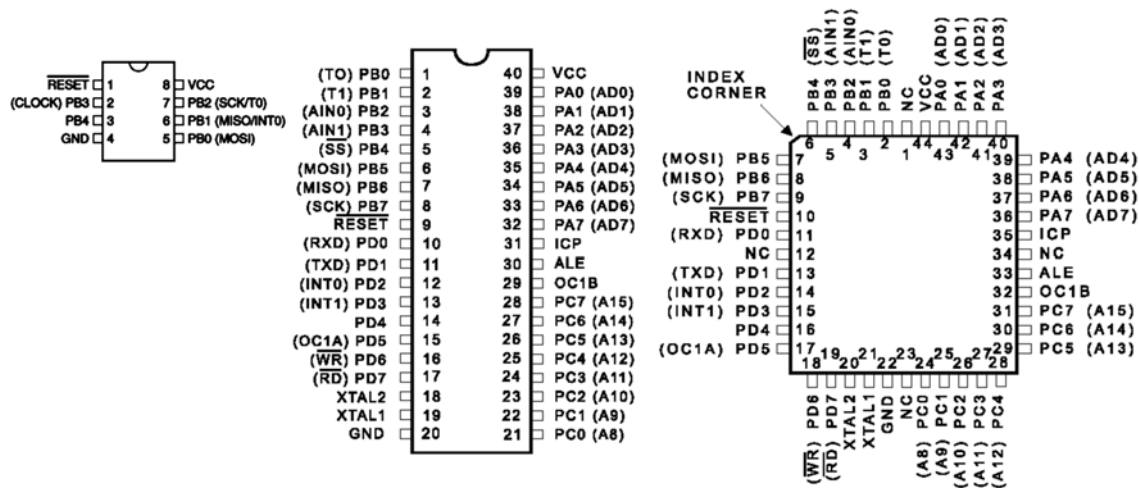


## **Einführung in die Mikrocontroller-Programmierung am Beispiel Atmel AVR**



**Abb. 1** Mikrocontroller in verschiedenen Bauformen (Atmel)

Mikrocontroller (Abb. 1) sind programmierbare Universalrechner, die zum Einsatz in Embedded Systems optimiert sind. Kennzeichnende Merkmale:

- die gängigen Mikrocontroller sind an sich nichts Revolutionäres – es sind zwar kleine und kostengünstige, im Grunde (Architektur, Programmiermodell usw.) aber herkömmliche Rechenmaschinen,
- kostenoptimierte Auslegung (möglichst wenig Siliziumfläche, möglichst kleine Gehäuse),
- Einbau in eine Anwendungsumgebung,
- Programmierung auf den jeweiligen Anwendungszweck hin (Controller wird durch Programmierung zur Einweckmaschine),
- Programme typischerweise vom Nutzer nicht änderbar,
- wir dürfen programmieren (Narrenfreiheit, potentielle Funktionsvielfalt),
- wir müssen programmieren (Zwang zur Rückführung der anwendungsseitig geforderten Informationswandlungen auf elementare Programmschritte; Serialisierung der Informationsverarbeitung, Problemlösung entspricht nicht mehr dem anwendungsseitigen Datenflußschema).

### *Anwendungsfälle:*

- Realzeitraster im ms-Bereich oder langsamer,
- es stehen keine anderweitigen Forderungen entgegen (extremes Stromsparen, EMV, Sicherheitsvorschriften).

### *Infrastruktur:*

- Stromversorgung,
- Takt,
- Rücksetzen,
- E-A-Anschaltung,
- ggf. Speicheranschaltung.

*Auswahlkriterien:*

1. Kosten,
2. frei nutzbare E-A-Anschlüsse (Ports),
3. eingebaute Peripherie,
4. Programmspeicherkapazität,
5. Datenspeicherkapazität,
6. Registersatz,
7. Leistungsvermögen der Befehlsliste<sup>1)</sup>,
8. Geschwindigkeit,
9. technische Einsatzbedingungen (Gehäuse, Speisespannung, Treibvermögen, Takterzeugung, Programmierverfahren usw.),
10. Kompatibilität,
11. Bezugsmöglichkeiten,
12. Entwicklungsunterstützung.

Der typische moderne (kleine) Mikrocontroller – mit dem wir uns zunächst ausschließlich beschäftigen wollen – hat eingebaute Programm- und Datenspeicher. Somit stehen alle E-A-Anschlüsse zum Lösen der Anwendungsaufgabe zur Verfügung (Abb. 2). Seine Festwertspeicher sind Flash-ROMs oder EEPROMs. Sie können in der Einsatzumgebung programmiert werden (In-System Programming ISP – kein Programmiergerät, kein UV-Licht zum Löschen).

*Lösen von Anwendungsaufgaben:*

Hard- und Software sind zunächst im Verbund zu betrachten. Wir beginnen mit der Hardware. Erstes Ziel: Zusatzbeschaltung vermeiden! Im Idealfall besteht – von der Infrastruktur (Takt, Rücksetzen, Stromversorgung) abgesehen – das System nur aus dem Mikrocontroller und der gemäß Anwendungsaufgabe anzuschließenden Hardware (Tasten, Schalter, Leuchtanzeigen, Sensoren usw.). Zweites Ziel: mit möglichst wenigen E-A-Anschlüssen auskommen (genauer: mit dem “kleinst-möglichen” Gehäuse (Kostenfrage)).

*Oftmals unvermeidlich:*

- Hardware zur Signalwandlung und-aufbereitung,
- Treiberstufen.

In diesem Rahmen kann (und sollte) getrickst und optimiert werden (Analogmultiplexer, Schieberegister-Interfaces usw.). Maßgebend ist stets Kostenrechnung „über alles“!

Externe Logikschaltungen sollten aber nicht über einzelne Gatter o. dergl. hinausgehen. CPLDs sind beträchtlich teurer als Mikrocontroller!

---

1): wir merken uns: alle Befehlslisten sind äußerst leistungsfähig (powerful), manche mehr, manche weniger.



Ergebnis: Schaltungslösung. Dokumentation: wenigstens Blockschaltbild (hinreichend detailliert).

*Beim professionellen Entwickeln:*

Jetzt schon prüfen:

- Beschaffung,
- Fertigung,
- Prüfung,
- EMV-Probleme.

Es nützt gar nichts, Bauelemente einzusetzen, die zwar gut aussehen, deren Beschaffung aber im voraussehenden Fertigungszeitraum nicht gewährleistet ist.

Entwurf muß sich auch wirklich fertigen lassen! Prüfen, ob Bedingungen der Leiterplattenfertigung erfüllt werden. Es nützt nichts, Bauelemente einzusetzen, die im gegebenen Kostenrahmen nicht zu bestücken sind (Anschlußabstände, Lötverfahren usw.).

Jede Hardware braucht den CE-Kuckuck! Jetzt schon an die erforderlichen Vorkehrungen denken! (Z. B. an Platz für Filter und für Suppressordioden.)

Es kann durchaus sein, daß wegen solcher Praxis- bzw. Trivialprobleme ein an sich guter Entwurf geändert werden muß – womöglich radikal. Aber besser jetzt als später!

## 2. Was ist – als Software – zu programmieren?

Das wichtigste: die auszuführenden Funktionen wirklich verstehen: Was soll der Apparat eigentlich leisten? – Wesentliche Zusammenhänge? – Funktionelle Abhängigkeiten? – Spitzfindigkeiten? – Gelegenheiten, sich in den Fuß zu schießen (Gotchas)?

Vorgehensweise ergibt sich aus den Anforderungen. Akademische Entwurfsprinzipien nur bedingt hilfreich. Von Anfang an vorsortieren:

- was muß unbedingt sein?
- was dient nur der Schönheit? – Was kann also geopfert werden, wenn Terminvorgaben drängen oder wenn nicht mehr alles in den Speicher paßt?<sup>1)</sup>
- was hat Zeit? (Hier ist die Laufzeit gemeint. Es geht darum, ob die Programmierung laufzeitkritisch ist oder nicht.)
- wobei sind bestimmte Laufzeitvorgaben einzuhalten? – Ausführbarkeit zeitkritischer Schleifen ggf. anhand von Probeprogrammen überprüfen! (Noch ehe wir in die eigentliche Programmierung einsteigen.)

Gleich am Anfang: die zeitkritischen Programmstücke erkennen (Erfahrungs- und Intuitionssache) und zusehen (Probeprogrammierung), ob es reicht oder nicht. Hieraus ergeben sich gelegentlich Rückwirkungen auf die Hardware (Auswahl des Mikrocontrollers). Anregungen: die Applikationsschriften der Hersteller und Nutzervereinigungen (Internet). Achtung – nicht blindlings vertrauen, sondern alles selbst erproben!

---

1): wir merken uns: Termine sind immer zu knapp, Speicher immer zu klein...

*Entwicklungsplattform wählen:*

1. Programmiersprache. Nicht in der Herde mitrennen. Selbständig denken! Es gibt nicht nur C, sondern auch Forth, Basic usw. Eine Sprache wie C hat nur dann wirkliche Vorteile, wenn die Bibliotheksfunktionen ausgiebig genutzt werden– wir müssen uns weder die Arithmetik selbst schreiben noch uns über die Parameterübergabe beim Unterprogrammruf Gedanken machen. C-Programme für einfache Steuerungsabläufe sehen aus wie Assemblerprogramme, nur ist die Syntax eine Zumutung.
2. Entwicklungsumgebung (nur PC mit Programmierschnittstelle (Download-Kabel) oder PC + In-Circuit-Emulator). Viele Aufgaben sind mit einfachen Hilfsmitteln (in Hard- und Software) zu lösen– man muß sich nur zu helfen wissen...
3. Erprobungsumgebung (fertiges Starterkit, Labormuster, V-Muster des eigentlichen Systems). Handverdrahtete Labormuster kritisch bei hohen Frequenzen, Analogsignalen usw.

– Aus didaktischen Gründen (= um wirklich was zu lernen) beschränken wir uns auf die Assemblerprogrammierung –

*Elementare Festlegungen (Programmierkonventionen):*

- Belegung der E-A-Schnittstellen,
- Registerbelegung. Daran denken, daß es meist keine echten Universalregister sind (es ist nicht so, daß mit jedem Register alles geht). Arbeitsregister und Register für ggf. benötigte Sonderfunktionen freihalten.
- Arbeitsspeicherbelegung. Ggf. an den Stack denken.
- EEPROM- Belegung (Speicher mit Datenerhalt),
- Programmspeicherbelegung (gelegentlich ist an zu speichende Konstanten, an Sprungweiten usw. zu denken),
- Parameterübergabe beim Unterprogrammruf,
- Debugging-Vorkehrungen.

Alles symbolisch deklarieren (EQU-Anweisungen)!

*Das grundsätzliche Programmschema:*

1. Initialisierung,
2. Abfrage, ob etwas zu tun ist,
3. wenn ja, dann das tun, was zu tun ist,
4. zurück zu 2.

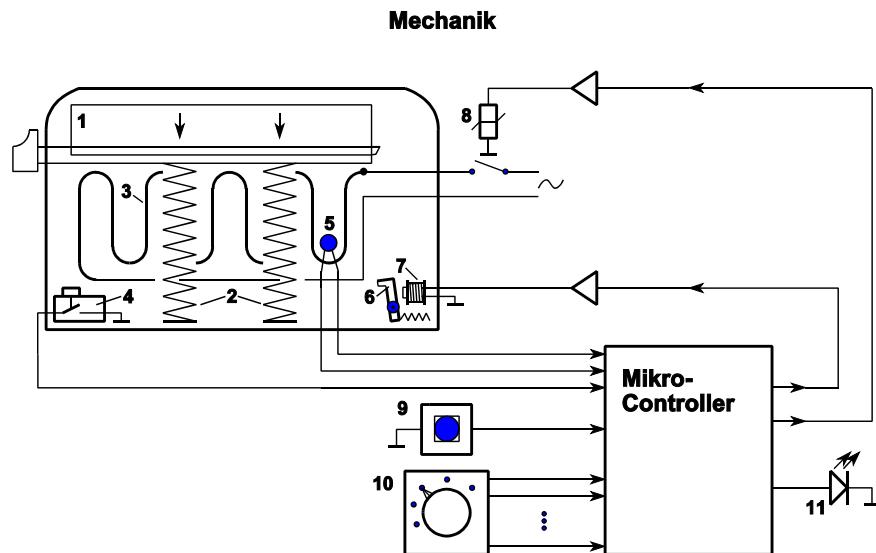
**Ein ganz einfaches Programmbeispiel**

Wir beginnen mit einem System, das jedermann nach kurzer Erläuterung versteht– und dem wir im täglichen Leben durchaus begegnen können: es handelt sich darum, einen Toaster mit einem Mikrocontroller zu steuern (Abb.n 3 bis 5).



1 – Knauf zum Herunterdrücken des Korbes; 2 – Stoptaste (zum Abbrechen des Toast-Vorgangs); 3 – Drehschalter zum Einstellen des Bräunungsgrades.

**Abb. 3** Die zu programmierende Einrichtung (Ansicht)



1 – Korb; 2 – Druckfeder; 3 – Heizwendel; 4 – Endlagenkontakt; 5 – Temperatursensor; 6 – Rastung; 7 – Auslösemagnet; 8 – Relais (oder Triac); 9 – Stoptaste (zum Abbrechen des Toast-Vorgangs); 10 – Drehschalter zum Einstellen des Bräunungsgrades; 11 – Kontrollanzeige (Leuchtdiode).

**Abb. 4** Der Toaster im Blockschaltbild

#### *Was ist zu steuern?*

Wir legen die Brotscheiben ein und drücken den Korb 1 nach unten. Er rastet in dieser Lage ein (Rastung 6). Dieser Betriebszustand wird mittels des Endlagenkontaktes 4 signalisiert. Dies bewirkt, daß der Toast-Vorgang beginnt. Um ihn zu beenden, wird der Auslösemagnet 7 erregt und somit Rastung 6 ausgelöst. Daraufhin drückt die Druckfeder 2 den Korb 1 wieder nach oben. Zur Beeinflussung des Ablaufs sind eine Stoptaste 9 (vorzeitiges Beenden) und ein Drehschalter 10 (zum Einstellen des Bräunungsgrades) vorgesehen. Das Toasten selbst beruht auf einer Erregung der Heizwendel 3. Hierzu muß das Relais 8 erregt werden.

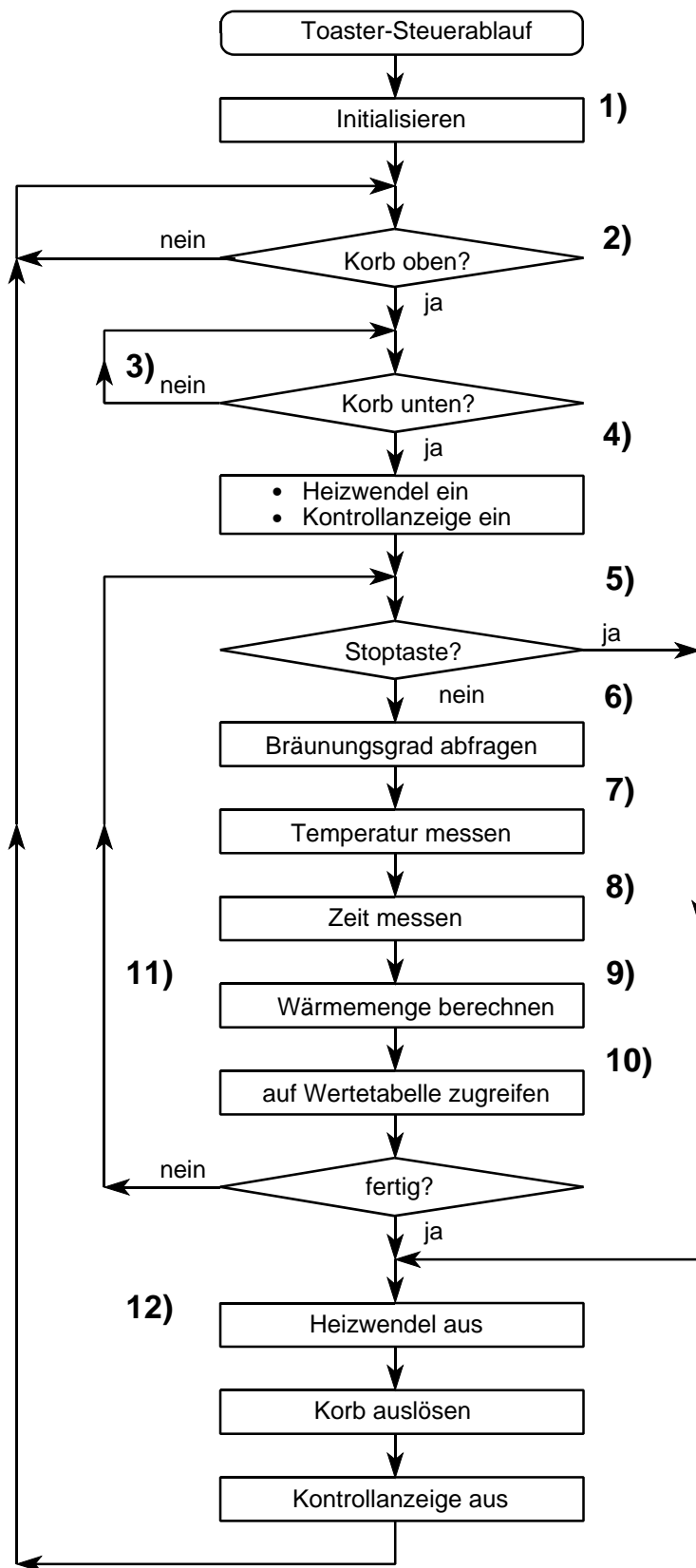
Das einfache Beispiel zeigt das Organisationsprinzip eines typischen Anwendungsprogramms:

- es wird gestartet (irgendwie muß es schließlich einmal losgehen (hier: mit dem Einschalten)),
- es richtet sich erst einmal ein (Initialisierung),
- es fragt ab, ob etwas zu tun ist,
- ist etwas zu tun, so wird dies ausgeführt,
- ist die Arbeit erledigt, wird wiederum abgefragt, ob etwas zu tun ist (Abfrage- bzw. Warteschleife).

All diese Funktionen sind aus den vergleichsweise einfachen Maschinenbefehlen gleichsam zusammenzstückeln – und das gelingt auch, sofern wir den Bedarf an Zeit und Speicherplatz zunächst vernachlässigen (Prinzip der Turing-Maschine).

*Erklärung zu Abb. 5:*

- 1) nach dem Einschalten wird alles in „Grundstellung“ versetzt (Fachbegriff: Initialisierung). Ggf. wird auch der Korb 1 ausgelöst.
- 2) der Korb 1 darf nicht in der unteren Lage eingerastet sein. Er wurde ggf. unter 1) oder 12) ausgelöst. Warten, bis der Endlagenkontakt 4 abgeschaltet hat.
- 3) Warteschleife im Ruhezustand,
- 4) mit dem Einrasten des Korbes 1 (Meldung über Endlagenkontakt 4) beginnt der Toast-Vorgang,
- 5) die Stoptaste 9 wird immer wieder abgefragt, um zu erkennen, ob der Vorgang abgebrochen werden soll,
- 6) der Drehschalter 10 wird immer wieder abgefragt, um den gewünschten Bräunungsgrad auch ggf. mitten im Ablauf ändern zu können (Bedienkomfort),
- 7) mittels Temperatursensor 5,
- 8) durch interne Zeitählung im Mikrocontroller,
- 9) die bisher umgesetzte Wärmemenge wird aus Temperatur und Zeit errechnet (und in jedem Schleifenumlauf aufsummiert). Es handelt sich um eine interne Hilfsgröße, die nicht normgerecht (als SI-Einheit J (Joule)) ermittelt werden muß.
- 10) das ist ein einfaches, oft angewendetes Prinzip: man rechnet nicht mit komplizierten Formeln, sondern man hat im Mikrocontroller eine Wertetabelle fest gespeichert, die zu jedem einstellbaren Bräunungsgrad die zugehörige Wärmemenge angibt (die Werte wurden während der Entwicklung durch Versuch bestimmt),
- 11) die Schleife des Toast-Vorgangs,
- 12) zurück zur Warteschleife – hat der Korb die untere Endlage verlassen, kann ein neuer Toast-Vorgang gestartet werden.

**Abb. 5** Der Programmablauf

**Adreßräume**

Die AVR-Mikrocontroller haben insgesamt 5 verschiedene Speichereinrichtungen und damit 5 Adreßräume:

1. Programmspeicher (Flash ROM). 16 Bits Zugriffsbreite. Speicherkapazität modellspezifisch. Obergrenze ohne Adreßverlängerung: 64k Worte (= Befehle). Mindestausstattung: einige k Worte. Obergrenze mit Adreßverlängerung: 16M Worte.
2. Datenspeicher (Data RAM). 8 Bits Zugriffsbreite. Speicherkapazität modellspezifisch. Obergrenze ohne Adreßverlängerung: 64 kBytes. Mindestausstattung: 256 Bytes. Einige Modelle unterstützen eine extern anzuschließende Speichererweiterung. Obergrenze mit Adreßverlängerung: 16 MBytes.
3. Universalregister. 8 Bits Zugriffsbreite. 32 Bytes.
4. E-A-Register. 8 Bits Zugriffsbreite. 64 Bytes. Die ersten 32 Bytes sind für Einzelbitzugriffe zugänglich. In manchen Modellen auf insgesamt 192 Bytes erweitert (über Datenspeicheradreßraum erreichbar). Belegung modellspezifisch. Steht jeweils im Datenblatt (Register Summary). Über den E-A-Adreßraum sind auch einige architekturseitige Register zugänglich.
5. EEPROM. 8 Bits Zugriffsbreite. Speicherkapazität modellspezifisch. Programmgesteuerter Zugriff. Mindestausstattung: 256 Bytes.

Die Universal- und E-A-Register sind auch über den Datenspeicheradreßraum zugänglich.

*a) herkömmliche bzw. kleinere Modelle:*

- Adressen 0...31 = 00H..1FH: Universalregister,
- Adressen 32...95 = 20H....5FH: E-A-Register (Einzelbitzugriffe: Adressen 32...63 = 20H...3FH),
- Adressen ab 96 = 60H: der eigentliche Datenspeicher.

*b) erweiterte Modelle:*

Der E-A-Adreßraum wurde auf insgesamt 192 Bytes erweitert. Davon sind lediglich die ersten 64 Bytes über E-A-Befehle erreichbar. Die verbleibenden 128 Bytes sind nur über den Datenspeicheradreßraum zugänglich.

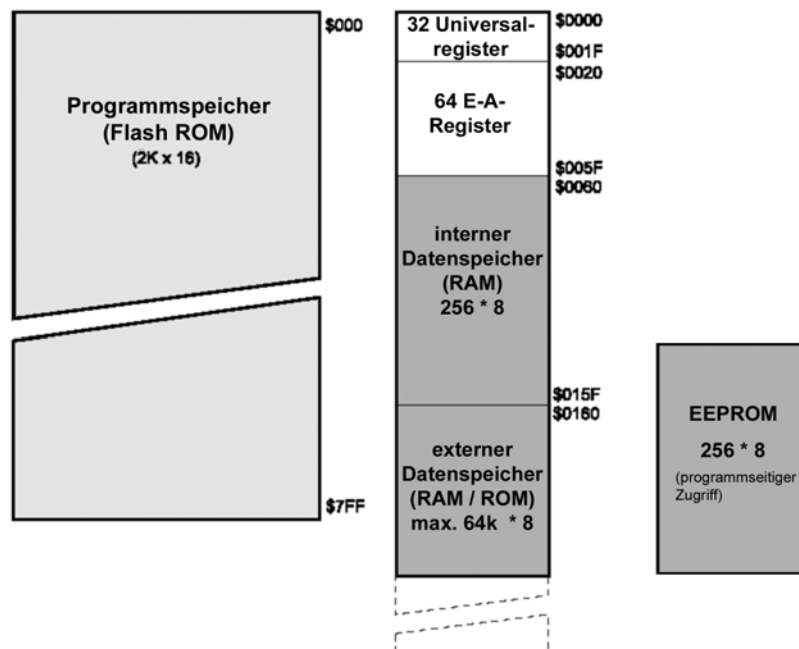
- Adressen 0...31 = 00H..1FH: Universalregister,
- Adressen 32...95 = 20H....5FH: E-A-Register der Grundausstattung (Einzelbitzugriffe: Adressen 32...63 = 20H...3FH),
- Adressen 96...255 = 60H...FFH: zusätzliche E-A-Register,
- Adressen ab 256 = 100H: der eigentliche Datenspeicher.

*Byteanordnung und Bitindizierung*

Rechtsadressierung und Rechtsindizierung. Adresse zeigt auf das niederwertige Byte. Bit 0 =  $2^0$ .

**Speicherausstattung und Adressierung am Beispiel des Atmel AVR 4414**

Aus Gründen der Überschaubarkeit beziehen wir uns auf ein älteres Modell (Abb. 6). *Zum richtigen Programmieren unbedingt das Datenmaterial des betreffenden Schaltkreises heranziehen!*



**Abb. 6** Die Speicherausstattung des AVR 4414 (nach Atmel)

*Registersatz (1): Universalregister*

7	0	Adresse
R0	\$00	
R1	\$01	
R2	\$02	
...		
R13	\$0D	
R14	\$0E	
R15	\$0F	
R16	\$10	
R17	\$11	
...		
R26	\$1A	X-Register, niederwertiges Byte
R27	\$1B	X-Register, höherwertiges Byte
R28	\$1C	Y-Register, niederwertiges Byte
R29	\$1D	Y-Register, höherwertiges Byte
R30	\$1E	Z-Register, niederwertiges Byte
R31	\$1F	Z-Register, höherwertiges Byte

**Abb. 7** Der Universalregistersatz (nach Atmel)

Nur die Register R16 bis R31 sind wirkliche Universalregister. In diesem Bereich sind die Arbeits- und Adreßregister unterzubringen.

Weitere Sondernutzungen:

- R0: im LPM-Befehl,
- R0 + R1: im SPM-Befehl.

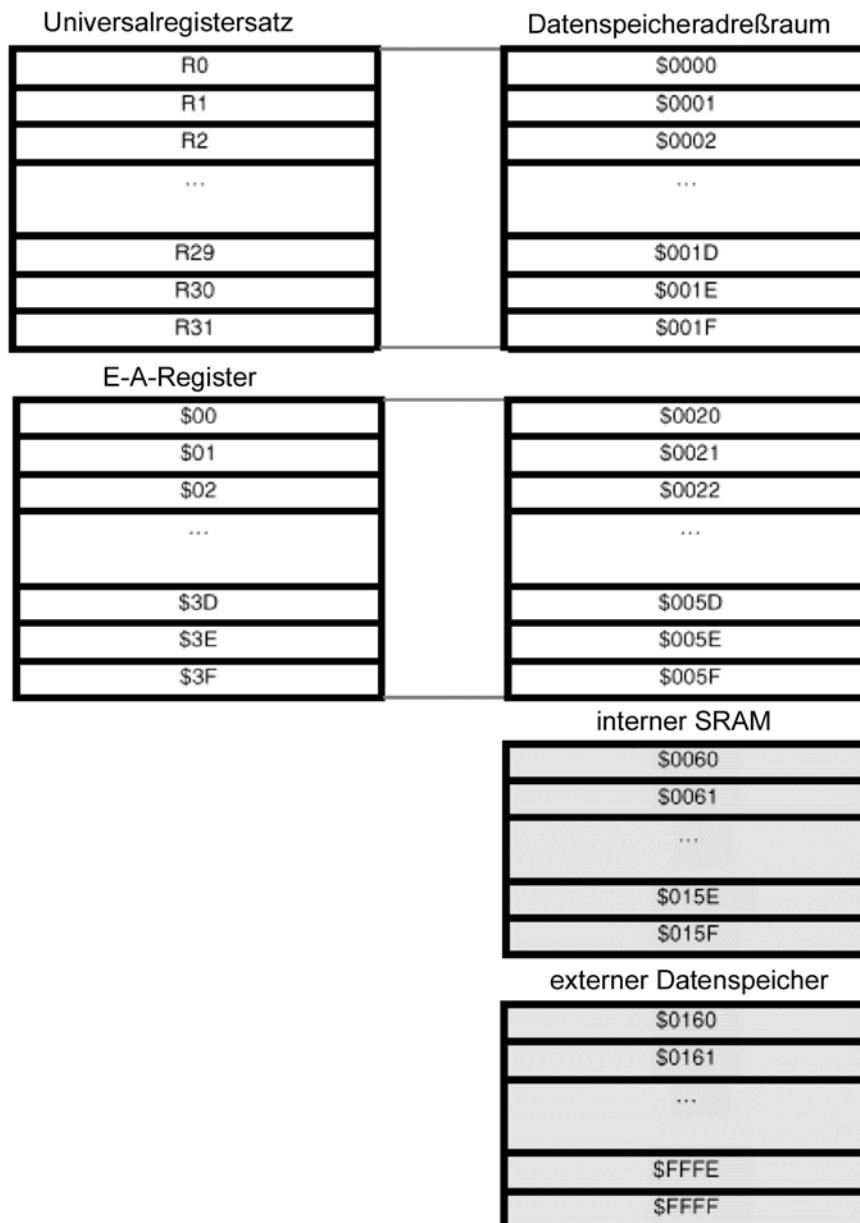
Registersatz (2): E-A-Register

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	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8
\$3D (\$5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
\$3C (\$5C)	Reserved								
\$3B (\$5B)	GIMSK	INT1	INT0	-	-	-	-	-	-
\$3A (\$5A)	GIFR	INTF1	INTF0						
\$39 (\$59)	TIMSK	TOIE1	OCIE1A	OCIE1B	-	TICIE1	-	TOIE0	-
\$38 (\$58)	TIFR	TOV1	OCF1A	OCF1B	-	ICF1	-	TOV0	-
\$37 (\$57)	Reserved								
\$36 (\$56)	Reserved								
\$35 (\$55)	MCUCR	SRE	SRW	SE	SM	ISC11	ISC10	ISC01	ISC00
\$34 (\$54)	Reserved								
\$33 (\$53)	TCCR0	-	-	-	-	-	CS02	CS01	CS00
\$32 (\$52)	TCNT0	Timer/Counter0 (8 Bit)							
\$31 (\$51)	Reserved								
\$30 (\$50)	Reserved								
\$2F (\$4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	-	-	PWM11	PWM10
\$2E (\$4E)	TCCR1B	ICNC1	ICES1	-	-	CTC1	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)	Reserved								
\$26 (\$46)	Reserved								
\$25 (\$45)	ICR1H	Timer/Counter1 - Input Capture Register High Byte							
\$24 (\$44)	ICR1L	Timer/Counter1 - Input Capture Register Low Byte							
\$23 (\$43)	Reserved								
\$22 (\$42)	Reserved								
\$21 (\$41)	WDTCR	-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0
\$20 (\$40)	Reserved								
\$1F (\$3F)	Reserved	-	-	-	-	-	-	-	-
\$1E (\$3E)	EEAR	EEPROM Address Register							
\$1D (\$3D)	EEDR	EEPROM Data Register							
\$1C (\$3C)	EEDR	-	-	-	-	-	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	-	-	-	-	-	-
\$0D (\$2D)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
\$0C (\$2C)	UDR	UART I/O Data Register							
\$0B (\$2B)	USR	RXC	TXC	UDRE	FE	OR	-	-	-
\$0A (\$2A)	UCR	RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8
\$09 (\$29)	UBRR	UART Baud Rate Register							
\$08 (\$28)	ACSR	ACD	-	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
...	Reserved								
\$00 (\$20)	Reserved								

Diese E-A-Adressen (0...31) sind für Einzelbitzugriffe zugänglich

Abb. 8 Die E-A-Register des AVR 4414 (Atmel)

Die Registersätze sind auch über den Datenspeicheradreßraum zugänglich:

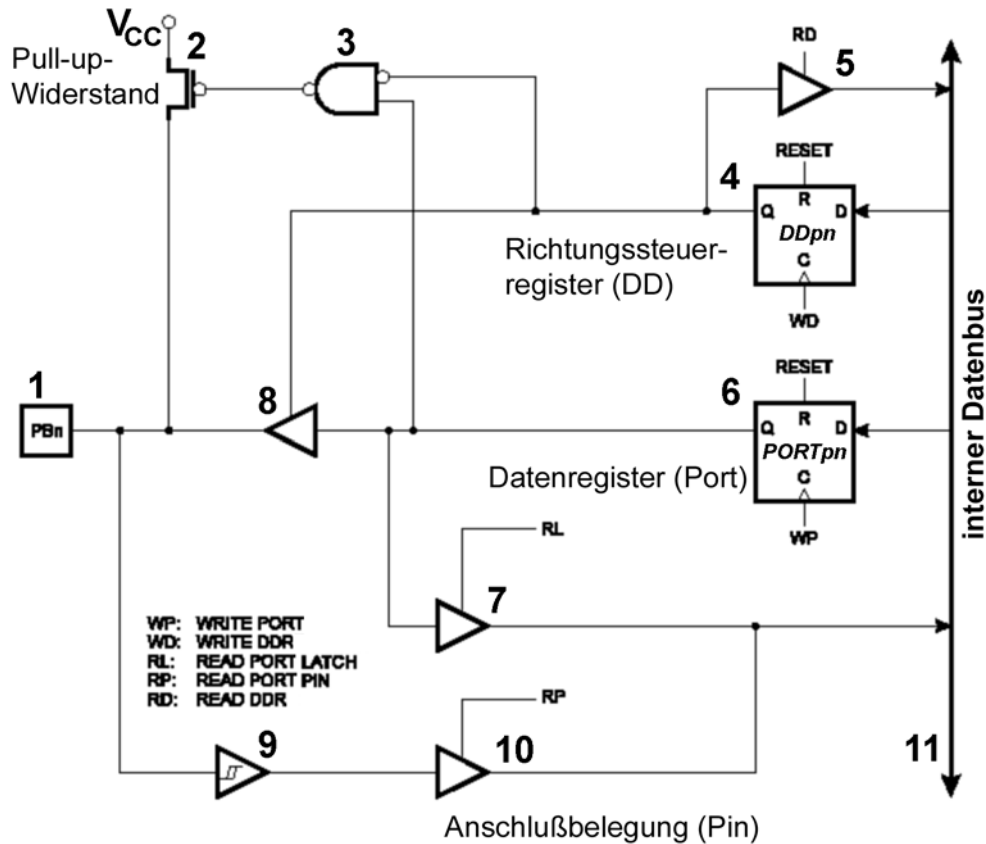


**Abb. 9** Registeradressierung über den Datenspeicheradreßraum (nach Atmel)

*Zugriffszeiten:*

- echte Registerzugriffe: 1 Taktzyklus (und zwar typischerweise alle Zugriffe gleichzeitig, die in einem Befehl zu erledigen sind). Somit dauern die meisten einschlägigen Befehle nur einen Taktzyklus.
- ein Zugriff auf den internen Datenspeicher: 2 Taktzyklen,
- ein Zugriff auf den externen Datenspeicher ohne Wartezustand: 3 Taktzyklen,
- ein Zugriff auf den externen Datenspeicher mit Wartezustand: 4 Taktzyklen.

Die Grundsaltung eines E-A-Anschlusses:



1 – Anschluß (Pin); 2 – Pull-up-Widerstand (als FET ausgeführt); 3 – Pull-up-Widerstand wird aktiviert, wenn Anschluß = Eingang und Datenbit = 1; 4 – Richtungssteuerregister (Data Direction Register DD); 5 – Rücklesetreiber für Richtungssteuerregister; 6 – Datenausgangsregister (Port-Register); 7 – Rücklesetreiber für Port-Register; 8 – Ausgangstreiber (Pin-Treiber); 9 – Schmitt-Trigger; 10 – Pin-Lesetreiber; 11 – interner Datenbus. Richtungssteuerung: DD = 0: Eingang, DD = 1: Ausgang.

**Abb. 10** Ein E-A-Anschluß im Blockschaltbild (nach Atmel)

*Asynchrone Eingangssignale und Metastabilität:*

Kein Problem. Eingangsbelegungen werden in der Hardware synchronisiert. Synchronisations-Flipflops sind vorhanden, aber hier nicht dargestellt.

*Partial Power Down (ausgeschalteter Controller in eingeschalteter Umgebung):*

Achtung! – Es gibt Controllertypen, die im ausgeschalteten Zustand die Pins eigens hochohmig schalten, so daß das Problem nicht mehr besteht. Ist aber in unserem Beispiel nicht der Fall.

*Richtwerte zur Treibfähigkeit (bei VCC = + 5 V):*

High: bis 3 mA, Low: 10 bis 20 mA (manche Ports). Auf das Kleingedruckte achten! Stichwort: Zulässiger Gesamtstrom durch Masse- bzw. Speisespannungsanschlüsse.

## Besondere Register der Prozessorarchitektur

Als E-A-Register sind programmseitig zugänglich:

- das Statusregister (SREG),
- der Stackpointer,
- die Adreßverlängerungs- bzw. Bankregister RAMPD, RAMPX, RAMPY, RAMP, EIND. Modell-spezifisch.

Adreßregister im Bereich der Universalregister:

- X-Register. 16 Bits. Register R27, R26.
- Y-Register. 16 Bits. Register R29, R28.
- Z-Register. 16 Bits. Register R31, R30.

Statusregister (SREG):

7	6	5	4	3	2	1	0
I	T	H	S	V	N	Z	C

Bit	Abkür-zung	Bezeichnung	Bedeutung
0	C	Carry Flag	Ausgangsübertrag oder Borgen
1	Z	Zero Flag	Ergebnis gleich Null
2	N	Negative Flag	Vorzeichen negativ (= Resultatbit 7)
3	V	Two's Complement Overflow	Zweierkomplement-Überlauf
4	S	Signed Flag ( $N \oplus V$ )	Vergleichsaussage $A < B$ (Zweierkomplement)
5	H	Half Carry Flag	Ausgangsübetrag aus Bit 3 nach Bit 4
6	T	Transfer Bit	Zwischenspeicher für Bittransportbefehle BLD, BST
7	I	Global Interrupt Enable/Disable	allgemeine Unterbrechungserlaubnis

Stackpointer (SP):

Muß auf Daten-RAM zeigen. Gleich zu Anfang etablieren! (Erforderlich zum Unterprogrammrufruf und zur Interruptbehandlung.)

Wachstumsrichtung des Stacks:

Abwärts (von höheren zu niedrigeren Adressen).

Prinzip der Adreßzählung:

- Push = Postdecrement,
- Pop = Präincrement.

SP zeigt stets auf das erste freie Byte im Stack.

*Stacknutzung:*

Für Unterprogrammrufruf, Unterbrechungsbehandlung und zeitweises Retten von Registerinhalten. SP ist nicht als Basisregister für gezielte Zugriffe in den Stack nutzbar (nur Push/Pop).

*Der übliche Ausweg:*

Einrichten eines zusätzlichen Parameter-Stacks mit Adreßregister X oder Y.

15	8   7	0
SP (high)		SP (low)
E-A-Register 3E		E-A-Register 3D

In Modellen mit nicht mehr als 256 Bytes Datenspeicher ist nur SP (low) implementiert.

*X-Register:*

15	8   7	0
X (high)		X (low)
R27		R26

*Y-Register:*

15	8   7	0
Y (high)		Y (low)
R29		R28

*Z-Register:*

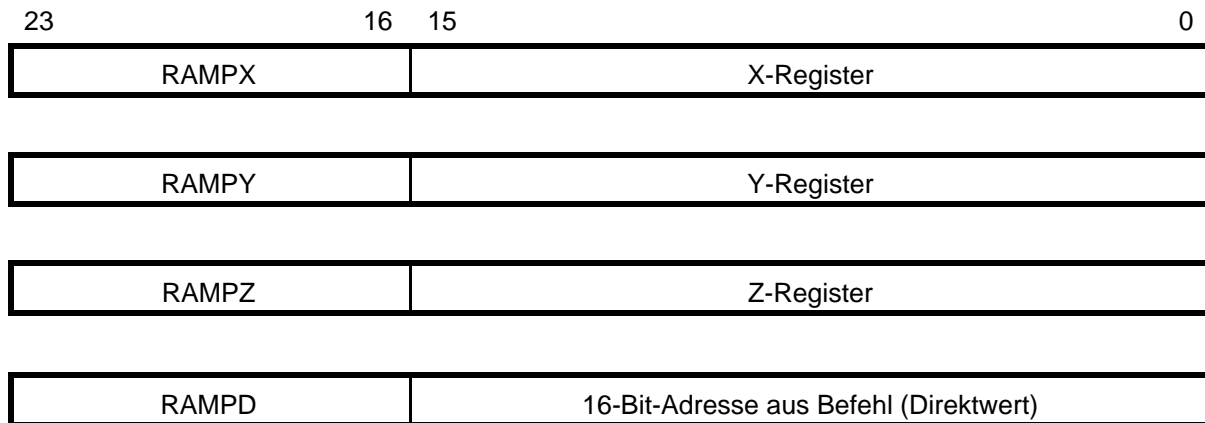
15	8   7	0
Z (high)		Z (low)
R31		R30

*Adreßverlängerung:*

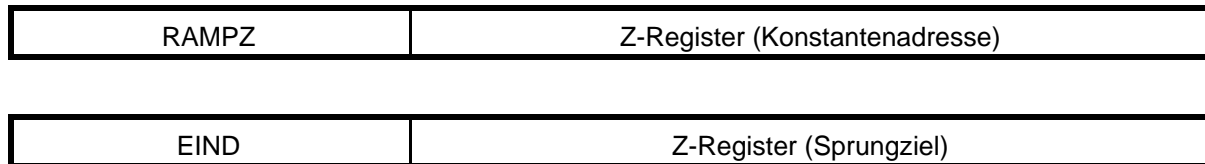
Hierfür sind Bankregister von jeweils maximal 1 Byte Länge vorgesehen. Es sind nur so viele Bits implementiert wie im jeweiligen Modell erforderlich.

Bankregister gibt es nur in Modellen mit mehr als 64k Programmspeicher bzw. mehr als 64 kBytes Datenspeicher (SRAM). Unser Beispiel (4414) hat *keine* Bankregister.

*Verlängerte Datenspeicheradresse (maximal 24 Bits):*

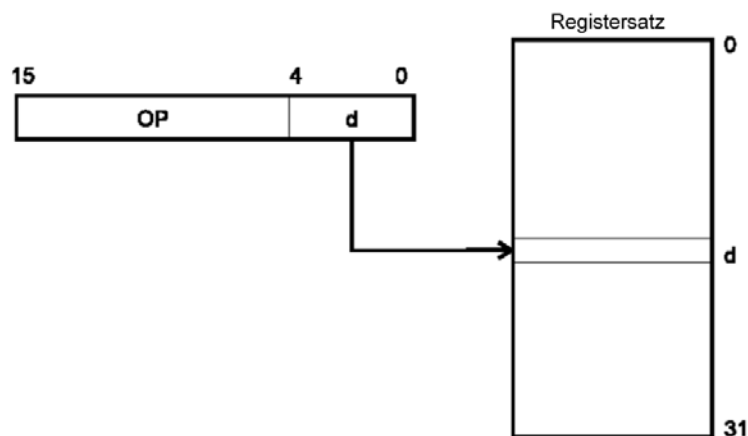


*Verlängerte Programmspeicheradresse:*



### Registeradressierung

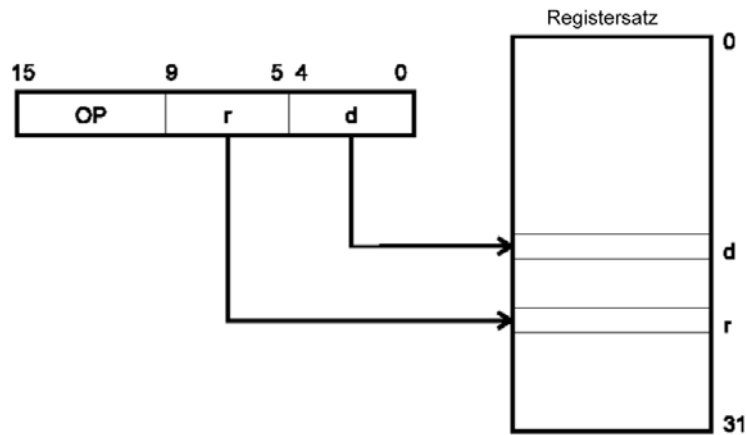
*Direktadressierung eines einzelnen Registers:*



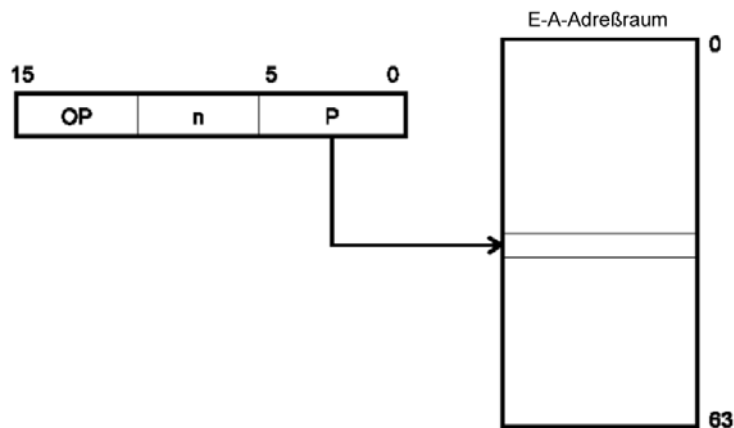
*Hinweis:*

Befehle mit Direktwerten können nur die Register R16...31 adressieren.

*Direktadressierung zweier Register:*



*Direktadressierung eines E-A-Registers:*



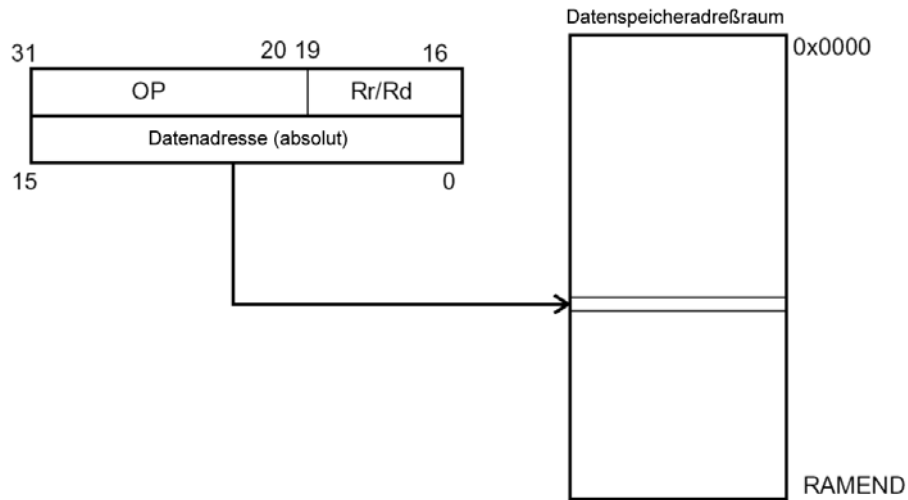
Nur die ersten 64 E-A-Register sind so zugänglich. Die in manchen Modellen vorgesehenen zusätzlichen E-A-Register (maximal 128) sind nur über Datenspeicherzugriffe erreichbar.

*Register (universelle und E-A) indirekt adressieren:*

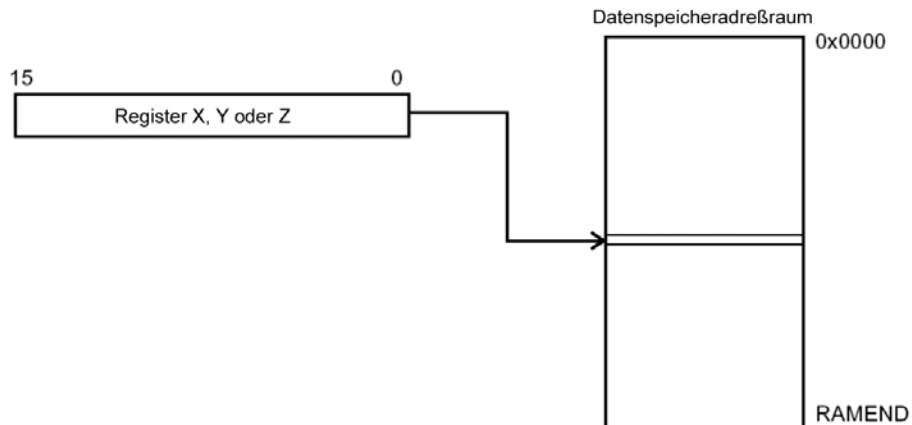
Zugang über Datenspeicheradreßraum ausnutzen.

### Datenspeicheradressierung

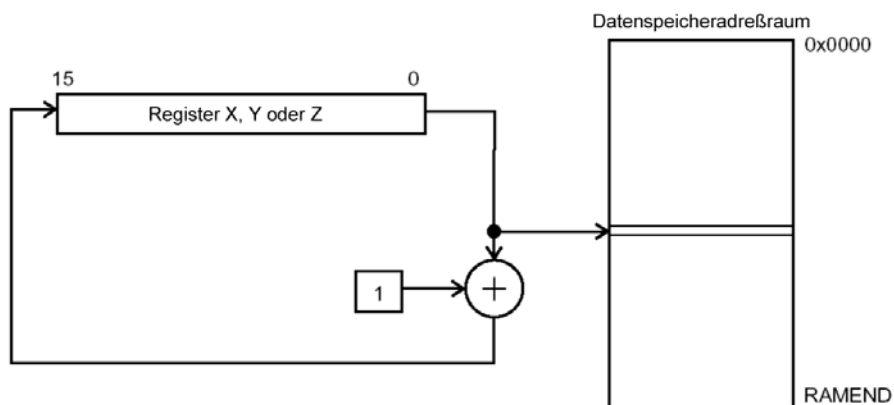
*Direktadressierung über 16-Bit-Absolutadresse:*



*Indirekte Adressierung über Adreßregister X, Y, Z:*

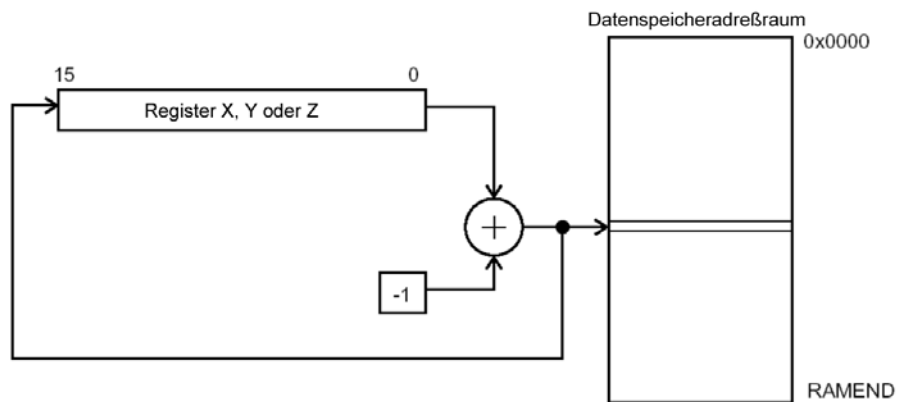


*Indirekte Adressierung mit Postincrement:*  
 Adreßerhöhung nach Zugriff (bei Stackzugriff: PUSH).

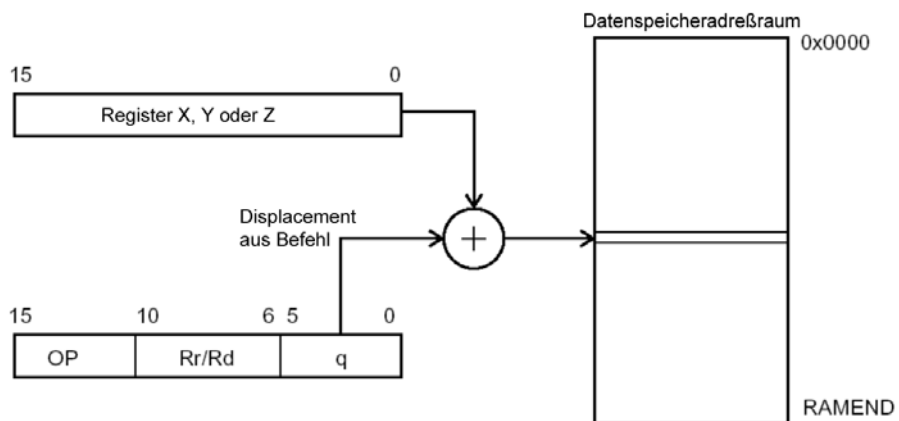


*Indirekte Adressierung mit Predecrement:*

Adreßverminderung vor Zugriff (bei Stackzugriff: POP).

*Indirekte Adressierung mit Displacement:*

Nur Adreßregister Y oder Z nutzbar. Displacement 0...+63.

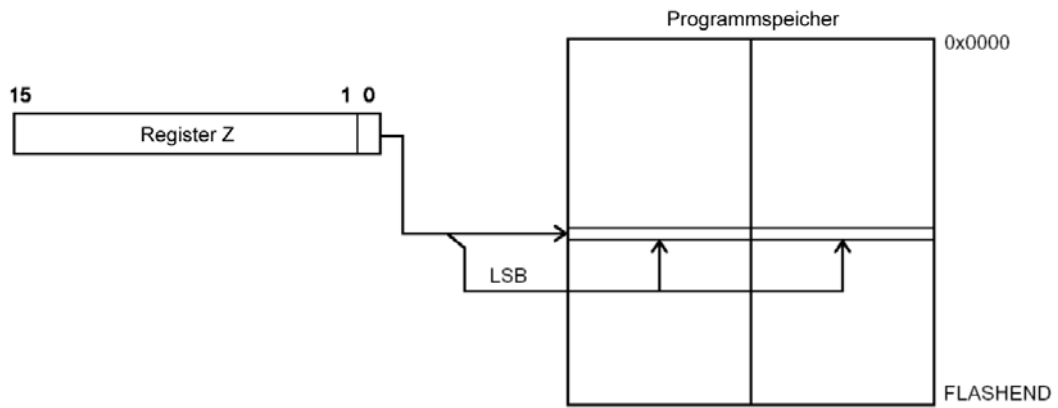
**Programmspeicheradressierung***Konstantenadressierung (LPM, ELPM, SPM):*

Datenbyte wird nach Register R0 transportiert. Nur die ersten 64 kBytes = 32k Befehle erreichbar.

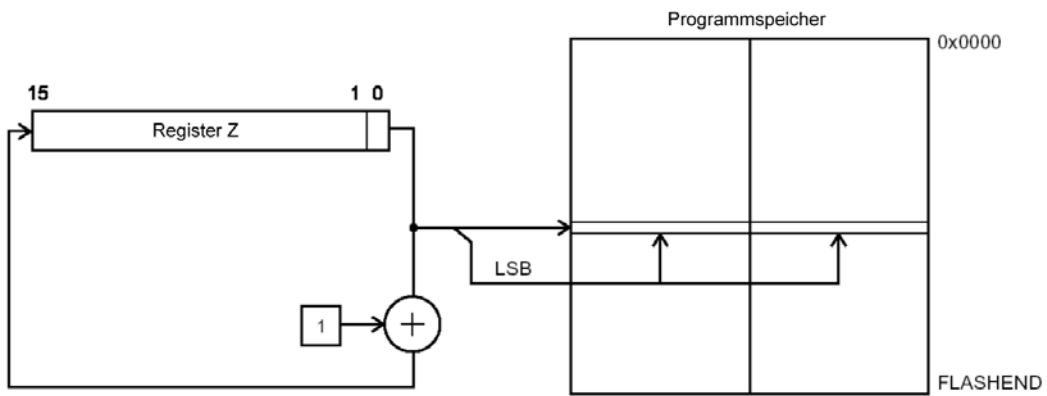
*Achtung:* Marken in Assemblerprogramm betreffen Befehle, keine Bytes. Bei Bezug auf Marken Adreßrechnung erforderlich (Befehlsadresse um 1 Bit nach links verschieben bzw. Befehlsadresse • 2). Marke zeigt auf niederer Byte (Rechtsadressierung).

SPM: Bit 0 im Z-Register löschen.

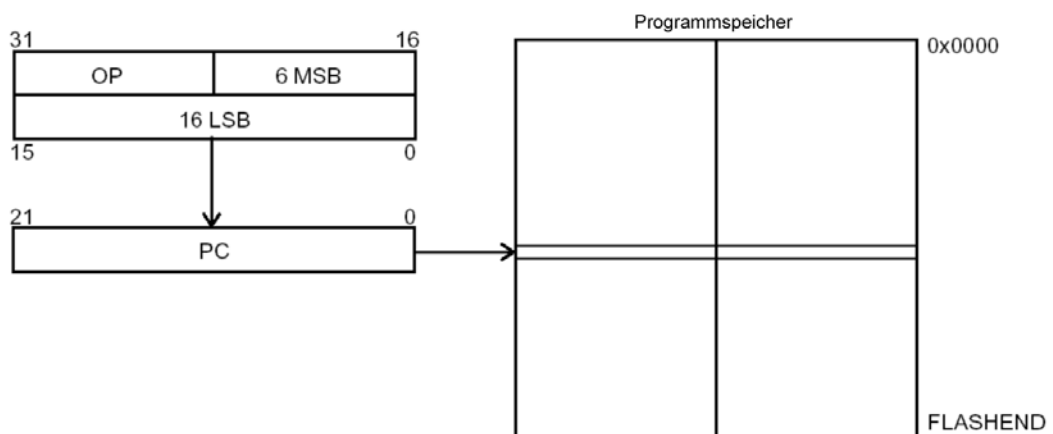
ELPM: Adreßverlängerung mit Register RAMPZ.



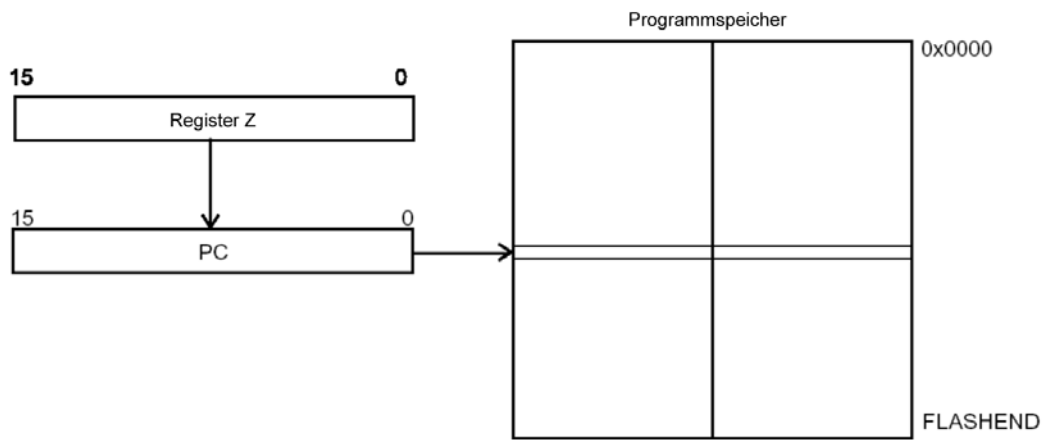
*Konstantenadressierung mit Postincrement (LPM Z+, ELPM Z+):*  
 ELPM Z+: Adreßverlängerung mit Register RAMPZ.



*Direkte Befehlsadressierung (JMP, CALL):*



*Indirekte Befehlsadressierung (IJMP, ICALL) über Adreßregister Z:*

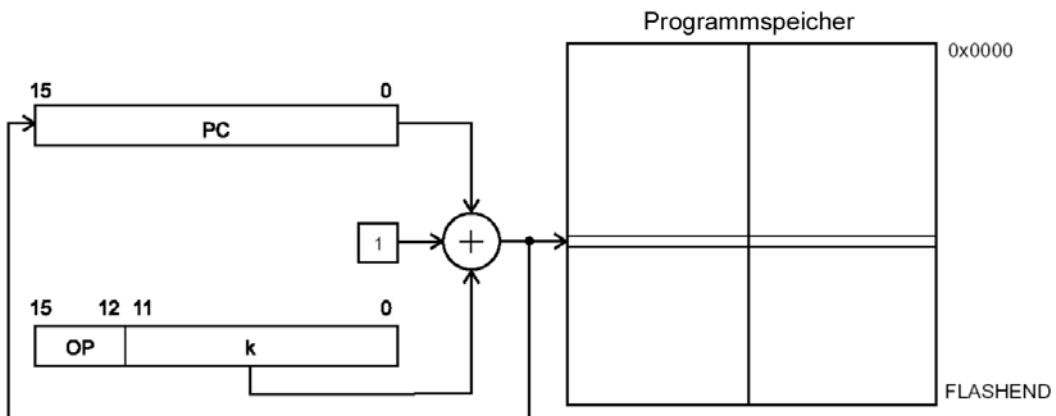


*Relative Befehlsadressierung (RJMP, RCALL, BRxx):*

Displacement:

- bei RJMP und RCALL: -2048...+2047
- bei BRxx: -64...+63.

Jeweils mit Wrap Around modulo Programmspeicherende (Endadresse + 1 = Adresse 0; Adresse 0 - 1 = Endadresse.)



# Transportbefehle

## Transportieren zwischen Registern

### **MOV – Transportieren**

MOV Rd,Rr

Transportiert Inhalt des Registers Rr in das Register Rd.

Register: R0...R31.

Flagbits: bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

### **MOVW – Transportieren Wort**

MOVW Rd,Rr

Transportiert Inhalt des Registerpaares Rr+1, Rr in das Registerpaar Rd+1,Rd.

Register: R0, R2, R4...R30. Nur gerade Registeradressen.

Flagbits: bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

## Laden (Datenspeicher => Register)

### **LDI – Laden Direktwert**

LDI Rd,K

Lädt das Direktwertbyte K in das Register Rd.

Register: R16...R31.

Flagbits: bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

### **LDS – Laden direkt**

LDS Rd,k

Der 16-Bit-Direktwert k adressiert ein Byte im Datenspeicher. Dieses wird in das Register Rd geladen. Hat das Modell mehr als 64 kBytes Datenspeicher, so wird die Adresse mittels Bankregister RAMPD verlängert.

Register: R0...R31.

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

**LD – Laden indirekt**

LD Rd,X	LD Rd,Y	LD Rd,Z
LD Rd,X+	LD Rd,Y+	LD Rd,Z+
LD Rd, -X	LD Rd, -Y	LD Rd, -Z

Der Inhalt des ausgewählten Adreßregisters (X, Y oder Z) adressiert ein Byte im Datenspeicher. Dieses wird in das Register Rd geladen.

Ablaufvarianten:

- Zugriff ohne Adreßveränderung (z. B. LD Rd,X),
- Adreßerhöhung nach dem Zugriff (Postincrement; z. B. LD Rd,Y+),
- Adreßverminderung vor dem Zugriff (Prädecrement; z. B. LD Rd,-Z).

Hat das Modell mehr als 64 kBytes Datenspeicher, so wird die Adresse mittels des entsprechenden Bankregisters (RAMPX, RAMPY, RAMPZ) verlängert. Die gesamte Adresse wird der Erhöhung bzw. Verminderung unterzogen.

Register: R0...R31.

Achtung: Varianten X+, -X nicht mit R26 und R27, Varianten Y+, -Y nicht mit R28 und R29, Varianten Z+, -Z nicht mit R30 und R31.

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

**LDD – Laden indirekt mit Displacement**

LDD Rd,Y+q

LDD Rd,Z+q

Zum Inhalt des ausgewählten Adreßregisters (Y oder Z) wird der Direktwert q (0...63) addiert. Das Additionsergebnis adressiert ein Byte im Datenspeicher. Dieses wird in das Register Rd geladen.

Hat das Modell mehr als 64 kBytes Datenspeicher, so wird die Adresse mittels des entsprechenden Bankregisters (RAMPY, RAMPZ) verlängert. Die gesamte Adresse wird in die Addition einbezogen. Es wird nur die Zugriffsadresse berechnet; Adreßregisterinhalte werden nicht verändert.

Register: R0...R31.

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

## Laden (Programmspeicher => Register)

### LPM – Laden aus Programmspeicher

LPM

LPM Rd,Z

LPM Rd,Z+

Der Inhalt des Adreßregisters Z adressiert ein Byte im Programmspeicher. Es können nur die ersten 32kBytes des Programmspeichers adressiert werden. (Keine Adreßverlängerung).

Ablaufvarianten:

- LPM: Byte wird in Register R0 geladen.
- LPM Rd,Z: byte wird in Register Rd geladen.
- LPM Rd,Z+. Byte wird in Register Rd geladen. Anschließend wird die Adresse im Register Z um 1 erhöht (Postincrement).

Register: R0...R31.

Achtung: Variante LPM Rd,Z+ nicht mit R30 und R31.

Flagbits: bleiben, wie sie sind.

Dauer: 3 Taktzyklen.

### ELPM – Laden aus Programmspeicher, erweitert

ELPM

ELPM Rd,Z

ELPM Rd,Z+

Aus den Inhalten des Adreßregisters Z und des Bankregisters RAMPZ wird eine Adresse gebildet, die ein Byte im Programmspeicher adressiert.

Ablaufvarianten:

- ELPM: Byte wird in Register R0 geladen.
- ELPM Rd,Z: byte wird in Register Rd geladen.
- ELPM Rd,Z+. Byte wird in Register Rd geladen. Anschließend wird die Adresse im Register Z um 1 erhöht (Postincrement).

Hiermit ist der gesamte Programmspeicher zugänglich. Beim Postincrement (ELPM Rd,Z+) wird die gesamte Adresse in die Addition einbezogen.

Register: R0...R31.

Achtung: Variante ELPM Rd,Z+ nicht mit R30 und R31.

Flagbits: bleiben, wie sie sind.

Dauer: 3 Taktzyklen.

## Speichern (Register => Datenspeicher)

### STS – Speichern direkt

STS k,Rr

Der 16-Bit-Direktwert k adressiert ein Byte im Datenspeicher. Der Inhalt des Registers Rr wird in diese Byteposition gespeichert.

Hat das Modell mehr als 64 kBytes Datenspeicher, so wird die Adresse mittels des Bankregisters RAMPD verlängert.

Register: R0...R31.

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

### ST – Speichern indirekt

ST X,Rr	ST Y,Rr	ST Z,Rr
ST X+,Rr	ST Y+,Rr	ST Z+,Rr
ST -X,Rr	ST -Y,Rr	ST -Z,Rr

Der Inhalt des ausgewählten Adreßregisters (X, Y oder Z) adressiert ein Byte im Datenspeicher. Der Inhalt des Registers Rr wird in diese Byteposition gespeichert.

Ablaufvarianten:

- Zugriff ohne Adreßveränderung (z. B. ST X,Rr),
- Adreßerhöhung nach dem Zugriff (Postincrement; z. B. ST Y+,Rr),
- Adreßverminderung vor dem Zugriff (Prädecrement; z. B. ST -Z,Rr).

Hat das Modell mehr als 64 kBytes Datenspeicher, so wird die Adresse mittels des entsprechenden Bankregisters (RAMPX, RAMPY, RAMPZ) verlängert. Die gesamte Adresse wird der Erhöhung bzw. Verminderung unterzogen.

Register: R0...R31.

Achtung: Varianten X+, -X nicht mit R26 und R27, Varianten Y+, -Y nicht mit R28 und R29, Varianten Z+, -Z nicht mit R30 und R31.

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

**STD – Speichern indirekt mit Displacement**

STD Y+q,Rr

STD Z+q,Rr

Zum Inhalt des ausgewählten Adreßregisters (Y oder Z) wird der Direktwert q (0...63) addiert. Das Additionsergebnis adressiert ein Byte im Datenspeicher. Der Inhalt des Registers Rr wird in diese Byteposition gespeichert.

Hat das Modell mehr als 64 kBytes Datenspeicher, so wird die Adresse mittels des entsprechenden Bankregisters (RAMPY, RAMPZ) verlängert. Die gesamte Adresse wird in die Addition einbezogen. Es wird nur die Zugriffsadresse berechnet; Adreßregisterinhalte werden nicht verändert.

Register: R0...R31.

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

**SPM – Speichern in Programmspeicher (neu programmieren)**

SPM

Die adressierte Seite im Programmspeicher wird gelöscht und neu programmiert. Hierzu gibt es verschiedene Funktionen (modellspezifisch).

Verwendete Register:

- Adressierung: Z,
- Daten: R1 und R0,
- Steuerung: Steuerregister im E-A-Adreßbereich.

Flagbits: bleiben, wie sie sind.

Dauer: funktionsabhängig.

**ESPM – Speichern in Programmspeicher, erweitert**

ESPM

Die adressierte Seite im Programmspeicher wird gelöscht und neu programmiert. Hierzu gibt es verschiedene Funktionen (modellspezifisch).

Verwendete Register:

- Adressierung: RAMPZ + Z,
- Daten: R1 und R0,
- Steuerung: Steuerregister im E-A-Adreßbereich.

Flagbits: bleiben, wie sie sind.

Dauer: funktionsabhängig.

## Stackbefehle

### **PUSH**

PUSH Rr

Der Inhalt des Registers Rr wird auf den Stack gelegt (gemäß Adresse im Stackpointer SP). Dann wird der Inhalt des Stackpointers um 1 vermindert.

Register: R0...R31.

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

### **POP**

POP Rd

Der der Inhalt des Stackpointers wird um 1 erhöht. Dann wird das von SP im Stack adressierte Byte in das Register Rd gebracht.

Register: R0...R31.

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

## E-A-Befehle

### **IN – Eingabe**

IN Rd,A

Transportiert Inhalt des E-A-Registers A in das Register Rd.

Register: R0...R31.

E-A-Register: 0...63.

Flagbits: bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

### **OUT – Ausgabe**

OUT A,Rr

Transportiert Inhalt des Registers Rr in das E-A-Register A.

Register: R0...R31.

E-A-Register: 0...63.

Flagbits: bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

**SBI – Setzen Bit in E-A-Register**

SBI A,b

Setzt Bit an Position b ( $b = 0...7$ ) in E-A-Register A.

E-A-Register: 0...31.

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

**CBI – Löschen Bit in E-A-Register**

CBI A,b

Löscht Bit an Position b ( $b = 0...7$ ) in E-A-Register A.

E-A-Register: 0...31.

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

**SBIC – Überspringen, wenn Bit in E-A-Register gelöscht**

SBIC A,b

Fragt Bit in Position b ( $b = 0...7$ ) des E-A-Registers A ab und steuert dementsprechend das Holen des nächsten Befehls:

- ist das Bit gesetzt, wird der Folgebefehl ausgeführt,
- ist das Bit gelöscht, wird der Folgebefehl übersprungen, also der übernächste Befehl ausgeführt.

(Entspricht Verzweigen bei gesetztem Bit.)

E-A-Register: 0...31.

Dauer:

- 1 Taktzyklus, wenn Bit gesetzt ist (kein Überspringen),
- 2 Taktzyklen, wenn Bit gelöscht ist und ein 16 Bits langer Befehl übersprungen werden muß,
- 2 Taktzyklen, wenn Bit gelöscht ist und ein 32 Bits langer Befehl übersprungen werden muß.

**SBIS – Überspringen, wenn Bit in E-A-Register gesetzt**

SBIS A,b

Fragt Bit in Position b ( $b = 0...7$ ) des E-A-Registers A ab und steuert dementsprechend das Holen des nächsten Befehls:

- ist das Bit gelöscht, wird der Folgebefehl ausgeführt,
- ist das Bit gesetzt, wird der Folgebefehl übersprungen, also der übernächste Befehl ausgeführt.

(Entspricht Verzweigen bei gelöschtem Bit.)

E-A-Register: 0...31.

Dauer:

- 1 Taktzyklus, wenn Bit gelöscht ist (kein Überspringen),
- 2 Taktzyklen, wenn Bit gesetzt ist und ein 16 Bits langer Befehl übersprungen werden muß,
- 2 Taktzyklen, wenn Bit gesetzt ist und ein 32 Bits langer Befehl übersprungen werden muß.

## Verarbeitungsbefehle

### Logische Operationen

#### **AND – UND-Verknüpfung**

AND Rd,Rr

Die Inhalte der Register Rr und Rd werden bitweise konjunktiv miteinander verknüpft. Das Ergebnis wird im Register Rd gespeichert.

$\langle Rd \rangle := \langle Rd \rangle \& \langle Rr \rangle$

Register: R0...R31.

Flagbits: S, N, Z werden gemäß Verknüpfungsergebnis gestellt. V wird Null. Die anderen bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

#### **ANDI – UND-Verknüpfung mit Direktwert**

ANDI Rd,K

Der Inhalt des Registers Rd wird mit dem Direktwert K bitweise konjunktiv verknüpft. Das Ergebnis wird im Register Rd gespeichert.

$\langle Rd \rangle := \langle Rd \rangle \& K$

Register: R16...R31.

Flagbits: S, N, Z werden gemäß Verknüpfungsergebnis gestellt. V wird Null. Die anderen bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

#### **OR – ODER-Verknüpfung**

OR Rd,Rr

Die Inhalte der Register Rr und Rd werden bitweise disjunktiv miteinander verknüpft. Das Ergebnis wird im Register Rd gespeichert.

$$\langle \text{Rd} \rangle = \langle \text{Rd} \rangle \text{ W } \langle \text{Rr} \rangle$$

Register: R0...R31.

Flagbits: S, N, Z werden gemäß Verknüpfungsergebnis gestellt. V wird Null. Die anderen bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

### **ORI – ODER-Verknüpfung mit Direktwert**

ORI Rd,K

Der Inhalt des Registers Rd wird mit dem Direktwert K bitweise disjunktiv verknüpft. Das Ergebnis wird im Register Rd gespeichert.

$$\langle \text{Rd} \rangle := \langle \text{Rd} \rangle \text{ W } K$$

Register: R16...R31.

Flagbits: S, N, Z werden gemäß Verknüpfungsergebnis gestellt. V wird Null. Die anderen bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

### **EOR – Exklusiv-ODER (XOR, Antivalenz)**

EOR Rd,Rr

Die Inhalte der Register Rr und Rd werden bitweise antivalent miteinander verknüpft. Das Ergebnis wird im Register Rd gespeichert.

$$\langle \text{Rd} \rangle := \langle \text{Rd} \rangle \uparrow \langle \text{Rr} \rangle$$

Register: R0...R31.

Flagbits: S, N, Z werden gemäß Verknüpfungsergebnis gestellt. V wird Null. Die anderen bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

### **COM – Bitweise Negation (Einerkomplement)**

COM Rd

Der Inhalt des Registers Rd wird bitweise invertiert. Das Ergebnis wird im Register Rd gespeichert.

Register: R0...R31.

Flagbits: S, N, Z werden gemäß Verknüpfungsergebnis gestellt. V wird Null. C wird Eins. Die anderen bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

## Alias-Befehle

Einige Verknüpfungsoperationen werden – zu besonderen Zwecken<sup>1)</sup> – unter anderen Assembler-Mnemonics angeboten.

### **SBR – Setzen Bits in Register**

SBR Rd,K

Einsen im Direktwert K führen zum Setzen der betreffenden Bits im Register Rd. Nullen im Direktwert K lassen die betreffenden Bitpositionen im Register Rd unverändert.

Register: R16...R31.

Alias zu ORI Rd,K.

### **CBR – Löschen Bits in Register**

CBR Rd,K.

Einsen im Direktwert K führen zum Löschen der betreffenden Bits im Register Rd. Nullen im Direktwert K lassen die betreffenden Bitpositionen im Register Rd unverändert.

Register: R16...R31.

Alias zu ANDI Rd,/K – der Assembler erzeugt für CBR einen ANDI-Befehl mit negiertem Direktwert.

*Achtung – Gotcha:*

SBR und CBR sind keine richtigen Einzelbitbefehle (anders als SBI und CBI):

- SBI A,5    setzt Bit 5 im E-A-Register A,
- SBR Rd,5    setzt die Bits 2 und 0 im Register Rd (gemäß Bitmuster 5H = 101).

*Praxistip:* Zum Setzen und Löschen von Einzelbits in Registern Makros definieren!

### **TST – Testen auf Null oder Minus**

TST Rd.

Prüft, ob ein Registerinhalt gleich Null oder ob er negativ ist (anhand der gesetzten Flagbits Z und N erkennbar). Registerinhalt wird dabei nicht verändert. Prinzip: konjunktive Verknüpfung “mit sich selbst”.

Alias zu AND Rd,Rd.

### **CLR – Register löschen**

CLR Rd

Der Inhalt des Registers Rd wird gelöscht (Null). Prinzip: Antivalenzverknüpfung (XOR) “mit sich selbst”.

---

1): nicht zuletzt: damit die Befehlsliste nach mehr aussieht ...

Flagbits: S, V, N werden Null, Z wird Eins. Die anderen bleiben, wie sie sind.

Register: R0...R31.

Alias zu EOR Rd,Rd.

**SER – Register setzen**  
SER Rd

Der Inhalt des Registers Rd wird gesetzt (FFH).

Register: R16...R31.

Flagbits: bleiben, wie sie sind.

Alias zu LDI Rd,FFH.

## Arithmetische Operationen

**ADD – Addieren**  
ADD Rd,Rr

Die Inhalte der Register Rr und Rd werden als Binärzahlen zueinander addiert. Das Ergebnis wird im Register Rd gespeichert.

$\langle Rd \rangle := \langle Rd \rangle + \langle Rr \rangle$

Register: R0...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt.

Dauer: 1 Taktzyklus.

**ADC – Addieren mit Eingangsübertrag**  
ADC Rd,Rr

Die Inhalte der Register Rr und Rd werden als Binärzahlen zueinander addiert. In die Addition fließt weiterhin die Belegung des Flagbits C als Eingangsübertrag ein. Das Ergebnis wird im Register Rd gespeichert.

$\langle Rd \rangle := \langle Rd \rangle + \langle Rr \rangle + C$

Register: R0...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt. Die anderen bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

**ADIW – Addieren Direktwert zu Wort in Registerpaar**

ADIW Rd,K

Der Inhalt des mit Rd angegebenen Registerpaares und der Direktwert K werden als Binärzahlen zueinander addiert. Das Ergebnis wird im besagten Registerpaar gespeichert.

$$\langle \text{Rd}, \text{Rd}+1 \rangle := \langle \text{Rd}, \text{Rd}+1 \rangle + K$$

Register: R24, R26, R28, R30.

Wertebereich des Direktwertes K: 0...63. Wird vorzeichenlos erweitert.

Ein Hilfsbefehl vor allem zum Umgang mit den Adreßregistern X, Y, Z.

Die Registerpaare: R24 – R25; R26 – R27 (X); R28 – R29 (Y); R30 – R31 (Z).

Flagbits: S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt. Die anderen bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

**SUB – Subtrahieren**

SUB Rd,Rr

Die Inhalte der Register Rr und Rd werden als Binärzahlen voneinander subtrahiert. Das Ergebnis wird im Register Rd gespeichert.

$$\langle \text{Rd} \rangle := \langle \text{Rd} \rangle - \langle \text{Rr} \rangle$$

Register: R0...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt. Die anderen bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

**SBC – Subtrahieren mit Eingangsübertrag**

SBC Rd,Rr

Die Inhalte der Register Rr und Rd werden als Binärzahlen voneinander subtrahiert. In die Subtraktion fließt weiterhin die Belegung des Flagbits C als Eingangsübertrag ein. Das Ergebnis wird im Register Rd gespeichert.

$$\langle \text{Rd} \rangle := \langle \text{Rd} \rangle - \langle \text{Rr} \rangle - C$$

Register: R0...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt. Die anderen bleiben, wie sie sind. Besonderheit Z-Flag: wird gelöscht, wenn Resultat  $\neq 0$ ; bleibt auf bisherigem Wert, wenn Resultat = 0 (Sticky Flag).

Dauer: 1 Taktzyklus.

**SUBI – Subtrahieren Direktwert**

SUBI Rd,K

Der Direktwert wird vom Inhalt des Registers Rd subtrahiert. Das Ergebnis wird im Register Rd gespeichert.

$$\langle \text{Rd} \rangle := \langle \text{Rd} \rangle - K$$

Register: R16...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt. Die anderen bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

**SBCI – Subtrahieren Direktwert mit Eingangsübertrag**

SBCI Rd,K

Der Direktwert wird vom Inhalt des Registers Rd subtrahiert. In die Subtraktion fließt weiterhin die Belegung des Flagbits C als Eingangsübertrag ein. Das Ergebnis wird im Register Rd gespeichert.

$$\langle \text{Rd} \rangle := \langle \text{Rd} \rangle - K - C$$

Register: R16...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt. Die anderen bleiben, wie sie sind.

Besonderheit Z-Flag: wird gelöscht, wenn Resultat  $\neq 0$ ; bleibt auf bisherigem Wert, wenn Resultat = 0 (Sticky Flag).

Dauer: 1 Taktzyklus.

*Hinweis:*

Es gibt keine entsprechenden Befehle zum Addieren von Direktwerten. Sind Direktwerte zu addieren, können Befehle SUBI und SBCI verwendet werden, wobei als Direktwert jeweils das Zweierkomplement des gewünschten Zahlenwertes anzugeben ist (Angabe als negative Zahl).

*Programmierbeispiel:*

Das Registerpaar r17, r16 wird mit dem Wert 10 000 geladen. Dann wird dieser Wert um 1000 erhöht:

```
ldi    r16, low(10000)
ldi    r17, high(10000)
subi   r16, low(-1000)
sbci   r17, high(-1000)
```

**SBIW – Subtrahieren Direktwert von Wort in Registerpaar**

SBIW Rd,K

Der Direktwert K und der Inhalt des mit Rd angegebenen Registerpaares werden als Binärzahlen voneinander subtrahiert. Das Ergebnis wird im besagten Registerpaar gespeichert.

$$\langle \text{Rd}, \text{Rd}+1 \rangle := \langle \text{Rd}, \text{Rd}+1 \rangle - K$$

Register: R24, R26, R28, R30.

Wertebereich des Direktwertes K: 0...63. Wird vorzeichenlos erweitert.

Ein Hilfsbefehl vor allem zum Umgang mit den Adreßregistern X, Y, Z. Die Registerpaare: R24 – R25; R26 – R27 (X); R28 – R29 (Y); R30 – R31 (Z).

Flagbits: S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt. Die anderen bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

### **NEG – Zweierkomplement** NEG Rd

Es wird das Zweierkomplement des Inhalts von Register Rd gebildet. Das Ergebnis wird im Register Rd gespeichert.

$\langle \text{Rd} \rangle := 0 - \langle \text{Rd} \rangle$

Register: R0...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt.

Dauer: 1 Taktzyklus.

### **INC – Erhöhen um 1** INC Rd

Der Inhalt des Registers Rd wird um Eins erhöht. Das Ergebnis wird im Register Rd gespeichert.

$\langle \text{Rd} \rangle := \langle \text{Rd} \rangle + 1$

Register: R0...R31.

Flagbits: S, V, N, Z werden gemäß Verknüpfungsergebnis gestellt.

Dauer: 1 Taktzyklus.

### **DEC – Vermindern um 1** DEC Rd

Der Inhalt des Registers Rd wird um Eins vermindert. Das Ergebnis wird im Register Rd gespeichert.

$\langle \text{Rd} \rangle := \langle \text{Rd} \rangle - 1$

Register: R0...R31.

Flagbits: S, V, N, Z werden gemäß Verknüpfungsergebnis gestellt.

Dauer: 1 Taktzyklus.

**MUL, MULS, MULSU – Multiplizieren**

MUL Rd,Rr

MULS Rd,Rr

MULSU Rd,Rr

Die Inhalte der Register Rd und Rr werden als Binärzahlen (ohne bzw. mit Vorzeichen) miteinander multipliziert. Das 16 Bits lange Ergebnis wird in den Registern R1 und R0 gespeichert.

$$\langle R1, R0 \rangle := \langle Rd \rangle \cdot \langle Rr \rangle$$

*Varianten:*

- MUL: vorzeichenlose Multiplikation,
- MULS: ganzzahlige Multiplikation,
- MULSU: Multiplikation ganzzahlig (Rd) und vorzeichenlos(Rr). Ergebnis ganzzahlig (mit Vorzeichen)

Register: R0...R31.

Flagbits: Z wird gesetzt, wenn das Ergebnis gleich null ist. C entspricht dem höchstwertigen Ergebnisbit (Bit 15 bzw. Bit 7, R1). Die anderen bleiben, wie sie sind.

Dauer: 2 Taktzyklen

**FMUL, FMULS, FMULSU – Multiplikation mit Linkverschiebung**

FMUL Rd,Rr

FMULS Rd,Rr

FMULSU Rd,Rr

Die Inhalte der Register Rd und Rr werden als Binärzahlen miteinander multipliziert. Die Belegung des höchstwertigen Ergebnisbits wird in das Flagbit C übernommen. Das Ergebnis wird um ein Bit nach links verschoben und in den Registern R1 und R0 gespeichert.

$$\langle R1, R0 \rangle := (\langle Rd \rangle \cdot \langle Rr \rangle) \cdot 2$$

*Varianten:*

- FMUL: vorzeichenlose Multiplikation,
- FMULS: ganzzahlige Multiplikation,
- FMULSU: Multiplikation ganzzahlig (Rd) und vorzeichenlos(Rr).

Register: R16...R31.

Flagbits: Z wird gesetzt, wenn das Ergebnis gleich Null ist. C entspricht dem (herausgeschobenen) höchstwertigen Ergebnisbit (Bit 15). Die anderen bleiben, wie sie sind.

Dauer: 2 Taktzyklen

Anwendung: vor allem um Binärbrüche des Formates 1.7 (Bit 7 = Einerstelle, Bits 6...0 = Stellen nach dem Binärkomma) miteinander zu multiplizieren (erzeugt Format 2.14) und auf das Format 1.15 zu bringen (Signalverarbeitung usw.)

## Vergleichsbefehle

### CP – Vergleichen

CP Rd,Rr

Die Inhalte der Register Rr und Rd werden als Binärzahlen voneinander subtrahiert. Die Registerinhalte werden nicht verändert.

Vergleichsablauf:  $\langle Rd \rangle - \langle Rr \rangle$

Register: R0...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt.

Dauer: 1 Taktzyklus.

### CPI – Vergleichen mit Direktwert

CPI Rd,K

Der Direktwert K und der Inhalt des Registers Rd werden als Binärzahlen voneinander subtrahiert. Der Registerinhalt wird nicht verändert.

Vergleichsablauf:  $\langle Rd \rangle - K$

Register: R16...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt.

Dauer: 1 Taktzyklus.

### CPC – Vergleichen mit Eingangsübertrag

CP Rd,Rr

Die Inhalte der Register Rr und Rd werden als Binärzahlen voneinander subtrahiert. In die Subtraktion fließt weiterhin die Belegung des Flagbits C als Eingangsübertrag ein. Die Registerinhalte werden nicht verändert.

Vergleichsablauf:  $\langle Rd \rangle - \langle Rr \rangle - C$

Register: R0...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Verknüpfungsergebnis gestellt. Besonderheit Z-Flag: wird gelöscht, wenn Resultat  $\neq 0$ ; bleibt auf bisherigem Wert, wenn Resultat = 0 (Sticky Flag).

Dauer: 1 Taktzyklus.

### CPSE – Vergleichen und Überspringen bei Gleichheit

CPSE Rd,Rr

Die Inhalte der Register Rr und Rd werden als Binärzahlen voneinander subtrahiert. Die Registerinhalte werden nicht verändert. Das Vergleichsergebnis steuert das Holen des nächsten Befehls:

- sind beide Registerinhalte ungleich, wird der Folgebefehl ausgeführt,
- sind beide Registerinhalte gleich, wird der Folgebefehl übersprungen, also der übernächste Befehl ausgeführt.

(Folge CPSE – JMP entspricht Verzweigen bei Ungleichheit.)

Vergleichsablauf:  $\langle Rd \rangle - \langle Rr \rangle$

Register: R0...R31.

Flagbits: bleiben, wie sie sind.

Dauer:

- 1 Taktzyklus, wenn Ungleichheit (kein Überspringen),
- 2 Taktzyklen, wenn Gleichheit und ein 16 Bits langer Befehl übersprungen werden muß,
- 2 Taktzyklen, wenn Gleichheit und ein 32 Bits langer Befehl übersprungen werden muß.

## Verschiebeoperationen

### LSL – Linksverschieben

LSL Rd

Der Inhalt des Registers Rd wird um eine Bitposition nach links verschoben. Das ursprüngliche Bit 7 gelangt dabei in das Flagbit C. Bit 0 wird gelöscht. Das Ergebnis wird im Register Rd gespeichert. Entspricht Multiplikation mit 2.

Register: R0...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Ergebnis gestellt.

Besonderheiten: H := Bitposition 3; C := Bitposition 7.

Dauer: 1 Taktzyklus.

### LSR – Rechtsverschieben

LSR Rd

Der Inhalt des Registers Rd wird um eine Bitposition nach rechts verschoben. Das ursprüngliche Bit 0 gelangt dabei in das Flagbit C. Bit 7 wird gelöscht. Das Ergebnis wird im Register Rd gespeichert. Entspricht der vorzeichenlosen Division durch 2.

Register: R0...R31.

Flagbits: S, V, N, Z, C werden gemäß Ergebnis gestellt.

Besonderheit: C := Bitposition 0.

Dauer: 1 Taktzyklus.

**ASR – Rechtsverschieben arithmetisch**

ASR Rd

Der Inhalt des Registers Rd wird um eine Bitposition nach rechts verschoben. Das ursprüngliche Bit 0 gelangt dabei in das Flagbit C. Bit 7 bleibt unverändert. Das Ergebnis wird im Register Rd gespeichert.

Register: R0...R31.

Flagbits: S, V, N, Z, C werden gemäß Ergebnis gestellt.  
Besonderheit: C := Bitposition 0.

Dauer: 1 Taktzyklus.

**ROL – Linksrotieren durch Übertrag**

ROL Rd

Der Inhalt des Registers Rd wird um eine Bitposition nach links verschoben. Dabei läuft das Flagbit C in Bitposition 0 ein, und das ursprüngliche Bit 7 gelangt in das Flagbit C. Das Ergebnis wird im Register Rd gespeichert.

Register: R0...R31.

Flagbits: H, S, V, N, Z, C werden gemäß Ergebnis gestellt.  
Besonderheiten: H := Bitposition 3; C := Bitposition 7.

Dauer: 1 Taktzyklus.

**ROR – Rechtsrotieren durch Übertrag**

ROR Rd

Der Inhalt des Registers Rd wird um eine Bitposition nach rechts verschoben. Dabei läuft das Flagbit C in Bitposition 7 ein, und das ursprüngliche Bit 0 gelangt in das Flagbit C. Das Ergebnis wird im Register Rd gespeichert.

Register: R0...R31.

Flagbits: S, V, N, Z, C werden gemäß Ergebnis gestellt.  
Besonderheit: C := Bitposition 0.

Dauer: 1 Taktzyklus.

**SWAP – Tetraden vertauschen**

SWAP Rd

Die Tetraden (Halbbytes) des Registers Rd werden miteinander vertauscht: Bits 7...4 => Bits 3...0; Bits 3...0 => Bits 7...4. Das Ergebnis wird im Register Rd gespeichert.

Register: R0...R31.

Flagbits: bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

# Programmsteuerbefehle

## Einzelbitbefehle

### **SBRC – Überspringen, wenn Bit in Register gelöscht**

SBRC Rd,b

Fragt Bit in Position b ( $b = 0..7$ ) des Registers Rd ab und steuert dementsprechend das Holen des nächsten Befehls:

- ist das Bit gesetzt, wird der Folgebefehl ausgeführt,
- ist das Bit gelöscht, wird der Folgebefehl übersprungen, also der übernächste Befehl ausgeführt.

(Folge SBRC – JMP entspricht Verzweigen bei gesetztem Bit.)

Register: 0...31.

Flagbits: bleiben, wie sie sind.

Dauer:

- 1 Taktzyklus, wenn Bit gesetzt ist (kein Überspringen),
- 2 Taktzyklen, wenn Bit gelöscht ist und ein 16 Bits langer Befehl übersprungen werden muß,
- 2 Taktzyklen, wenn Bit gelöscht ist und ein 32 Bits langer Befehl übersprungen werden muß.

### **SBRS – Überspringen, wenn Bit in Register gesetzt**

SBRS Rd,b

Fragt Bit in Position b ( $b = 0..7$ ) des Registers Rd ab und steuert dementsprechend das Holen des nächsten Befehls:

- ist das Bit gelöscht, wird der Folgebefehl ausgeführt,
- ist das Bit gesetzt, wird der Folgebefehl übersprungen, also der übernächste Befehl ausgeführt.

(Folge SBRS – JMP entspricht Verzweigen bei gelöschtem Bit.)

Register: 0...31.

Flagbits: bleiben, wie sie sind.

Dauer:

- 1 Taktzyklus, wenn Bit gelöscht ist (kein Überspringen),
- 2 Taktzyklen, wenn Bit gesetzt ist und ein 16 Bits langer Befehl übersprungen werden muß,
- 2 Taktzyklen, wenn Bit gesetzt ist und ein 32 Bits langer Befehl übersprungen werden muß.

**BLD – Bit laden in T-Flag**

BLD Rd,b

Transportiert Bit in Position b (b = 0...7) des Registers Rd in das Flagbit T.

Register: 0...31.

Flagbits: T := Bit b aus Register Rd. Die anderen bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

**BST – Bit speichern aus T-Flag**

BST Rd,b

Transportiert Flagbit T in Position b (b = 0...7) des Registers Rd.

Register: 0...31.

Flagbits: bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

## Verzweigen

**RJMP – Verzweigen relativ**

RJMP k

Unbedingte Verzweigung zu einer Adresse, die aus dem Inhalt des Befehlszählers und der Adreßangabe k ( $-2048 \leq k \leq 2047$ ) errechnet wird. In Programmspeichern von höchstens 4k Worten (8 kBytes) kann jedes Wort mittels RJMP erreicht werden (Adreßrechnung mit Wrap Around modulo Speichergröße).

$$\langle PC \rangle := \langle PC \rangle + k + 1$$

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

**IJMP – Verzweigen indirekt**

IJMP

Unbedingte Verzweigung zu einer Adresse, die aus dem Adreßregister Z (R31, R30) entnommen wird. Es können nur die ersten 64k Worte erreicht werden (ist der Programmspeicher größer als 64k Worte, so werden die höherwertigen Adreßbits von Bit 17 an mit Nullen geladen).

$$\langle PC \rangle := \langle R31, R30 \rangle \quad (\langle PC (21:17) \rangle := 0)$$

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

### **EIJMP – Verzweigen indirekt, erweitert**

#### **EIJMP**

Unbedingte Verzweigung zu einer Adresse, die aus dem Bankregister EIND und aus dem Adreßregister Z (R31, R30) entnommen wird. Es kann der gesamte Programmspeicher erreicht werden.

$\langle PC(15:0) \rangle := \langle R31, R30 \rangle$ ;  $\langle PC(21:17) \rangle := \langle EIND \rangle$

Flagbits: bleiben, wie sie sind.

Dauer: 2 Taktzyklen.

### **JMP – Verzweigen im gesamten Programmadreßraum**

#### **JMP k**

Unbedingte Verzweigung zu Adresse k ( $0 \leq k < 4M$ ). 32-Bit-Befehl.

Flagbits: bleiben, wie sie sind.

Dauer: 3 Taktzyklen.

### **BRBC, BRBS, Rxx – bedingtes relatives Verzweigen**

BRBC b,k

BRBS b,k

BRxx k

Die Maschinenbefehle können nur jeweils eines der Bitpositionen des Statusregisters abfragen:

- BRBC b,k: Verzweigen, wenn Bit b gelöscht,
- BRBS b,k: Verzweigen, wenn Bit b gesetzt.

Die Befehle BRxx sind lediglich entsprechende Alias-Befehle. Ist die Bedingung xx nicht erfüllt, so wird der nächste Befehl von der folgenden Adresse geholt. Ist die Bedingung xx erfüllt, wird zu einer Adresse verzweigt, die aus dem Inhalt des Befehlszählers und der Adreßangabe k ( $-64 \leq k \leq 63$ ) errechnet wird. Zu den Verzweigungsbedingungen siehe die folgenden Tabellen.

$\langle PC \rangle := \langle PC \rangle + 1$  oder  $\langle PC \rangle := \langle PC \rangle + k + 1$

Flagbits: bleiben, wie sie sind.

Dauer:

- 1 Taktzyklus, wenn Bedingung nicht erfüllt (kein Verzweigen),
- 2 Taktzyklen, wenn Bedingung erfüllt (Verzweigen).

a) *Flagbits*

Bezeichnung		Benennung	Bedeutung
allgemein	Atmel		
ZF	Z	Zero Flag	Ergebnis = 0
CF	C	Carry Flag	Addition: Ausgangsübertrag = 1, Subtraktion: Ausgangsübertrag = 0 (Borgen)
OF	V	Overflow Flag	Overflow = Ausgangsübertrag $\neq$ Übertrag in die Vorzeichenstelle = Ausgangsübertrag $\oplus$ Übertrag in die Vorzeichenstelle
SF	N	Sign Flag (Atmel: Negative Flag)	Vorzeichen (= höchstwertige Bitposition) = 1. Wert ist negativ

Die AVR-Prozessoren haben als zusätzliches Flagbit S (Signed Flag) die Verknüpfung  $N \vee V$ .

b) *das Statusregister (SREG):*

7	6	5	4	3	2	1	0
I	T	H	S	V	N	Z	C

c) *Vergleichen vorzeichenloser Binärzahlen. Rechengang: A - B. Zweierkomplement-Arithmetik*

Vergleichsaussage	Bedingung	Flagbits	typische Bezeichnung
$A = B$	Ergebnis = 0 (sowie Ausgangsübertrag)	$Z = 1$	Equal
$A \neq B$	Ergebnis $\neq 0$	$Z = 0$	Not Equal
$A < B$	kein Ausgangsübertrag (Borgen; $C = 1$ )	$C = 1$	Below (Atmel: Lower)
$A > B$	Ergebnis $\neq 0$ und Ausgangsübertrag (kein Borgen; $C = 0$ )	$\bar{Z} \wedge \bar{C} = 1$ bzw. $Z \vee C = 0$	Above (Atmel: Higher)
$A \leq B$	Ergebnis = 0 oder kein Ausgangsübertrag (Borgen; $C = 1$ )	$Z \vee C = 1$	Below or Equal
$A \geq B$	Ausgangsübertrag (kein Borgen; $C = 0$ )	$C = 0$	Above or Equal (Atmel: Same or Higher)

d) *Vergleichen ganzer Binärzahlen. Rechengang:  $A - B$ . Zweierkomplement-Arithmetik*

Vergleichsaussage	Bedingung	Flagbits	typische Bezeichnung
$A = B$	Ergebnis = 0	$Z = 1$	Equal
$A \neq B$	Ergebnis $\neq 0$	$Z = 0$	Not Equal
$A < B$	Ergebnis negativ und Überlauf oder Ergebnis positiv und kein Überlauf	$N \neq V$ $N \oplus V = 1$ $S = 1$	Less
$A > B$	Ergebnis $\neq 0$ und Ergebnis negativ und kein Überlauf oder Ergebnis positiv und Überlauf	$Z = 0$ und $N = V$ $Z \cdot (N \oplus V) = 0$ $Z \cdot S = 0$	Greater
$A \leq B$	Ergebnis = 0 oder Ergebnis negativ und Überlauf oder Ergebnis positiv und kein Überlauf	$Z$ oder $(N \neq V)$ $Z \vee (N \oplus V) = 1$ $Z \vee S = 1$	Less or Equal
$A \geq B$	Ergebnis negativ und kein Überlauf oder Ergebnis positiv und Überlauf	$N = V$ $N \oplus V = 0$ $S = 0$	Greater or Equal

e) *die Atmel-Bedingungen*

Mnemonic	Verzweigen, wenn...	Anmerkungen			
BRBC b	Flagbit b gelöscht (b = 0...7)	3: V	2: N	1: Z	0: C
BRBS b	Flagbit b gesetzt (b = 0...7)	7: I	6: T	5: H	4: S
BRCC	C-Flag gelöscht	C = 0. Alias zu BRBC 0			
BRCS	C-Flag gesetzt	C = 1. Alias zu BRBS 0			
BREQ	Gleichheit	Z = 1. Alias zu BRBS 1			
BRGE	größer oder gleich (ganzzahlig)	S = 0. Alias zu BRBC 4			
BRHC	H-Flag gelöscht	H = 0. Alias zu BRBC 5			
BRHS	H-Flag gesetzt	H = 1. Alias zu BRBS 5			
BRID	Interrupts verhindert	I = 0. Alias zu BRBC 7			
BRIE	Interrupts erlaubt	I = 1. Alias zu BRBS 7			
BRLO	kleiner (vorzeichenlos)	C = 1. Alias zu BRBS 0			
BRLT	kleiner (ganzzahlig)	S = 1. Alias zu BRBS 4			
BRMI	negativ	N = 1. Alias zu BRBS 2			
BRNE	ungleich	Z = 0. Alias zu BRBC 1			
BRPL	positiv	N = 0. Alias zu BRBC 2			
BRSH	größer oder gleich (vorzeichenlos)	C = 0. Alias zu BRBC 0			
BRTC	T-Flag gelöscht	T = 0. Alias zu BRBC 6			
BRTS	T-Flag gesetzt	T = 1. Alias zu BRBS 6			
BRVC	kein Überlauf	V = 0. Alias zu BRBC 3			
BRVS	Überlauf	V = 1. Alias zu BRBS 3			

## f) Vergleichen vorzeichenloser und ganzer Binärzahlen (Rechengang A-B):

Allgemeinbezeichnung	Wirkung: Verzweigen, wenn...	AVR	Abhilfe
Branch on Above	$A > B$ vorzeichenlos	-	B-A; verzw. auf $B < A$ (BRLO)
Branch on Above or Equal	$A \geq B$ vorzeichenlos	BRSH	-
Branch on Below	$A < B$ vorzeichenlos	BRLO	-
Branch on Below or Equal	$A \leq B$ vorzeichenlos	-	B-A; verzw. auf $B \geq A$ (BRSH)
Branch on Greater	$A > B$ ganzzahlig	-	B-A; verzw. auf $B < A$ (BRLT)
Branch on Greater or Equal	$A \geq B$ ganzzahlig	BRGE	-
Branch on Less	$A < B$ ganzzahlig	BRLT	-
Branch on Less or Equal	$A \leq B$ ganzzahlig	-	B-A; verzw. auf $B \geq A$ (BRGE)
Branch on Equal	$A = B$	BREQ	-
Branch on Not Equal	$A \neq B$	BRNE	-

Die AVR-Prozessoren unterstützen nicht alle Verzweigungsbedingungen, die zum Vergleichen von Zahlenwerten erforderlich sind. Abhilfe: wenn wir A mit B vergleichen wollen, rechnen wir nicht A-B, sondern B-A und wenden die jeweils inverse Verzweigungsbedingung an.

*Praxistip:* Entsprechende Makros schreiben.

## g) Handbuchauszug (Atmel). Gleiche Aussage wie vorstehende Tabelle.

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

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

## Unterprogrammrufruf und Rückkehr

### RCALL – Unterprogrammrufruf relativ

RCALL k

Die Adresse des Folgebefehls wird auf den Stack gelegt (2 oder 3 Bytes, je nach Länge des Befehlszählers). Dann Verzweigung zu einer Adresse, die aus dem Inhalt des Befehlszählers und der Adreßangabe  $k$  ( $-2048 \leq k \leq 2047$ ) errechnet wird. In Programmspeichern von höchstens 4k Worten (8 kBytes) kann jedes Wort mittels RCALL erreicht werden (Adreßrechnung mit Wrap Around modulo Speichergröße).

Flagbits: bleiben, wie sie sind.

Dauer:

- bei PC mit maximal 16 Bits: 3 Taktzyklen,
- bei PC mit 17...22 Bits: 4 Taktzyklen.

### ICALL – Unterprogrammrufruf indirekt

ICALL

Die Adresse des Folgebefehls wird auf den Stack gelegt (2 oder 3 Bytes, je nach Länge des Befehlszählers). Dann Verzweigung zu einer Adresse, die aus dem Adreßregister Z (R31, R30) entnommen wird. Es können nur die ersten 64k Worte erreicht werden (ist der Programmspeicher größer als 64k Worte, so werden die höherwertigen Adreßbits von Bit 17 an mit Nullen geladen).

Flagbits: bleiben, wie sie sind.

Dauer:

- bei PC mit maximal 16 Bits: 3 Taktzyklen,
- bei PC mit 17...22 Bits: 4 Taktzyklen.

### EICALL – Unterprogrammrufruf indirekt, erweitert

EICALL

Die Adresse des Folgebefehls wird auf den Stack gelegt (2 oder 3 Bytes, je nach Länge des Befehlszählers). Dann Verzweigung zu einer Adresse, die aus dem Bankregister EIND und aus dem Adreßregister Z (R31, R30) entnommen wird. Es kann der gesamte Programmspeicher erreicht werden.

Flagbits: bleiben, wie sie sind.

Dauer:

- bei PC mit maximal 16 Bits: 3 Taktzyklen,
- bei PC mit 17...22 Bits: 4 Taktzyklen.

### CALL – Unterprogrammrufruf im gesamten Programmadreßraum

CALL k

Die Adresse des Folgebefehls wird auf den Stack gelegt (2 oder 3 Bytes, je nach Länge des Befehlszählers). Dann Verzweigung zu Adresse  $k$  ( $0 \leq k < 4M$ ). 32-Bit-Befehl.

Flagbits: bleiben, wie sie sind.

Dauer:

- bei PC mit maximal 16 Bits: 3 Taktzyklen,
- bei PC mit 17...22 Bits: 4 Taktzyklen.

### **RET – Rückkehr aus Unterprogramm**

RET

Die Rückkehradresse (2 oder 3 Bytes, je nach Länge des Befehlszählers) wird vom Stack geholt und in den Befehlszähler (PC) geladen.

Flagbits: bleiben, wie sie sind.

Dauer:

- bei PC mit maximal 16 Bits: 4 Taktzyklen,
- bei PC mit 17...22 Bits: 5 Taktzyklen.

## **Flagbitbefehle**

Diese Befehle dienen zum Setzen bzw. Löschen einzelner Flagbits im Zustandsregister. Die jeweils verbleibenden Flagbits bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

*Grundbefehle:*

### **BSET – Flagbit setzen**

BSET s

Bit in Position s ( $s = 0...7$ ) des Zustandsregisters wird gesetzt. Die anderen Flagbits bleiben, wie sie sind.

### **BCLR – Flagbit löschen**

BCLR s

Bit in Position s ( $s = 0...7$ ) des Zustandsregisters wird gelöscht. Die anderen Flagbits bleiben, wie sie sind.

s	Flagbit	s	Flagbit
0	C	4	S
1	Z	5	H
2	N	6	T
3	V	7	I

*Alias-Befehle:*

Flagbit	Setzen	Löschen
C	SEC	CLC
Z	SEZ	CLZ
N	SEN	CLN
V	SEV	CLV
S	SES	CLS
H	SEH	CLH
T	SET	CLT
I	SEI	CLI

*Unterbrechungserlaubnissteuerung:*

- SEI = Enable Interrupt (Unterbrechung annehmen),
- CLI = Disable Interrupt (Unterbrechung nicht annehmen).

## Sonstige Befehle

### **RETI – Rückkehr aus Unterbrechungsbehandlung**

RETI

Die Rückkehradresse (2 oder 3 Bytes, je nach Länge des Befehlszählers) wird vom Stack geholt und in den Befehlszähler (PC) geladen.

Flagbits: bleiben, wie sie sind.

Dauer:

- bei PC mit maximal 16 Bits: 4 Taktzyklen,
- bei PC mit 17...22 Bits: 5 Taktzyklen.

### **NOP – keine Operation (wirkungsloser Befehl)**

NOP

Keine Wirkung, außer dem Weiterzählen des Befehlszählers.

Flagbits: bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

### **SLEEP – Stromsparezustand einleiten**

SLEEP

Versetzt den Prozessor in einen Stromsparezustand (modellspezifisch). Danach keine weitere Befehlsausführung.

Flagbits: bleiben, wie sie sind.

Dauer: 1 Taktzyklus.

### **WDR – Zeitkontrolle rücksetzen**

WDR

Setzt Kontrollzeitgeber (Watchdog Timer) zurück und verhindert somit dessen Überlaufen (was zu einem Watchdog Reset führen würde).

Ist die Zeitkontrolle aktiviert, so müssen in hinreichend kurzen Zeitabständen WDR-Befehle gegeben werden. Einzelheiten sind modellspezifisch.

Flagbits: bleiben, wie sie sind.

Dauer: 1 Taktzyklus.