; **Logical AND**
$0 \le d \le 31, 0 \le r \le 31$; **Rd ← Rd • Rr**

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

Flags: S, V ← 0, N, Z Cycles: 1

Example:

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

ANDI Rd, K ; **Logical AND with Immediate**
$16 \le d \le 31, 0 \le K \le 255$; **Rd ← Rd • K**

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

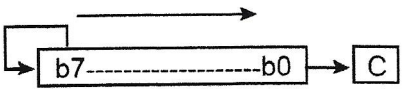
Flags: S, V ← 0, N, Z Cycles: 1

Example:

```
andi r17,$0F            ;Clear upper nibble of r17
andi r18,$10            ;Isolate bit 4 in r18
```

ASR Rd ; **Arithmetic Shift Right**
$0 \le d \le 31$

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



Flags: S, V, N, Z, C Cycles: 1

Example:

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

BCLR s ; **Bit Clear in SREG**
$0 \le s \le 7$; **SREG(s) ← 0**

Clears a single flag in SREG (Status Register).

Flags: I, T, H, S, V, N, Z, C Cycles: 1

Example:

```
bclr 0                  ;Clear Carry flag
bclr 7                  ;Disable interrupts
```

BLD Rd, b ; **Bit Load from the T Flag in SREG to a Bit in Register**
$0 \le d \le 31, 0 \le b \le 7$; **Rd(b) ← T**

Copies the T flag in the SREG (Status Register) to bit b in register Rd.

Flags: --- Cycles: 1

Example:

```
bst r1,2                ;Store bit 2 of r1 in T flag
bld r0,4                ;Load T flag into bit 4 of r0
```

BRBC s, k ; **Branch if Bit in SREG is Cleared**
$0 \le s \le 7, -64 \le k \le +63$; **If SREG(s) = 0 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests a single bit in SREG (Status Register) and branches relatively to PC if the bit is set.

Flags: --- Cycles: 1or 2

Example:

```
        cpi r20,5       ;Compare r20 to the value 5
        brbc 1,noteq    ;Branch if Zero flag cleared
        ...
noteq:nop               ;Branch destination (do nothing)
```

BRBS s, k ; **Branch if Bit in SREG is Set**
$0 \le s \le 7, -64 \le k \le +63$; **If SREG(s) = 1 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests a single bit in SREG (Status Register) and branches relatively to PC if the bit is set.

Flags: --- Cycles: 1 or 2

Example:

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

BRCC k ; **Branch if Carry Cleared**

$-64 \le k \le +63$; **If C = 0 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Carry flag (C) and branches relatively to PC if C is cleared.

Flags: --- Cycles: 1 or 2

Example:

```
            add r22,r23     ;Add r23 to r22
            brcc nocarry    ;Branch if carry cleared
            ...
nocarry:    nop             ;Branch destination (do nothing)
```

BRCS k ; **Branch if Carry Set**

$-64 \le k \le +63$; **If C = 1 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Carry flag (C) and branches relatively to PC if C is set.

Flags: --- Cycles: 1 or 2

Example:

```
            cpi r26,$56     ;Compare r26 with $56
            brcs carry      ;Branch if carry set
            ...
carry:      nop             ;Branch destination (do nothing)
```

BREAK ; **Break**

The BREAK instruction is used by the on-chip debug system, and is normally not used in the application software. When the BREAK instruction is executed, the AVR CPU is set in the stopped mode. This gives the on-chip debugger access to internal resources.

Flags: --- Cycles: 1

Example: ---

BREQ k ; **Branch if Equal**

$-64 \le k \le +63$; **If Rd = Rr (Z = 1) then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Zero flag (Z) and branches relatively to PC if Z is set. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the unsigned or signed binary number represented in Rd was equal to the unsigned or signed binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:

```
            ccp r1,r0       ;Compare registers r1 and r0
            breq equal      ;Branch if registers equal
            ...
equal:      nop             ;Branch destination (do nothing)
```

BRGE k ; **Branch if Greater or Equal (Signed)**
$-64 \le k \le +63$; **If Rd ≥ Rr (N⊕V = 0) then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Signed flag (S) and branches relatively to PC if S is cleared. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the signed binary number represented in Rd was greater than or equal to the signed binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:

```
            cp r11,r12          ;Compare registers r11 and r12
            brge greateq        ;Branch if r11 ≥ r12 (signed)
            ...
greateq:    nop                 ;Branch destination (do nothing)
```

BRHC k ; **Branch if Half Carry Flag is Cleared**
$-64 \le k \le +63$; **If H = 0 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Half Carry flag (H) and branches relatively to PC if H is cleared.

Flags: --- Cycles: 1 or 2

Example:

```
            brhc hclear         ;Branch if Half Carry flag cleared
            ...
hclear:     nop                 ;Branch destination (do nothing)
```

BRHS k ; **Branch if Half Carry Flag is Set**
$-64 \le k \le +63$; **If H = 1 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Half Carry flag (H) and branches relatively to PC if H is set.

Flags: --- Cycles: 1 or 2

Example:

```
            brhs hset           ;Branch if Half Carry flag set
            ...
hset:       nop                 ;Branch destination (do nothing)
```

BRID k ; **Branch if Global Interrupt is Disabled**
$-64 \le k \le +63$; **If I = 0 then PC←PC + k + 1, else PC←PC + 1**

Conditional relative branch. Tests the Global Interrupt flag (I) and branches relatively to PC if I is cleared.

Flags: --- Cycles: 1 or 2

Example:

```
            brid intdis         ;Branch if interrupt disabled
            ...
intdis:     nop                 ;Branch destination (do nothing)
```

BRIE k ; **Branch if Global Interrupt is Enabled**
$-64 \le k \le +63$; **If I = 1 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Global Interrupt flag (I) and branches relatively to PC if I is set.

Flags: --- Cycles: 1 or 2

Example:

```
            brie inten          ;Branch if interrupt enabled
            ...
inten:      nop                 ;Branch destination (do nothing)
```

BRLO k ; **Branch if Lower (Unsigned)**
$-64 \le k \le +63$; **If Rd < Rr (C = 1) then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Carry flag (C) and branches relatively to PC if C is set. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the unsigned binary number represented in Rd was smaller than the unsigned binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:

```
            eor r19,r19         ;Clear r19
loop:       inc r19             ;Increment r19
            ...
            cpi r19,$10         ;Compare r19 with $10
            brlo loop           ;Branch if r19 < $10 (unsigned)
            nop                 ;Exit from loop (do nothing)
```

BRLT k ; **Branch if Less Than (Signed)**
$-64 \le k \le +63$; **If Rd < Rr (N ⊕ V = 1) then PC← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Signed flag (S) and branches relatively to PC if S is set. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the signed binary number represented in Rd was less than the signed binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:

```
            bcp r16,r1          ;Compare r16 to r1
            brlt less           ;Branch if r16 < r1 (signed)
            ...
less:       nop                 ;Branch destination (do nothing)
```

BRMI k ; **Branch if Minus**
$-64 \le k \le +63$; **If N=1 then PC←PC + k + 1, else PC←PC + 1**

Conditional relative branch. Tests the Negative flag (N) and branches relatively to PC if N is set.

Flags: --- Cycles: 1 or 2

Example:

```
            subi r18,4          ;Subtract 4 from r18
            brmi negative       ;Branch if result negative
            ...
negative:   nop                 ;Branch destination (do nothing)
```

BRNE k ; **Branch if Not Equal**
$-64 \le k \le +63$; **If Rd ≠ Rr (Z = 0) then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Zero flag (Z) and branches relatively to PC if Z is cleared. If the instruction is executed immediately after any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the unsigned or signed binary

number represented in Rd was not equal to the unsigned or signed binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:

```
            eor r27,r27        ;Clear r27
loop:       inc r27            ;Increment r27
            ...
            cpi r27,5          ;Compare r27 to 5
            brne loop          ;Branch if r27 not equal 5
            nop                ;Loop exit (do nothing)
```

BRPL k ; **Branch if Plus**

$-64 \le k \le +63$; **If N = 0 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Negative flag (N) and branches relatively to PC if N is cleared.

Flags: --- Cycles: 1 or 2

Example:

```
            subi r26,$50       ;Subtract $50 from r26
            brpl positive      ;Branch if r26 positive
            ...
positive:   nop                ;Branch destination (do nothing)
```

BRSH k ; **Branch if Same or Higher (Unsigned)**

$-64 \le k \le +63$; **If Rd ≥Rr (C = 0) then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Carry flag (C) and branches relatively to PC if C is cleared. If the instruction is executed immediately after execution of any of the instructions CP, CPI, SUB, or SUBI, the branch will occur if and only if the unsigned binary number represented in Rd was greater than or equal to the unsigned binary number represented in Rr.

Flags: --- Cycles: 1 or 2

Example:

```
            subi r19,4         ;Subtract 4 from r19
            brsh highsm        ;Branch if r19 >= 4 (unsigned)
            ...
highsm:     nop                ;Branch destination (do nothing)
```

BRTC k ; **Branch if the T Flag is Cleared**

$-64 \le k \le +63$; **If T = 0 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the T flag and branches relatively to PC if T is cleared.

Flags: --- Cycles: 1 or 2

Example:

```
            bst r3,5           ;Store bit 5 of r3 in T flag
            brtc tclear        ;Branch if this bit was cleared
            ...
tclear:     nop                ;Branch destination (do nothing)
```

BRTS k ; **Branch if the T Flag is Set**
$-64 \le k \le +63$; **If T = 1 then PC←PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the T flag and branches relatively to PC if T is set.

Flags: --- Cycles: 1 or 2

Example:

```
            bst r3,5        ;Store bit 5 of r3 in T flag
            brts tset       ;Branch if this bit was set
            ...
tset:       nop             ;Branch destination (do nothing)
```

BRVC k ; **Branch if Overflow Cleared**
$-64 \le k \le +63$; **If V = 0 then PC ← PC + k + 1, else PC ← PC + 1**

Conditional relative branch. Tests the Overflow flag (V) and branches relatively to PC if V is cleared.

Flags: --- Cycles: 1 or 2

Example:

```
            add r3,r4       ;Add r4 to r3
            brvc noover     ;Branch if no overflow
            ...
noover:     nop             ;Branch destination (do nothing)
```

BRVS k ; **Branch if Overflow Set**
$-64 \le k \le +63$; **If V=1 then PC←PC + k + 1, else PC←PC + 1**

Conditional relative branch. Tests the Overflow flag (V) and branches relatively to PC if V is set.

Flags: --- Cycles: 1 or 2

Example:

```
            add r3,r4       ;Add r4 to r3
            brvs overfl     ;Branch if overflow
            ...
overfl:     nop             ;Branch destination (do nothing)
```

BSET s ; **Bit Set in SREG**
$0 \le s \le 7$; **SREG(s) ← 1**

Sets a single flag or bit in SREG (Status Register).

Flags: Any of the flags. Cycles: 1

Example:

```
            bset 6          ;Set T flag
            bset 7          ;Enable interrupt
```

BST Rd,b ; **Bit Store from Register to T Flag in SREG**
$0 \le d \le 31, 0 \le b \le 7$; **T ← Rd(b)**

Stores bit b from Rd to the T flag in SREG (Status Register).

Flags: T Cycles: 1

Example:

```
                            ;Copy bit
            bst r1,2        ;Store bit 2 of r1 in T flag
            bld r0,4        ;Load T into bit 4 of r0t
```

CALL k ; **Long Call to a Subroutine**
$0 \leq k < 64K$ (Devices with 16 bits PC) or $0 \leq k < 4M$ (Devices with 22 bits PC)

Calls to a subroutine within the entire program memory. The return address (to the instruction after the CALL) will be stored onto the stack. (See also RCALL.) The stack pointer uses a post-decrement scheme during CALL.

Flags: --- Cycles: 4

Example:

```
        mov r16,r0      ;Copy r0 to r16
        call check      ;Call subroutine
        nop             ;Continue (do nothing)
        ...
check:  cpi r16,$42     ;Check if r16 has a special value
        breq error      ;Branch if equal
        ret             ;Return from subroutine
        ...
error:  rjmp error      ;Infinite loop
```

CBI A, b ; **Clear Bit in I/O Register**
$0 \leq A \leq 31, 0 \leq b \leq 7$; **I/O(A,b) ← 0**

Clears a specified bit in an I/O Register. This instruction operates on the lower 32 I/O registers (addresses 0–31).

Flags: --- Cycles: 2

Example:

```
        cbi $12,7       ;Clear bit 7 in Port D
```

CBR Rd, k ; **Clear Bits in Register**
$16 \leq d \leq 31, 0 \leq K \leq 255$; **Rd ← Rd • ($FF – K)**

Clears the specified bits in register Rd. Performs the logical AND between the contents of register Rd and the complement of the constant mask K.

Flags: S, N, V ← 0, Z Cycles: 1

Example:

```
        cbr r16,$F0     ;Clear upper nibble of r16
        cbr r18,1       ;Clear bit 0 in r18
```

CLC ; **Clear Carry Flag**
; **C ← 0**

Clears the Carry flag (C) in SREG (Status Register).

Flags: C ← 0. Cycles: 1

Example:

```
        add r0,r0       ;Add r0 to itself
        clc             ;Clear Carry flag
```

CLH ; **Clear Half Carry Flag**
; **H ← 0**

Clears the Half Carry flag (H) in SREG (Status Register).

Flags: H ← 0. Cycles: 1

Example:

```
        clh             ;Clear the Half Carry flag
```

CLI ; **Clear Global Interrupt Flag**
; **I ← 0**

Clears the Global Interrupt flag (I) in SREG (Status Register). The interrupts will be immediately disabled. No interrupt will be executed after the CLI instruction, even if it occurs simultaneously with the CLI instruction.

Flags: I ← 0. Cycles: 1

Example:

```
in temp, SREG      ;Store SREG value
                   ;(temp must be defined by user)
cli                ;Disable interrupts during timed sequence
sbi EECR, EEMWE    ;Start EEPROM write
sbi EECR, EEWE     ;
out SREG, temp     ;Restore SREG value (I-flag)
```

CLN ; **Clear Negative Flag**
; **N ← 0**

Clears the Negative flag (N) in SREG (Status Register).

Flags: N ← 0. Cycles: 1

Example:

```
add r2,r3          ;Add r3 to r2
cln                ;Clear Negative flag
```

CLR Rd ; **Clear Register**
0 ≤ d ≤ 31 ; **Rd ← Rd ⊕ Rd**

Clears a register. This instruction performs an Exclusive-OR between a register and itself. This will clear all bits in the register..

Flags: S ← 0 , N ← 0, V ← 0, Z ← 0 Cycles: 1

Example:

```
        clr r18            ;Clear r18
loop:   inc r18            ;Increment r18
        ...
        cpi r18,$50        ;Compare r18 to $50
        brne loop
```

CLS ; **Clear Signed Flag**
; **S ← 0**

Clears the Signed flag (S) in SREG (Status Register).

Flags: S ← 0. Cycles: 1

Example:

```
add r2,r3          ;Add r3 to r2
cls                ;Clear Signed flag
```

CLT ; **Clear T Flag**
; **T ← 0**

Clears the T flag in SREG (Status Register).

Flags: T ← 0. Cycles: 1

Example:

```
clt                ;Clear T flag
```

CLV ; **Clear Overflow Flag**
; **V ← 0**

Clears the Overflow flag (V) in SREG (Status Register).
Flags: V ← 0. Cycles: 1

Example:

```
        add r2,r3       ;Add r3 to r2
        clv             ;Clear Overflow flag
```

CLZ ; **Clear Zero Flag**
; **Z ← 0**

Clears the Zero flag (Z) in SREG (Status Register).
Flags: Z ← 0. Cycles: 1

Example:

```
        clz             ;Clear zero
```

COM Rd ; **One's Complement**
0 ≤ d ≤ 31 ; **Rd ← $FF – Rd**

This instruction performs a one's complement of register Rd.
Flags: S, V ← 0, N , Z ← 1, C. Cycles: 1

Example:

```
        com r4          ;Take one's complement of r4
        breq zero       ;Branch if zero
        ...
zero:   nop             ;Branch destination (do nothing)
```

CP Rd,Rr ; **Compare**
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 ; **Rd – Rr**

This instruction performs a compare between two registers, Rd and Rr. None of the registers are changed. All conditional branches can be used after this instruction.
Flags: H, S,V, N, Z, C. Cycles: 1

Example:

```
        cp r4,r19       ;Compare r4 with r19
        brne noteq      ;Branch if r4 not equal r19
        ...
noteq:  nop             ;Branch destination (do nothing)
```

CPC Rd,Rr ; **Compare with Carry**
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 ; **Rd – Rr – C**

This instruction performs a compare between two registers, Rd and Rr, and also takes into account the previous carry. None of the registers are changed. All conditional branches can be used after this instruction.
Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
                        ;Compare r3:r2 with r1:r0
        cp r2,r0        ;Compare low byte
        cpc r3,r1       ;Compare high byte
        brne noteq      ;Branch if not equal
        ...
noteq:  nop             ;Branch destination (do nothing)
```

CPI Rd,K ; **Compare with Immediate**
$16 \le d \le 31, 0 \le K \le 255$; **Rd – K**

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

Flags: H, S,V, N, Z, C. Cycles: 1

Example:

```
        cpi r19,3       ;Compare r19 with 3
        brne error      ;Branch if r19 not equal 3
        ...
error:  nop             ;Branch destination (do nothing)
```

CPSE Rd,Rr ; **Compare Skip if Equal**
$0 \le d \le 31, 0 \le r \le 31$; **If Rd = Rr then PC ← PC + 2 or 3 else PC ← PC + 1**

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

Flags:--- Cycles: 1, 2, or 3

Example:

```
        inc r4          ;Increment r4
        cpse r4,r0      ;Compare r4 to r0
        neg r4          ;Only executed if r4 not equal r0
        nop             ;Continue (do nothing)
```

DEC Rd ; **Decrement**
$0 \le d \le 31$; **Rd ← Rd – 1**

Subtracts one from the contents of register Rd and places the result in the destination register Rd.

The C flag in SREG is not affected by the operation, thus allowing the DEC instruction to be used on a loop counter in multiple-precision computations.

When operating on unsigned values, only BREQ and BRNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

Flags: S,V, N, Z. Cycles: 1

Example:

```
        ldi r17,$10     ;Load constant in r17
loop:   add r1,r2       ;Add r2 to r1
        dec r17         ;Decrement r17
        brne loop       ;Branch if r17 not equal 0
        nop             ;Continue (do nothing)
```

EOR Rd,Rr ; **Exclusive OR**
$0 \le d \le 31, 0 \le r \le 31$; **Rd ← Rd ⊕ Rr**

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

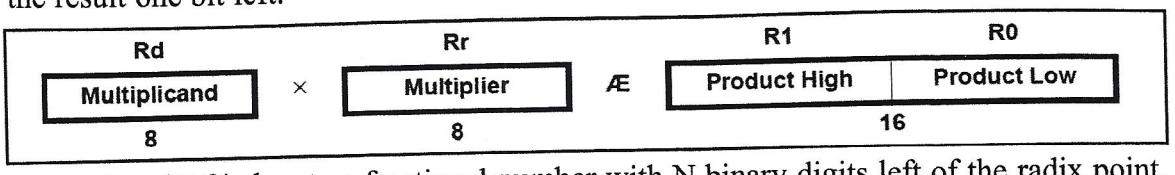
Flags: S, V, Z ← 0, N, Z. Cycles: 1

Example:

```
        eor r4,r4       ;Clear r4
        eor r0,r22      ;Bitwise XOR between r0 and r22
```

FMUL Rd,Rr ; **Fractional Multiply Unsigned**
16 ≤ d ≤ 23, 16 ≤ r ≤ 23 ; **R1:R0 ← Rd × Rr (unsigned ← unsigned × unsigned)**

This instruction performs 8-bit × 8-bit → 16-bit unsigned multiplication and shifts the result one bit left.



Let (N.Q) denote a fractional number with N binary digits left of the radix point, and Q binary digits right of the radix point. A multiplication between two numbers in the formats (N1.Q1) and (N2.Q2) results in the format ((N1 + N2).(Q1 + Q2)). For signal processing applications, the (1.7) format is widely used for the inputs, resulting in a (2.14) format for the product. A left shift is required for the high byte of the product to be in the same format as the inputs. The FMUL instruction incorporates the shift operation in the same number of cycles as MUL.

The (1.7) format is most commonly used with signed numbers, while FMUL performs an unsigned multiplication. This instruction is therefore most useful for calculating one of the partial products when performing a signed multiplication with 16-bit inputs in the (1.15) format, yielding a result in the (1.31) format. (Note: The result of the FMUL operation may suffer from a 2's complement overflow if interpreted as a number in the (1.15) format.) The MSB of the multiplication before shifting must be taken into account, and is found in the carry bit. See the following example.

The multiplicand Rd and the multiplier Rr are two registers containing unsigned fractional numbers where the implicit radix point lies between bit 6 and bit 7. The 16-bit unsigned fractional product with the implicit radix point between bit 14 and bit 15 is placed in R1 (high byte) and R0 (low byte).

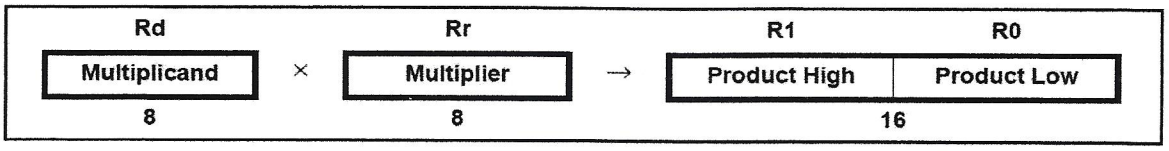
Flags: Z, C. Cycles: 2

Example:

```
;*****************************************************************
;* DESCRIPTION
;* Signed fractional multiply of two 16-bit numbers with 32-bit result.
;* r19:r18:r17:r16 = ( r23:r22 * r21:r20 ) << 1
;*****************************************************************
            fmuls 16x16_32:
            clr r2
            fmuls r23, r21          ;((signed)ah *(signed)bh) << 1
            movw r19:r18, r1:r0
            fmul r22, r20           ;(al * bl) << 1
            adc r18, r2
            movwr17:r16, r1:r0
            fmulsu r23, r20         ;((signed)ah * bl) << 1
            sbc r19, r2
            add r17, r0
            adc r18, r1
            adc r19, r2
            fmulsu r21, r22         ;((signed)bh * al) << 1
            sbc r19, r2
            add r17, r0
            adc r18, r1
            adc r19, r2
```

FMULS Rd,Rr ; **Fractional Multiply Signed**
$16 \le d \le 23$, $16 \le r \le 23$; **R1:R0 ← Rd × Rr (signed ← signed × signed)**

This instruction performs 8-bit × 8-bit → 16-bit signed multiplication and shifts the result one bit left.



Let (N.Q) denote a fractional number with N binary digits left of the radix point, and Q binary digits right of the radix point. A multiplication between two numbers in the formats (N1.Q1) and (N2.Q2) results in the format ((N1 + N2).(Q1 + Q2)). For signal processing applications, the (1.7) format is widely used for the inputs, resulting in a (2.14) format for the product. A left shift is required for the high byte of the product to be in the same format as the inputs. The FMULS instruction incorporates the shift operation in the same number of cycles as MULS.

The multiplicand Rd and the multiplier Rr are two registers containing signed fractional numbers where the implicit radix point lies between bit 6 and bit 7. The 16-bit signed fractional product with the implicit radix point between bit 14 and bit 15 is placed in R1 (high byte) and R0 (low byte).

Note that when multiplying 0x80 (–1) with 0x80 (–1), the result of the shift operation is 0x8000 (–1). The shift operation thus gives a two's complement overflow. This must be checked and handled by software.

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

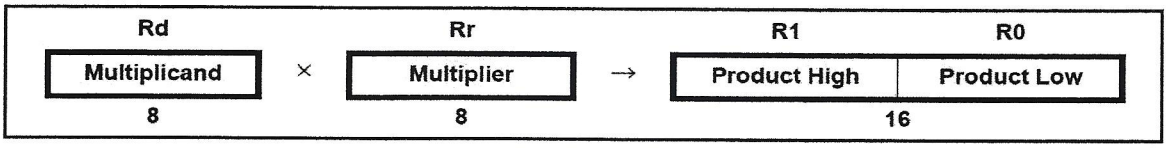
Flags: Z, C. Cycles: 2

Example:

```
fmuls r23,r22          ;Multiply signed r23 and r22 in
                       ;(1.7) format, result in (1.15) format
movw r23:r22,r1:r0     ;Copy result back in r23:r22
```

FMULSU Rd,Rr ; **Fractional Multiply Signed with Unsigned**
$16 \le d \le 23$, $16 \le r \le 23$; **R1:R0 ← Rd × Rr**

This instruction performs 8-bit × 8-bit → 16-bit signed multiplication and shifts the result one bit left.



Let (N.Q) denote a fractional number with N binary digits left of the radix point, and Q binary digits right of the radix point. A multiplication between two numbers in the formats (N1.Q1) and (N2.Q2) results in the format ((N1 + N2).(Q1 + Q2)). For signal processing applications, the (1.7) format is widely used for the inputs, resulting in a (2.14) format for the product. A left shift is required for the high byte of the product to be in the same format as the inputs. The FMULSU instruction incorporates the shift operation in the same number of cycles as MULSU.

The (1.7) format is most commonly used with signed numbers, while FMULSU

performs a multiplication with one unsigned and one signed input. This instruction is therefore most useful for calculating two of the partial products when performing a signed multiplication with 16-bit inputs in the (1.15) format, yielding a result in the (1.31) format. (Note: The result of the FMULSU operation may suffer from a 2's complement overflow if interpreted as a number in the (1.15) format.) The MSB of the multiplication before shifting must be taken into account, and is found in the carry bit. See the following example.

The multiplicand Rd and the multiplier Rr are two registers containing fractional numbers where the implicit radix point lies between bit 6 and bit 7. The multiplicand Rd is a signed fractional number, and the multiplier Rr is an unsigned fractional number. The 16-bit signed fractional product with the implicit radix point between bit 14 and bit 15 is placed in R1 (high byte) and R0 (low byte).

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

Flags: Z, C. Cycles: 2

Example:

```
;*****************************************************************
;* DESCRIPTION
;* Signed fractional multiply of two 16-bit numbers with 32-bit result.
;* r19:r18:r17:r16 = ( r23:r22 * r21:r20 ) << 1
;*****************************************************************
fmuls16x16_32:
            clrr2
            fmuls r23, r21     ;((signed)ah * (signed)bh) << 1
            movwr19:r18, r1:r0
            fmul r22, r20      ;(al * bl) << 1
            adc r18, r2
            movwr17:r16, r1:r0
            fmulsu r 23, r20   ;((signed)ah * bl) << 1
            sbc r19, r2
            add r17, r0
            adc r18, r1
            adc r19, r2
            fmulsu r21, r22    ;((signed)bh * al) << 1
            sbc r19, r2
            add r17, r0
            adc r18, r1
            adc r19, r2
```

ICALL ; Indirect Call to Subroutine

Indirect call of a subroutine pointed to by the Z (16 bits) pointer register in the register file. The Z-pointer register is 16 bits wide and allows calls to a subroutine within the lowest 64K words (128K bytes) section in the program memory space. The stack pointer uses a post-decrement scheme during ICALL.

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

Flags: --- Cycles: 3

Example:

```
mov r30,r0          ;Set offset to call table
icall               ;Call routine pointed to by r31:r30
```

IJMP **; Indirect Jump**

Indirect jump to the address pointed to by the Z (16 bits) pointer register in the register file. The Z-pointer register is 16 bits wide and allows jumps within the lowest 64K words (128K bytes) of the program memory.

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

Flags:--- Cycles: 2

Example:

```
        mov r30,r0      ;Set offset to jump table
        ijmp            ;Jump to routine pointed to by r31:r30
```

IN Rd,A **; Load an I/O Location to Register**

$0 \le d \le 31$, $0 \le A \le 63$ **; Rd ← I/O(A)**

Loads data from the I/O space (ports, timers, configuration registers, etc.) into register Rd in the register file.

Flags:--- Cycles: 1

Example:

```
        in r25,$16      ;Read Port B
        cpi r25,4       ;Compare read value to constant
        breq exit       ;Branch if r25=4
        ...
exit:   nop             ;Branch destination (do nothing)
```

INC Rd **; Increment**

$0 \le d \le 31$ **; Rd ← Rd + 1**

Adds one to the contents of register Rd and places the result in the destination register Rd.

The C flag in SREG is not affected by the operation, thus allowing the INC instruction to be used on a loop counter in multiple-precision computations.

When operating on unsigned numbers, only BREQ and BRNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

Flags: S, V, N, Z. Cycles: 1

Example:

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

JMP k **; Jump**

$0 \le k < 4M$ **; PC ← k**

Jump to an address within the entire 4M (words) program memory. See also RJMP.

Flags:--- Cycles: 3

Example:

```
        mov r1,r0          ;Copy r0 to r1
        jmp farplc         ;Unconditional jump
        ...
farplc: nop                ;Jump destination (do nothing)
```

LD **; Load Indirect from Data Space to Register ; using Index X**

Loads one byte indirect from the data space to a register. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the X (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPX in register in the I/O area has to be changed.

The X-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented.

These features are especially suited for accessing arrays, tables, and stack pointer usage of the X-pointer register. Note that only the low byte of the X-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPX register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/ decrement is added to the entire 24-bit address on such devices.

Syntax:	Operation:	Comment:
(i) LD Rd, X	Rd ← (X)	X: Unchanged
(ii) LD Rd, X+	Rd ← (X) , X ← X + 1	X: Post-incremented
(iii) LD Rd, –X	X ← X – 1, Rd ← (X)	X: Pre-decremented

Flags:--- Cycles: 2

Example:

```
        clr r27            ;Clear X high byte
        ldi r26,$60        ;Set X low byte to $60
        ld r0,X+           ;Load r0 with data space loc. $60
                           ;X post inc)
        ld r1,X            ;Load r1 with data space loc. $61
        ldi r26,$63        ;Set X low byte to $63
        ld r2,X            ;Load r2 with data space loc. $63
        ld r3,-X           ;Load r3 with data space loc.
                           ;$62(X pre dec)
```

LD (LDD) **; Load Indirect from Data Space to Register ; using Index Y**

Loads one byte indirect with or without displacement from the data space to a register. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the Y (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPY in register in the I/O area has to be changed.

The Y-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented. These features are especially suited for accessing arrays, tables, and stack pointer usage of the Y-pointer register. Note that only the low byte of the Y-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPY register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/ decrement/displacement is added to the entire 24-bit address on such devices.

Syntax:	Operation:	Comment:
(i) LD Rd, Y	Rd ← (Y)	Y: Unchanged
(ii) LD Rd, Y+	Rd ← (Y) ,Y ← Y + 1	Y: Postincremented
(iii) LD Rd, –Y	Y ← Y – 1, Rd ← (Y)	Y: Predecremented
(iiii) LDD Rd, Y + q	Rd ← (Y + q)	Y: Unchanged, q: Displacement

Flags:--- Cycles: 2

Example:

```
clr r29          ;Clear Y high byte
ldi r28,$60      ;Set Y low byte to $60
ld r0,Y+         ;Load r0 with data space loc. $60(Y post inc)
ld r1,Y          ;Load r1 with data space loc. $61
ldi r28,$63      ;Set Y low byte to $63
ld r2,Y          ;Load r2 with data space loc. $63
ld r3,-Y         ;Load r3 with data space loc. $62(Y pre dec)
ldd r4,Y+2       ;Load r4 with data space loc. $64
```

LD (LDD) **; Load Indirect from Data Space to Register ; using Index Z**

Loads one byte indirect with or without displacement from the data space to a register. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the Z (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPZ in register in the I/O area has to be changed.

The Z-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented. These features are especially suited for stack pointer usage of the Z-pointer register, however because the Z-pointer register can be used for indirect subroutine calls, indirect jumps, and table lookup, it is often more convenient to use the X or Y-pointer as a dedicated stack pointer. Note that only the low byte of the Z-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPZ register in the I/O area is updated in parts with more than 64K bytes

data space or more than 64K bytes program memory, and the increment/decrement/displacement is added to the entire 24-bit address on such devices.

Syntax:	Operation:	Comment:
(i) LD Rd, Z	Rd ← (Z)	Z: Unchanged
(ii) LD Rd, Z+	Rd ← (Z) Z ← Z + 1	Z: Postincrement
(iii) LD Rd, –Z	Z ← Z – 1 Rd ← (Z)	Z: Predecrement
(iiii) LDD Rd, Z + q	Rd ← (Z + q)	Z: Unchanged, q: Displacement

Flags:--- Cycles: 2

Example:

```
clr r31          ;Clear Z high byte
ldi r30,$60      ;Set Z low byte to $60
ld r0,Z+         ;Load r0 with data space loc.$60(Z postinc.)
ld r1,Z          ;Load r1 with data space loc. $61
ldi r30,$63      ;Set Z low byte to $63
ld r2,Z          ;Load r2 with data space loc. $63
ld r3,-Z         ;Load r3 with data space loc. $62(Z predec.)
ldd r4,Z+2       ;Load r4 with data space loc. $64
```

LDI Rd,K ; **Load Immediate**

16 ≤ d ≤ 31, 0 ≤ K ≤ 255 ; **Rd ← K**

Loads an 8-bit constant directly to registers 16 to 31.

Flags:--- Cycles: 1

Example:

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

LDS Rd,k ; **Load Direct from Data Space**

0 ≤ d ≤ 31, 0 ≤ k ≤ 65535 ; **Rd ← (k)**

Loads one byte from the data space to a register. The data space consists of the register file, I/O memory, and SRAM.

Flags:--- Cycles: 2

Example:

```
lds r2,$FF00     ;Load r2 with the contents of
                 ;data space location $FF00
add r2,r1        ;add r1 to r2
sts $FF00,r2     ;Write back
```

LPM ; **Load Program Memory**

Loads one byte pointed to by the Z-register into the destination register Rd. This instruction features a 100% space effective constant initialization or constant data fetch. The program memory is organized in 16-bit words while the Z-pointer is a byte address. Thus, the least significant bit of the Z-pointer selects either the low byte (ZLSB = 0) or the high byte (ZLSB = 1). This instruction can address the first 64K bytes (32K words) of

program memory. The Z-pointer register can either be left unchanged by the operation, or it can be incremented. The incrementation does not apply to the RAMPZ register.

Devices with self-programming capability can use the LPM instruction to read the Fuse and Lock bit values. Refer to the device documentation for a detailed description.

Syntax:	Operation:	Comment:
(i) LPM	R0 ← (Z)	Z: Unchanged, R0 implied Rd
(ii) LPM Rd, Z	Rd ← (Z)	Z: Unchanged
(iii) LPM Rd, Z+	Rd ← (Z), Z ← Z + 1	Z: Postincremented

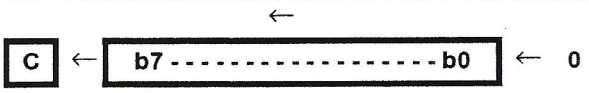
Flags:--- Cycles: 3

Example:

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

LSL Rd **; Logical Shift Left**
$0 \le d \le 31$

Shifts all bits in Rd one place to the left. Bit 0 is cleared. Bit 7 is loaded into the C flag of the SREG (Status Register). This operation effectively multiplies signed and unsigned values by two.



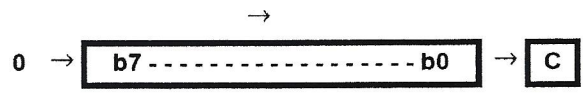
Flags: H, S, V, N, Z, C. Cycles: 1

Example:

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

LSR Rd **; Logical Shift Left**
$0 \le d \le 31$

Shifts all bits in Rd one place to the right. Bit 7 is cleared. Bit 0 is loaded into the C flag of the SREG. This operation effectively divides an unsigned value by two. The C flag can be used to round the result.



Flags: S, V, N ← 0, Z, C. Cycles: 1

Example:

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

MOV Rd,Rr **; Copy Register**
$0 \le d \le 31, 0 \le r \le 31$ **; Rd ← Rr**

This instruction makes a copy of one register into another. The source register Rr is left unchanged, while the destination register Rd is loaded with a copy of Rr.

Flags: --- Cycles: 1

Example:

```
        mov r16,r0          ;Copy r0 to r16
        call check          ;Call subroutine
        ...
check:  cpi r16,$11         ;Compare r16 to $11
        ...
        ret                 ;Return from subroutine
```

MOVW Rd + 1:Rd,Rr + 1:Rrd ; **Copy RegisterWord**
d ∈ {0,2,...,30}, r ∈ {0,2,...,30} ; **Rd + 1:Rd ← Rr + 1:Rr**

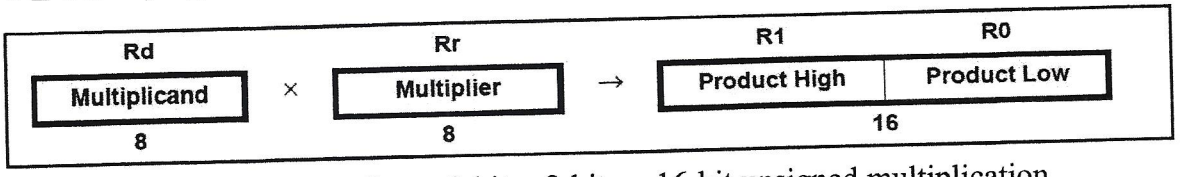
This instruction makes a copy of one register pair into another register pair. The source register pair Rr + 1:Rr is left unchanged, while the destination register pair Rd + 1:Rd is loaded with a copy of Rr + 1:Rr.

Flags: --- Cycles: 1

Example:

```
        movw r17:16,r1:r0   ;Copy r1:r0 to r17:r16
        call check          ;Call subroutine
        ...
check:  cpi r16,$11         ;Compare r16 to $11
        ...
        cpi r17,$32         ;Compare r17 to $32
        ...
        ret                 ;Return from subroutine
```

MUL Rd,Rr ; **Multiply Unsigned**
0 ≤ d ≤ 31, 0 ≤ r ≤ 31 ; **R1:R0 ← Rd × Rr(unsigned ← unsigned × unsigned)**



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

The multiplicand Rd and the multiplier Rr are two registers containing unsigned numbers. The 16-bit unsigned product is placed in R1 (high byte) and R0 (low byte). Note that if the multiplicand or the multiplier is selected from R0 or R1 the result will overwrite those after multiplication.

Flags: Z, C. Cycles: 2

Example:

```
        mul r5,r4           ;Multiply unsigned r5 and r4
        movw r4,r0          ;Copy result back in r5:r4
```

MULS Rd,Rr ; **Multiply Signed**
16 ≤ d ≤ 31, 16 ≤ r ≤ 31 ; **R1:R0 ← Rd × Rr(signed ← signed × signed)**

This instruction performs 8-bit × 8-bit → 16-bit signed multiplication.

The multiplicand Rd and the multiplier Rr are two registers containing signed numbers. The 16-bit signed product is placed in R1 (high byte) and R0 (low byte).

Flags: Z, C. Cycles: 2

Example:

```
        muls r21,r20        ;Multiply signed r21 and r20
        movw r20,r0         ;Copy result back in r21:r20
```

MULSU Rd,Rr ; **Multiply Signed with Unsigned**
$16 \le d \le 31$, $16 \le r \le 31$; **R1:R0 ← Rd × Rr (signed ← signed × unsigned)**

This instruction performs 8-bit × 8-bit → 16-bit multiplication of a signed and an unsigned number.

The multiplicand Rd and the multiplier Rr are two registers. The multiplicand Rd is a signed number, and the multiplier Rr is unsigned. The 16-bit signed product is placed in R1 (high byte) and R0 (low byte).

Flags: Z, C. Cycles: 2

Example:---

NEG Rd ; **Two's Complement**
$0 \le d \le 31$; **Rd ← $00 – Rd**

Replaces the contents of register Rd with its two's complement; the value $80 is left unchanged.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
            sub r11,r0          ;Subtract r0 from r11
            brpl positive       ;Branch if result positive
            neg r11             ;Take two's complement of r11
positive:   nop                 ;Branch destination (do nothing)
```

NOP ; **No Operation**

This instruction performs a single-cycle No Operation.

Flags: ---. Cycles: 1

Example:

```
            clr r16             ;Clear r16
            ser r17             ;Set r17
            out $18,r16         ;Write zeros to Port B
            nop                 ;Wait (do nothing)
            out $18,r17         ;Write ones to Port B
```

OR Rd,Rr ; **Logical OR**
$0 \le d \le 31$, $0 \le r \le 31$; **Rd ← Rd OR Rr**

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

Flags: S, V ← 0, N, Z. Cycles: 1

Example:

```
            or r15,r16          ;Do bitwise or between registers
            bst r15,6           ;Store bit 6 of r15 in T flag
            brts ok             ;Branch if T flag set
            ...
ok:         nop                 ;Branch destination (do nothing)
```

ORI Rd,K ; **Logical OR with Immediate**
$16 \leq d \leq 31, 0 \leq K \leq 255$; **Rd ← Rd OR K**

Performs the logical OR between the contents of register Rd and a constant and places the result in the destination register Rd.

Flags: S, V ← 0, N, Z. Cycles: 1

Example:

```
ori r16,$F0        ;Set high nibble of r16
ori r17,1          ;Set bit 0 of r17
```

OUT A,Rr ; **Store Register to I/O Location**
$0 \leq r \leq 31, 0 \leq A \leq 63$; **I/O(A) ← Rr**

Stores data from register Rr in the register file to I/O space (ports, timers, configuration registers, etc.).

Flags: ---. Cycles: 1

Example:

```
clr r16            ;Clear r16
ser r17            ;Set r17
out $18,r16        ;Write zeros to Port B
nop                ;Wait (do nothing)
out $18,r17        ;Write ones to Port B
```

POP Rd ; **Pop Register from Stack**
$0 \leq d \leq 31$; **Rd ← STACK**

This instruction loads register Rd with a byte from the STACK. The stack pointer is pre-incremented by 1 before the POP.

Flags: ---. Cycles: 2

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
```

PUSH Rr ; **Push Register on Stack**
$0 \leq d \leq 31$; **STACK ← Rr**

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

Flags: ---. Cycles: 2

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
```

RCALL k ; **Relative Call to Subroutine**
$-2K \leq k < 2K$; **PC ← PC + k + 1**

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.

Flags: ---. Cycles: 3

Example:

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

RET ; **Return from Subroutine**

Returns from subroutine. The return address is loaded from the stack. The stack pointer uses a pre-increment scheme during RET.

Flags: ---. Cycles: 4

Example:

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

RETI ; **Return from Interrupt**

Returns from interrupt. The return address is loaded from the stack and the Global Interrupt flag is set.

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

Flags: ---. Cycles: 4

Example:

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

RJMP k ; **Relative Jump**
$-2K \le k < 2K$; **PC ← PC + k + 1**

Relative jump to an address within PC – 2K +1 and PC + 2K (words). 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.

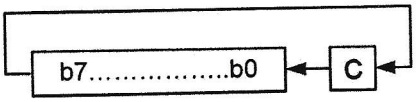
Flags: ---. Cycles: 2

Example:

```
            cpi r16,$42       ;Compare r16 to $42
            brne error        ;Branch if r16 not equal $42
            rjmp ok           ;Unconditional branch
error:      add r16,r17       ;Add r17 to r16
            inc r16           ;Increment r16
ok:         nop               ;Destination for rjmp (do nothing)
```

ROL Rd ; **Rotate Left through Carry**
$0 \le d \le 31$

Shifts all bits in Rd one place to the left. The C flag is shifted into bit 0 of Rd. Bit 7 is shifted into the C flag. This operation combined with LSL effectively multiplies multibyte signed and unsigned values by two.



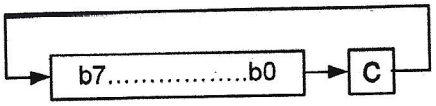
Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
            lsl r18           ;Multiply r19:r18 by two
            rol r19           ;r19:r18 is a signed or unsigned word
            brcs oneenc       ;Branch if carry set
            ...
oneenc:     nop               ;Branch destination (do nothing)
```

ROR Rd ; **Rotate Right through Carry**
$0 \le d \le 31$

Shifts all bits in Rd one place to the right. The C flag is shifted into bit 7 of Rd. Bit 0 is shifted into the C flag. This operation combined with ASR effectively divides multibyte signed values by two. Combined with LSR, it effectively divides multibyte unsigned values by two. The Carry flag can be used to round the result.



Flags: S, V, N, Z, C. Cycles: 1

Example:

```
            lsr r19           ;Divide r19:r18 by two
            ror r18           ;r19:r18 is an unsigned two-byte integer
            brcc zeroenc1     ;Branch if carry cleared
            asr r17           ;Divide r17:r16 by two
            ror r16           ;r17:r16 is a signed two-byte integer
            brcc zeroenc2     ;Branch if carry cleared
            ...
zeroenc1:   nop               ;Branch destination (do nothing)
            ...
zeroenc2:   nop               ;Branch destination (do nothing)
```

SBC Rd,Rr ; **Subtract with Carry**
$0 \le d \le 31, 0 \le r \le 31$; **Rd ← Rd – Rr – C**

Subtracts two registers and subtracts with the C flag and places the result in the destination register Rd.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:

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

SBCI Rd,K ; **Subtract Immediate with Carry**
$0 \le d \le 31, 0 \le r \le 31$; **Rd ← Rd – K – C**

Subtracts a constant from a register and subtracts with the C flag and places the result in the destination register Rd.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
                        ;Subtract $4F23 from r17:r16
        subi r16,$23    ;Subtract low byte
        sbci r17,$4F    ;Subtract with carry high byte
```

SBI A,b ; **Set Bit in I/O Register**
$0 \le A \le 31, 0 \le b \le 7$; **I/O(A,b) ← 1**

Sets a specified bit in an I/O register. This instruction operates on the lower 32 I/O registers.

Flags: ---. Cycles: 2

Example:

```
        out $1E,r0      ;Write EEPROM address
        sbi $1C,0       ;Set read bit in EECR
        in r1,$1D       ;Read EEPROM data
```

SBIC A,b ; **Skip if Bit in I/O Register is Cleared**
$0 \le d \le 31, 0 \le r \le 31$; **If I/O(A,b) = 0 then PC ← PC + 2 (or 3) else PC ← PC + 1**

This instruction tests a single bit in an I/O register and skips the next instruction if the bit is cleared. This instruction operates on the lower 32 I/O registers.

Flags:---. Cycles: 1/2/3

Example:

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

SBIS A,b ; **Skip if Bit in I/O Register is Set**
$0 \le d \le 31, 0 \le r \le 31$; **If I/O(A,b) = 1 then PC ← PC + 2 (or 3) else PC ← PC + 1**

This instruction tests a single bit in an I/O register and skips the next instruction if the bit is set. This instruction operates on the lower 32 I/O registers.

Flags: ---. Cycles: 1/2/3

Example:

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

SBIW Rd + 1:Rd,K ; **Subtract Immediate from Word**
d ∈ {24,26,28,30}, 0 ≤ K ≤ 63 ; **Rd + 1:Rd ← Rd + 1:Rd – K**

Subtracts an immediate value (0–63) from a register pair and places the result in the register pair. This instruction operates on the upper four register pairs, and is well suited for operations on the pointer registers.

Flags: S, V, N, Z, C. Cycles: 2

Example:

```
sbiw r25:r24,1      ;Subtract 1 from r25:r24
sbiw YH:YL,63       ;Subtract 63 from the Y-pointer
```

SBR Rd,K ; **Set Bits in Register**
16 ≤ d ≤ 31, 0 ≤ K ≤ 255 ; **Rd ← Rd OR K**

Sets specified bits in register Rd. Performs the logical ORI between the contents of register Rd and a constant mask K and places the result in the destination register Rd.

Flags: S,V←0, N, Z. Cycles: 1

Example:

```
sbr r16,3           ;Set bits 0 and 1 in r16
sbr r17,$F0         ;Set 4 MSB in r17
```

SBRC Rr,b ; **Skip if Bit in Register is Cleared**
0 ≤ r ≤ 31, 0 ≤ b ≤7 ; **If Rr(b) = 0 then PC ← PC + 2 or 3 else PC ← PC + 1**

This instruction tests a single bit in an I/O register and skips the next instruction if the bit is set. This instruction operates on the lower 32 I/O registers.

Flags: --- Cycles: 1/2/3

Example:

```
sub r0,r1           ;Subtract r1 from r0
sbrc r0,7           ;Skip if bit 7 in r0 cleared
sub r0,r1           ;Only executed if bit7 in r0 not cleared
nop                 ;Continue (do nothing)
```

SBRS Rr,b ; **Skip if Bit in Register is Set**
0 ≤ r ≤ 31, 0 ≤ b ≤7 ; **If Rr(b) = 1 then PC ← PC + 2 or 3 else PC ← PC + 1**

This instruction tests a single bit in a register and skips the next instruction if the bit is set.

Flags: H, S, V, N, Z, C. Cycles: 1/2/3

Example:

```
sub r0,r1           ;Subtract r1 from r0
sbrs r0,7           ;Skip if bit 7 in r0 set
neg r0              ;Only executed if bit 7 in r0 not set
nop                 ;Continue (do nothing)
```

SEC ; **Set Carry Flag**
; **C ← 1**

Sets the Carry flag (C) in SREG (Status Register).

Flags: C ← 1. Cycles: 1

Example:

```
sec                 ;Set Carry flag
adc r0,r1           ;r0=r0+r1+1
```

SEH ; **Set Half Carry Flag**
; **H ← 1**

Sets the Half Carry (H) in SREG (Status Register).
Flags: H ← 1. Cycles: 1
Example:

```
seh                ;Set Half Carry flag
```

SEI ; **Set Global Interrupt Flag**
; **I ← 1**

Sets the Global Interrupt flag (I) in SREG (Status Register). The instruction following SEI will be executed before any pending interrupts.
Flags: I ← 1. Cycles: 1
Example:

```
sei                ;Set global interrupt enable
sec                ;Set Carry flag
;Note: will set Carry flag before any pending interrupt
```

SEN ; **Set Negative Flag**
; **N ← 1**

Sets the Negative flag (N) in SREG (Status Register).
Flags: N ← 1. Cycles: 1
Example:

```
add r2,r19         ;Add r19 to r2
sen                ;Set Negative flag
```

SER Rd ; **Set all Bits in Register**
16 ≤ d ≤ 31 ; **Rd ← $FF**

Loads $FF directly to register Rd.
Flags: ---. Cycles: 1
Example:

```
ser r17            ;Set r17
out $18,r17        ;Write ones to Port B
```

SES ; **Set Signed Flag**
; **S ← 1**

Sets the Signed flag (S) in SREG (Status Register).
Flags: S ← 1. Cycles: 1
Example:

```
add r2,r19         ;Add r19 to r2
ses                ;Set Negative flag
```

SET ; **Set T Flag**
; **T ← 1**

Sets the T flag in SREG (Status Register).
Flags: T ← 1. Cycles: 1
Example:

```
set                ;Set T flag
```

SEV ; **Set Overflow Flag**
; **V ← 1**

Sets the Overflow flag (V) in SREG (Status Register).
Flags: V ← 1. Cycles: 1

Example:

```
sev                    ;Set Overflow flag
```

SEZ ; **Set Zero Flag**
; **Z ← 1**

Sets the Zero flag (Z) in SREG (Status Register).
Flags: Z ← 1. Cycles: 1

Example:

```
sez                    ;Set Z flag
```

SLEEP

This instruction sets the circuit in sleep mode defined by the MCU control register.
Flags: ---. Cycles: 1

Example:

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

SPM ; **Store Program Memory**

SPM can be used to erase a page in the program memory, to write a page in the program memory (that is already erased), and to set Boot Loader Lock bits. In some devices, the program memory can be written one word at a time, in other devices an entire page can be programmed simultaneously after first filling a temporary page buffer. In all cases, the program memory must be erased one page at a time. When erasing the program memory, the RAMPZ and Z-register are used as page address. When writing the program memory, the RAMPZ and Z-register are used as page or word address, and the R1:R0 register pair is used as data(1). When setting the Boot Loader Lock bits, the R1:R0 register pair is used as data.

Refer to the device documentation for detailed description of SPM usage. This instruction can address the entire program memory.

Flags: ---. Cycles: depends on the operation

	Syntax:	Operation:	Comment:
(i)	SPM	(RAMPZ:Z) ← $ffff	Erase program memory page
(ii)	SPM	(RAMPZ:Z) ← R1:R0	Write program memory word
(iii)	SPM	(RAMPZ:Z) ← R1:R0	Write temporary page buffer
(iv)	SPM	(RAMPZ:Z) ← TEMP	Write temporary page buffer to program memory
(v)	SPM	BLBITS ← R1:R0	Set Boot Loader Lock bits

ST ; Store Indirect From Register to Data Space ; using Index X

Stores one byte indirect from a register to data space. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the X (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPX register in the I/O area has to be changed.

The X-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented.These features are especially suited for accessing arrays, tables, and stack pointer usage of the X-pointer register. Note that only the low byte of the X-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPX register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/ decrement is added to the entire 24-bit address on such devices.

Flags: ---. Cycles: 2

	Syntax:	Operation:	Comment:
(i)	ST X, Rr	(X) ← Rr	X: Unchanged
(ii)	ST X+, Rr	(X) ← Rr X ← X + 1	X: Postincremented
(iii)	ST –X, Rr	X ← X – 1 (X) ← Rr	X: Predecremented

Example:

```
clr r27          ;Clear X high byte
ldi r26,$60      ;Set X low byte to $60
st X+,r0         ;Store r0 in data space loc. $60(X post inc)
st X,r1          ;Store r1 in data space loc. $61
ldi r26,$63      ;Set X low byte to $63
st X,r2          ;Store r2 in data space loc. $63
st -X,r3         ;Store r3 in data space loc. $62(X pre dec)
```

ST (STD) ; Store Indirect From Register to Data Space ; using Index Y

Stores one byte indirect with or without displacement from a register to data space. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the Y (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPY register in the I/O area has to be changed.

The Y-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented. These features are especially suited for accessing

arrays, tables, and stack pointer usage of the Y-pointer register. Note that only the low byte of the Y-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPY register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/ decrement/displacement is added to the entire 24-bit address on such devices.

Flags: ---. Cycles:2

	Syntax:	Operation:	Comment:
(i)	ST Y, Rr	(Y) ← Rr	Y: Unchanged
(ii)	ST Y+, Rr	(Y) ← Rr Y ← Y + 1	Y: Postincremented
(iii)	ST –Y, Rr	Y ← Y – 1 (Y) ← Rr	Y: Predecremented
(iiii)	STD Y + q, Rr	(Y + q) ← Rr	Y: Unchanged q: Displacement

Example:

```
clr r29          ;Clear Y high byte
ldi r28,$60      ;Set Y low byte to $60
st Y+,r0         ;Store r0 in data space loc. $60 (Y postinc.)
st Y,r1          ;Store r1 in data space loc. $61
ldi r28,$63      ;Set Y low byte to $63
st Y,r2          ;Store r2 in data space loc. $63
st -Y,r3         ;Store r3 in data space loc. $62 (Y predec.)
std Y+2,r4       ;Store r4 in data space loc. $64
```

ST (STD) ; Store Indirect From Register to Data Space using Index Z

Stores one byte indirect with or without displacement from a register to data space. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

The data location is pointed to by the Z (16 bits) pointer register in the register file. Memory access is limited to the current data segment of 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPZ register in the I/O area has to be changed.

The Z-pointer register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented. These features are especially suited for stack pointer usage of the Z-pointer register; however, because the Z-pointer register can be used for indirect subroutine calls, indirect jumps and table lookup, it is often more convenient to use the X or Y-pointer as a dedicated stack pointer. Note that only the low byte of the Z-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes. The RAMPZ register in the I/O area is updated in parts with more than 64K bytes data space or more than 64K bytes program memory, and the increment/decrement/displacement is added to the entire 24-bit address on such devices.

Flags: ---. Cycles: 2

	Syntax:	Operation:	Comment:
(i)	ST Z, Rr	(Z) ←Rr	Z: Unchanged
(ii)	ST Z+, Rr	(Z) ← Rr Z ← Z + 1	Z: Postincremented
(iii)	ST –Z, Rr	Z ← Z – 1 (Z) ← Rr	Z: Predecremented
(iiii)	STD Z + q, Rr	(Z + q) ← Rr	Z: Unchanged, q: Displacement

Example:

```
clr r31          ;Clear Z high byte
ldi r30,$60      ;Set Z low byte to $60
st Z+,r0         ;Store r0 in data space loc. $60 (Z postinc.)
st Z,r1          ;Store r1 in data space loc. $61
ldi r30,$63      ;Set Z low byte to $63
st Z,r2          ;Store r2 in data space loc. $63
st -Z,r3         ;Store r3 in data space loc. $62 (Z predec.)
std Z+2,r4       ;Store r4 in data space loc. $64
```

STS k,Rr ; **Store Direct to Data Space**
$0 \le r \le 31, 0 \le k \le 65535$; **(k) ← Rr**

Stores one byte from a register to the data space. For parts with SRAM, the data space consists of the register file, I/O memory, and internal SRAM (and external SRAM if applicable). For parts without SRAM, the data space consists of the register file only. The EEPROM has a separate address space.

A 16-bit address must be supplied. Memory access is limited to the current data segment of 64K bytes. The STS instruction uses the RAMPD register to access memory above 64K bytes. To access another data segment in devices with more than 64K bytes data space, the RAMPD register in the I/O area has to be changed.

Flags:---. Cycles: 2

Example:

```
lds r2,$FF00     ;Load r2 with the contents of location $FF00
add r2,r1        ;Add r1 to r2
sts $FF00,r2     ;Write back
```

SUB Rd,Rr ; **Subtract without Carry**
$0 \le d \le 31, 0 \le r \le 31$; **Rd ← Rd – Rr**

Subtracts two registers and places the result in the destination register Rd.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
          sub r13,r12     ;Subtract r12 from r13
          brne noteq      ;Branch if r12 not equal r13
          ...
noteq:    nop             ;Branch destination (do nothing)
```

SUBI Rd,K ; **Subtract Immediate**
$16 \le d \le 31, 0 \le K \le 255$; **Rd ← Rd – K**

Subtracts a register and a constant and places the result in the destination register Rd. This instruction works on registers R16 to R31 and is very well suited for operations on the X, Y, and Z-pointers.

Flags: H, S, V, N, Z, C. Cycles: 1

Example:

```
        subi r22,$11        ;Subtract $11 from r22
        brne noteq          ;Branch if r22 not equal $11
        ...
noteq:  nop                 ;Branch destination (do nothing)
```

SWAP Rd ; **Swap Nibbles**
$0 \le d \le 31$; **R(7:4) ← Rd(3:0), R(3:0) ← Rd(7:4)**

Swaps high and low nibbles in a register.

Flags:---. Cycles: 1

Example:

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

TST Rd ; **Test for Zero or Minus**
$0 \le d \le 31$; **Rd ← Rd • Rd**

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

Flags: S, V ← 1, N, Z. Cycles: 1

Example:

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

WDR ; **Watchdog Reset**

This instruction resets the watchdog timer. This instruction must be executed within a limited time given by the WD prescaler.

Flags:---. Cycles: 1

Example:

```
        wdr                 ;Reset watchdog timer
```

SECTION A.3: AVR REGISTER SUMMARY

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$3F ($5F)	SREG	I	T	H	S	V	N	Z	C
$3E ($5E)	SPH	–	–	–	–	SP11	SP10	SP9	SP8
$3D ($5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
$3C ($5C)	OCR0	Timer/Counter0 Output Compare Register							
$3B ($5B)	GICR	INT1	INT0	INT2	–	–	–	IVSEL	IVCE
$3A ($5A)	GIFR	INTF1	INTF0	INTF2	–	–	–	–	–
$39 ($59)	TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
$38 ($58)	TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
$37 ($57)	SPMCR	SPMIE	RWWSB	–	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN
$36 ($56)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE
$35 ($55)	MCUCR	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
$34 ($54)	MCUCSR	JTD	ISC2	–	JTRF	WDRF	BORF	EXTRF	PORF
$33 ($53)	TCCR0	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
$32 ($52)	TCNT0	Timer/Counter0 (8 Bits)							
$31 ($51)	OSCCAL	Oscillator Calibration Register							
	OCDR	On-Chip Debug Register							
$30 ($50)	SFIOR	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10
$2F ($4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
$2E ($4E)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10
$2D ($4D)	TCNT1H	Timer/Counter1 – Counter Register High Byte							
$2C ($4C)	TCNT1L	Timer/Counter1 – Counter Register Low Byte							
$2B ($4B)	OCR1AH	Timer/Counter1 – Output Compare Register A High Byte							
$2A ($4A)	OCR1AL	Timer/Counter1 – Output Compare Register A Low Byte							
$29 ($49)	OCR1BH	Timer/Counter1 – Output Compare Register B High Byte							
$28 ($48)	OCR1BL	Timer/Counter1 – Output Compare Register B Low Byte							
$27 ($47)	ICR1H	Timer/Counter1 – Input Capture Register High Byte							
$26 ($46)	ICR1L	Timer/Counter1 – Input Capture Register Low Byte							
$25 ($45)	TCCR2	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
$24 ($44)	TCNT2	Timer/Counter2 (8 Bits)							
$23 ($43)	OCR2	Timer/Counter2 Output Compare Register							
$22 ($42)	ASSR	–	–	–	–	AS2	TCN2UB	OCR2UB	TCR2UB
$21 ($41)	WDTCR	–	–	–	WDTOE	WDE	WDP2	WDP1	WDP0
$20 ($40)	UBRRH	URSEL	–	–	–	UBRR[11:8]			
	UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
$1F ($3F)	EEARH	–	–	–	–	–	–	EEAR9	EEAR8
$1E ($3E)	EEARL	EEPROM Address Register Low Byte							
$1D ($3D)	EEDR	EEPROM Data Register							
$1C ($3C)	EECR	–	–	–	–	EERIE	EEMWE	EEWE	EERE
$1B ($3B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
$1A ($3A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0
$19 ($39)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
$18 ($38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
$17 ($37)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
$16 ($36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
$15 ($35)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
$14 ($34)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
$13 ($33)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
$12 ($32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
$11 ($31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
$10 ($30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
$0F ($2F)	SPDR	SPI Data Register							
$0E ($2E)	SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X
$0D ($2D)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
$0C ($2C)	UDR	USART I/O Data Register							
$0B ($2B)	UCSRA	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
$0A ($2A)	UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
$09 ($29)	UBRRL	USART Baud Rate Register Low Byte							
$08 ($28)	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
$07 ($27)	ADMUX	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
$06 ($26)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
$05 ($25)	ADCH	ADC Data Register High Byte							
$04 ($24)	ADCL	ADC Data Register Low Byte							
$03 ($23)	TWDR	Two-wire Serial Interface Data Register							
$02 ($22)	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	TWA1	TWA0	TWGCE
$01 ($21)	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	–	TWPS1	TWPS0
$00 ($20)	TWBR	Two-wire Serial Interface Bit Rate Register							