SREG: Status Register

		Bit
C	Carry flag in Status Register	0
Z	Zero flag in Status Register	1
N	Negative flag in Status Register	2
V	Two's complement overflow indicator	3
S	N $ V, for signed tests	4
H	Half carry flag in Status Register	5
T	Transfer bit used by BLD and BST instructions	6
I	Global interrupt enable/disable flag	7

Logic Operations

&: AND #: OR $: exclusive OR

Hinweis für die Immediate-Befehle

* Befehl nicht erlaubt für Register R0...R15

Registers and Operands

PC	Program Counter
Rd	Destination (and source) Working Register in the register file
Rd(n)	Bit n of the destination Working Register
Rr	Source Working Register in the register file
P	I/O-Register and Port-Register
K	Constant data
k	Constant address
b	Bit in the Working Register or I/O-Register
s	Bit in the Status Register
X,Y,Z	Indirect Address Register (X=R27:R26, Y=R29:R28, Z=R31:R30)
(X), (Y), (Z)	Contents at indirect address X, Y or Z
q	Displacement for direct addressing

DATA TRANSFER INSTRUCTIONS

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
MOV	Rd, Rr	Move Between Working Registers	Rd:=Rr	None	1	
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd := Rr+1:Rr	None	1	
LDI	Rd, K	Load Immediate *	Rd:=K	None	1	
LD	Rd, X	Load Indirect	Rd:=(X)	None	2	
LD	Rd, X+	Load Indirect and Post-increment	Rd:= (X),X:= X+1	None	2	
LD	Rd, -X	Load Indirect and Pre-decrement	X:= X-1,Rd:=(X)	None	2	
LD	Rd, Y	Load Indirect	Rd:=(Y)	None	2	
LD	Rd, Y+	Load Indirect and Post-increment	Rd:= (Y),Y:= Y+1	None	2	
LD	Rd, -Y	Load Indirect and Pre-decrement	Y:= Y-1,Rd:=(Y)	None	2	
LDD	Rd, Y+q	Load Indirect with Displacement	Rd:=(Y+q)	None	2	
LD	Rd, Z	Load Indirect	Rd:=(Z)	None	2	
LD	Rd, Z+	Load Indirect and post-increment	Rd:= (Z),Z:= Z+1	None	2	
LD	Rd, -Z	Load Indirect and pre-decrement	Z:= Z-1,Rd:=(Z)	None	2	
LDD	Rd, Z+q	Load Indirect with Displacement	Rd:=(Z+q)	None	2	
LDS	Rd, k	Load Direct from SRAM	Rd:=(k)	None	2	
ST	X, Rr	Store Indirect	(X):=Rr	None	2	
ST	X+, Rr	Store Indirect and post-increment	(X):=Rr,X:= X+1	None	2	
ST	-X, Rr	Store Indirect and pre-decrement	X:= X-1,(X):= Rr	None	2	
ST	Y, Rr	Store Indirect	(Y):=Rr	None	2	
ST	Y+, Rr	Store Indirect and post-increment	(Y):=Rr,Y:= Y+1	None	2	
ST	-Y, Rr	Store Indirect and pre-decrement	Y:= Y-1,(Y):= Rr	None	2	
STD	Y+q, Rr	Store indirect with Displacement	(Y+q):=Rr	None	2	
ST	Z, Rr	Store Indirect	(Z):=Rr	None	2	
ST	Z+, Rr	Store Indirect and post-increment	(Z):=Rr,Z:= Z+1	None	2	
ST	-Z, Rr	Store Indirect and pre-decrement	Z:= Z-1,(Z):= Rr	None	2	
STD	Z+q, Rr	Store indirect with Displacement	(Z+q):=Rr	None	2	
STS	k, Rr	Store Direct to SRAM	(k):= Rr	None	2	
LPM		Load Program Memory	R0:=(Z)	None	3	
LPM	Rd, Z	Load Program Memory	Rd := (Z)	None	3	
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd := (Z), Z := Z+1	None	3	
SPM		Store Program Memory	(Z) := R1:R0	None	-	
IN	Rd, P	In Port	Rd:=P	None	1	
OUT	P, Rr	Out Port	P:=Rr	None	1	

BIT AND BIT-TEST INSTRUCTIONS

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
SBI	P, b	Set Bit in Port-Register	I/O(P,b):= 1	None	2	
CBI	P, b	Clear Bit in Port-Register	I/O(P,b):=0	None	2	
SBR	Rd, K	Set Bit(s) in Working Register	Rd:= Rd # K	Z,N,V,S	1	
CBR	Rd, K	Clear Bit(s) in Working Register	Rd:= Rd & (0xFF-K)	Z,N,V,S	1	
LSL	Rd	Logical Shift Left	Rd(n+1):= Rd(n), Rd(0):=0, C:=Rd(7)	Z,C,N,V,H	1	
LSR	Rd	Logical Shift Right	Rd(n):= Rd(n+1), Rd(7):=0, C:=Rd(0)	Z,C,N,V	1	

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
ROL	Rd	Rotate Left through Carry	Rd(0):= C, Rd(n+1):= Rd(n), C:=Rd(7)	Z,C,N,V,H	1	
ROR	Rd	Rotate Right through Carry	Rd(7):= C, Rd(n):= Rd(n+1), C:=Rd(0)	Z,C,N,V	1	
ASR	Rd	Arithmetic Shift Right	Rd(n):=Rd(n+1), n=0...6	Z,C,N,V	1	
SWAP	Rd	Swap Nibbles	Rd(3...0):=Rd(7...4), Rd(7...4):=Rd(3...0)	None	1	
BSET	S	Flag Set	SREG(s):=1	SREG(s)	1	
BCLR	S	Flag Clear	SREG(s):=0	SREG(s)	1	
BST	Rr, b	Bit Store from Work Register to T	T:=Rr(b)	T	1	
BLD	Rd, b	Bit Load from T to Work Register	Rd(b):=T	None	1	
SEC		Set Carry	C:=1	C	1	
CLC		Clear Carry	C:=0	C	1	
SEN		Set Negative Flag	N:=1	N	1	
CLN		Clear Negative Flag	N:=0	N	1	
SEZ		Set Zero Flag	Z:=1	Z	1	
CLZ		Clear Zero Flag	Z:=0	Z	1	
SEI		Global Interrupt Enable	I:=1	I	1	
CLI		Global Interrupt Disable	I:=0	I	1	
SES		Set Signed Test Flag	S:=1	S	1	
CLS		Clear Signed Test Flag	S:=0	S	1	
SEV		Set Twoís Complement Overflow	V:=1	V	1	
CLV		Clear Twoís Complement Overflow	V:=0	V	1	
SET		Set T in SREG	T:=1	T	1	
CLT		Clear T in SREG	T:=0	T	1	
SEH		Set Half-carry Flag in SREG	H:=1	H	1	
CLH		Clear Half-carry Flag in SREG	H:=0	H	1	
NOP		No Operation		None	1	
SLEEP		Sleep		None	3	
WDR		Watchdog Reset		None	1	
BREAK		Break	For On-Chip Debug Only	None	N/A	

ARITHMETIC AND LOGIC INSTRUCTIONS

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
ADD	Rd, Rr	Add Two Working Registers	Rd:=Rd + Rr	Z,C,N,V,H,S	1	
ADC	Rd, Rr	Add with Carry Two Working Registers	Rd:=Rd + Rr + C	Z,C,N,V,H,S	1	
ADIW#	Rdl, K	Add immediate to Word	Rdh:Rdl:=Rdh:Rdl + K	Z,C,N,V,S	2	
SUB	Rd, Rr	Subtract Two Working Registers	Rd:=Rd - Rr	Z,C,N,V,H,S	1	
SUBI	Rd, K	Subtract Constant from Working Register*	Rd:=Rd - K	Z,C,N,V,H,S	1	
SBC	Rd, Rr	Subtract with Carry Two Working Registers	Rd:=Rd - Rr - C	Z,C,N,V,H,S	1	
SBCI	Rd, K	Subtract with Carry Constant from Work Registers*	Rd:=Rd - K - C	Z,C,N,V,H,S	1	
SBIW#	Rdl, K	Subtract Immediate from Word	Rdh:Rdl:=Rdh:Rdl - K	Z,C,N,V,S	2	
AND	Rd, Rr	Logical AND Working Registers	Rd:= Rd & Rr	Z,N,V,S	1	
ANDI	Rd, K	Logical AND Working Register and Constant*	Rd:= Rd & K	Z,N,V,S	1	
OR	Rd, Rr	Logical OR Working Registers	Rd:= Rd # Rr	Z,N,V,S	1	
ORI	Rd, K	Logical OR Working Register and Constant*	Rd:= Rd # K	Z,N,V,S	1	
EOR	Rd, Rr	Exclusive OR Working Registers	Rd:= Rd $ Rr	Z,N,V,S	1	
COM	Rd	One's Complement (Inversion)	Rd:= 0xFF - Rd	Z,C,N,V,S	1	
NEG	Rd	Two's Complement	Rd:= 0x00 - Rd	Z,C,N,V,H,S	1	
INC	Rd	Increment	Rd:= Rd + 1	Z,N,V,S	1	
DEC	Rd	Decrement	Rd:= Rd - 1	Z,N,V,S	1	
TST	Rd	Test for Zero or Minus	Rd:= Rd & Rd	Z,N,V,S	1	
CLR	Rd	Clear Working Register	Rd:= 0x00	Z,N,V,S	1	
SER	Rd	Set Working Register	Rd:= 0xFF	None	1	
MUL	Rd, Rr	Multiply Unsigned	R1:R0 := Rd x Rr	Z,C	2	X
MULS	Rd, Rr	Multiply Signed	R1:R0 := Rd x Rr	Z,C	2	X
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 := Rd x Rr	Z,C	2	X
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 := (Rd x Rr) << 1	Z,C	2	X
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 := (Rd x Rr) << 1	Z,C	2	X
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 := (Rd x Rr) << 1	Z,C	2	X

#) Wortoperationen nur mit Wortregistern R25:R24, R27:R26, R29:R28, R31:R30 und 6 Bit Konstante 0..63. adiw R24,13

BRANCH INSTRUCTIONS

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
RJMP	k	Relative Jump	PC:=PC+k+1	None	2	
JMP	k	Jump	PC:= k	None	2/3	X
IJMP		Indirect Jump to (Z)	PC:= Z	None	2	
RCALL	k	Relative Subroutine Call	PC:=PC+k+1	None	3	
ICALL		Indirect Call to (Z)	PC:=Z	None	3	
CALL	k	Direct Subroutine Call	PC := k	None	4	X
RET		Subroutine Return	PC:=STACK	None	4	
RETI		Interrupt Return	PC:=STACK, I:=1	I	4	
CPSE	Rd, Rr	Compare, Skip if Equal	If(Rd=Rr) THEN PC:=PC+2 or 3	None	1/2/3	
CP	Rd, Rr	Compare	Rd-Rr	Z,N,V,C,H,S	1	
CPC	Rd, Rr	Compare with Carry	Rd-Rr-C	Z,N,V,C,H,S	1	
CPI	Rd, K	Compare Working Register with Immediate *	Rd-K	Z,N,V,C,H,S	1	
SBRC	Rr, b	Skip if Bit in Working Register is Cleared	If(Rr(b)=0) THEN PC:=PC+2 or 3	None	1/2/3	
SBRS	Rr, b	Skip if Bit in Working Register is Set	If(Rr(b)=1) THEN PC:=PC+2 or 3	None	1/2/3	
SBIC	P, b	Skip if Bit in I/O-Register is Cleared	If(P(b)=0) THEN PC:=PC+2 or 3	None	1/2/3	
SBIS	P,b	Skip if Bit in I/O-Register is Set	If(P(b)=1) THEN PC:=PC+2 or 3	None	1/2/3	
BRBS	s, k	Branch if Status Flag Set	If(SREG(s)=1) THEN PC:=PC+k+1	None	1/2	
BRBC	s, k	Branch if Status Flag Cleared	If(SREG(s)=0) THEN PC:=PC+k+1	None	1/2	
BREQ	k	Branch if Equal	If (Z=1) THEN PC:=PC+k+1	None	1/2	
BRNE	k	Branch if Not Equal	If (Z=0) THEN PC:=PC+k+1	None	1/2	
BRCS	k	Branch if Carry Set	If (C=1) THEN PC:=PC+k+1	None	1/2	
BRCC	k	Branch if Carry Cleared	If (C=0) THEN PC:=PC+k+1	None	1/2	
BRSH	k	Branch if Same or Higher	If (C=0) THEN PC:=PC+k+1	None	1/2	
BRLO	k	Branch if Lower	If (C=1) THEN PC:=PC+k+1	None	1/2	
BRMI	k	Branch if Minus	If (N=1) THEN PC:=PC+k+1	None	1/2	
BRPL	k	Branch if Plus	If (N=0) THEN PC:=PC+k+1	None	1/2	
BRGE	k	Branch if Greater or Equal, Signed	If (N$V=0) THEN PC:=PC+k+1	None	1/2	
BRLT	k	Branch if Less Than Zero, Signed	If (N$V=1) THEN PC:=PC+k+1	None	1/2	
BRHS	k	Branch if Half-Carry Flag Set	If (H=1) THEN PC:=PC+k+1	None	1/2	
BRHC	k	Branch if Half-Carry Flag Cleared	If (H=0) THEN PC:=PC+k+1	None	1/2	
BRTS	k	Branch if T-Flag Set	If (T=1) THEN PC:=PC+k+1	None	1/2	
BRTC	k	Branch if T-Flag Cleared	If (T=0) THEN PC:=PC+k+1	None	1/2	
BRVS	k	Branch if OverflowFlag is Set	If (V=1) THEN PC:=PC+k+1	None	1/2	
BRVC	k	Branch if Overflow Flag is Cleared	If (V=0) THEN PC:=PC+k+1	None	1/2	
BRIE	k	Branch if Interrupt Enabled	If (I=1) THEN PC:=PC+k+1	None	1/2	
BRID	k	Branch if Interrupt Disabled	If (I=0) THEN PC:=PC+k+1	None	1/2	

Assembler-Direktiven

Element	Bedeutung	Beispiel	Erklärung zum Beispiel
;	Einleiten eines Kommentars	`; Kommentar bla bla`	
.include	Einbinden einer Datei	`.include "2313def.inc"`	Die 2313-Definitionen werden eingebunden
.equ	Deklaration von Konstanten. Wert ist nicht mehr änderbar Im weiteren Quelltext	`.equ papagei = 1`	Der Bezeichner papagei hat nun den Wert 1
		`.equ fisch = papagei * 2`	Werte können auch duch Ausdrücke (Expression) berechnet werden
.set	Deklaration / Definition einer Variablen. Erneute Zuweisung eines Wertes ist möglich	`.set cpuclock = 6000`	
		`.set cycle = cpuclock*20/8`	
.def	Weist einem Register einen Symbolischem Namen zu	`.def tmp = R16`	tmp ist R16
.org	Legt Adresse fest ab der folgender Code hinterlegt werden soll	`.org $2A` `Ldi R16,3`	Der ldi-Befehl wird in Adresse $2A Geschrieben
label:	Eine Einsprungmarke	`init:`	Die Marke (engl. label) **init**

Darstellung von Werten				
Dezimal	255		10	
Hexadezimal	0xFF	$FF	0xA	$0A
Binär	0b11111111		0b00001010	
Oktal	0377		012	

Hinweise zu .equ, .set: Der Assembler kann mühselige Rechenarbeit übernehmen: Bestimmte Werte, die öfter im Programm gebraucht werden, z.B. die Frequenz mit der der Kontroller arbeitet können unter ***symbolischen Namen*** gespeichert werden. Die Werte lassen sich direkt oder durch einfache Ausdrücke beschreiben. Der Assembler errechnet die Werte und setzt diese an den entsprechenden Stellen ein.

Speicher-Direktiven

Element	Bedeutung	Beispiel	Erklärung zum Beispiel
.cseg	Es folgt ein Code-Segment	`.cseg ; Flash`	Der folgende Code soll ins Flash

.dseg	Variablen im SRAM	`.dseg ; SRAM`
		`x: .byte 1 ; 1 Byte Variable`
		`feld: .byte 6 ; Feld mit 6 Byte`
			Speicherbereiche werden so im SRAM reserviert

.eseg	Es folgt ein EEPROM-Segment	`.eseg ; EEPROM`	Die folgenden Daten sollen ins EEPROM

.db	Liste mit Bytekonstanten	`hein: .db 1,2,3,4 ; Zahlen`
		`.db 'a','A' ; Zeichen`
		`text: .db "Lara" ; String`
			8bit-Werte für Flash oder EEPROM

.dw	Liste mit Wortkonstanten	`rage: .dw 4711 ; Zahl` `.dw hein ; Adresse`	16bit-Werte für Flash oder EEPROM
.byte	Reserviert n Bytes	`werte: .byte 10 ; 10 Bytes`	Bytebereich in SRAM oder EEPROM

Assembler Funktionen

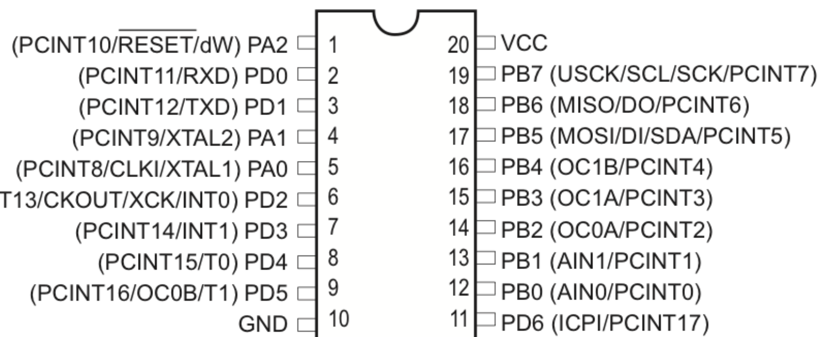
Funktion	Bedeutung	Beispiel
LOW(Ausdruck)	liefert Low-Byte = Bit 0-7	`LOW($123456) -> $56`
HIGH(Ausdruck)	liefert High-Byte = Bit 8-15	`HIGH($123456) -> $34`
BYTE2(Ausdruck)	liefert High-Byte = Bit 8-15	`BYTE2($123456) -> $34`
PAGE(Ausdruck)	liefert Bit 16-21	`PAGE($23456) -> $2`
BYTE3(Ausdruck)	liefert Bit 16-23	`BYTE3($12345678) -> $34`
BYTE4(Ausdruck)	liefert Bit 24-31	`BYTE4($12345678) -> $12`
LWRD(Ausdruck)	liefert Low-Wort = Bit 0-15	`LWRD($12345678) -> $5678`
HWRD(Ausdruck)	liefert High-Wort = Bit 16-31	`HWRD($12345678) -> $1234`
EXP2(Ausdruck)	liefert $2^{Ausdruck}$	`EXP2(3) -> 8`
LOG2(Ausdruck)	liefert $\log_2$(Ausdruck) ganzzahlig	`LOG2(17) -> 4`

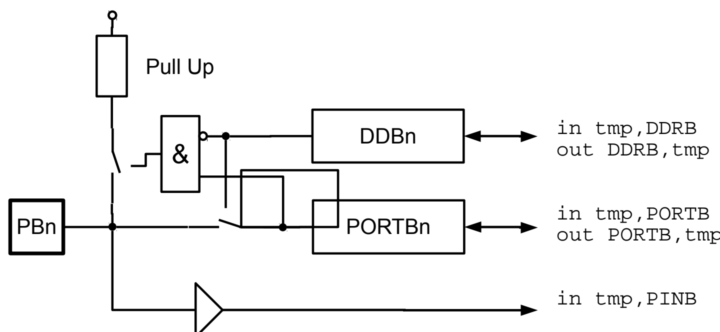
Umgang mit Ports (Datenrichtung, Ein- Ausgabe)

Der Mikrokontroller hat mehrere I/O Pins (Anschlüsse) die als 8Bit Ports gruppiert sind. Beim ATtiny2313A gibt es PORTB (PBn) und PORTD (PDn). Jedes Pin (z.B. PB0) kann als Ein- oder Ausgang geschaltet werden, die Richtung wird in dem Data Direction FlipFlop (z.B. DDB0) festgelegt.

Nach einem Reset sind alle Pins Eingänge weil die FlipFlops den Wert 0 (Inital Value) haben.

Um ein Pin als Ausgang zu schalten, muss in seinem DD-FlipFlop der Wert 1 gespeichert werden. Der Ausgangswert wird dann in dem PORT-FlipFlop (z.B. PORTB0) bestimmt.





Die Pins können zusätzliche Funktionen haben, siehe die Bezeichner in den Klammern bei der Zeichnung oben, z.B. serielle Schnittstelle.

Hier begnügen wir uns mit der normalen I/O-Funktion, dazu links ein Prinzip-Schaltbild eines Pins.

Die einzelnen FlipFlops (z.B. PORTBn) werden zu Registern (z.B. PORTB) zusammengefasst.

Primäre Pinfunktion

DDBn	PORTBn	I/O	Pull-up	Kommentar
0	0	Input	No	Tri-State (Hochohmig)
0	1	Input	Yes	Der Ausgang liefert einen Strom für z.B. Taster auf GND
1	0	Output	No	Push-pull Zero Output (Ausgang ist 0)
1	1	Output	No	Push-pull One Output (Ausgang ist 1)

Befehle, die mit Ports zu tun haben

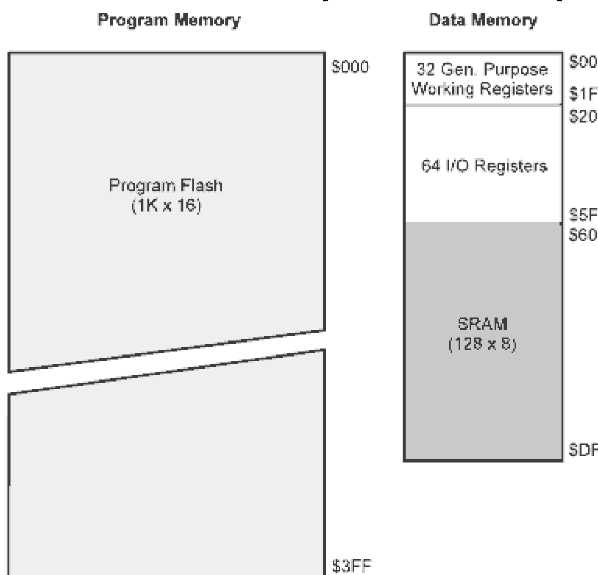
Befehl	Operand	Beschreibung	Beispiel	Erklärung zum Beispiel
IN	Rd,P	Einlesen eines Port in Register	`in R16,PIND`	PortD einlesen in R16
OUT	P, Rd	Ausgeben eines Register in Port	`out PORTB,R16`	R16 an PortB ausgeben
SBI	P, b	Setze Bit b in Port P	`sbi PORTB,2`	Das Bit 2 an PortB setzen
CBI	P, b	Lösche Bit b in Port P	`cbi PORTB,2`	Das Bit 2 an PortB löschen
SBIC	P, b	Überspringe, wenn Bit b in Port P gelöscht	`sbic PIND,4`	wenn Bit gelöscht ist, wird der folgende Befehl übersprungen. D.h. der folgende Befehl wird ausgeführt wenn Bit gesetzt..
SBIS	P, b	Überspringe, wenn Bit b in Port P gesetzt	`sbis PIND,4` `rjmp testmode`	Gehe in Tesmode wenn Pin 4 von PD 0 ist.

Initialisierungsbeispiel

Assembler	C Code

`ldi r16,0b00000011`
`out PORTB,r16`
`ldi r16,0b00000100`
`out DDRB,r16`
	`// PB2 ist Ausgang PB1 und PB0 Eingang mit PullUp`
	`PORTB = (1<<PB1)|(1<<PB0); // PullUps ein`
	
	`DDRB  = (1<<DDB2); // oder (1<<2)`

Atmel-RISC-8Bit Speicherkonzept am Beispiel des ATtiny2313



Der Speicher ist gemäß der Harvard-Architektur in einen Programm- und einen Datenspeicher unterteilt. Der Programmspeicher ist wortweise (16 Bit) organisiert, d.h. Den 2 KiByte Flash-Speicher stehen nur 1 Ki Adressen $000..$3FF gegenüber. Der Datenspeicher ist byteweise organisiert:

- $00 .. $1F sind die Register R0 bis R31
- $20 .. $5F ist den I/O Komponenten (Ports, Timer, UART, usw.) zugeordnet
- $60 .. $DF sind 128 Byte SRAM

Anmerkung: Das EEPROM (ohne Abbildung) wird über einen besonderen Mechanismus angesprochen, es gibt spezielle Befehle..

Wichtig: Zur Vereinfachung und zum Einsparen von Programmspeicher werden den Registern und I/O Adressen besondere Kurz-Zugriffe zugeordnet.

Beispiel: PORTB hat innerhalb des I/O Blocks die Adresse $18, bezogen auf den geammten Datenbereich die Adresse $38.
D.h. für manche Daten-Adressen gibt es zwei Zugriffsmöglichkeiten.

C Datentypen, Konstanten und Variablen

Datentyp	Bit	Dezimalbereich	Bemerkung
unsigned char	8	0..255	vorzeichenlose kleine Zahlen
char	8	-128..+127	kleine Zahlen mit Vorzeichen
unsigned int	16	0..65535	
int	16	-32768..+32767	
unsigned long	32	0..4294967295	
long	32	-2147483648..+2147483647	
float	32	$\pm 10^{-37}$ .. $\pm 10^{+38}$	
void			leer oder unbestimmt

Variablen werden vom Compiler in Registern oder im SRAM angelegt. Konstanten und vorbesetzte Variablen werden üblicherweise in einem Prolog aus dem Flash-Speicher in den SRAM kopiert -der Compiler erzeugt eine Befehlsfolge zur Initialisierung.

Die Art wie Werte gespeichert werden kann durch Kennwörter manuell beeinflusst werden:

Speicherklassen für Variablen

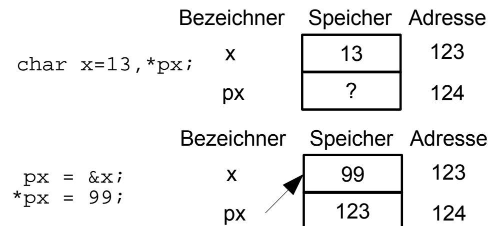
Kennwort	Beschreibung
static	auf fester Adresse (statisch) anlegen. Lokale Variablen behalten ihren Wert auch nach Verlassen des Blocks.
const	Daten sind konstant und dürfen nicht geändert werden. Compiler legt keine Speicherstelle an, sondern setzt die zugewiesenen Werte ein, vgl. Imediate-Befehle (ldi usw.).
volatile	Daten können von außen geändert werden (flüchtig) -z.B. Ports sind so vereinbart.

C-Code	Entsprechung in Assembler

`const unsigned char ZORA = 7; // nicht Variable angelegen`
`unsigned char bernd = ZORA; // global statisch vor main`
 	
`void main() {`
`  unsigned char temp = 123; // lokal dynamisch in main`
`  PORTB++; // Port B um 1 erhöhen`
`}`
	`.def bernd = R20`
	`ldi bernd,7`
	
	`.def temp = R16`
	`ldi temp,123; Initialisierung`
	`in R24, PORTB`
	`inc R24`
	`out PORTB,R24`

Zeiger

Datentyp *Pointername	`char *p`	Zeiger deklarieren
Pointer = &Variablenname	`p = &variable`	Zeiger Adresse von Variable zuweisen
*Pointername	`*p = 1`	Speicherstelle verwenden de-referenzieren



Call by Reference

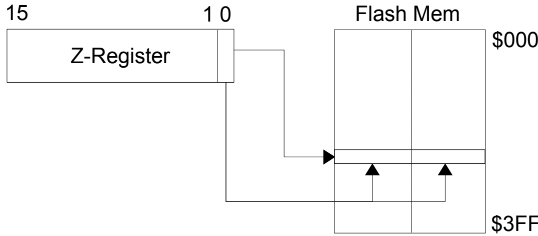
```
void test(char a, char *b){ // Call by Value bei a, Wert wird kopiert, b ist Zeiger auf char Wert
  a=3;                      // Lokaler Variablen a wird 3 zugewiesenen
  *b=3;                     // Zeiger b wird dereferenziert und der Speicherstelle 3 zugewiesen
}
void main(){
  char x=7,y=7;             // Lokale Variablen x,y in main
  test(x,&y);   // der Wert von x und die Referenz auf y (die Adresse von y) als Parameter von test
}               // nun ist x immer noch 7 aber y hat den Wert 3!
```

C-Operatoren

Rang	Operator	Syntax	Richtung	Bedeutung
1	->	Zeiger -> Element	–>	Elementselektion
1	[]	Zeiger [Ausdruck], Arrayelement	–>	Indizierung
1	()	Ausdruck, Typ (Ausdrucksliste)	–>	Funktionsaufruf, Werterzeugung
1	++, –	Lvalue++, Lvalue–	–>	Post- In-,De-krement
2	++, –	++Lvalue, –Lvalue	<–	Prä- In-,De-krement
2	˜ , !	˜Ausdruck, !Ausdruck	<–	Komplement, Negation
2	- , +	-Ausdruck, +Ausdruck	<–	unäres Minus, Plus
2	()	(Typ) Ausdruck	<–	Typkonvertierung
2	&	&Variablenbezeichner	<–	unär: Adressoperator
2	*	*Zeigerbezeichner	<–	unär: Indirektionsoperator
2	sizeof()	sizeof(Bezeichner)	<–	Operandengrösse in Byte
3	*, /	Ausd. * Ausd., Ausd. / Ausd.	–>	Multiplikation, Division
3	%	Ausdruck % Ausdruck	–>	Modulo, Divisionsrest
4	+, -	Ausd. + Ausd., Ausd. - Ausd.	–>	Addition, Subtraktion
5	<<, >>	Ausd. << Ausd., Ausd. >> Ausd.	–>	Linksshift, Rechtsshift
6	<, >	Ausd. < Ausd., Ausd. > Ausd.	–>	kleiner, größer als
6	<=, >=	Ausd. <= Ausd., Ausd. >= Ausd.	–>	kleiner, größer gleich als
7	==, !=	Ausd. == Ausd., Ausd. != Ausd.	–>	gleich, ungleich
8	&	Ausdruck & Ausdruck	–>	bitweises Und
9	^	Ausdruck ^ Ausdruck	–>	bitweises EXOR
10	|	Ausdruck | Ausdruck	–>	bitweises Oder
11	&&	Ausdruck && Ausdruck	–>	logisches Und
12	||	Ausdruck || Ausdruck	–>	logisches Oder
13	?:	Ausdruck ? Ausdruck : Ausdruck	<–	Bedingte Zuweisung
14	=	Lvalue = Ausdruck	<–	einfache Zuweisung
14	+=, -=, *=, /=, %=	Lvalue op Ausdruck	<–	Kombinierte Zuweisung (arithmetisch)
14	<<=, >>=, &=, ^=, |=	Lvalue op Ausdruck	<–	Kombinierte Zuweisung (bitweise)
15	,	Ausdruck , Ausdruck	–>	Folge von Ausdrücken

Adressierung von Konstanten im Flash

Im Programm-Speicher können Konstanten abgelegt werden mit der Assembler-Direktive *.db*. Mit dem LPM-Befehl kann ein Byte aus dem Programmspeicher via Z-Register (Z=R31:R30) in ein Register kopiert werden. Da der Programmspeicher 16 Bit organisiert ist wird mit dem LSB des Z-Registers zwischen dem High- und Low-Byte einer Programm-Speicher-Adresse unterschieden.

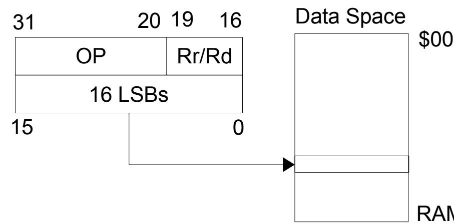



```
ldi R31, high(kzeiger*2) ; High-Byte in Z
ldi R30, low(kzeiger*2)  ; Low-Byte in Z
lpm R16, Z+ ; lade erste Konstante und Z++
; usw.
kzeiger: .db "ABI",0 ; die Konstante
```

Mnemonic	Operands	Description	Operation	Flags	# Clocks	Mega
LPM		Load Program Memory	R0:=(Z)	None	3	
LPM	Rd, Z	Load Program Memory	Rd := (Z)	None	3	X
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd := (Z), Z := Z+1	None	3	X
SPM		Store Program Memory	(Z) := R1:R0	None	-	X

Adressierung von Variablen im SRAM

Data Direct

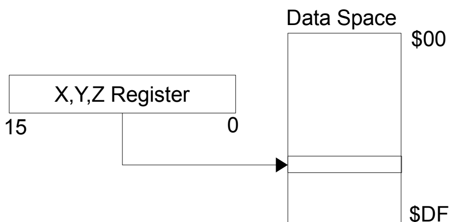


Mnemonic	Operands	Description	Operation	Flags	# Clocks
LDS	Rd, k	Load Direct from SRAM	Rd:=(k)	None	2
STS	k, Rr	Store Direct to SRAM	(k):= Rr	None	2

Beispiel: *lds R16, 0x60* ; Lade Inhalt des ersten SRAM-Bytes in R16. Dieser Befehl braucht zwei Wörter Platz wegen der 16 Bit SRAM Adresse (es gibt auch größere Controller)

Data Indirect

Möglich mit (X=R27:R26, Y=R29:R28, Z=R31:R30) Register



Mnemonic	Operands	Description	Operation	Flags	# Clocks
LD	Rd, X	Load Indirect	Rd:=(X)	None	2
LD	Rd, X+	Load Indirect and Post-increment	Rd:= (X),X:= X+1	None	2
LD	Rd, -X	Load Indirect and Pre-decrement	X:= X-1,Rd:=(X)	None	2
ST	X, Rr	Store Indirect	(X):=Rr	None	2
ST	X+, Rr	Store Indirect and post-increment	(X):=Rr,X:= X+1	None	2
ST	-X, Rr	Store Indirect and pre-decrement	X:= X-1,(X):= Rr	None	2

Beispiel: Tabelle tab[2] = {2,4} im SRAM anlegen:

<table>
<tr><th>Assembler</th><th>C-Code</th></tr>
<tr><td><pre>
  ldi R27, HIGH(tab); Prolog zur
  ldi R26, LOW(tab); Initialisierung
  ldi R24,2
  st x+,R24
  ldi R24,4
  st x+, R24
  .dseg; Datenbereich
tab: .byte 2; Feld mit 2 Byte
</pre></td><td><pre>unsigned char tab[2] = {2,4}; // mit Anfangswerten</pre></td></tr>
</table>

Data Indirect with Displacement

Mnemonic	Operands	Description	Operation	Flags	# Clocks
LDD	Rd, Y+q	Load Indirect with Displacement	Rd:=(Y+q)	None	2
STD	Y+q, Rr	Store indirect with Displacement	(Y+q):=Rr	None	2

Möglich mit (Y=R29:R28, Z=R31:R30) Register.

Bei komplexeren Datenstrukturen z.B. Listen von Records bzw. Objekten wäre es praktisch, zu der Basisadresse des Objekts (Zeiger, Pointer auf Record bzw. Objekt) ein bestimmtes Feld bzw. Attribut ansprechen zu können. Der Versatz der Feld- bzw. Attributadresse wird auch als "Displacement" bezeichnet.
Beachte: Displacement hat nur 6Bit -> 0 .. 63

Interrupts

ATtiny2313

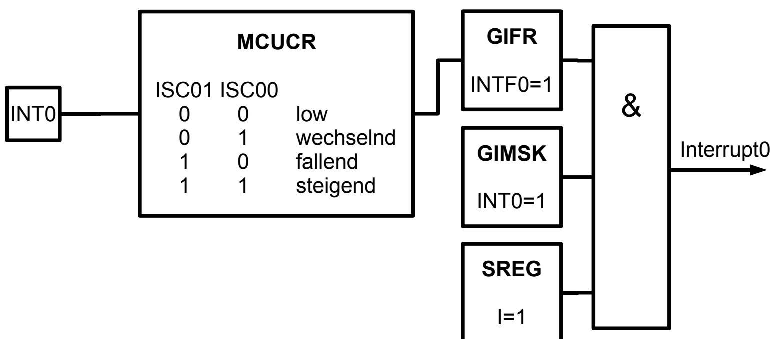
Adresse	Interrupt	Assembler-Bezeichner	C-Bezeichner
0x001	External Interrupt Request 0	INT0addr	INT0_vect
0x002	External Interrupt Request 1	INT1addr	INT1_vect
0x003	Timer/Counter1 Capture Event	ICP1addr	TIMER1_CAPT1_vect
0x004	Timer/Counter1 Compare Match	OC1addr	TIMER1_COMP1_vect
0x005	Timer/Counter1 Overflow	OVF1addr	TIMER1_OVF1_vect
0x006	Timer/Counter0 Overflow	OVF0addr	TIMER0_OVF0_vect
0x007	UART, Rx Complete	URXCaddr	UART_RX_vect
0x008	UART Data Register Empty	UDREaddr	UART_UDRE_vect
0x009	UART, Tx Complete	UTXCaddr	UART_TX_vect
0x00A	Analog Comparator	ACIaddr	ANA_COMP_vect

.equ INT_VECTORS_SIZE = 11 ; size in words

Assemblerbefehle, die mit Interrupts zu tun haben

Mnemonic	Operands	Description	Operation	Flags	# Clocks
SEI		Global Interrupt Enable	I:=1	I	1
CLI		Global Interrupt Disable	I:=0	I	1
RETI		Interrupt Return	PC:=STACK, I:=1	I	4
IN	Rd, SREG	In Port	Rd:=SREG	None	1
OUT	SREG, Rr	Out Port	SREG:=Rr	ALL	1
BRIE	k	Branch if Interrupt Enabled	If (I=1) THEN PC:=PC+k+1	None	1 oder 2
BRID	k	Branch if Interrupt Disabled	If (I=0) THEN PC:=PC+k+1	None	1 oder 2
SLEEP		Sleep		None	3
WDR		Watchdog Reset		None	1

Externe Interrupts



Bit	7	6	5	4	3	2	1	0
MCUCR	-	-	SE	SM	ISC11	ISC10	ISC01	ISC00
GIFR	INTF1	INTF0	-	-	-	-	-	-
GIMSK	INT1	INT0	-	-	-	-	-	-

Externe Interrupts können durch die INT0 (PD2) und INT1 (PD3) Pins ausgelöst werden. Sind die Pins als Ausgang geschaltet, lassen sich hierdurch auch Softwareinterrupts realisieren. Ob Low-Pegel oder steigende bzw. fallende Flanke den Interrupt auslösen wird über die Bits ISC des MCU Control Register MCUCR definiert. Ein Interrupt wird ausgeführt, wenn er im General Interrupt MaSK Register GIMSK freigeschaltet wurde und im SREG eine generelle Interruptfreigabe existiert.

Beispiel -externer Interrupt INT0 mit fallender Flanke:

Assembler

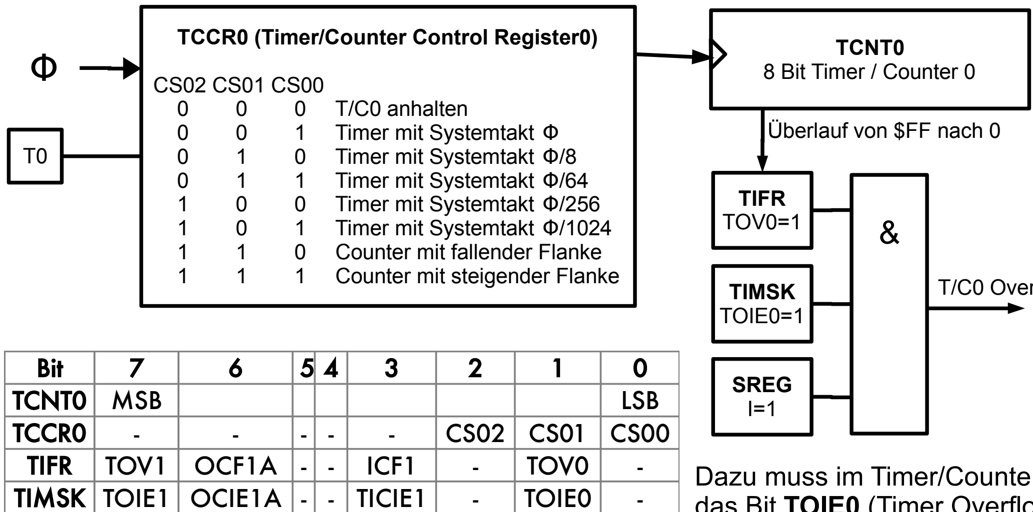
```
.include "2313def.inc"
  rjmp reset;
  .org INT0addr; Adresse des Vektors
  rjmp isrINT0; Behandeln von INT0 Interrupt

  .org INT_VECTORS_SIZE; weiter nach
Interrupttabelle
reset: ; Initialisierung
  ldi r16,high(RAMEND)
  out SPH,r16 ; Set Stack Pointer to top of RAM
  ldi r16,low(RAMEND)
  out SPL,r16
  in R16,MCUCR ; altes Steuerregister
  sbr R16,ISC01 ; setze Bit ISC01
  out MCUCR,R16
  in R16,GIMSK; altes Freigaberegister
  sbr R16,INT0; setze Bit INT0
```

C-Code

```
#include <avr/io.h>      // Definitionen laden
#include <util/delay.h> // Delay-Bibliothek laden
#include <avr/interrupt.h>

ISR(INT0_vect){
// Befehle für Interrupt-Behandlung
}

void main(){
// Befehle für Initialisierung
  MCUCR |= (1 << ISC01); // INT0 fallende Flanke
  GIMSK |= (1 << INT0);  // INT0 freigeschaltet
  sei(); alle Interrupts frei
  while (1){
    // weiteres Hauptprogramm
  }
}
```

```
  out GIMSK, R16
  sei ; Enable interrupts
  ; -- weitere Befehle --

isrINT0:
  push r16; Register retten
  in r16,SREG; Statusregister laden
  push r16; Statusregister retten
  ; -- weitere Befehle --
  pop r16; Statusregister vom Stapel holen
  out SREG,r16; Statusregister wieder herstellen
  pop r16
  reti; Interrupt behandelt
```

Timer / Zähler



Bit	7	6	5	4	3	2	1	0
TCNT0	MSB							LSB
TCCR0	-	-	-	-	-	CS02	CS01	CS00
TIFR	TOV1	OCF1A	-	-	ICF1	-	TOV0	-
TIMSK	TOIE1	OCIE1A	-	-	TICIE1	-	TOIE0	-

Der ATtiny2313 verfügt über zwei Zähler, die auch als Timer verwendet werden können.

Der **TCNT0** hat 8 Bit, und zählt aufwärts. Mit dem Timer Control Register 0 **TCCR0** kann die Art des Taktsignals bestimmt werden. Neben verschiedenen Teilverhältnissen des Systemtaktes Φ, kann auch mit externem Signal an PD4 gezählt werden. Bei einem Überlauf (Zählstand $FF -> 0) kann ein Interrupt **Timer/Counter 0 Overflow** ausgelöst werden. Dazu muss im Timer/Counter-Interruptmaskenregister **TIMSK** das Bit **TOIE0** (Timer Overflow Interrupt Enable) und die generelle Interruptfreigabe gesetzt sein.

Beispiel Timer / Counter0 als Zähler mit externem Signal fallende Flanke, Ausgabe auf PortB

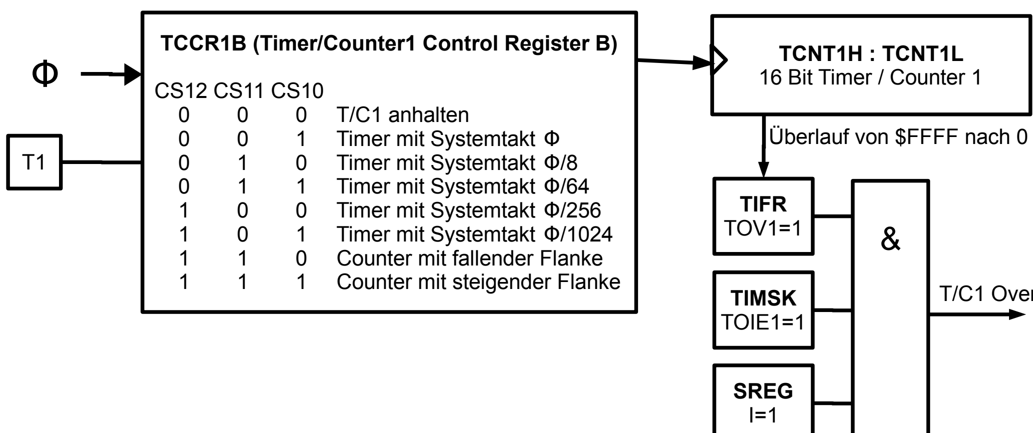
Assembler

```
  rjmp reset;
  .org INT_VECTORS_SIZE; nach Interrupttabelle
reset:
  ldi r16,low(RAMEND)
  out SPL,r16
  ser R16 ; R16 <- $FF
  out DDRB,R16; PortB als Ausgang
  in R16, TCCR0; altes Steuerregister
  ori R16,0b110; ext Takt fallende Flanke
  out TCCR0,R16; neues Steuerregister
loop:
  in R16,TCNT0; laufenden Zählerstand
  out PORTB,R16; auf PortB ausgeben
  rjmp loop
```

C-Code

```
#include <avr/io.h>      // Definitionen laden
#include <util/delay.h> // Delay-Bibliothek laden
#include <avr/interrupt.h>

void main(){
  DDRB = 0xff; // PortB als Ausgang
  TCCR0 |= 6;  // 110 ext. Takt fallende Flanke
  while (1){
    PORTB = TCNT0; // Zähler ausgeben
  }
}
```

16 Bit Timer / Counter1



	Bit	7	6	5	4	3	2	1	0
ICR1L	Timer/Counter1 Input Capture Register Low								LSB
ICR1H	Timer/Counter1 Input Capture Register High	MSB							
OCR1AL	Timer/Counter1 Output Compare Register A Low								LSB
OCR1AH	Timer/Counter1 Output Compare Register A High	MSB							
TCNT1L	Timer/Counter1 Low								LSB
TCNT1H	Timer/Counter1 High	MSB							
TCCR1B	Timer/Counter1 Control Register B	ICNC1	ICES1	-	-	CTC1	CS12	CS11	CS10
TCCR1A	Timer/Counter1 Control Register A	COM1A1	COM1A0	-	-	-	-	PWM11	PWM10
TIFR	Timer/Counter Interrupt Flag Register	TOV1	OCF1A	-	-	ICF1	-	TOV0	-
TIMSK	Timer/Counter Interrupt Mask Register	TOIE1	OCIE1A	-	-	TICIE1	-	TOIE0	-

Praktische Assembler Makros / Unterprogramme

Quelle: Günter Schmitt, Mikrocomputertechnik mit Controllern der Atmel AVR_RISC-Familie, Oldenbourg-Verlag
Download: http://www.oldenbourg-wissenschaftsverlag.de/fm/694/3-486-58790_1.zip

```
; Mmuldiv.h Headerdatei Makros Multiplikation und Division
; 16bit unsigned Multiplikation und Division
        .INCLUDE    "Mmul16.asm"     ; R3:R2:R1:R0 <- @0:@1 * @2:@3 mul-Befehle
        .INCLUDE    "Mmulx8.asm"     ; R1:R0 <- @0 * @1 Software
        .INCLUDE    "Mmulx16.asm"    ; R3:R2:R1:R0 <- @0:@1 * @2:@3 Software
        .INCLUDE    "Mdivx8.asm"     ; @0 / @1  @0 <- Quotient  @1 <- Rest
        .INCLUDE    "Mdivx16.asm"    ; @0:@1 <- @0:@1 / @2:@3   @2:@3 <- Rest
; 16bit signed Multiplikation und Division
        .INCLUDE    "Mmuls16.asm"    ; R3:R2:R1:R0 <- @0:@1 * @2:@3 mul-Befehle
        .INCLUDE    "Mmulsx16.asm"   ; R3:R2:R1:R0 <- @0:@1 * @2:@3 Software
        .INCLUDE    "Mdivsx16.asm"   ; @0:@1 <- @0:@1 / @2:@3   @2:@3 <- Rest
; 8bit nach drei BCD-Stellen zerlegen
        .INCLUDE    "Mdual2bcd.asm" ; @0=dual @1 <- Hund @2 <- Zehn @3 <- Ein
; Festpunktmultiplikation emuliert fmul-Befehl
        .INCLUDE    "Mfmulx8.asm"    ; R1:R0 <- @0 * @1
```