Inhaltsverzeichnis

9 Maschinensprache und Assembler **1**
9.1 RISC und CISC . . . 1
9.2 Der betrachtete Beispiel-Mikroprozessor . . . 2
9.3 Register . . . 6
9.3.1 Statusregister . . . 6
9.3.2 General Purpose Working Register . . . 7
9.3.3 Stack-Pointer . . . 7
9.4 Assembler . . . 9
9.4.1 Assemblerbefehle . . . 9
9.4.2 Ablauf eines Assemblerprogramms . . . 10
9.5 Adressierungsarten . . . 11
9.5.1 Sofort-Operanden-Befehle . . . 11
9.5.2 Direkte Adressierung . . . 12
9.5.3 Relative Adressierung . . . 13
9.5.4 Indexregister . . . 13
9.5.5 Indexregister mit Offset . . . 14
9.6 Untergliederung der Assemblerbefehle . . . 15
9.6.1 Datentransferbefehle . . . 16
9.6.2 Arithmetische Operationen . . . 18
9.6.3 Logische Operationen . . . 19
9.6.4 Vergleichsbefehle (`CP` und `CPI`) . . . 19
9.6.5 Bitbefehle (`SBR`, `CBR`, `BSET` und `BCLR`) . . . 20
9.6.6 Verzweigebefehle . . . 20
9.7 Programmierung von Schleifen . . . 22
9.8 Rekursion . . . 23
9.9 Befehlsübersicht . . . 24
9.10 Weiterführende Links . . . 28

Literaturverzeichnis **29**

Index **31**

Kapitel 9

Maschinensprache und Assembler

Maschinensprache ist charakterisiert durch Befehlsformat und -umfang. Sie ist hardwareabhängig, wenn auch weitgehend konsistent innerhalb einer Rechnerfamilie (z.B. *Atmel AVR*; IBM 360/370/390; Intel 8080, 80x86; Motorola 680x0).

Im *von Neumann-Rechner* befinden sich *Maschinenbefehle* zusammen mit den zur Programmausführung benötigten Daten im Hauptspeicher, codiert als 0-1-Folgen. Während der Programmausführung werden die Maschinenbefehle in das *Befehlsregister* der CPU geladen und von der *Steuereinheit* (Control Unit) interpretiert.

Die im Rahmen der Vorlesung betrachteten AVR-Mikroprozessoren basieren auf der *Harvard-Architektur*. Hier sind *Programmspeicher* sowie *Datenspeicher* strikt voneinander getrennt. Dies hat den Vorteil, dass innerhalb eines einzigen Taktzyklus Befehle und ihre dazugehörigen Daten aus separaten Speichern gelesen werden können. Die ursprüngliche von Neumann-Architektur benötigt für die gleiche Operation zwei Taktzyklen.

Grundsätzlich besteht ein Maschinenprogramm für einen Mikroprozessor aus einer Folge von Zahlen in dualer oder bestenfalls hexadezimaler Darstellung und ist deshalb nur schwer zu lesen oder gar zu erstellen. Aus diesem Grund wird statt der reinen Maschinensprache in der Regel die sog. Assemblersprache verwendet. Die *Assemblersprache* verbessert die Lesbarkeit von Maschinenprogrammen durch eine mnemotechnische (d.h. leicht zu merkende) Darstellung des Befehlscodes und die Möglichkeit, Symbole für Speicheradressen und Daten zu verwenden. Vor der Ausführung muss ein Assemblerprogramm natürlich in ein reines Maschinenprogramm, d.h. in eine Folge sog. *Operationscodes* (häufig kurz OpCodes genannt) umgewandelt werden. Das hierzu benötigte Übersetzungsprogramm wird Assembler genannt. Man beachte, dass der Begriff *Assembler* häufig auch als Kürzel für die Assemblersprache selbst benutzt wird.

9.1 RISC und CISC

Bevor wir in die Tiefen der AVR-Architektur eintauchen, soll zunächst der Begriff RISC geklärt werden. Diese Abkürzung steht für Reduced Instruction Set Computer. RISC ist eine Prozessorarchitektur mit einem einfachen Befehlssatz, also das Gegenteil einer CISC-Architektur (CISC = Complex Instruction Set Computer, IA-32-Prozessoren sind

typische CISC-CPUs).
CISC war lange Zeit die dominante Architektur beim Entwurf von Prozessoren. Das Entwicklungsziel für CISC-CPUs war, mit einem einzigen Maschinenbefehl möglichst komplexe Operationen ausführen zu können. Das dabei auftretende Problem einer CISC-Architektur ist, dass zur Ausführung eines einzigen CISC-Befehls viele Taktzyklen benötigt werden.
CISC-Befehlssätze werden durch Mikroprogramme („interne Automaten“) in den CPUs umgesetzt. Mit anderen Worten: ein einziger CISC-Befehl wurde CPU-intern durch eine Folge von Mikrobefehlen abgearbeitet.

CISC-Befehle verwenden unterschiedliche Befehlsformate. Es gibt zum Beispiel Anweisungen mit einem 8-, 12-, 16- oder auch 24-bit-Operationscode. Die Ursprünge der CISC-Technologie liegen bei den Minicomputern der späten 60er und frühen 70er Jahre (PDP8 und PDP11 von DEC usw.). Die Prozessoren dieser Rechner wurden mit Hilfe einer großen Zahl nur wenig integrierter Schaltkreise aufgebaut (die erste PDP8-Variante bestand sogar nur aus einzelnen Transistoren). Damals war externer Speicher langsam und sehr teuer. Aus diesem Grund wurden CPUs mit Mikrocode-ROMs für die bereits angesprochenen Mikroprogramme ausgestattet, die dann die jeweiligen CISC-Maschinenbefehle implementierten. Mikrocode-ROMs waren deutlich schneller als der Hauptspeicher einer CPU.

Mit der Verbreitung des Mikroprozessors wurde das Mikrocode-ROM zum Problem: Es benötigte relativ viel Silizium-Fläche, die für andere Funktionen fehlte. Die Befehlssätze vieler Mikroprozessoren wurden fest verdrahtet und damit im Vergleich zu den Minicomputern auch wieder etwas einfacher. Weiterhin machte die Speichertechnologie mit der Verbreitung komplexer integrierter Schaltkreise sehr große Fortschritte.

Ausgangsbasis für die RISC-Entwicklung war eine IBM-Studie der 80er-Jahre, aus der hervorging, dass bei einem CISC-Befehlssatz von ca. 200 einzelnen Befehlen eines IBM-Minicomputers ungefähr 10 Befehle 80% des Codes der meisten Programme, 21 Befehle 95% und 30 Befehle 99% ausmachten. Die verbleibenden 170 Befehle kommen gerade einmal für 1% des von einem Compiler erzeugten Codes zum Einsatz.
Als Folge dieser Studie reifte die Erkenntnis, den Befehlssatz einer CPU auf ein Minimum zu reduzieren und die einzelnen Befehle in einem Schaltwerk fest zu verdrahten. Aus diesen Grundlagen sind RISC-Architekturen entstanden. Das Ziel beim Entwurf eines RISC-Prozessor ist es, die meisten Befehle mit einem einzigen Taktzyklus ausführen zu können. RISC bewirkt somit auch eine Leistungssteigerung für Prozessoren. AVR-Prozessoren gehören zur Familie der RISC-Prozessoren.

9.2 Der betrachtete Beispiel-Mikroprozessor

Die in diesem Kapitel betrachteten Befehle und Strukturen beziehen sich auf den Mikrocontroller AVRmega8 der Firma Atmel und dessen Assembler. Hierbei handelt es sich um einen *8-Bit-RISC-Prozessor* mit geringer Energieaufnahme.

Die technischen Daten im Überblick:

- 130 Assemblerbefehle
- 32 8-Bit-Register
- 8k interner Speicher
- 512 Bytes EEPROM
- 1k SRAM
- zwei 8-Bit-Timer / Counter
- ein 16-Bit-Timer / Counter
- sechs Kanal 10-Bit-A/D-Wandler (Analog / Digital Wandler)
- Echtzeitzähler mit separatem Oszilator
- drei Pulsweitenmodulationskanäle (PWM-Kanäle)
- analoger Comparator
- serielles Interface

Die einzelnen Komponenten können im Blockdiagramm (siehe Abbildung 9.2) identifiziert werden.
Der 8kB interne Speicher ist als *SRAM* realisiert und interagiert unter anderem mit dem 8-bit *Datenbus*.
Das *Steuerwerk* (*Instruction Decoder*) beinhaltet das Leitwerk des Mikroprozessors. Hier werden die einzelnen Befehle des Programms dekodiert und, entsprechend des Befehls, die Operanden geladen, auf die der Befehl angewandt werden soll.
Die arithmetisch-logische Einheit (*ALU*) stellt das *Rechenwerk* dar und dient beispielsweise der Addition.
Um eine Kommunikation mit externen und internen Komponenten zu ermöglichen, verfügt der Mikroprozessor über eine Vielfalt von *Eingabeports* sowie *Ausgabeports*. Diese sind im Blockdiagramm in Abb. 9.2 unter anderem mit PORTB, PORTC und PORTD bezeichnet.
Beim *Analog-Digitalwandler* (ADC) handelt es sich um einen Konverter, der das analoge Eingangssignal (eine Spannung) in ein digitales Signal wandelt, welches dann vom Mikrocontroller weiterverarbeitet werden kann.
Der *Stack-Pointer* stellt ein Hilfsregister dar. In ihm ist der Zeiger hinterlegt, der auf die nächste freie Adresse des *Stack*s (häufig auch als Stapel- oder Kellerspeicher bezeichnet) zeigt. Eine ausführliche Erläuterung des Stacks wird im Abschnitt 9.3.3 gegeben.
Der interne *Timer* (TIMER/COUNTER) ist ein integrierter Schaltkreis des Chips, der beim Messen von Zeitabständen, beim wiederkehrenden Ausführen von Programmteilen oder beim Zählen von Ereignissen zum Einsatz kommt. Der *Universal Asynchronous Receiver Transmitter* (kurz UART) versieht den seriellen digitalen Datenstrom mit einem fixen Rahmen. Dazu zählt unter anderem ein Start- und ein Stopp-Bit. Der UART realisiert damit z.B. die Grundlagen einer seriellen Übertragung. Eine serielle

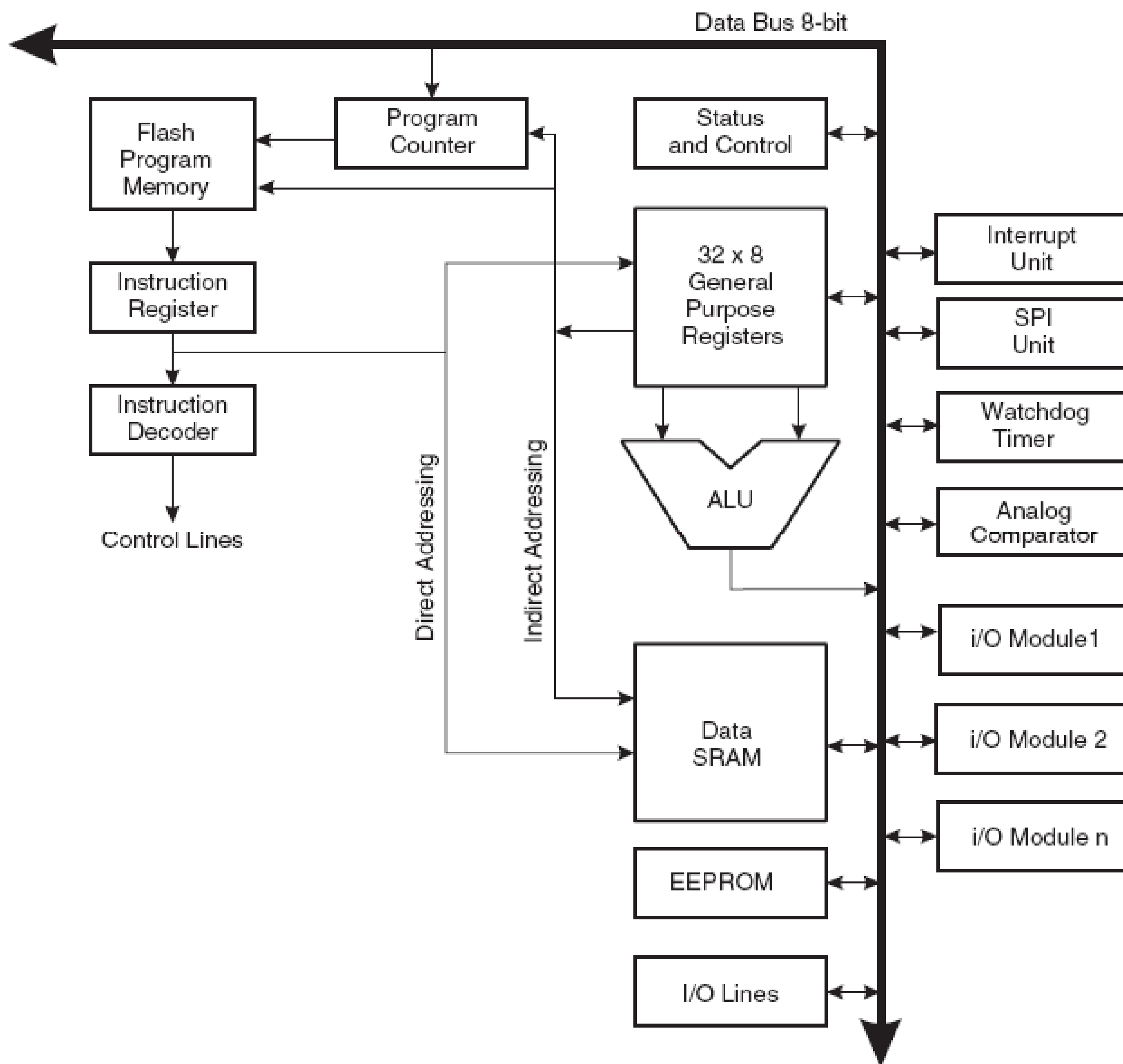

Abbildung 9.1: Die AVR-Mikrocontroller-Architektur im Überblick

Datenübertragung ist auch Grundlage des *Inter-Integrated Circuit* (I^2C) Standards. Hier handelt es sich um einen seriellen 2-Draht-Datenbus zur Kommunikation digitaler Schaltkreise. Ein weiteres serielles Bussystem mit ähnlichen Anwendunden stellt das *Serial Peripheral Interface* (SPI) dar.
Der *Komparator* (COMPARATOR) ermöglicht den Vergleich zweier analoger Werte.

Alle oben genannten Komponenten können im folgenden Blockdiagramm 9.2 identifiziert werden.

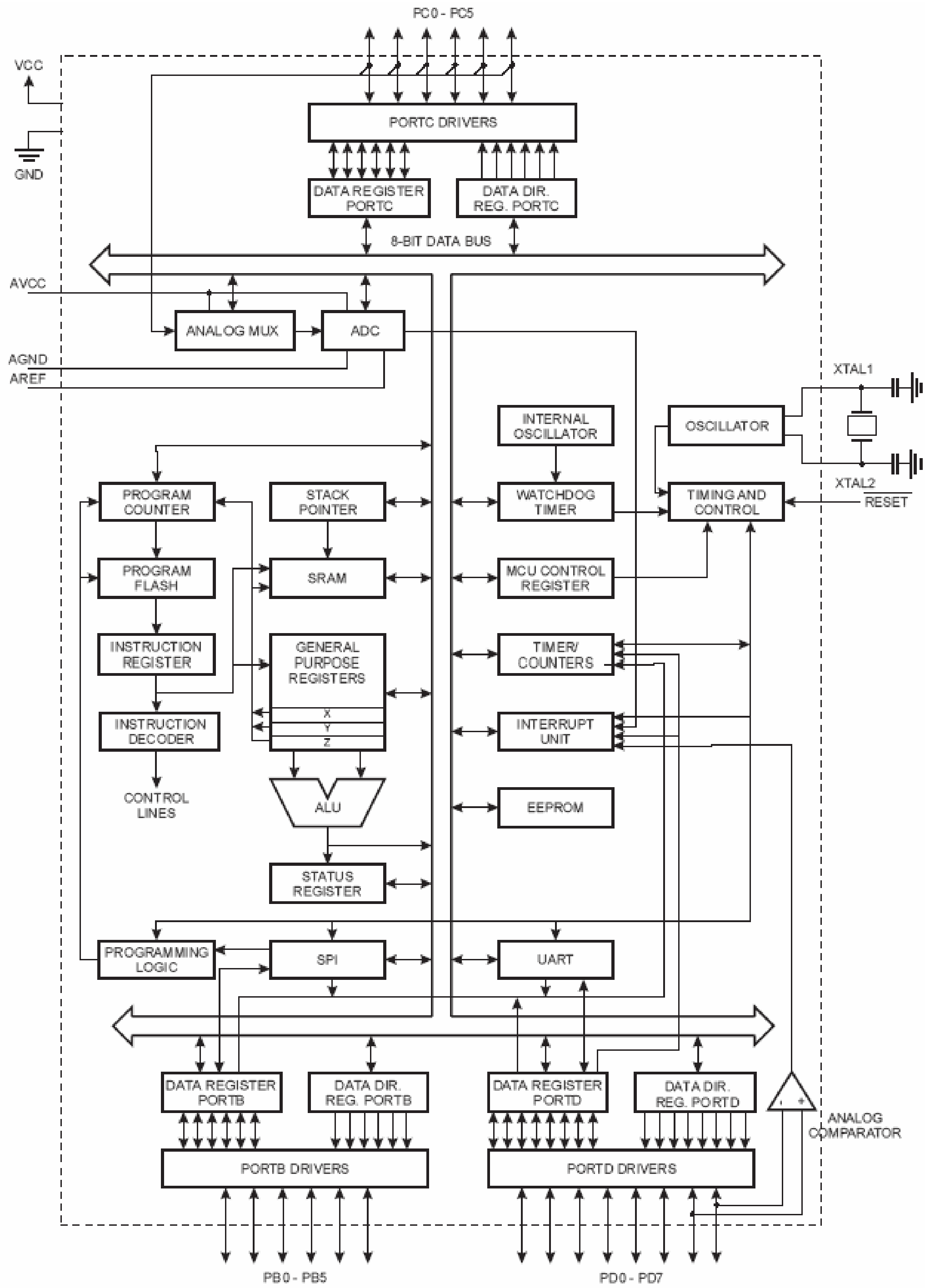

Abbildung 9.2: Blockdiagramm des AVRmega8

9.3 Register

Als Registersatz bezeichnet man zu Gruppen zusammengefasste Speicherzellen innerhalb des Speichers eines Prozessors. Dabei stellen die Register, sozusagen als RAM innerhalb der CPU, den Speichertyp dar, der mit der größten Zugriffsgeschwindigkeit angesprochen werden kann.
Weiterhin haben die Register innerhalb der Prozessorarchitektur insofern eine Sonderstellung, als dass sie häufig Quell- oder Zieloperand für Maschinencode sind.

Der Aufbau, die Größe und die Art der Register sind dabei abhängig vom eingesetzten Prozessortyp. Im folgenden soll anhand der realen Hardware des Atmel AVRmega8 der Aufbau der verschiedenen Register näher gebracht werden.

9.3.1 Statusregister

Als *Statusregister* wird ein Register im *Steuerwerk* bezeichnet. Über die dort gesetzten Bits sind Rückschlüsse auf die letzten Rechenoperationen der ALU möglich. So kann zum Beispiel geprüft werden, ob ein Ergebnis gleich 0 war. In diesem Fall ist das Zero Flag gesetzt. Dies kann dann wiederum Einfluß auf den Ablauf des Programms haben (siehe hierzu das Beispiel aus Abschnitt 9.7).
Da die einzelnen Bits logisch voneinander unabhängig sind, können sie in einem Register zusammen gefasst werden.
Der Aufbau des Statusregisters des Atmel AVRmega8 ist dabei wie folgt:

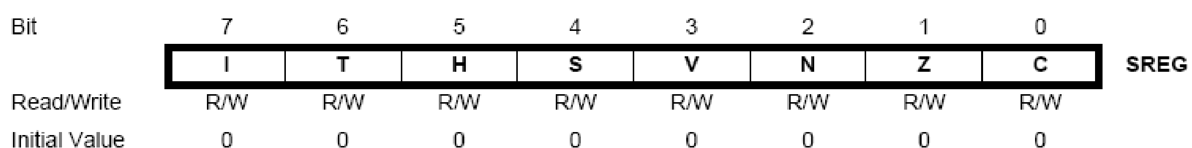

Abbildung 9.3: Statusregister des Atmelmega8

Dabei handelt es sich um die in Tabelle 9.1 aufgelisteten Bits bzw. Flags:

C	Carry Flag (Übertragsbit)
Z	Zero Flag (Ergebnis war 0)
N	Negative Flag (negatives Ergebnis der letzten Operation)
V	Two's Complement Flag / Overflow Flag (Überlaufsbit)
S	Sign-Bit (Das Sign Bit stellt immer das Ergebnis eines exklusiven Oders, das auf das V-Bit und das N-Bit angewandt wird. S=N⊕V)
H	Half Carry Flag (Wird gesetzt, wenn ein Übertrag von Bit 3 auf Bit 4 bei einer Addition erfolgt.)
T	Bit Copy Storage (dient als Speicher für einzelne Bits)
I	Global Interrupt Enable (steuert die Unterbrechungsverarbeitung)

Tabelle 9.1: Bits bzw. Flags des Statusregisters des Atmelmega8

9.3.2 General Purpose Working Register

Wie bereits einleitend erwähnt, verfügt der Mikrocontroller AVRmega8 über 32 8-Bit-*Register* zur internen Speicherung von Daten. Diese werden als Register R0 bis R31 bezeichnet. Um im Einzelfall auf ein 16-Bit-Register zurückgreifen zu können, sind sechs der 8-Bit-Register zusammenfassbar zu drei 16-bit-Registern.
Dies veranschaulichen die Abbildungen 9.4 und 9.5.

	7 0	Addr.	
	R0	0x00	
	R1	0x01	
	R2	0x02	
	...		
	R13	0x0D	
General	R14	0x0E	
Purpose	R15	0x0F	
Working	R16	0x10	
Registers	R17	0x11	
	...		
	R26	0x1A	X-register Low Byte
	R27	0x1B	X-register High Byte
	R28	0x1C	Y-register Low Byte
	R29	0x1D	Y-register High Byte
	R30	0x1E	Z-register Low Byte
	R31	0x1F	Z-register High Byte

Abbildung 9.4: Register des Atmel AVRmega8

Dabei werden die Register R26 bis R31 zu 16-bit-Registern zusammengezogen. Bezeichnet werden sie anschließend als *X-, Y- und Z-Register* und setzen sich aus einem hohen und einem niedrigen Byte zusammen.

9.3.3 Stack-Pointer

Der *Stack* (auch: *Kellerspeicher*, *Keller* oder *Stapel*) ist ein spezieller Speicher oder Speicherbereich, der nach dem *LIFO*-Prinzip (*Last In, First Out*) aufgebaut ist: Eine Leseoperation (*POP*) bezieht sich immer auf das sog. oberste Kellerelement, d.h. auf denjenigen Wert, der zuletzt (mit dem *PUSH*-Befehl) in diesen Speicher geschrieben wurde. Bei der Durchführung des POP-Befehls wird der gelesene Wert automatisch aus dem Keller entfernt, so dass der vorletzte in den *Keller* geschriebene Wert zum obersten Kellerelement wird.
Der Stack wird dabei zum Beispiel genutzt, um beim Aufruf einer *Subroutine* die *Rücksprungadresse* sichern zu können. Ohne einen Stack könnten keine Unterfunktionen aufgerufen und anschließend aus diesen zurückgekehrt werden, da keine Information über den Befehlszähler und somit den nächsten auszuführenden Befehl nach der Subroutine

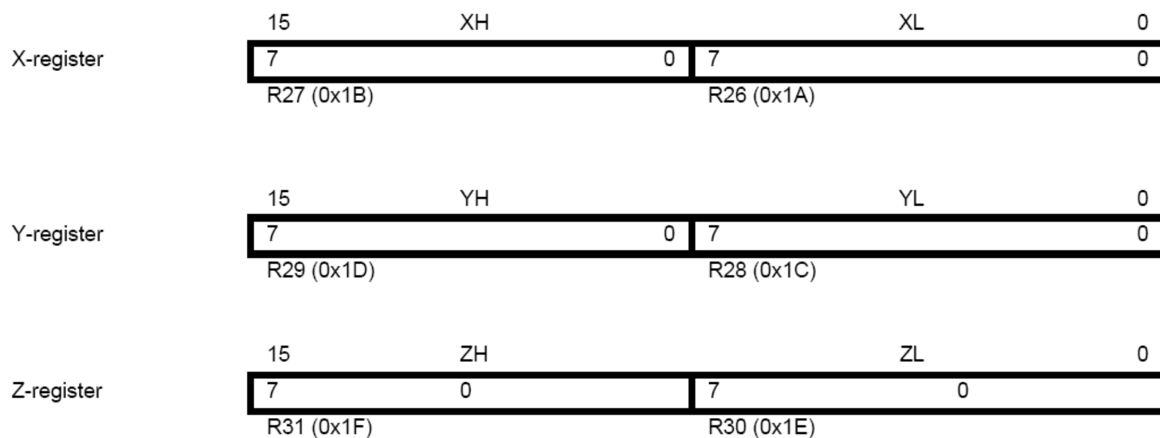

Abbildung 9.5: Die drei 16-bit-Register des Atmel AVRmega8 mit hohem und niedrigem Byte

vorhanden wäre.
Darüber hinaus können auf dem Stack auch temporäre Daten abgelegt werden, beispielsweise Inhalte von Registern. Diese würden ansonsten bei einer erneuten Belegung des Registers überschrieben.
Um mit dem Stack arbeiten zu können, benötigt man den so genannten *Stack-Pointer*, ein Zeiger der auf die Stapelspitze und somit auf die nächste freie Adresse nach dem zuletzt hinzugefügten Element zeigt. Von diesem ausgehend kann man dem Stapel ein Element hinzufügen (PUSH) oder eines herunternehmen (POP).

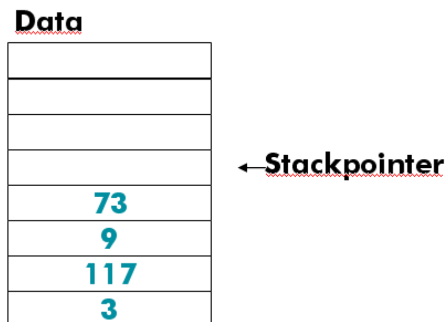

Abbildung 9.6: Stackpointer

Da der Mikrocontroller AVRmega8 über einen Speicher größer 256 Byte verfügt, reicht ein 8Bit-Register zur Adressierung nicht mehr aus. Hier wird für das obere Adressbyte ein zweites hinzugenommen, welches mit SPH (für Stack-Pointer / high byte) bezeichnet wird. Das erste, SPL, enthält entsprechend das untere Byte des Stack-Pointer-Register. Eine korrekte Initialisierung des Stackpointers erfolgt, indem jeweils das Lowbyte und das Hibyte von RAMEND, der höchsten Speicheradresse des verfügbaren Prozessor-RAMS, geladen werden und in die entsprechenden Register geschrieben werden.

Der nachfolgende Code zeigt die korrekte Initialisierung des Stack-Pointers.

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

Abbildung 9.7: Der Stack-Pointer

```
.include "m8def.inc"

LDI r16, LOW(RAMEND)   ; Lowbyte von RAMEND laden
OUT SPL, r16           ; in Stackpointer schreiben
LDI r16, HIGH(RAMEND)  ; Hibyte von RAMEND laden
OUT SPH, r16           ; in Stackpointer schreiben
```

Abbildung 9.8: Assemblercode zur Initialisierung des Stack-Pointers

9.4 Assembler

Assemblersprache stellt eine spezielle Programmiersprache dar, in welcher die *Maschinensprache* einer speziellen *Prozessorarchitektur* in einer für den Menschen lesbaren Form repräsentiert wird. Dabei stellen die einzelnen Befehle der Assemblersprache *Symbole des Maschinencodes* dar, die auch als Operationscodes oder kurz Opcodes bezeichnet werden. Die Abbildung zwischen Befehlsworten (*Mnemonic*) und Maschinencodes ist üblicherweise bijektiv, allerdings macht der Atmel-Assembler hier eine Ausnahme.
Da unterschiedliche Prozessoren unterschiedliche Befehlssätze unterstützen, ist bei Assembler *nicht* grundsätzlich die Möglichkeit gegeben, ein in einer Assemblersprache geschriebenes Programm für andere Prozessoren zu nutzen. Das heißt, dass üblicherweise Assembler-Programme nur schwer zwischen verschiedenen *Prozessorfamilien* portiert werden können.
Assembler ist somit *keine Hochsprache*. Da es aber sehr nah an der *Maschinenebene* angesiedelt ist, ermöglicht es, die Arbeitsweise eines Prozessors besser zu verstehen und dient dem Erlernen sehr effizienter Programmiermethoden.

9.4.1 Assemblerbefehle

Wie bereits zum Ende des letzten Abschnitts erwähnt, ist ein Assembler immer nur für einen Prozessor oder eine Prozessorfamilie gültig. Somit unterscheiden sich die einzelnen *Befehlsworte* für die einzelnen Prozessorfamilien.
Exemplarisch soll nun anhand des Befehlssatzes des Atmel AVRmega8 eine Einführung in die Assemblersprache für AVR-Prozessoren gegeben werden.

Die einzelnen Befehle des Assemblers des Atmel-Mikroprozessors sind hierbei immer 16-Bit lang. Die Position und Länge des *Codeanteils* unterscheidet sich und ist

abhängig vom einzelnen Befehl. Dabei kann ein solcher maximal zwei *Parameter* enthalten.
Da ein Mikrocontroller auf Dauer- oder zumindest zyklischen Betrieb ausgelegt ist, verfügt er zwar über Befehle, die ihn in einen Warte- oder *Sleep-Modus* versetzen, doch über keine Befehle für ein *Programmende*.
Einen einführenden Überblick gibt Abbildung 9.9.

Maschinencode	Assembler	
1110 1000 0000 0000	ldi	r16, 128
0000 1101 0000 0000	add	r16, r0
0001 1101 0001 0001	adc	r17, r1
0010 0011 0000 0001	and	r16, r17
1001 0101 0000 0101	asr	r16
0000 0000 0000 0000		

Abbildung 9.9: Gegenüberstellung von Maschinencode und Assembler

9.4.2 Ablauf eines Assemblerprogramms

Ein Assemblerprogramm läuft nach dem folgenden Schema ab: Der *Programmcounter* ist ein Zeiger, der auf den nächsten auszuführenden Befehl verweist. Nach dem Einschalten oder nach einem Reset des Prozessors steht dieser auf 0000.
Der Befehl, der sich an dieser Stelle befindet, wird gelesen (*fetch*). Im zweiten Schritt wird der Befehl ausgeführt (*execute*). Der Programmcounter wird anschließend um eins erhöht und zeigt auf den nächsten Befehl. Bei einem linearen Programm wäre dies der Befehl 0001. Dieser Vorgang wiederholt sich fortwährend.

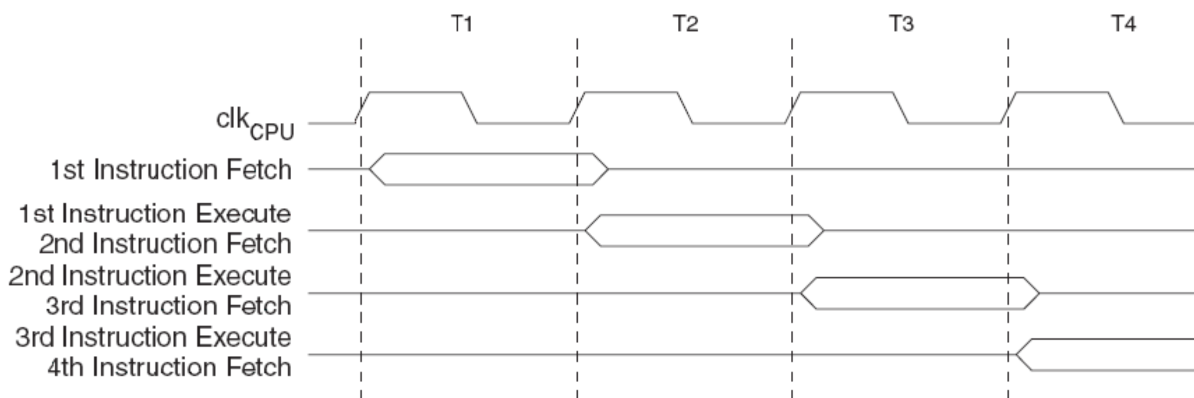

Abbildung 9.10: Taktzyklen eines Programms

Im Rahmen einer sehr einfachen Pipelining-Strategie wird der nächste Befehl bereits zur *Ausführungszeit* des vorhergehenden Befehls gelesen.
Abbildung 9.10 zeigt den Ablauf eines Programms mit Lade- und Ausführungszyklen.

Betrachtet man dabei einen Takt genauer, stellt man fest, dass innerhalb der Ausführungszeit eines solchen Befehls die *Register fetch*, *ALU-Operation* und *write*

back sequentiell ausgeführt werden.

Zu Beginn werden die benötigten Operanden in die Register geladen, dann die *ALU-Berechnung* durchgeführt und anschließend das Ergebnis in das entsprechende Register zurückgeschrieben. (Dies ist abhängig vom verwendeten Befehl; siehe hierzu Abschnitt 9.6.) Die Abbildung 9.11 zeigt den Ablauf grafisch.

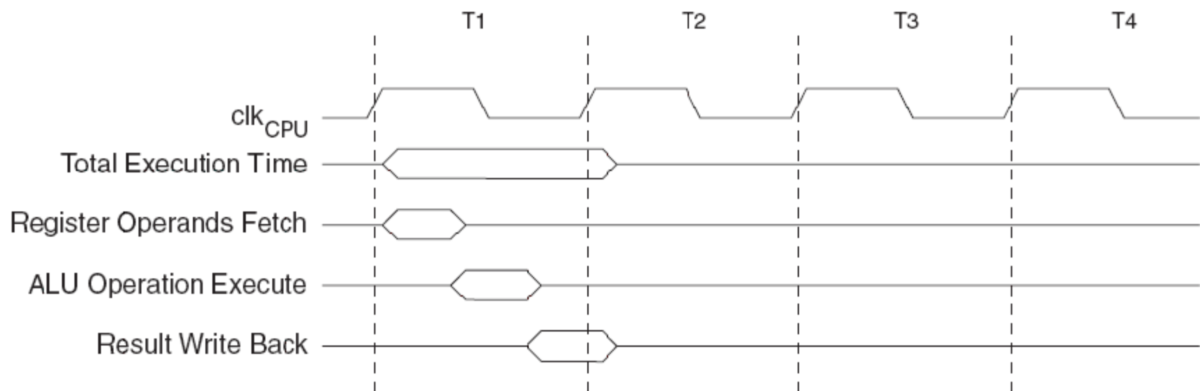

Abbildung 9.11: Sequentieller Ablauf von fetch, ALU-Operation und write back innerhalb eines Taktes

9.5 Adressierungsarten

Der *Maschinencode* eines Atmel Mikrocontrollers lässt sich in bis zu drei Bereiche aufgliedern: den Code für den eigentlichen *Befehl*, sowie Codierungen für maximal zwei *Operanden* (vgl. Abbildung 9.12). Abhängig vom Befehlscode werden die Codierungen für die Operanden verschieden interpretiert. Man spricht hier von verschiedenen *Adressierungsarten*. Abweichend von vielen anderen Assemblersprachen werden die Adressierungsarten bei der vorgestellten Architektur nicht durch *Symbole* an den Operanden, sondern durch abweichende Mnemonics codiert.

Maschinencode	**Assembler**	
1110 1000 0000 0000	ldi	r16, 128
0000 1101 0000 0000	add	r16, r0
0001 1101 0001 0001	adc	r17, r1

Abbildung 9.12: Aufbau des Maschinencodes des Atmel Mikrocontrollers am Beispiel einiger ausgewählter Befehle

9.5.1 Sofort-Operanden-Befehle

Die angegebenen Bits werden als Wert interpretiert, der unmittelbar zur Verfügung steht.

LDI Rd,K   1110 KKKK dddd KKKK

Abbildung 9.13: `LDI Rd, K` und zugehöriger Maschinencode

Im Beispiel werden die Bits 5 bis 8 und 13 bis 16 als Zahl aufgefasst, die Bits 9 bis 12 geben eine *Registernummer* an, in die dieser Wert geladen werden soll. Werden die Bits eines Maschinencodes direkt als Wert interpretiert, spricht man von einem *Sofort-Operanden-Befehl*. Sofort-Operanden werden in der Atmel-Befehlsübersicht mit einem großen `K` gekennzeichnet.

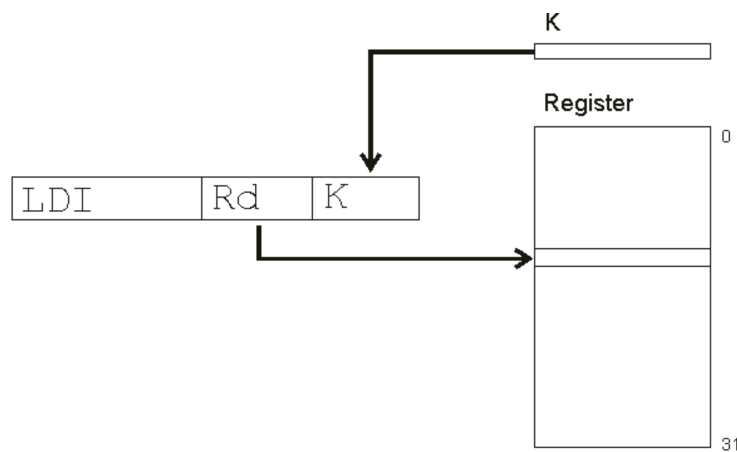

Abbildung 9.14: Sofort-Operanden-Befehl (`LDI`)

9.5.2 Direkte Adressierung

Die Codierung wird als Adresse im Speicher aufgefasst. Der eigentliche Operand wird zunächst aus dieser Speicheradresse geladen bzw. abschließend in diese Speicheradresse geschrieben.

LDS Rd,k   1001 000d dddd 0000
kkkk kkkk kkkk kkkk

Abbildung 9.15: `LDS Rd, k` und zugehöriger Maschinencode

Bei diesem Befehl handelt es sich um einen *Doppelwortbefehl*. Die Bits des zweiten Wortes werden als Adresse im Speicher interpretiert, aus der der Operand gelesen wird. Geben die Bits eines Maschinencodes eine Speicheradresse an, in der der Wert steht, so spricht man von *direkter Adressierung*. Kennzeichen für die direkte Adressierung in der Atmel-Befehlsübersicht ist ein kleines k.

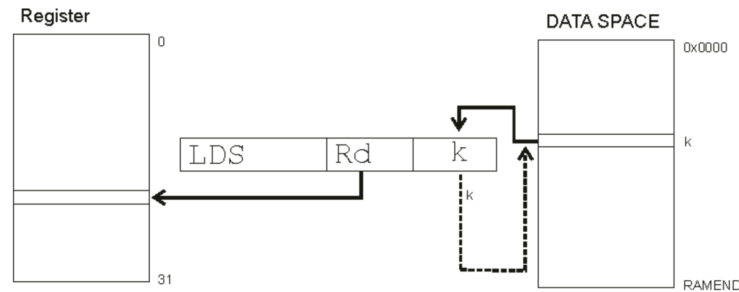

Abbildung 9.16: Direkte Adressierung am Beispiel des Befehls `LDS`

9.5.3 Relative Adressierung

Bei der *relativen Adressierung* werden die angegebenen Bits der Codierung als Adresse relativ zum aktuellen Wert des Programmcounters aufgefaßt. Da der Atmel AVR-Mikrocontroller auf der Harvard-Architektur basiert, ist es nicht möglich, Daten aus einer Adresse relativ zum Programmcounter zu laden. Das einzige Beispiel einer relativen Adressierung findet sich daher bei den *Sprungbefehlen*.

RJMP k **1100 kkkk kkkk kkkk**

Abbildung 9.17: `RJMP k` und zugehöriger Maschinencode

Der neue Wert des Programmcounters berechnet sich aus `PC = PC + 1 + k`. Kennzeichen in der Atmel Befehlsübersicht ist wiederum ein kleines `k`. Bei den Sprungbefehlen des AVRmega8 ist nur der relative Sprung oder Sprung über ein Indexregister möglich.

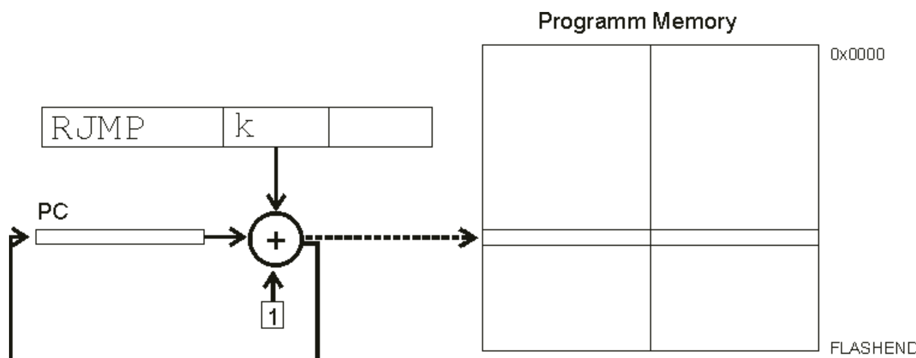

Abbildung 9.18: Relativer Sprung mittels `RJMP` zur relativen Adressierung

9.5.4 Indexregister

Bei der *Adressierung über ein Indexregister* wird der Wert des angegebenen Indexregisters als Adresse aufgefasst. Der Operand wird aus dieser Speicheradresse bezogen.

Im Beispiel wird der aktuelle Wert des Z-Registers als Adresse interpretiert. Der Wert, der im Speicher an der entsprechenden Stelle steht, wird in das Register Rd geladen. Der AVR-Mikrocontroller unterstützt drei verschiedene Indexregister (`X`, `Y` und `Z`), zum Teil

LD Rd, Z 1000 000d dddd 0000

Abbildung 9.19: `LD Rd, Z` und zugehöriger Maschinencode

mit verschiedenen Funktionen. Im Rahmen dieser Vorlesung wird nur das Indexregister `Z` betrachtet.

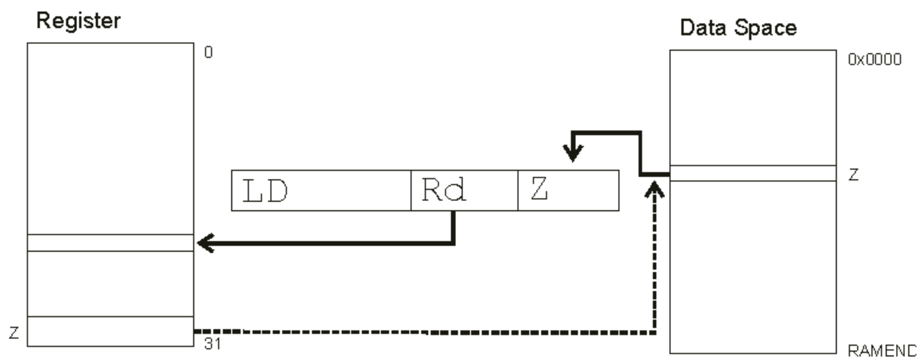

Abbildung 9.20: Der Befehl `LD` als Beispiel für die Adressierung über Indexregister

9.5.5 Indexregister mit Offset

Die *Adressierung über Indexregister mit Offset* gleicht der Adressierung über das Indexregister, jedoch wird zunächst ein Offset auf das Indexregister addiert. Das Resultat wird als Adresse interpretiert, aus der der Operand bezogen wird.

LDD Rd, Z+q 10q0 qq0d dddd 0qqq

Abbildung 9.21: `LDD Rd, Z+q` und zugehöriger Maschinencode

Im Beispiel wird der Wert von `q` zu `Z` hinzuaddiert und der Operand aus der Adresse `Z+q` im Speicher bezogen. Im AVR Assembler wird deutlich, dass die Adressierung über das Indexregister eine Sonderform der Adressierung über Indexregister mit Offset darstellt. Die Mnemonics sind zwar unterschiedlich, im Opcode ist `LD` jedoch ein `LDD` mit einem Offset von `0`. Offsets werden in der AVR Befehlsübersicht mit einem `q` gekennzeichnet.

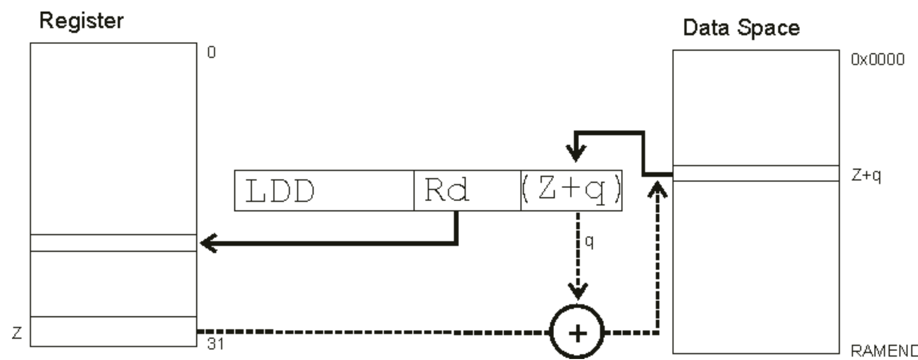

Abbildung 9.22: Laden über Indexregister mit Offset (LDD)

9.6 Untergliederung der Assemblerbefehle

In einem *Digitalrechner* sind verschiedene Arten von Maschinenbefehlen implementiert:

- *Datentransferbefehle*
  Befehle zum Transport von Daten zwischen Speicherzellen des Hauptspeichers oder anderer Speicher (z.B. des *Stack*) und *Registern* der *CPU*.

- *Arithmetische Operationen*
  Addition, *Subtraktion*, *Multiplikation*, *Division*.

- *Logische Operationen*
  Bitweise logische Verknüpfung (z.B. *OR*, *AND*) von zwei Operanden

- *Vergleichsbefehle*
  Testen des Inhalts zweier Register auf Gleichheit (*Compare*)

- *Bit-Befehle*
  Setzen, Löschen und Lesen einzelner Bits im Akku sowie *bitweises Vergleichen*.

- Status-Befehle
  Setzen, Löschen und Lesen einzelner Bits des Statuswortes.

- *Verzweigebefehle*
  Beeinflussung des Programmablaufs durch *bedingte* und *unbedingte Sprünge* (Jump, Call, Skip).

- Befehle zur *Programmablaufsteuerung*
  Befehle wie der Warte-Befehl gehören zu dieser Kategorie.

- Befehle zur *Unterbrechungsverarbeitung*
  Verarbeitung von Unterbrechungen durch innere Ereignisse (programmierte Unterbrechung) oder äußere Ereignisse (Interruptsignal). *Interrupts* spielen bei der Steuerung des Datenflusses von und zu externen Geräten eine wichtige Rolle. Der AVR unterstützt diverse Interrupt-Arten. Diese werden aber im Rahmen der Vorlesung nicht betrachtet.

9.6.1 Datentransferbefehle

Mittels *Datentransferbefehlen* können Wertzuweisungen zwischen einzelnen Registern, aber auch zwischen Registern und angegliedertem Speicher und der I/O stattfinden.

Datenverschiebung zwischen Registern (`MOV`)

Um eine Information von einem Register in ein anderes zu verschieben, muss der Assemblerprogrammierer den Assemblerbefehl ***MOV*** nutzen. Dieser erfordert noch zwei weitere Parameter, nämlich aus welchem Register gelesen (***Rr***) und in welches geschrieben (***Rd***) werden soll.

`MOV Rd, Rr` Rd←Rr

Unmittelbares Laden (`LDI`)

Mittels des Assemblerbefehls ***LDI*** ist es möglich, unmittelbar eine *Konstante* in ein Register zu laden. Hierzu müssen die Parameter `K` für die definierte Konstante und `Rd` für das *Zielregister* bestimmt werden. Dieses muss beim Befehl `LDI` eines der Register ***R16-R31*** sein. Die Folge ist ein sofortiges Laden in das Register `Rd`.

`LDI Rd, K` Rd←K

Direktes Laden aus dem Speicher (`LDS`)

Zudem ist es möglich, ein Datum direkt aus dem Speicher zu laden. Hierzu wird als Parameter des Befehls `LDS` die Speicheradresse `k` benötigt, aus der gelesen werden soll, sowie das Register `Rd`, in das geschrieben werden soll.

`LDS Rd, k` Rd←(k)

Indirektes Laden (`LD`)

Darüber hinaus unterstützt der betrachtete Mikrocontroller zwei Arten des indirekten Ladens. Beim ersten Befehl wird der Inhalt eines Registers in ein anderes verschoben. Hierbei handelt es sich um eine Speicheradresse `Z` des Datenspeichers, die ausgewertet wird und deren Inhalt dann in das vorgesehene Register `Rd` geschrieben wird.

`LD Rd, Z` Rd←Z

Indirektes Laden mit Offset (`LDD`)

Beim zweiten Befehl wird auf die ermittelte Speicheradresse ein Wert aufaddiert. Dies ermöglicht das Lesen von Daten ab einer bestimmten Adresse.
Das Datum der auf diese Weise ermittelten Speicheradresse `(Z+q)` wird dann in das Register `Rd` übertragen.

`LDD Rd, Z+q` Rd←(Z+q)

Speichern (STS, ST und STD)

Das Speichern von Informationen wird durch die Befehle ***STS***, ***ST***, ***STD*** sichergestellt, die sich analog zu den Lade-Befehlen verhalten.

- *direktes Speichern*

  STS k, Rr   (k)←Rr

- *indirekten Speichern*

  ST Z, Rr   (Z)←Rr

- *indirekten Speichern mit Verschiebung*

  STD Z+q, Rr   (Z+q)←Rr

Lesen und Schreiben auf Ports (IN und OUT)

Sofern man mittels des Assemblers für den Atmel Mikrocontroller *Ports* auslesen möchte, so geschieht dies mit dem Befehl ***IN***. Hierbei wird der Wert des Ports in das Register Rd übertragen.

IN Rd, P   Rd←P

Umgekehrt kann man auch den Wert eines Registers in einen Port schreiben. Hierzu ist der Assemblerbefehl ***OUT*** zu nutzen.

OUT P, Rr   P←Rr

Ports werden in diesem Zusammenhang benötigt, um Zugriff auf *E/A-Funktionen* zu erlangen . Zudem geschieht der Datenaustausch des *Timers*, des *Comparator*s, des *A/D-Wandlers* und des Stackpointers über Ports.

Schreiben auf und Lesen vom Stack PUSH und POP

Um auf den Stack zu schreiben, wird der Assemblerbefehl ***PUSH*** genutzt, der als Parameter das zu sichernde Register Rr enthält. Dieses wird auf dem Stack abgelegt. Anschließend wird der *Stackpointer* dekrementiert. Dieser zeigt nun auf die nächst niedrigere Adresse.

PUSH Rr   Stack←Rr, Stackpointer←Stackpointer - 1

Um vom Stack zu lesen, wird der Assemblerbefehl ***POP*** genutzt. Hier wird zunächst der Stackpointer inkrementiert und im Anschluss das Element vom Stack in das angegebene Register Rd verschoben.

POP Rd   Stackpointer←Stackpointer + 1, Rd←Stack,

Beachtet werden muss aber in jedem Fall, dass vor der Verwendung des Stackpointers zuerst seine *Initialisierung* stattfinden muss, damit dieser einen gültigen Wert hat und nicht durch die *Stackoperationen* andere gespeicherten Werte (Variablen etc.) überschrieben werden. Zudem muss der Programmierer dafür Sorge tragen, dass vor jedem Aufruf des Befehls POP der Befehl PUSH ausgeführt wurde.

9.6.2 Arithmetische Operationen

Um arithmetische Operationen durchzuführen, sind im Falle des Atmel-Mikroprozessors unter anderem folgende Operationen als Befehle implementiert.

Addition (ADD)

Der Assemblerbefehl ***ADD*** summiert die Register `Rd` und `Rr` und speichert das Ergebnis in `Rd`.

```
ADD Rd, Rr      Rd←Rd + Rr
```

Subtraktion (SUB)

Um ein Register `Rr` von einem Register `Rd` zu subtrahieren, wird der Befehl ***SUB*** genutzt. Dieser speichert das Ergebnis im Register `Rd`.

```
SUB Rd, Rr      Rd←Rd - Rr
```

Multiplikation (MUL)

Die Multiplikation (***MUL***) verhält sich analog zur Addition und Subtraktion.

```
MUL Rd, Rr      R1:R0←Rd × Rr
```

Linksshift des Registers Rd (LSL)

Um den Inhalt eines Registers nach links zu shiften, stellt der Assembler des Mikrocontrollers den Befehl ***LSL*** bereit. Die *Bit-Stelle* einer jeden Stelle des Registers wird um eins erhöht. Die Bit-Stelle `0` wird mit einer `0` gefüllt.

```
LSL Rd      Rd(n + 1)←Rd(n), Rd(0)←0
```

Rechtsshift des Registers Rd (LSR)

Der Rechtsshift mittels ***LSR*** verläuft analog. Jede Bit-Stelle des Registers wird auf die nächst kleinere umgesetzt. Das führende Bit `Rd(7)` wird mit einer `0` gesetzt.

```
LSR Rd      Rd(n)←Rd(n + 1), Rd(7)←0
```

Inkrementieren und Dekrementieren des Registers (INC und DEC)

Mittels dieser Befehle ist eine einfache Manipulation der Werte des Registers möglich. Soll der Wert um eins erhöht werden, so geschieht dies durch den Aufruf des Befehls ***INC***. Umgekehrt ist eine Dekrementierung durch Verwendung des Befehls ***DEC*** möglich.

```
INC Rd      Rd←Rd + 1
DEC Rd      Rd←Rd - 1
```

9.6.3 Logische Operationen

Neben den *arithmetischen Verknüpfungen* können auch logische Operationen zur Manipulation des Inhalts eines Registers zum Einsatz kommen.

Bitweise Und-Verknüpfung (AND)

Soll eine bitweise Und-Verknüpfung zweier *8-Bit-Register* stattfinden, wird der Befehl ***AND*** genutzt. So werden jeweils entsprechende Bit-Stellen miteinander verknüpft und bei gleichem Inhalt einer Bit-Stelle eine 1 im Register `Rd` gesetzt. Bei verschiedenem Inhalt wird eine 0 gesetzt.

```
AND Rd, Rr      Rd←Rd AND Rr
```

Bitweise Oder-Verknüpfung/Exklusiv-Oder-Verknüpfung (OR und EOR)

Analog geschieht die bitweise *Oder-Verknüpfung* und *Exklusiv-Oder-Verknüpfung*.

```
 OR Rd, Rr      Rd←Rd OR Rr
EOR Rd, Rr      Rd←Rd XOR Rr
```

Bitweises Negieren (COM)

Das bitweise Negieren eines Registers `Rd` geschieht in diesem Zuge durch *Invertieren* der einzelnen Bits. Hierzu dient der Befehl ***COM***.

```
COM Rd      Rd←$FF - Rd
```

9.6.4 Vergleichsbefehle (CP und CPI)

Um zu prüfen, ob in zwei Registern gleich große Werte enthalten sind, kann dies über den Befehl ***CP*** des Assemblers abgefragt werden. Somit wird intern der Wert des Registers `Rr` vom Wert des Registers `Rd` subtrahiert und auf `0` geprüft.

```
CP Rd, Rr      Rd - Rr
```

Zudem kann der Wert eines Registers auch mit einer *Konstanten* `K` verglichen werden. Dies geschieht analog zum Vergleich mit einem zweiten Register über den Befehl ***CPI***.

```
CPI Rd, K      Rd - K
```

Zu beachten ist in beiden Fällen, dass es sich beim Compare um eine Subtraktion handelt, deren Ergebnis nicht gespeichert wird, sondern nur die entsprechenden Statusregister gesetzt werden.

9.6.5 Bitbefehle (SBR, CBR, BSET und BCLR)

Assemblerbefehle dieser Gruppe ermöglichen das Setzen und Löschen einzelner Bits in Registern.

So kann zum Beispiel durch den Befehl ***SBR*** ein einzelnes oder mehrere Bits im Register gesetzt werden indem eine `OR`-Verknüfung mit einer entsprechenden Variablen durchgeführt wird.

```
SBR Rd, K      Rd←Rd OR K
```

Das Löschen entsprechender Bits im Register geschieht durch Verwendung des Befehls ***CBR***.

```
CBR Rd, K      Rd←Rd AND (0xFF - K)
```

Um speziell im *Statusregister* Bits zu setzen oder zu löschen, müssen die Befehle ***BSET*** und ***BCLR*** verwendet werden.
Beim Befehl `BSET` wird über den Befehl `SREG` an der Stelle `s` eine `1` gesetzt; zum Löschen wird an der Stelle `s` eine `0` gesetzt.

```
BSET s      SREG(s)←1
BCLR s      SREG(s)←0
```

9.6.6 Verzweigebefehle

Um neben rein *sequenziellen Programm*en auch solche mit *Verzweigung*en zu unterstützen, bietet der Assembler des Mikrocontrollers der Firma Atmel *Verzweigebefehle*. Dabei unterscheidet man zwischen zwei Arten, nämlich der Verwendung von Unterprogrammen und dem Verzweigen durch Sprünge.

Eine Befehlssequenz kann als Unterprogramm aus dem eigentlichen (Haupt-)Programm ausgelagert werden. Das Unterprogramm kann dann von beliebiger Stelle des *Hauptprogramm*s aus aufgerufen werden.
Die Verwendung von Unterprogrammen macht das Hauptprogramm übersichtlicher. Außerdem wird Speicherplatz gespart, falls das Unterprogramm mehrfach aufgerufen wird.

Beim Sprung in das Unterprogramm muss der Wert des *Befehlszähler*s zwischengespeichert werden. Das Zwischenspeichern wird automatisch vom Sprungbefehl (z.B. `RCALL`) durchgeführt. Beim Verlassen des Unterprogramms wird dieser Wert dann wieder in den Befehlszähler geschrieben, damit das *Hauptprogramm* an der richtigen Stelle fortgesetzt wird. Den *Rücksprung* und das Setzen des Befehlszählers erledigt der `RET`-Befehl.

Für das Zwischenspeichern des Befehlszählers verwendet der `RCALL`-Befehl den *Stack*. Durch das *LIFO*-Konzept des Kellers sind dann auch geschachtelte *Unterprogrammaufrufe* möglich, d.h. ein Unterprogramm kann wiederum ein Unterprogramm aufrufen.
Dabei ist es auch möglich, dass sich ein Unterprogramm wiederholt selbst aufruft. Dies

nennt man *Rekursion* (vgl. Abschnitt 9.8).

Darüber hinaus ist es möglich, im Programmablauf durch Sprünge zu verzweigen. Hierbei wird aber nicht, wie oben beschrieben, die Rücksprungadresse gesichert. Es ist also somit nur möglich, zu einer definierten *Sprungmarke* bzw. zu einem Befehl, auf den der *Programmcounter* dann zeigt, zu gelangen.
Sprungmarken sind hierbei Speicherzellen mit einem symbolischen Namen, der definiert und einer Speicherzelle zugeordnet wurde.

Relativer Aufruf eines Unterprogramms (`RCALL`)

Um den Aufruf eines Unterprogramms zu ermöglichen, wird der Programmcounter inkrementiert und auf dem Stack gesichert. Anschließend wird ein relativer Sprung (siehe relativer Sprung, Abschnitt 9.6.6) durchgeführt.

`RCALL k` `Stack←PC + 1`, `PC←PC + k + 1`

Rücksprung aus dem Unterprogramm (`RET`)

Um anschließend wieder aus dem Unterprogramm zurückzukehren und den „sequenziellen“ Programmablauf aufzunehmen, wird der gesicherte *Programmcounter* wiederhergestellt. Dies geschieht mittels des Befehls ***`RET`***. Der Programmcounter zeigt hierbei auf die nächste auszuführende Befehlszeile.
Der Programmierer muss aber darauf achten, dass er vor dem Aufruf des Befehls `RET` den Befehl `RCALL` oder `ICALL` ausgeführt hat.

`RET` `PC←Stack`

Relativer Sprung (`RJMP`)

Mit dem Befehl ***`RJMP`*** kann ein relativer Sprung durchgeführt werden. Der Programmcounter wird in diesem Fall nicht nur um 1 inkrementiert, sondern zusätzlich um den Wert `k` erhöht.

Mit relativer Adressierung können Programme so geschrieben werden, dass sie beliebig im Speicher verschiebbar sind und die Adressangaben trotzdem immer stimmen. Diese Eigenschaft wird für Programmteile benötigt, die als Teil anderer Programme (die in beliebigen Speicherbereichen liegen können) ablaufen sollen.

`RJMP k` `PC←PC + k + 1`

Bedingter Sprung (`BREQ` und `BRLO`)

Darüber hinaus ist es möglich, im Programmablauf zu springen, wenn ein Registerwert gleich 0 ist. Dies ist mit dem Befehl ***`BREQ`*** möglich, der testet, ob das *Zero-Flag* gesetzt wurde (Gleichheit). In diesem Fall wird ein relativer Sprung durchgeführt.

`BREQ k` `if (Z = 1) then PC←PC + k + 1`

Analog kann auch über das *Carry-Flag* (Übertragsbit) getestet werden, ob ein Wert kleiner ist als ein anderer und in diesem Fall verzweigen. Diese Funktionalität wird über den Assemblerbefehl *BRLO* bereitgestellt.

`BRLO k      if (C = 1) then PC←PC + k + 1`

Indirekter Sprung (IJMP)

Beim *indirekten Sprung* wird aus dem Register *Z* der neue Wert des Programmcounters gelesen. Hierbei handelt es sich um eine Speicheradresse des Programmspeichers. Diese wird gelesen und das Programm wird an dieser Stelle fortgeführt. Das heißt, dass bei der indirekten Adressierung der Inhalt der im Maschinenbefehl angegebenen Adresse nicht als der *Operand* selbst, sondern wiederum als Adresse interpretiert wird, in der dann der Operand steht.

`IJMP      PC←Z`

Indirekter Aufruf eines Unterprogramms (ICALL)

Mittels des Befehls *ICALL* kann ein Unterprogramm indirekt angesprungen werden. Der Programmcounter wird auf dem Stack gesichert und ein indirekter Sprung (s.o.) durchgeführt. Nach Ablauf des Unterprogramms kann der Programmcounter aus dem Stack geholt und für den weiteren Programmablauf gesetzt werden.

`ICALL      Stack←PC + 1, PC←Z`

9.7 Programmierung von Schleifen

In höheren Programmiersprachen können Schleifen in der Form

`WHILE <Bedingung> DO <Anweisung>`

programmiert werden.
In Assembler können diese *Schleifen* mit Hilfe von bedingten Sprüngen und *Marken* realisiert werden. Hierzu wird während des Programmablaufs ein Wert in ein Register geschrieben, der anschließend in einer *Schleife* beispielsweise dekrementiert werden soll.
Die Schleife selbst ist durch eine entsprechende *Sprungmarke* gekennzeichnet.
Soll, wie bereits erwähnt, der Wert dekrementiert werden, geschieht dies über den Befehl DEC. Der Inhalt des Registers wird um eins erniedrigt und das Z-Flag (*Zero-Flag*) berechnet. Sofern der Wert des Z-Flags gleich 0 ist, wird die Schleife durch den anschließenden Befehl BREQ mit der entsprechenden Sprungmarke (im unteren Beispiel end; siehe Abbildung 9.23) verlassen. Ist der Wert größer 0, wird ein relativer Sprung zum Beginn der Schleife durchgeführt und erneut durchlaufen. Dies geschieht so lange, bis das Z-Flag gleich 1 ist, d.h. der Wert des Registers gleich 0.
Dann wird der Code der Sprungmarke ausgeführt, über die die Schleife verlassen wurde. Im Beispiel wird der Befehl NOP ausgeführt. Die Abbildung 9.23 zeigt eine solche Schleife und die Sprungmarken.

```
        LDI r16,10  ; lade 10 in R16
loop:   NOP         ; Schleifeninhalt
        DEC r16     ; Verringere R16 um 1
        BREQ end    ; Falls 0: zu end springen
        RJMP loop   ; zu loop springen
end:    NOP         ; Code nach der Schleife
```

Abbildung 9.23: Beispielhafte Implementierung einer Schleife in Assembler

9.8 Rekursion

Wie bereits in Kapitel 9.6.6 erwähnt, ist es mit Assembler möglich, ein *Unterprogramm* von einem anderen Unterprogramm aufrufen zu lassen. Einen speziellen Fall stellt hierbei der Aufruf eines Unterprogramms durch sich selbst dar.
Definiert der Assemblerprogrammierer nun zur Berechnung eines Problems das Unterprogramm so, dass zum Beispiel der *Funktionswert* f(n) durch die Berechnung des Funktionswertes von f(n - 1) abhängt und dies fortgeführt wird, bis eine gegebene Abbruchbedingung dieser Verschachtelung erreicht ist, spricht man von *Rekursion*.
Das Unterprogramm wird somit solange erneut aufgerufen, bis ein durch den Programmierer definierter Wert zur Berechnung herangezogen werden kann.
Erfolgt keine korrekte Definition der *Abbruchbedingung*, so berechnet das Programm zu viele Durchläufe des Unterprogramms und beendet unter Umständen die Berechnung nie. Daher ist auf eine korrekte Abbruchbedingung zu achten, die erreicht werden kann.

Soll zum Beispiel das Produkt der Zahlen von 1 bis n berechnet werden, so ist dies mathematisch definiert durch:

$$fak(n) = 1 \cdot 2 \cdot 3 \cdot 4 \cdot ... \cdot (n-1) \cdot n$$

Die *rekursive Beschreibung* der Produktfunktion (Fakultät) sieht vor, dass das Produkt aller Zahlen von 1 bis n durch das Produkt aller vorhergehenden, also des Produktes der Zahlen von 1 bis n-1, multipliziert mit der aktuellen Zahl erhalten werden kann.

$$fak(n) = fak(n-1) \cdot n$$

Um eine korrekte Abbruchbedingung zu erhalten, wird die *Abbruchbedingung* definiert als

$$fak(1) = 1$$

Somit ist die Produktfunktion rekursiv definiert als:

$$fak(\mathrm{n}) = \begin{cases} 1, & wenn\ \mathrm{n} = 1 (Abbruchbedingung) \\ fak(n-1) \cdot n, & wenn\ \mathrm{n} > 1 (Rekursionsschritt) \end{cases}$$

Das Produkt der Zahlen von 1 bis 4 berechnet sich beispielsweise wie folgt:

$$\begin{aligned} fak(4) &= fak(3) \cdot 4 \\ &= (fak(2) \cdot 3) \cdot 4 \\ &= ((fak(1) \cdot 2) \cdot 3) \cdot 4 \\ &= 1 \cdot 2 \cdot 3 \cdot 4 \\ &= 24 \end{aligned}$$

Um eine gleichwertige Beschreibung der Produktfunktion in Assembler zu erhalten, muss der Code so aufgebaut sein, dass innerhalb einer Unterfunktion die Zerlegung des Problems in ein *Teilproblem* geschieht (hier fak(n) = fak(n-1) · n). Dies darf aber nur so lange geschehen, bis der Rekursionsanfang erreicht ist. Dann ist das Unterprogramm durch einen Sprung zu verlassen. Die ermittelten Faktoren werden dann miteinander multipliziert. Nachfolgend ist ein solches Assemblerprogramm für den Atmel AVRmega8 aufgelistet:

```
.include "m8def.inc"

        LDI r16, LOW(RAMEND)     ; LOW-Byte der obersten RAM-Adresse
        OUT SPL, r16
        LDI r16, HIGH(RAMEND)    ; HIGH-Byte der obersten RAM-Adresse
        OUT SPH, r16

        LDI r16,4
        RCALL fak
        NOP
fak:    PUSH r16
        CPI r16,1
        BREQ done
        DEC r16
        RCALL fak
done:   POP r17
        MUL r16,r17
        MOV r16,r0
        RET
```

Abbildung 9.24: Berechnung der Fakultät mittels Assembler

9.9 Befehlsübersicht

Die nachfolgende Tabelle gibt noch einmal zusammenfassend einen Überblick über die vorgestellten Befehle des Atmel AVRmega8 in alphabetischer Ordnung.

Die Tabelle listet die Syntax, die Operation, den Wertebereich der Operanden, die Veränderung des Programmcounters sowie optional des Stackpointers und den 16-bit Opcode auf.

Syntax	Operation 16-bit Opcode	Operanden	Programmcounter (opt. Stack-Pointer)
ADD Rd, Rr	Rd←Rd + Rr	0≤d≤31, 0≤r≤31	PC←PC + 1
	0000 11rd dddd rrrr		
AND Rd, Rr	Rd←Rd • Rr	0≤d≤31, 0≤r≤31	PC←PC + 1
	0010 00rd dddd rrrr		
BCLR s	SREG(s)←0	0≤s≤7	PC←PC + 1
	1001 0100 1sss 1000		
BREQ k	if Rd = Rr (Z = 1) then PC←PC + k + 1, else PC←PC + 1	-64≤k≤+63	PC←PC + 1
	1111 00kk kkkk k001		
BRLO k	if Rd < Rr (C = 1) then PC←PC + k + 1, else PC←PC + 1	-64≤k≤+63	PC←PC + k + 1; PC←PC + 1, if condition is false
	1111 00kk kkkk k000		
BSET s	SREG(s)←1	0≤s≤7	PC←PC + 1
	1001 0100 0ssss 1000		
CBR Rd, K	Rd←Rd • ($FF - K)	16≤d≤31, 0≤K≤255	PC←PC + 1
	0111 $\bar{K}\bar{K}\bar{K}\bar{K}$ dddd $\bar{K}\bar{K}\bar{K}\bar{K}$		
COM Rd	Rd←$FF - Rd	0≤d≤31	PC←PC + 1
	1001 010d dddd 0000		
CP Rd, Rr	Rd - Rr	0≤d≤31, 0≤r≤31	PC←PC + 1
	0001 01rd dddd rrrr		
CPI Rd, K	Rd - K	16≤d≤31, 0≤K≤255	PC←PC + 1
	0011 KKKK dddd KKKK		
DEC Rd	Rd←Rd - 1	0≤d≤31	PC←PC + 1
	1001 010d dddd 1010		
EOR Rd, Rr	Rd←Rd ⊕ Rr	0≤d≤31, 0≤r≤31	PC←PC + 1
	0010 01rd dddd rrrr		
ICALL	PC(15:0)←Z(15:0)		See Operation. Stack: Stack← PC + 1; SP←SP - 2
	1001 0101 0000 1001		
IJMP	PC(15:0)←Z(15:0)		See Operation.
	1001 0100 0000 1001		

Syntax	Operation	Operanden	Programmcounter (opt. Stack-Pointer)
	16-bit Opcode		
IN Rd, A	Rd←I/O(A)	0≤d≤31, 0≤A≤63	PC←PC + 1
	1011 0AAd dddd AAAA		
INC Rd	Rd←Rd + 1	0≤d≤31	PC←PC + 1
	1001 010d dddd 0011		
LD Rd, Z	Rd←(Z)	0≤d≤31	PC←PC + 1
	1000 000d dddd 0000		
LDD Rd, Z+q	Rd←(Z+q)	0≤d≤31, 0≤q≤63	PC←PC + 1
	10q0 qq0d dddd 0qqq		
LDI Rd, K	Rd←K	16≤d≤31, 0≤K≤255	PC←PC + 1
	1110 KKKK dddd KKKK		
LDS Rd, k	Rd←(k)	0≤d≤31, 0≤k≤65535	PC←PC + 2
	1001 000d dddd 0000 kkkk kkkk kkkk kkkk (Doppelwort)		
LSL Rd	Rd(n + 1)←Rd(n), Rd(0)←0	0≤d≤31	PC←PC + 1
	0000 11dd dddd dddd (intern ADD Rd, Rd)		
LSR Rd	Rd(n)←Rd(n + 1), Rd(7)←0	0≤d≤31	PC←PC + 1
	1001 010d dddd 0110		
MOV Rd, Rr	Rd←Rr	0≤d≤31, 0≤r≤31	PC←PC + 1
	0010 11rd dddd rrrr		
MUL Rd, Rr	R1:R0←Rd × Rr	0≤d≤31, 0≤r≤31	PC←PC + 1
	1001 11rd dddd rrrr		
NOP			PC←PC + 1
	0000 0000 0000 0000		
OR Rd, Rr	Rd←Rd ∨ Rr	0≤d≤31, 0≤r≤31	PC←PC + 1
	0010 10rd dddd rrrr		
OUT A, Rr	I/O(A)←Rr	0≤r≤31, 0≤A≤63	PC←PC + 1
	1011 1AAr rrrr AAAA		
POP Rd	Rd←Stack	0≤d≤31	PC←PC + 1; SP←SP + 1
	1001 000d dddd 1111		

Syntax	Operation / 16-bit Opcode	Operanden	Programmcounter (opt. Stack-Pointer)
PUSH Rr	Stack←Rr	0≤r≤31	PC←PC + 1; SP←SP − 1
	1001 001r rrrr 1111		
RCALL k	PC←PC + k + 1	-2048≤k<2048	PC←PC + k + 1; SP←SP - 2; Stack←PC + 1
	1101 kkkk kkkk kkkk		
RET	PC(15:0)←Stack		See Operation; SP←SP + 2
	1001 0101 0000 1000		
RJMP k	PC←PC + k + 1	-2048≤k<2048	PC←PC + k + 1
	1100 kkkk kkkk kkkk		
SBR Rd, K	Rd←Rd ∨ K	16≤d≤31, 0≤K≤255	PC←PC + 1
	0110 KKKK dddd KKKK		
ST Z, Rr	(Z)←Rr	0≤r≤31	PC←PC + 1
	1000 001r rrrr 0000 (intern: STD Z+0, Rr)		
STD Z+q, Rr	(Z+q)←Rr	0≤r≤31, 0≤q≤63	PC←PC + 1
	10q0 qq1r rrrr 0qqq		
STS k, Rr	(k)←Rr	0≤r≤31, 0≤k≤65535	PC←PC + 2
	1001 001r rrrr 0000 kkkk kkkk kkkk kkkk (Doppelwort)		
SUB Rd, Rr	Rd←Rd - Rr	0≤d≤31, 0≤r≤31	PC←PC + 1
	0001 10rd dddd rrrr		

Tabelle 9.2: Übersicht über alle vorgestellten Assemblerbefehle des Atmel AVRmega8

9.10 Weiterführende Links

Um einen tieferen Einblick in die Programmierung mit Assembler (von *AVR*) zu erhalten, bieten sich die folgenden Links an:

- Tutorial:
  - http://www.avr-asm-tutorial.net/
  - http://www.mikrocontroller.net/articles/AVR-Tutorial
- Chip-Dokumentation:
  - http://www.atmel.com/dyn/resources/prod_documents/doc2486.pdf
- Sprach-Dokumentation:
  - http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf
- Development-Umgebung:
  - http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725
- Dokumentation zur Entwicklungsumgebung:
  - http://www.atmel.com/dyn/resources/prod_documents/novice.pdf
  - http://www.atmel.com/dyn/resources/prod_documents/doc2510.pdf

Literaturverzeichnis

[Am90] Ameling, W.
Digitalrechner – Grundlagen und Anwendungen. Technische Informatik 1
Vieweg, Braunschweig

[Bl92] Blieberger, J., Schildt, G.-H., Schmid, U., Stöckler, S.
Informatik
Springer-Verlag, Wien New York

[Bo96] Borucki, L.
Digitaltechnik
B. G. Teubner, Stuttgart, 4. Auflage

[Co92] Coy, W.
Aufbau und Arbeitsweise von Rechenanlagen. Eine Einführung in Rechnerarchitektur und Rechnerorganisation für das Grundstudium der Informatik
Vieweg, Braunschweig, 2. Auflage

[Kl83] Klar, R.
Digitale Rechenautomaten
Sammlung Göschen 2050, de Gruyter, Berlin

[Ma01] Märtin, C.
Rechnerarchitekturen
Fachbuchverlag Leipzig/Carl Hanser Verlag, München

[Me82] Mendelson, E.
Boolesche Algebra und logische Schaltungen — Theorie und Anwendung
Schaums Outline, McGraw-Hill, Hamburg

[Mu99] Müller-Schloer, C., Schallenberger, B., et al.
Vom Arbeitsplatzrechner zum ubiquitären Computer
VDE Verlag, Berlin

[Re87] Rembold, U.
Einführung in die Informatik für Naturwissenschaftler und Ingenieure
Hanser Verlag, München

[Sc92] Schiffmann, W., Schmitz, R.
Technische Informatik 1: Grundlagen der digitalen Elektronik
Springer-Lehrbuch, Springer-Verlag Berlin

[Sc92a] Schiffmann, W., Schmitz, R.
Technische Informatik 2: Grundlagen der Computertechnik
Springer-Lehrbuch, Springer-Verlag Berlin

[Ti93] Tietze, U., Schenk, Ch.
Halbleiter-Schaltungstechnik
Springer-Verlag Berlin, 10. erweiterte Auflage

[Wa84] Waldschmidt, E., Walter K.
Grundzüge der Informatik I
Bibliographisches Institut, Mannheim

[Wa84a] Waldschmidt, E., Walter K.
Grundzüge der Informatik II
Bibliographisches Institut, Mannheim

Index

8-Bit-RISC-Prozessor, **2**
8-Bit-Register, **19**

A/D-Wandlers, **17**
Abbruchbedingung, **23**
ADD, **18**
Addition, **15**, **18**
Adressierung über ein Indexregister, **13**
Adressierung über Indexregister mit Offset, **14**
Adressierungsarten, **11**
ALU, **3**, **6**
ALU-Berechnung, **11**
ALU-Operation, **10**
Analog-Digitalwandler, **3**
AND, **15**, **19**
Arithmetische Operationen, **15**
arithmetischen Verknüpfungen, **19**
Assembler, **1**
Assemblersprache, **1**
Atmel AVR, **1**
Ausführungszeit, **10**
Ausgabeports, **3**
AVR, **28**

BCLR, **20**
Befehl, **11**
Befehlsregister, **1**
Befehlsworte, **9**
Befehlszähler, **20**
Bit-Befehle, **15**
Bit-Stelle, **18**
Bitweise logische Verknüpfung, **15**
Bitweise Und-Verknüpfung, **19**
Bitweises Negieren, **19**
bitweises Vergleichen, **15**
BREQ, **21**
BRLO, **22**
BSET, **20**

Carry-Flag, **22**
CBR, **20**
Codeanteils, **9**
COM, **19**
Comparator, **17**
Compare, **15**
Control Unit, 1
CP, **19**
CPI, **19**
CPU, 1, **15**

Datenbus, **3**
Datenspeicher, **1**
Datentransferbefehle, **15, 16**
DEC, **18**
Digitalrechner, **15**
direkter Adressierung, **12**
direktes Speichern, **17**
Division, **15**
Doppelwortbefehl, **12**

E/A-Funktionen, **17**
Eingabeports, **3**
execute, **10**
Exklusiv-Oder-Verknüpfung, **19**

fetch, **10**
Funktionswert, **23**

Harvard-Architektur, **1**
Hauptprogramm, **20**
Hochsprache, **9**

ICALL, **22**
IN, **17**
INC, **18**
indirekten Speichern, **17**
indirekten Speichern mit Verschiebung, **17**
indirekten Sprung, **22**
Initialisierung, **17**
Instruction Decoder, **3**
Inter-Integrated Circuit, **4**
Interrupts, **15**
Invertieren, **19**

Keller, **7**
Kellerspeicher, **7**
Komparator, **4**

Konstante, **16**
Konstanten, **19**

Last In, First Out, **7**
LDI, **16**
LIFO, **7**, **20**
Linksshift, **18**
Logische Operationen, **15**
LSL, **18**
LSR, **18**

Marken, **22**
Maschinenbefehle, **1**
Maschinencode, **11**
Maschinenebene, **9**
Maschinensprache, **1**, **9**
Mnemonic, **9**
MOV, **16**
MUL, **18**
Multiplikation, **15**, **18**

Oder-Verknüpfung, **19**
Operand, **22**
Operanden, **11**
Operationscodes, **1**
OR, **15**
OUT, **17**

Parameter, **10**
POP, **7**, **17**
Ports, **17**
Programmablaufsteuerung, **15**
Programmcounter, **10**, **21**
Programmende, **10**
Programmspeicher, **1**
Prozessorarchitektur, **9**
Prozessorfamilien, **9**
PUSH, **7**, **17**

R16, **16**
R31, **16**
Rücksprung, **20**
Rücksprungadresse, **7**
Rd, **16**, **20**
Rechenwerk, **3**
Rechtsshift, **18**
Register, **7**, **10**
Registern, **15**
Registernummer, **12**
Rekursion, **21**, **23**
Rekursionsschritt, **23**
rekursive Beschreibung, **23**
relativen Adressierung, **13**
RET, **21**
RJMP, **21**
Rr, **16**

SBR, **20**
Schleife, **22**
Schleifen, **22**
sequenziellen Programm, **20**
Serial Peripheral Interface, **4**
Sleep-Modus, **10**
Sofort-Operanden-Befehl, **12**
Sprünge, **15**, **21**
Sprungbefehlen, **13**
Sprungmarke, **21, 22**
SRAM, **3**
ST, **17**
Stack, **3**, **7**, **15**, **20**
Stack-Pointer, **3**, **8**
Stackoperationen, **17**
Stackpointer, **17**
Stapel, **7**
Statusregister, **6**, **20**
STD, **17**
Steuereinheit, **1**
Steuerwerk, **3**, **6**
STS, **17**
SUB, **18**
Subroutine, **7**
Subtraktion, **15**, **18**
Symbole, **11**
Symbole des Maschinencodes, **9**

Teilproblem, **24**
Timer, **3**
Timers, **17**

Universal Asynchronous Receiver Transmitter, **3**
Unterbrechungsverarbeitung, **15**
Unterprogramm, **20**, **23**
Unterprogrammaufrufe, **20**

Vergleichsbefehle, **15**
Verzweigebefehle, **15**, **20**

Verzweigung, **20**
von Neumann-Rechner, **1**

write back, **11**

X-, Y- und Z-Register, **7**

Z, **22**
Zero-Flag, **21, 22**
Zielregister, **16**