5. Programmierung in Maschinensprache

5.1 Der Prozessor MSP430

5.2 Die Programmierumgebung zum MSP430-Assembler

5.3 Die Adressierungsarten des MSP430

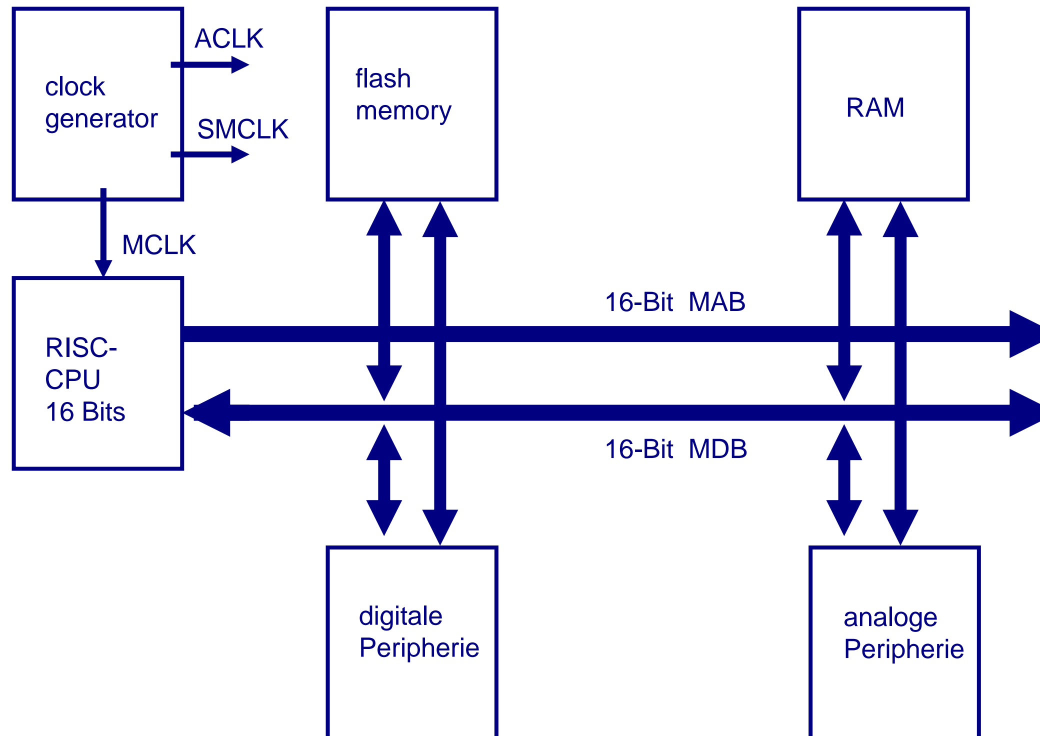
5.4 Der Befehlssatz des MSP 430

5.5 Unterprogrammtechnik

5.1 Der Prozessor MSP 430

- 16-Bit CPU
- niedrige Energieaufnahme (mehrere Jahre Laufzeit mit einer Batterie):
  - 0.1 µA RAM-Auffrischung
  - 0.8 µA real-time clock läuft
  - 250 µA normaler Betrieb
- von-Neumann-Architektur (Programm und Daten in demselben Adressraum)
- einfache Programmierbarkeit in Assembler und C
- jede Adressierungsart für jeden Operanden in jedem Befehl verwendbar!
- Haupt-Anwendungsbereich: embedded systems

Architektur des MSP 430



MCLK = Main CLock
ACLK = Auxiliary Clock
SMCLK = System Clock (für die Peripherie)

Die Register des MSP430

Der MSP 430 hat 16 Register, die mit R0 - R15 bezeichnet werden. Davon haben die ersten vier eine spezielle Bedeutung. Die ersten drei können im Simulator wahlweise als R0, R1, R2 oder als PC, SP, SR bezeichnet werden.

Der Befehlszähler

R0: **Program Counter**, kurz PC. Schreibt man einen Wert in R0, wird dieser vom Prozessor als die Adresse interpretiert, an der der nächste auszuführende Befehl steht. Ein Laden von R0 mit einer Adresse entspricht einem nachfolgenden Sprung an diese Adresse im Speicher.

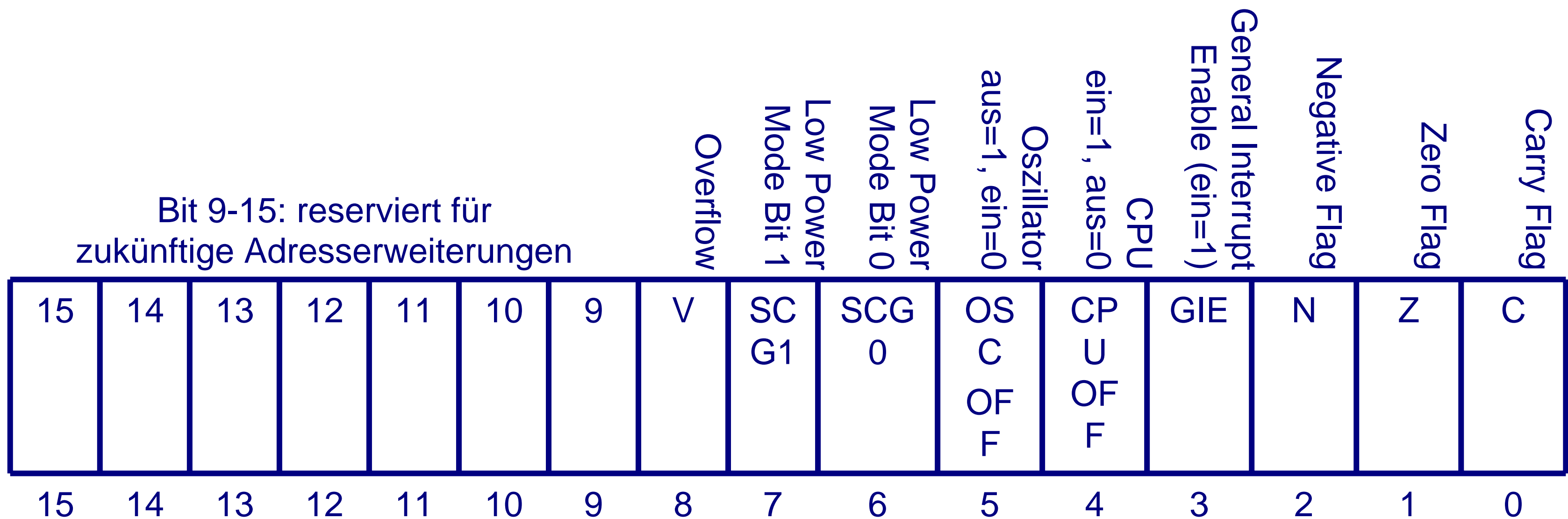
Regel: Der PC ist immer gerade: Befehle stehen im Speicher immer auf 16-Bit-Wortgrenzen.

Der Stackpointer

R1: Der **Stackpointer**, kurz SP. Er ist ebenfalls immer gerade. Der Stack wird mit seiner höchsten Adresse initialisiert und *wächst zu niedrigeren Adressen*. Der Stack steht nicht ausschließlich unter der Kontrolle des Programms, Interrupt-Routinen können ihn jederzeit verändern (“push”), müssen die Änderungen aber vor der Rückkehr revidieren (“pop”).

Das Statusregister

R2: **Status Register**, kurz SR. Jedes Bit hat hier eine spezielle Bedeutung.



Das Konstantenregister

R3: **Konstantenregister**.
Das Konstantenregister wird verwendet, um die Konstanten #-1, #0, #1, #2, #4 und #8 effizient bereit zu stellen.

Eine größere Konstante wie #61 muss hinter dem Befehlswort im Speicher abgelegt werden. Die sechs genannten Konstanten werden jedoch sehr häufig in Programmen verwendet. Sie finden bei Verwendung des Pseudo-Registers R3 in Form von Flags noch im eigentlichen Befehlswort Platz oder werden unter Verwendung von nicht benutzen Adressierungsarten des SR generiert, erfordern also keinen extra Speicherzugriff.

Folge: kürzeres Programm und schnellere Ausführung des Befehls

Konstante im Konstantenregister

Konstante	Hexadezimal	Rs	As	Verwendung
-1	FFFFh	R3	11b	alle Bits sind 1
0	0000h	R3	00b	alle Bits sind 0
1	0001h	R3	01b	Inkrement für Bytes
2	0002h	R3	10b	Inkrement für Worte
4	0004h	R2	10b	für Bit-Tests
8	0008h	R2	11b	für Bit-Tests

Der Aufbau des Speichers

0xFFE0-0xFFFF	16 Adressen für Unterprogramme	Interrupt-Vektoren
0x1100-0xFFDF	Wird in der Regel einmal vor Inbetriebnahme des Rechners geschrieben, kann jedoch auch in 512- Byte-Bänken während des Betriebes verändert werden.	ca. 60 kByte **Flash-ROM** für Firmware, Programme, Daten, Tabellen. Bei ausreichend Spannung auch vom Programm aus schreibbar!
0x1000-0x10FF	zwei kleine Bänke	2x128 Byte **Flash-ROM**
0x0A00-0x0FFF	für Programme via Scatter Flasher	Boot-Loader **ROM** (fix)
0x0200-0x09FF	nur 2kB schnelles RAM	**RAM** (für Variable, Stack)
0x0100-0x01FF	kein echter Speicher hinter diesen Adressen, sondern Verbindung mit der „Außenwelt“ („memory-mapped device buffers“)	**16-Bit-Peripherie** (memory-mapped). Nur wortweise (16 Bit) lesen!
0x0000-0x00FF		**8-Bit-Peripherie** (memory-mapped) Nur byteweise lesen!

Anschluss von Sensoren/Peripherie

- Einfache Datenübergabe durch „memory mapping“ der Register der Peripherie
- direkte Unterstützung für 8-Bit- und 16-Bit-Peripherie
- einfache Programmierung von Ausgaben auf externe Geräte ebenfalls durch „memory mapping“; also Schreiben der Steuerbits in Speicheradressen, die in Wirklichkeit externe Register sind.

Interrupts

Zum Vergleich: Klassische "proaktive" Programmierung

Die gesamte Ablaufsteuerung liegt vollständig in der Verantwortung des Hauptprogramms.

```
void main(...) // wird nach Programmstart zuerst angesprungen
{
    while(endless) // Läuft kontinuierlich bis Programmende
    {
            if hardware_event   then HandleHardwareEvent(...);
            if idle             then Sleep(100 ms);
    } // end while
} // end main
```

Interrupts als Alternative zum “busy waiting”

Es ist guter Stil, den Prozessor nicht die gesamte Zeit mit der while-Schleife zu be-schäftigen! Dies verbraucht in jedem Fall 100% der Prozessorleistung, egal wie schnell dieser ist.

Im Beispiel entschärft die Anweisung `sleep(100 ms)` diesen Sachverhalt, wenn kein Ereignis vorliegt. Der Prozess wird vom Scheduler des Betriebssystems dann für 100 ms nicht bedient, und es können andere Prozesse die Rechenressourcen erhalten. Dadurch verschlechtert sich die Reaktionszeit auf das Ereignis aber auch auf 50 ms im Mittel.

Interrupt-Struktur

Der MSP 430 hat eine mächtige Interrupt-Unterstützung in Hardware:

- Vektorisierung, kein Abfragen (polling) nötig
- Unterbrechbarkeit während der Interrupt-Bearbeitung ein-/ausschaltbar (Interrrupt-Schachtelung möglich)
- Sicherung der Rückkehr an die richtige Stelle über den Stack
- Prioritäten möglich

Reaktive Programmierung (1)

Das erwarten viele moderne Betriebssysteme:

```
void HandleHardwareEvent(...)
{
    ...
} // end HandleHardwareEvent


void main(...) // wird nach Programmstart angesprungen
{
    InitAllYouNeed(...);
    OperatingSystem.RegisterHWEventHandler(HandleHardwarEvent);
} // end main
```

Reaktive Programmierung (2)

Nach dem Programmstart kann die Anwendung alles Nötige initialisieren. Ansonsten definiert das Programm Funktionen für Ereignisse, die es abarbeiten möchte. Diese Funktionen werden beim Betriebssystem angemeldet. Tritt später ein Ereignis tatsächlich ein, so ruft das Betriebssystem die zuvor registrierte Programmfunktion auf.

Grundsätzlich funktioniert die Programmierung des MSP 430 reaktiv. Wenn nichts geschieht, schaltet sich der Prozessor so schnell wie möglich ab, um Strom zu sparen.

Der Watchdog

Das Programm kann sich von einer Reihe von Sensor-Zuständen über so genannte **Komparatoren** wecken lassen, wenn z. B. bestimmte Pegel einen Sollwert überschritten haben.

Der **Watchdog** (“Wachhund”) hat eine Sonderstellung innerhalb der Interrupt-Logik. Er soll überwachen, ob sich das Programm aufgehängt hat. Nach dem *Reset* des Prozessors ist er aktiv, kann jedoch vom Programm aus auch deaktiviert werden.

Der Watchdog wartet eine vorgegebene maximale Anzahl von Taktzyklen. Wenn er bis dahin nicht zurückgesetzt wurde, können abhängig vom Modus des Watchdog zwei Dinge passieren: Er kann einen Interrupt oder einen Reset auslösen. Damit soll verhindert werden, dass selten auftretende Probleme zum Totalausfall des Systems führen.

Der Watchdog arbeitet ähnlich wie der “Totmannknopf” in Zügen, der beim Einschlafen des Fahrers Katastrophen verhindern soll.

Steuerung des Watchdog

Das 16-Bit Word-Register ab Adresse 0x0120 steuert die genaue Funktionsweise des Watchdog. Wird der Watchdog aktiv, so wird der Watchdog-Interrupt ausgelöst.

WD Timer (Vec 0xFFF4)

WG HW Interrupt (Vec xFFFE)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Password (wird vergl. mit 0x5A)								WDTHOLD (0=aktiv / 1=inaktiv)	WDTNMIES	WDTNMI	WDTTMSEL (0=HW / 1=Timer)	WDTCNTCL (1 = Counter reset)	WDTSSEL	WDTTIS1	WDTTIS0

Timer-Einstellungen: WDTSSEL, WDTTIS1, WDTTIS0

5.3 Die Programmierumgebung für den MSP430-Assembler

Als Programmierumgebung verwenden wir in diesem Kurs den MSP IAR Assembler. Diese Umgebung erlaubt es uns, die Assembler-Programme zu editieren und dann gleich die Ausführung zu simulieren, d.h. ohne Verwendung eines „echten" MSP 430-Prozessors auszuführen.

Notation für Assembler-Befehle

```
[label[:]]   [operation]   [operanden]      [;Kommentar]
```

Alle Teile sind optional.

`[label[:]]` ist eine Marke (ein Sprungziel). Der Doppelpunkt ist optional, wenn die Marke in Spalte 1 beginnt.

`[operation]` ist ein Assemblerbefehl oder eine Assembler-Direktive (siehe später).

`[operanden]` sind die Operanden des Befehls oder der Direktive.

`[;Kommentar]` ist ein Kommentar. Es wird kein Maschinencode erzeugt.

Konstante (1)

Integer

Binär	1010b oder b‘1010‘
Oktal	1234q oder q‘1234‘
Dezimal	1234, -1, d‘1234‘
Hexadezimal	0ffffh oder 0xFFFF oder h‘ffff‘

Gleitkommazahlen

ohne Exponent	10.23, -43.21
mit Exponent	1.023E1, -4.321E20, 123.4E-5

Konstante (2)

ASCII-Zeichen

‘ABCD‘	einzelne Buchstaben
“ABCD“	ein String aus fünf Zeichen mit einer ASCII-Null am Ende (als Delimiter)
‘A“B‘	die Zeichen A‘B

Asssembler-Direktiven

Assembler-Direktiven werden nicht in Maschinenbefehle übersetzt, sondern steuern die Verhaltensweise des Assemblers. Beispiele sind

ORG	500	beginne mit der Generierung von Befehlen ab Adresse 500	
END		Ende des Assembler-Programms	
DS			define storage; reserviere Speicher
DS8		reserviere 8 Bits	
DS16		reserviere 16 Bits	
DC		0xFF	definiere eine Konstante
DC8	0xFF	definiere eine Konstante mit 8 Bits	
DC16	0xF8F4	definiere eine Konstante mit 16 Bits	

Es gibt noch viele andere Direktiven, siehe Manual.

5.3 Die Adressierungsarten des MSP 430

- Übersicht über die Adressierungsarten
- Register-Operanden
- Indexregister mit Distanz
- Symbolische Adresse (relativ zum PC)
- Absolute Adresse
- Indirekte Adresse
- Indirekte Adressierung mit Postinkrement
- Konstante als Operand
- Beispiele für die Programmierung mit dem Stackpointer

Übersicht über die Adressierungsarten

Unterschiedliche Adressierungsarten erlauben es, den Speicher in unterschiedlicher Art und Weise mit Hardwareunterstützung zu lesen und zu schreiben.

Einadress-Maschinen haben oft nur zwei explizite Adressierungsbefehle, LOAD und STORE, die den Speicher benutzen; alle anderen Operationen können nur auf die Register zugreifen.

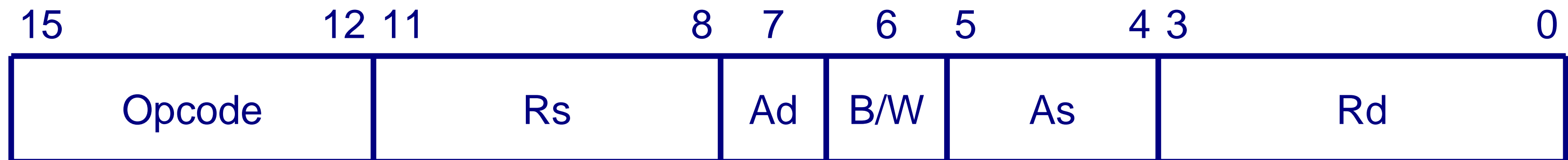
Zweiadress-Maschinen erlauben in der Regel sowohl Operanden in Registern als auch Operan-den im Speicher. Mit *einem* Befehl können *zwei* Operanden geladen, miteinander verknüpft und an einer der beiden Adressen wieder *abgespeichert* wer-den (Beispiel: ADD src,dst).

Der MSP430 kennt **sieben Adressierungsarten**. Alle sieben können für den Quell-operanden in Zweiadressbefehlen und für den Operanden in Einadressbefehlen ver-wendet werden, aber nur vier davon für den Zieloperanden in Zweiadressbefehlen.

Aufbau eines Befehls im MSP 430

Aufbau

Die Befehle mit zwei Operanden bestehen aus den folgenden sechs Feldern:



- Opcode: Das 4-Bit-Feld definiert den auszuführenden Befehl
- Rs: Das 4-Bit-Feld definiert das Register (R0-R15) der Quelle (*source*)
- Ad: Das Bit definiert die Adressierungsart des Ziels (*destination*)
- B/W: Das Bit definiert, ob Wort-Befehl (0) oder Byte-Befehl (1)
- As: Das 2-Bit-Feld definiert die Adressierungsart der Quelle (*source*)
- Rd: Das 4-Bit-Feld definiert das Register (R0-R15) des Ziels (*destination*).

Tabelle der Adressierungsarten

As/Ad	Adressierungsart	Syntax	Beschreibung
00/0	Register	Rn	der Operand steht im Register
01/1	Indexregister mit Distanz (indexed)	X(Rn)	(Rn + X) zeigt auf den Operanden
01/1	symbolisch	ADDR	(PC + X) zeigt auf den Operanden (Adressierung relativ zum Befehlzähler)
01/1	absolut	&ADDR	Das Speicherwort nach dem Befehl enthält die absolute Adresse des Op.
10/-	indirekt	@Rn	Rn wird als Zeiger auf den Operanden benutzt (Adressregister)
11/-	indirekt mit Postinkrement	@Rn+	Rn wird als Zeiger auf den Operanden benutzt, danach wird Rn inkrementiert
11/-	immediate	#N	Das Wort nach dem Befehl enthält die Konstante N.

Register-Operanden

Assembler-Code

MOV R10, R11

Beschreibung

Kopiere den Inhalt von R10 nach R11; R10 bleibt unverändert

	Vorher		Nachher
R10	0A023h	R10	0A023h
R11	0FA15h	R11	0A023h
PC	PC_{old}	PC	$PC_{old}+2$

Indexregister mit Distanz (1)

Assembler-Code

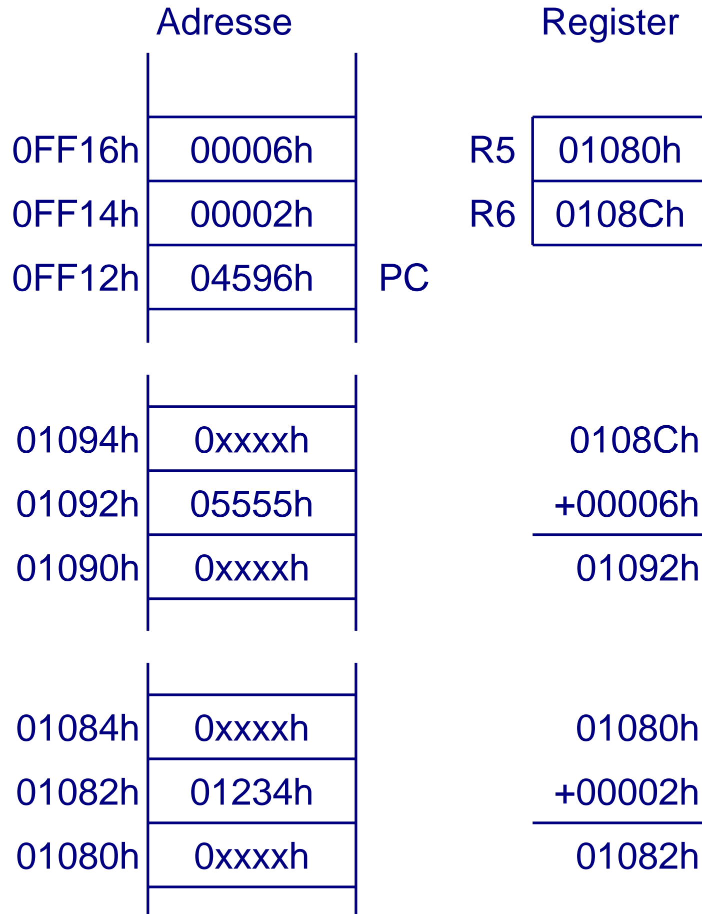
MOV 2(R5), 6(R6)

Beschreibung

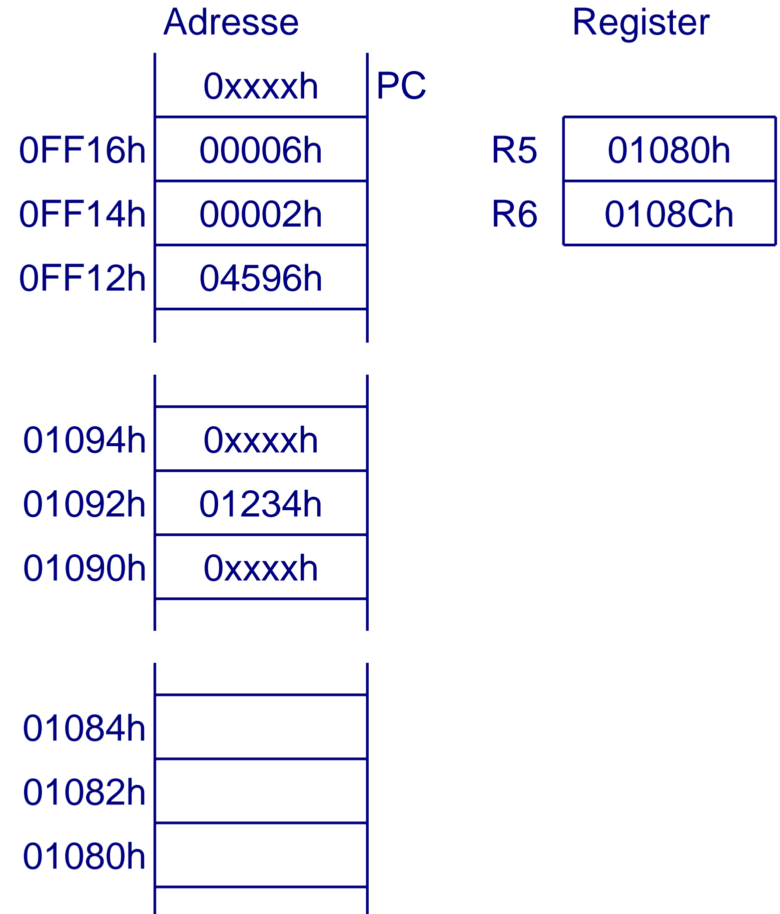
Kopiere den Inhalt von der Quelladresse (Inhalt von R5+2) zur Zieladresse (Inhalt von R6+6). Die Quell- und Zielregister (R5 und R6) werden nicht verändert.

Indexregister mit Distanz: Beispiel

Vorher:



Nachher:



Symbolische Adresse (relativ zum PC)

Assembler-Code

MOV EDE, TONI

entspricht:

MOV x(PC), y(PC)

Beschreibung

Kopiere den Inhalt von der Quelladresse EDE (Inhalt von PC+x) an die Zieladresse TONI (Inhalt von PC+y). Die Speicherwörter nach der Instruktion enthalten die gewünschte *Distanz* (Offset, relative Adresse) zwischen dem PC und dem Quell- bzw. Zieloperanden. Der Assembler berechnet beim Assemblieren die Offsets x und y automatisch und fügt sie in den Befehl ein.

Symbolische Adresse: Beispiel

Vorher:

Adresse		Register
0FF16h	011FEh	
0FF14h	0F102h	
0FF12h	04090h	PC

Adresse		Register
0F018h	0xxxxh	0FF14h
0F016h	0A123h	+0F102h
0F014h	0xxxxh	0F016h

Adresse		Register
01116h	0xxxxh	0FF16h
01114h	01234h	+011FEh
01112h	0xxxxh	01114h

Nachher:

Adresse		Register
	0xxxxh	PC
0FF16h	011FEh	
0FF14h	0F102h	
0FF12h	04090h	

Adresse	
0F018h	0xxxxh
0F016h	0A123h
0F014h	0xxxxh

Adresse	
01116h	0xxxxh
01114h	0A123h
01112h	0xxxxh

Absolute Adresse

Assembler-Code

MOV &EDE, &TONI

entspricht:

MOV x(0), y(0)

Beschreibung

Kopiere den Inhalt von der Quelladresse EDE zur Zieladresse TONI. Die Speicherwörter nach der Instruktion enthalten die *absolute* Adresse des Quell- bzw. Zieloperanden.

Absolute Adresse: Beispiel

Vorher:

Adresse	Speicher	Register
0FF16h	01114h	
0FF14h	0F016h	
0FF12h	04292h	PC
0F018h	0xxxxh	
0F016h	0A123h	
0F014h	0xxxxh	
01116h	0xxxxh	
01114h	01234h	
01112h	0xxxxh	

Nachher:

Adresse	Speicher	Register
	0xxxxh	PC
0FF16h	01114h	
0FF14h	0F016h	
0FF12h	04292h	
0F018h	0xxxxh	
0F016h	0A123h	
0F014h	0xxxxh	
01116h	0xxxxh	
01114h	0A123h	
01112h	0xxxxh	

Indirekte Adresse

Assembler Code

```
MOV.B @R10, 0(R11)
```

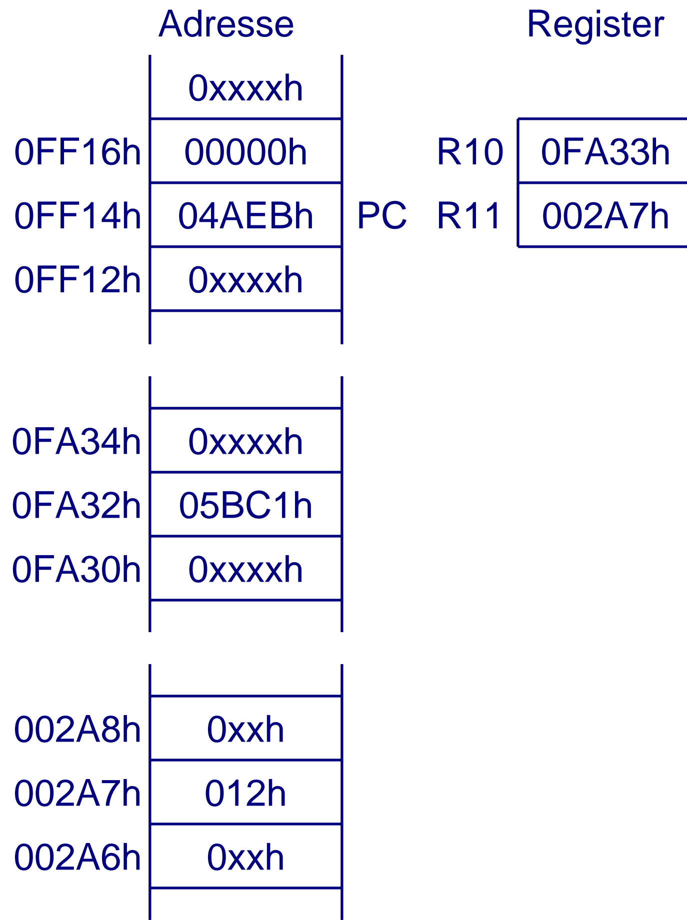
Beschreibung

Kopiere den Inhalt von der Quelladresse (R10 als Adressregister) zur Zieladresse (R11 als Adressregister). Die Register werden nicht verändert.

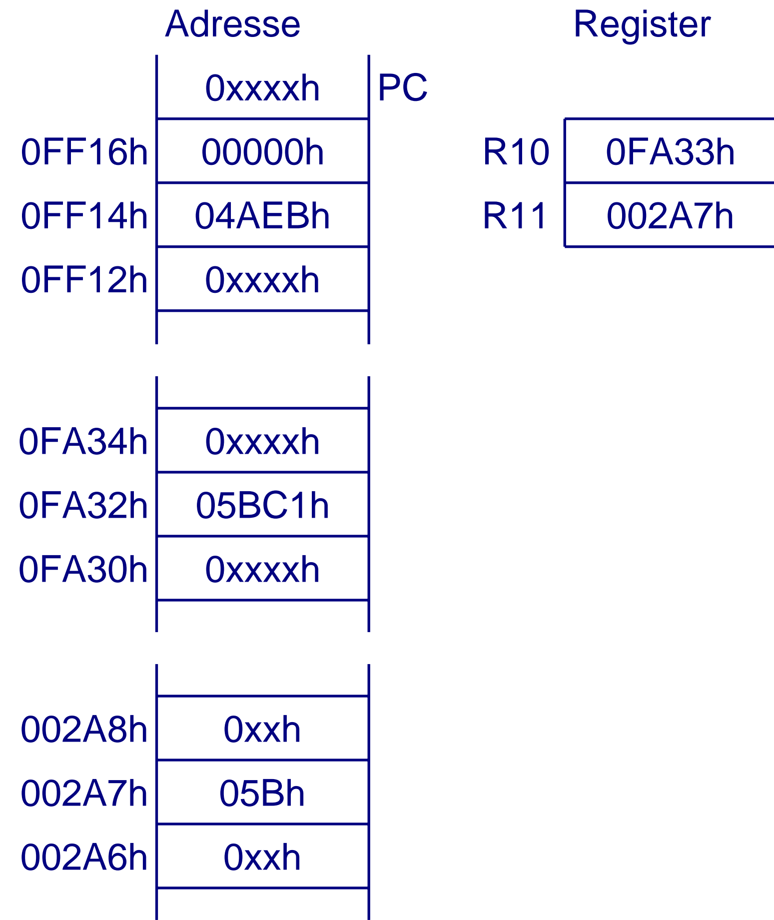
Bei 2-Operanden-Befehlen nur für den Quelloperanden möglich! Deshalb wird beim Zieloperanden der Modus „Indexregister mit Distanz“ mit einer Distanz von 0 verwen-det.

Indirekte Adresse: Beispiel

Vorher:



Nachher:



Register (indirekt)

`@Rn` Register n enthält die *Adresse* des Operanden

Beispiel:

```
mov #0FFFEh, R15  // lädt Register 15 mit Adresse FFFEh
mov @R15, R5      // schreibt den Wert an der
                  // Speicheradresse FFFEh in das
                  // Register 5
```

Register (indexiert)

`offset(Rn)` Die Adresse des Operanden ist die Summe aus dem Inhalt von Rn und dem Offset.

Beispiel:

```
mov #0FFFFh,R15       // lädt Reg. 15 mit Adresse
                      //   (bzw. Zahl) FFFFh
mov #00123h,-1(R15)// schreibt 123h ab Adresse FFFEh
```

Indirekte Adressierung mit Postinkrement

Assembler-Code

MOV @R10+, 0(R11)

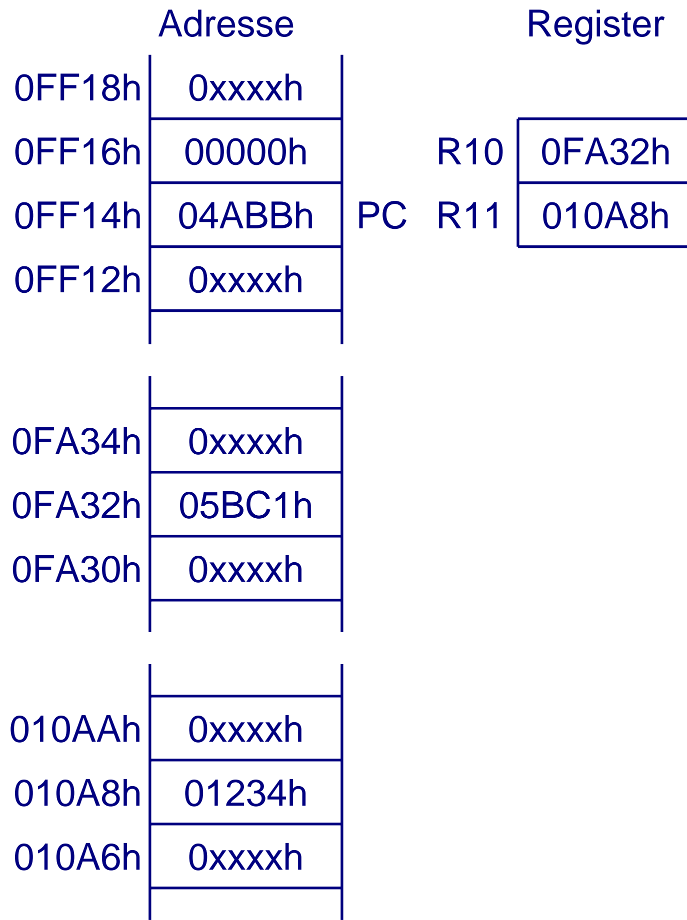
Beschreibung

Kopiere den Inhalt von der Quelladresse (Inhalt von R10) zur Zieladresse (Inhalt von R11). Register R10 wird bei einer Byte-Operation um 1, bei einer Wort-Operation um 2 inkrementiert. Das Inkrementieren erfolgt nach der Operation; das Register (hier R10) zeigt auf die nächste Adresse im Speicher, ohne einen zusätzlichen Befehl zu erfordern. Dies ist zum Beispiel bei der Verarbeitung von Arrays sehr nützlich.

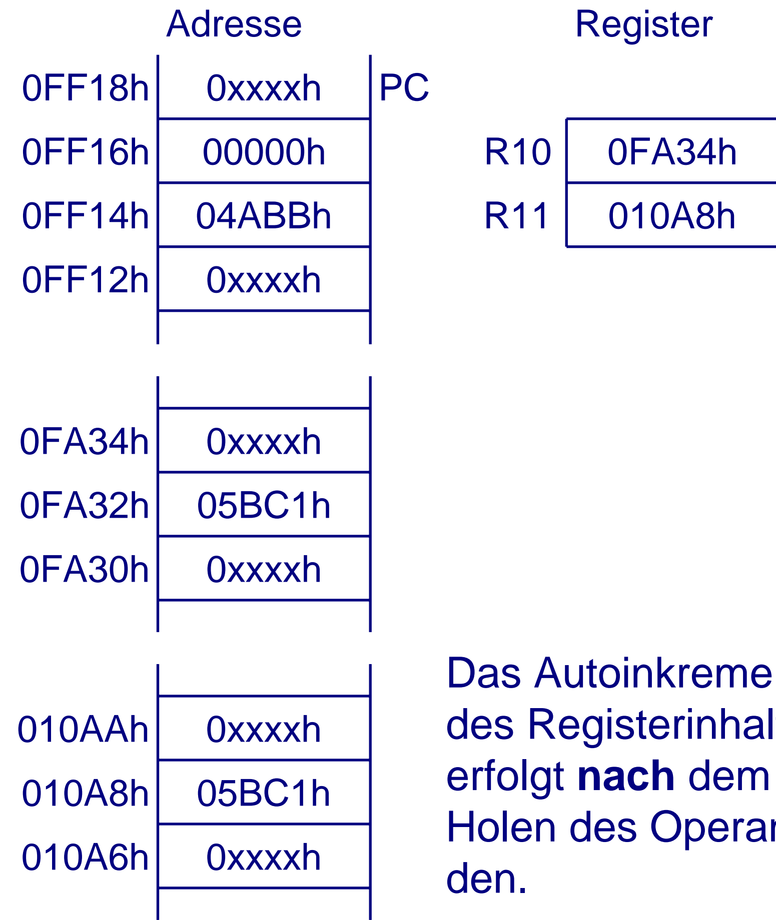
Bei 2-Operanden-Befehlen nur für den Quelloperanden möglich.

Indirekte Adressierung mit Postinkrement: Beispiel 1

Vorher:



Nachher:



Das Autoinkrement des Registerinhalts erfolgt **nach** dem Holen des Operanden.

Indirekte Adressierung mit Postinkrement: Beispiel 2

```
mov #01234h,&00200h     // schreibt 1234h ab Speicherzelle 200h
mov #00200h,R7          // lädt Register 7 mit Wert (Adresse) 200h
mov @R7+,R1             // lädt Register R1 mit 1234h
                        // danach steht 202h in R7
```

Was bewirkt folgender Ausdruck? `add @R15+,-2(R15)`

Der Ausdruck ist äquivalent zu den Anweisungen

add @R15, 0(R15)

add #2, R15

Konstante als Operand (immediate)

Assembler-Code

MOV #45h, TONI

Beschreibung

Kopiere die Konstante 45h, die sich im Wort nach der Instruktion befindet, an die Zieladresse TONI.

Bei 2-Operanden-Befehlen nur für den Quelloperanden möglich.

Konstante als Operand: Beispiel

Vorher:

Adresse		Register
0FF16h	01192h	
0FF14h	00045h	
0FF12h	040B0h	PC

Adresse		Register
010AAh	0xxxxh	0FF16h
010A8h	01234h	+01192h
010A6h	0xxxxh	010A8h

Nachher:

Adresse		Register
0FF18h	0xxxxh	PC
0FF16h	01192h	
0FF14h	00045h	
0FF12h	040B0h	

Adresse		Register
010AAh	0xxxxh	
010A8h	00045h	
010A6h	0xxxxh	

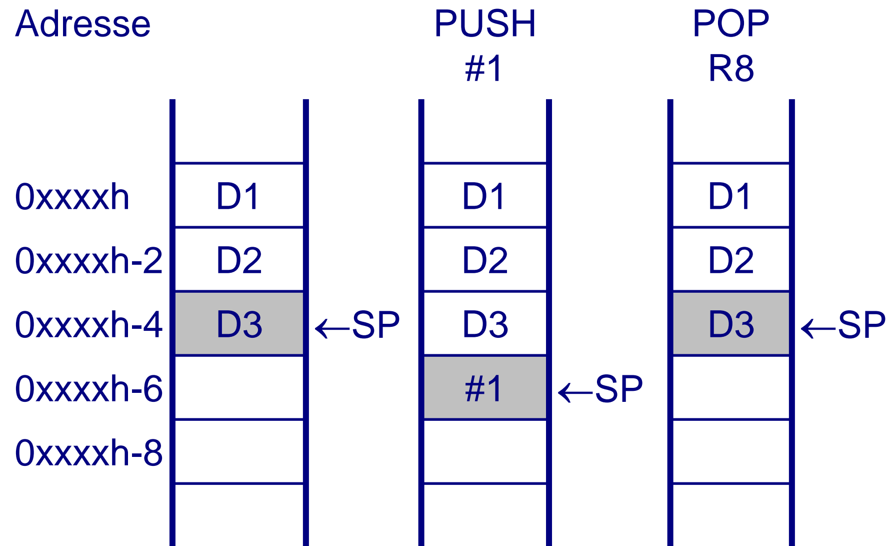
Die “Konstantenregister” R2 und R3

Register	Bits für Adressierungsart (As)	Konstante	Bemerkung
R2	00	------	Registermodus für SR
R2	01	(0)	absolute Adressierung
R2	10	00004h	+4
R2	11	00008h	+8
R3	00	00000h	0
R3	01	00001h	+1
R3	10	00002h	+2
R3	11	0FFFFh	-1

Vorteile dieser Art der Konstantengenerierung:

- keine speziellen Befehle erforderlich
- geringere Programmspeicheranforderungen, da kein zusätzliches Wort für die sechs meistbenutzten Konstanten erforderlich ist.
- geringere Ausführungszeit für die Befehle, da kein separater Speicherzugriff zum Holen der Konstanten nötig ist.

Beispiele für die Programmierung mit dem SP



Merke: Der Stackpointer ist R1

```
MOV     R1, R4       ;SP -> R4
MOV     @R1, R5      ;D3 (TOS) ->R5
MOV     2 (R1), R6   ;D2 ->R6
MOV     R7, 0 (R1)   ;überschreibe TOS mit R7
MOV     R8, 4 (R1)   ;verändere D1
PUSH    R12          ;R12→0xxxxh-6, SP an dieselbe Adresse
POP     R12          ;0xxxxh-6 → R12, SP zeigt auf 0xxxxh-4
MOV     @R1+, R5     ;D3 → R5 (POP vom Stack), bewirkt dasselbe  wie POP
```

5.4 Der Befehlssatz des MSP430

Die RISC-Struktur der MSP430-Familie ist am deutlichsten sichtbar, wenn man die Zahl der Befehlsformate betrachtet: **Nur drei Formate** existieren, bei anderen 16-bit-Rechnern kommen oft zehn oder mehr Befehlsformate zusammen.

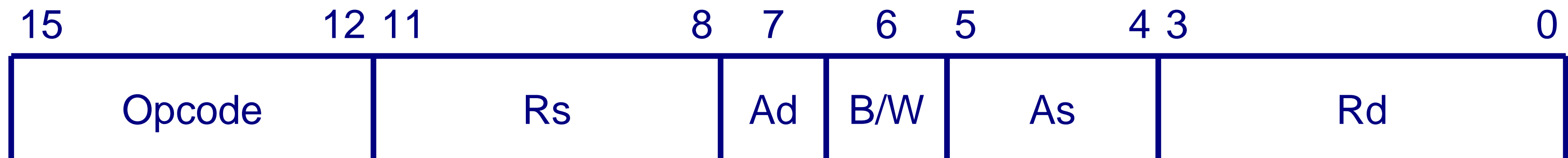
Die verwendeten Zeichen bei der Behandlung der Status-Bits haben auf den nachfolgenden Folien folgende Bedeutung:

*	Das Bit wird beeinflusst
1	Das Bit wird gesetzt
0	Das Bit wird zurück gesetzt
@Z	Das Bit ist das negierte Zero Bit (= .not. Z)
-	Das Bit wird nicht beeinflusst, es behält den vorherigen Wert.

Zweiadressbefehle (1)

Aufbau

Die Befehle mit zwei Operanden bestehen aus den folgenden sechs Feldern:



- Opcode: Das 4-Bit-Feld definiert den auszuführenden Befehl
- Rs: Das 4-Bit-Feld definiert das Register (R0-R15) der Quelle (*source*)
- Ad: Das Bit definiert die Adressierungsart des Ziels (*destination*)
- B/W: Das Bit definiert, ob Wort-Befehl (0) oder Byte-Befehl (1)
- As: Das 2-Bit-Feld definiert die Adressierungsart der Quelle (*source*)
- Rd: Das 4-Bit-Feld definiert das Register (R0-R15) des Ziels (*destination*).

Zweiadressbefehle (2)

Die in den Befehlsbeschreibungen verwendeten Begriffe *src* und *dst* sind Kombinationen von Registern mit einer Adressierungsart:

src: Source-Operand. Register Rs zusammen mit der Adressierungsart definiert in As

dst: Destination-Operand. Register Rd zusammen mit der Adressierungsart definiert in Ad.

Nur zwei Bits für die Adressierungsart der Quelle?

Im MSP 430 sind nur vier Adressierungsarten wirklich in Hardware implementiert:

1. Das Register ist der Operand (register mode).
2. Das Register ist ein Adressregister, d. h., es enthält die Adresse des Operanden im Speicher (indirect mode).
3. Das Register ist ein Adressregister mit Auto-Inkrement (indirect mode with auto-increment).
4. Das Register ist ein Indexregister, also ein Adressregister mit Distanz (indexed mode).

Alle anderen Adressierungsarten werden durch Verwendung des PC und des SR als Register im Befehl gewonnen! Beispiele: Der „symbolic mode“ (also Adressierung re-lativ zum PC) wird durch „indexed mode“ mit dem PC als Register gewonnen. Der „absolute mode“ wird durch Adressierung mit dem SR als Register gewonnen, das durch eine Gatterschaltung in der CPU kurzzeitig abgesperrt (auf null gehalten) wird.

Tabelle der Zweiadressbefehle

Die zwölf implementierten Zweiadressbefehle sind:

			V	N	Z	C
ADD	src,dst	Addiere src zu dst	*	*	*	*
ADDC	src,dst	Addiere src + carry zu dst	*	*	*	*
AND	src,dst	src .and. dst →dst	0	*	*	@Z
BIC	src,dst	.not. src .and. dst →dst	-	-	-	-
BIS	src,dst	src .or. dst →dst	-	-	-	-
BIT	src,dst	src .and. dst → SR	0	*	*	@Z
CMP	src,dst	Vergleiche src und dst: (dst-src)→SR	*	*	*	*
DADD	src,dst	Addiere src + carry dezimal zur dst	*	*	*	*
MOV	src,dst	Kopiere src nach dst	-	-	-	-
SUB	src,dst	Subtrahiere src von dst (dst-src→dst)	*	*	*	*
SUBC	src,dst	Subtrahiere src mit carry von dst (dst + .not. src + C → dst)	*	*	*	*
XOR	src,dst	src .xor. dst → dst	*	*	*	@Z

Zweiadressbefehle als Byte-Befehle

Die implementierten Zweiadressbefehle sind alle auch als Byte-Befehle verwendbar. Als Operand im Speicher wird genau das adressierte Byte verwendet, bei den Registern R0 bis R15 das untere Byte (least significant byte).

Der Befehl MOV

MOV	Kopiere Source-Wort in Destination-Wort
MOV.B	Kopiere Source-Byte in Destination-Byte

Name	Move source to destination
Syntax	MOV src,dst oder MOV.W src,dst MOV.B src,dst
Operation	src → dst
Beschreibung	Der Source-Operand wird in den Destination-Operanden kopiert. Der Source-Operand wird nicht verändert. **Das Statusregister wird nicht verändert!**

Beispiel für MOV

Eine Wort-Tabelle, auf die R13 zeigt, soll in einen RAM-Teil beginnend bei Label RAMT koopiert werden. Es sollen 20h Worte übertragen werden. R14 zählt Bytes! (Der Befehl INCD ist "increment double".)

```
        MOV    #BEGIN,R13          ;Beginn der Source-Tabelle
        MOV    #0,R14              ;0 → R14 (Tabellenoffset)
LOOP    MOV    @R13+,RAMT(R14)     ;nächstes Wort kopieren
        INCD   R14                 ;Index = Index +2
        CMP    #2*20h,R14          ;schon 20 Worte kopiert?
        JLO    LOOP                ;nein, weiter kopieren
        …                          ;ja, fertig
```

Der Befehl ADD (1)

ADD	Addiere das Source-Wort auf das Destination-Wort
ADD.B	Addiere das Source-Byte auf das Destination-Byte

Name	Add source to destination
Syntax	ADD src,dst oder ADD.W src, dst ADD.B src, dst
Operation	src+dst → dst
Beschreibung	Der Source-Operand wird auf den Destination-Operanden addiert. Der vorherige Inhalt des Destination-Operanden wird mit dem Ergebnis überschrieben. Der Source-Operand bleibt unverändert.

Der Befehl ADD (2)

Statusbits

- N: Wird gesetzt, falls das Ergebnis negativ ist (MSB=1), wird zurückgesetzt, falls das Ergebnis positiv ist (MSB=0)
- Z: Wird gesetzt, falls das Ergebnis Null ist, wird zurückgesetzt, falls das Ergebnis nicht null ist
- C: Wird gesetzt, wenn bei der Operation ein Übertrag entsteht, wird andernfalls zurückgesetzt
- V: Wird gesetzt, wenn das Ergebnis der Addition zweier positiver Zahlen negativ ist oder wenn das Ergebnis der Addition zweier negativer Zahlen positiv ist. Zeigt also an, ob wegen des Zweierkomplements ein Rechenfehler entstanden ist. Wird andernfalls zurückgesetzt.

Beispiel für ADD

Beispiel: R5 wird um 10 erhöht. Falls dabei ein Übertrag (carry) auftritt, wird zur Marke TONI gesprungen.

```
ADD   #10,R5          ;Addiere 10 zu R5
JC    TONI            ;Carry=1: weiter bei TONI
…                     ;Carry =0: hier weiter
```

Der Befehl AND (1)

AND	Logische UND-Verknüpfung von Source- und Destination-Wort
AND.B	Logische UND-Verknüpfung von Source- und Destination-Bytes
Name	AND
Syntax	AND src,dst oder AND.W src,dst AND.B src,dst
Operation	src .and. dst → dst
Beschreibung	Der Source-Operand und der Destination-Operand werden bitweise mit der logischen Funktion UND verbunden. Der vorherige Inhalt des Destination-Operanden wird mit dem Ergebnis überschrieben. Der Source-Operand bleibt unverändert.

Der Befehl AND (2)

Statusbits

Statusbits	N	Wird gesetzt, falls das Ergebnis negativ ist (MSB=1), wird zurück-gesetzt, falls das Ergebnis positiv ist (MSB=0)
	Z	Wird gesetzt, falls das Ergebnis null ist, wird zurückgesetzt, falls das Ergebnis nicht null ist
	C	Wird gesetzt, wenn das Ergebnis nicht null ist, wird zurückgesetzt, falls das Ergebnis null ist. Das Carry-Bit entspricht damit dem invertierten Zero-Bit! C = NOT Z
	V	Wird immer zurückgesetzt (gelöscht)

Beispiel für AND

Beispiel: Die Bits, die in R5 gesetzt sind, werden als Maske für das Wort TOM im Speicher verwendet. Falls das Resultat null ist, wird zur Marke TONI gesprungen.

```
AND    R5,TOM          ;TOM .and. R5 → TOM
JZ     TONI            ;Resultat = 0
...                    ;Resultat ist nicht null
```

Der Befehl CMP (1)

CMP	Vergleiche Source- und Destination-Worte
CMP.B	Vergleiche Source- und Destination-Bytes
Name	Compare
Syntax	CMP src,dst oder CMP.W src, dst CMP.B src, dst
Operation	(dst-src) (intern: dst + .not. src +1)
Beschreibung	Der Source-Operand wird vom Destination-Operanden dst subtrahiert. Dies geschieht durch Addition des Zweierkomplements von src (dargestellt durch .not. src +1). Die beiden Operanden werden nicht verändert, das Resultat wird nicht gespeichert, nur die Status-Bits in SR werden entsprechend verändert! Der Byte-Befehl CMP.B löscht das obere Byte eines als Destination verwendeten Registers **nicht**, da kein Schreibbefehl auf das Register erfolgt (es wird nur gelesen).

Der Befehl CMP (2)

Statusbits

N Wird gesetzt, falls das Ergebnis negativ ist (src > dst), wird zurückgesetzt, falls das Ergebnis positiv oder null ist (src $\leq$ dst)

Z Wird gesetzt, falls das Ergebnis null ist (src = dst). Wird zurückgesetzt, falls das Ergebnis nicht null ist (src $\neq$ dst)

C Wird gesetzt, wenn ein Übertrag vom MSB entsteht. Wird zurückgesetzt, falls kein Übertrag vom MSB entsteht.

V Wird gesetzt, falls die Subtraktion eines negativen Source-Operanden von einem positiven Destination-Operanden ein negatives Resultat ergibt. Wird auch gesetzt, falls die Subtraktion eines positiven Source-Operanden von einem negativen Destination-Operanden ein positiven Resultat ergibt. Andernfalls wird es zurück gesetzt (gelöscht).

Beispiele für CMP (1)

Beispiel 1: R5 und R6 werden verglichen. Falls die Inhalte der beiden Register gleich sind, fährt das Programm an der Marke EQUAL fort.

```
CMP   R5,R6          ;R5=R6?
JEQ   EQUAL          ;Ja, weiter bei Marke EQUAL
…                    ;Nein, hier weiter
```

Beispiele für CMP (2)

Beispiel 2: Zwei Tasten, die an den Port 1 angeschlossen sind, werden getestet:

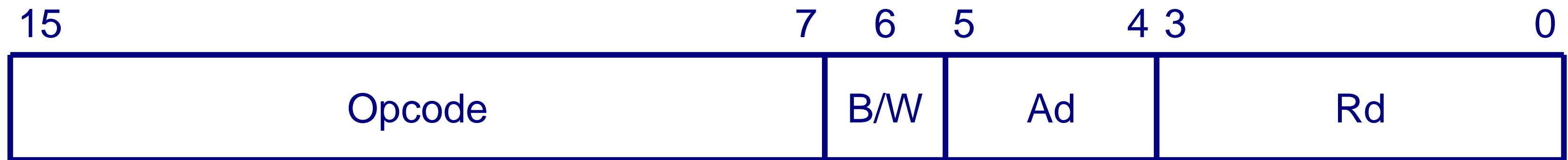
- Falls Taste KEY1 gedrückt ist, soll zur Marke MENU1 gesprungen werden.
- Falls Taste KEY2 gedrückt ist, soll zur Marke MENU2 gesprungen werden.
- Falls beide oder keine Tasten gedrückt sind, erfolgt keine Aktion.

```
CMP.B   #KEY1,&P1IN     ;ist KEY1 betätigt?
JNZ     MENU1           ;Ja, weiter bei MENU1
CMP.B   #KEY2,&P1IN     ;Nein, ist KEY2 betätigt?
JNZ     MENU2           ;Ja, weiter bei MENU2
...                     ;nichts tun
```

Einadressbefehle

Aufbau

Die sieben implementierten Befehle mit einem Operanden (Single-Operand Instructions) sind aus den folgenden vier Feldern des Befehlswortes aufgebaut:



- Opcode: Das 9-bit-Feld definiert den auszuführenden Befehl.
- B/W: Das Bit definiert, ob Wort-Befehl (0) oder Byte-Befehl (1).
- Ad: Das 2-Bit-Feld definiert die Adressierungsart der Destination.
- Rd: Das 4-Bit-Feld definiert das Register (R0-R15) der Destination.

Tabelle der Einadressbefehle

Die implementierten Einadressbefehle sind:

			V	N	Z	C
CALL	dst	Subroutinen-Aufruf	-	-	-	-
PUSH	dst	Operand auf dem Stack speichern	-	-	-	-
RETI	dst	Rückkehr von einem Interrupt	*	*	*	*
RRA	dst	Arithmetisches Rechtsschieben	0	*	*	*
RRC	dst	Logisches Rechtsschieben durch den carry	*	*	*	*
SWPB	dst	Auswechseln der beiden Bytes eines Worts	-	-	-	-
SXT	dst	Vorzeichen in oberes Byte erweitern	0	*	*	@Z

PUSH.B	src	Kopiere src auf den Stack	-	-	-	-
RRA.B	dst	Arithmetisches Rechtsschieben	0	*	*	*
RRC.B	dst	Logisches Rechtsschieben durch den carry	*	*	*	*

Der Befehl PUSH

PUSH	Kopiere Source-Wort auf den Stack
PUSH.B	Kopiere Source-Byte auf den Stack
Name	push
Syntax	PUSH src oder PUSH.W src PUSH.B src
Operation	SP – 2 → SP src → @SP
Beschreibung	Wort-Befehl: Der SP wird um 2 erniedrigt. Der Inhalt des Source-Wortes wird in das Speicherwort geschrieben, auf das der SP zeigt. Der Source-Operand bleibt unverändert. Byte-Befehl: der SP wird um 2 erniedrigt, da er immer auf Worte zeigt! Der Inhalt des Source-Bytes wird in das untere Byte des Speicherwortes geschrieben, auf das der SP zeigt. **Das obere Byte bleibt unverändert.** Das Statusregister bleibt unverändert.

Beispiele für PUSH

Beispiel 1:

```
PUSH      R8          ;legt den Inhalt von R8 auf dem Stack
                       ab
```

Beispiel 2:

```
PUSH.B    TONI        ;oberes Byte vom TOS (top of stack)
                       wird nicht verändert!
```

Der Befehl CALL (1)

CALL	Unterprogrammaufruf	
Name	Call Subroutine	
Syntax	CALL	dst
Operation	dst → tmp	Destination wird berechnet und gespeichert
	SP–2 → SP	der Stack-Pointer wird dekrementiert
	PC → @SP	Die Rückkehradresse wird auf den Stack geschrie-ben. Details siehe weiter unten.
	tmp → PC	Destination wird in den PC geschrieben: Das Pro-gramm fährt an der Startadresse des Unterpro-gramms fort.

Der Befehl CALL (2)

Beschreibung

Es wird ein Unterprogrammsprung an eine Adresse, die an beliebiger Stelle des Adressraums liegen kann, ausgeführt. Alle sieben Adressierungsarten können verwendet werden. Die berechnete Rückkehradresse (die Adresse des Befehls, der auf den CALL-Befehl folgt) wird auf dem Stack gespeichert (der RET-Befehl verwendet diese Adresse zur Rückkehr aus dem Unterprogramm).

Wichtige Anmerkung: Es wird nicht zu der Adresse *dst* gesprungen, sondern zu der Adresse, die in dem Wort an Adresse *dst* steht! Diese indirekte Adressierung ist unerwartet und führt oft zu Programmierfehlern!

Statusbits

Die Statusbits im Statusregister werden nicht verändert. Das Unterprogramm kann also den Status des Hauptprogramms auswerten.

Wohin zeigt der PC gerade?

Beim Ablauf des Programms ist zu beachten, dass im MSP 430 der Befehlszähler (PC) immer sofort nach dem Lesen eines Befehlswortes (hier des CALLs) um 2 erhöht wird. Er zeigt also bereits auf den nachfolgenden Befehl, wenn der CALL-Befehl die Adressierungsarten Register Mode, Indirect Mode oder Indirect Auto-increment Mode benutzt, oder aber auf das zweite Wort des CALL-Befehls selbst, wenn dieser den Immediate Mode, Symbolic Mode, Absolute Mode oder Indexed Mode benutzt. In den letzten vier Fällen wird der PC vor dem Speichern auf dem Stack nochmals und 2 erhöht, um das zweite Wort des CALL-Befehls zu übersprin-gen.

Beispiel für CALL

Beispiel : Immediate Mode: Die häufigste Adressierungsart für den CALL-Befehl: Die Marke, an der die Subroutine startet, wird damit adressiert. Auch die absolute Angabe einer Startadresse, wie z. B. #0AA00h, ist möglich.

```
CALL  #EXEC            ;Sprung zur Subroutine EXEC
CALL  #0AA00h          ;Sprung zur Adresse 0AA00h
```

Software-"Emulation" des CALL-Befehls

CALL ist eine effiziente Har*dware-Implementierung* der folgenden Befehlsfolge:

```
call dst            entspricht:

sub 2,SP            ; SP zum Retten der Rücksprungadresse setzen
mov PC,@SP          ; Rücksprungadresse retten
mov dst,PC          ; der Sprung erfolgt durch direkte Manipulation des
                    ; PC
```

Der Befehl RETI (1)

RETI	Rückkehr vom Interrupt
Name	Returm from Interrupt
Syntax	RETI
Operation	@SP+2 → SR Statusregister wieder herstellen SP+2 → SP SP auf nächstes Wort @SP+2 → PC Rücksprungadresse in den PC laden SP+2 → SP SP auf nächstes Wort

Die Operation besteht aus zwei POP-Befehlen, die in der CPU im Ablauf zusammengefasst sind. Dadurch kann ein anstehender Interrupt nicht zwischen den beiden Befehlen aktiv werden.

Der Befehl RETI (2)

Beschreibung

Das Statusregister SR wird auf den Wert gebracht, den es zum Zeitpunkt der Unterbrechung durch den Interrupt hatte. Dazu wird es mit dem Wort überschrieben, auf das der Stack Pointer zeigt. Der Stack Pointer wird danach um zwei erhöht (und zeigt nun auf die gespeicherte Rücksprungadresse).

Der Program Counter PC wird auf den Wert gebracht, den er bei der Unterbrechung durch den Interrupt hatte. Es ist die Adresse nach dem letzten noch ausgeführten Befehl vor der Zulassung des Interrupts. Dazu wird der PC mit dem Wort überschrieben, auf das der Stack Pointer zeigt. Der Stack Pointer SP wird danach um 2 erhöht.

Statusbits

Werden vom Stack geladen, wie oben beschrieben.

Beispiel für RETI

Eine Interrupt-Routine INTR rettet zu Beginn die Register R8 und R7, um sie für eigene Zwecke zu verwenden. Vor der Rückkehr werden die beiden Register wieder vom Stack geladen. Achtung: umgekehrte Reihenfolge bei PUSH und POP!

```
INTR   PUSH   R8        ;R8 auf Stack retten
       PUSH   R7        ;R7 auf Stack retten
       …
       POP    R7        ;R7 wieder herstellen
       POP    R8        ;R8 wieder herstellen
       RETI             ;Rückkehr ins Hauptprogramm
```

Anmerkung: Der Befehl RET (return from subroutine) ist ein *emulierter* Befehl! Er ist das Pendant zum CALL-Befehl für normale Unterprogrammsprünge.

Sprungbefehle

Aufbau

Opcode: Das 3-Bit-Feld mit dem Wert 001b definiert den Sprungbefehl.

Bedingung: Das 3-Bit-Feld definiert die Bedingung für den auszuführenden Sprung. **Dazu werden die Status-Bits im SR ausgewertet.**

Offset: Das 10-Bit-Feld definiert den Wort-Offset im Zweierkomplement (Bit 9 ist dabei das Vorzeichen)

Der Offset ist der Wert (in Worten), der zum Befehlszähler addiert wird, falls die Sprungbedingung erfüllt ist. Es sind also Sprünge über –511 bis +512 Worte möglich. Das Offsets wird im Zweierkomplement angegeben, kann also auch negativ sein.

Tabelle der Sprungbefehle

Die Sprungbefehle sind:

Preisfrage: Wie kann man 12 Bedingungen mit 3 Bits codieren?

JC	Marke	Springe falls Übertrag-Bit = 1 (C=1)
JNC	Marke	Springe falls Übertrag-Bit = 0 (C=0)
JLO	Marke	Springe falls dst < src (C=0) ohne Vorzeichen
JHS	Marke	Springe falls dst ≥ src (C=1) ohne Vorzeichen
JEQ	Marke	Springe falls dst = src (Z=1)
JZ	Marke	Springe falls Zero-Bit = 1 (Z=1)
JGE	Marke	Springe falls dst ≥ src (N .xor. V = 0) mit Vorz.
JL	Marke	Springe falls dst < src (N .xor. V = 1) mit Vorz.
JMP	Marke	Springe immer
JN	Marke	Springe falls Negativ-Bit = 1 (N=1)
JNE	Marke	Springe falls dst ≠ src (Z=0)
JNZ	Marke	Springe falls Zero-Bit = 0 (Z=0)

Warum wird manchmal das C-Bit einbezogen?

Beispiel 1 für Überlauf:

`cmp.b 1,-127` berechnet wird -127-1 = -128

-128 ist bereits außerhalb des Wertebereichs. Gesetzt wird daher nur das Carry-Bit.

Beispiel 2 für Überlauf:

`JL` „Jump on Less than" Jump if N==1 xor C==1 (if dest < src)

Auch hier kann wegen des Überlaufs das Ergebnis der Subtraktion falsch sein, jedoch wird in Abhängigkeit von den Bits wenigstens richtig gesprungen.

Der Befehl JZ/JEQ (1)

JZ	Springe, falls Zero-Bit gesetzt
JEQ	Springe , falls gleich
Name	Jump if Zero Jump if Equal
Syntax	JZ label JEQ label
Operation	falls Z=1: PC + 2 x Offset → PC falls Z=0: Den folgenden Befehl ausführen

Der Befehl JZ/JEQ (2)

Beschreibung

Das Zero-Bit Z des Statusregisters SR wird getestet. Falls es gesetzt ist, wird der vorzeichenbehaftete 10-Bit-Offset, der in den LSBs des Befehls enthalten ist, mit Zwei multipliziert und zum Befehlszähler PC addiert. Falls Z nicht gesetzt ist, wird der Befehl nach dem JZ/JEQ ausgeführt.

JZ wird gerne für den Vergleich mit Null verwendet, wenn es sich um *Bitmuster* handelt, zum Beispiel nach dem Befehl TST.

JEQ wird gerne nach dem Vergleich zweier *arithmetischer Operanden* verwendet, zum Beispiel nach dem Befehl CMP.

Statusbits

Die Statusbits werden nicht verändert! Es können also noch weitere bedingte Sprünge folgen, die sich auf denselben Inhalt des Statusregisters beziehen.

Beispiel für JEQ

Falls R6 den gleichen Inhalt wie das adressierte Tabellenwort hat, soll zur Marke LABEL gesprungen werden.

```
CMP     R6,TAB(R5)          ;R6 und Tabellenwort gleich?
JEQ     LABEL               ;Ja: zu LABEL springen
…                           ;Nein: normal weiter
```

Emulierte Befehle

Die emulierten Befehle sind nicht in der MSP430-Hardware implementiert. Sie werden durch die Verwendung des Konstantengenerators zusammen mit den implementierten Befehlen gebildet.

So wird beispielsweise der Befehl

```
INV   dst           ;invertiere den dst-Operanden
```

durch den implementierten Befehl

```
XOR   #0FFFFh,dst   ;invertiere den dst-Operanden
```

emuliert. Die Konstante 0FFFFh (-1) ist bekanntlich im Konstantengenerator verfügbar.

Tabelle der emulierten Befehle (1)

Die 24 emulierten Befehle für Wort-Operanden sind:

			V	N	Z	C
ADC	dst	Addiere Carry zu dst	*	*	*	*
BR	dst	Springe indirekt dst	-	-	-	-
CLR	dst	Setze dst auf Null	-	-	-	-
CLRC		Lösche Carry-Bit	-	-	-	0
CLRN		Lösche Negativ-Bit	-	0	-	-
CLRZ		Lösche Zero-Bit	-	-	0	-
DADC	dst	Addiere carry zu dst (dezimal)	*	*	*	*
DEC	dst	Dekrementiere dst um 1	*	*	*	*
DECD	dst	dst–2 → dst	*	*	*	*
DINT		Interrupts ausschalten	-	-	-	-
EINT		Interrupts einschalten	-	-	-	-
INC	dst	Inkrementiere dst um 1	*	*	*	*

Tabelle der emulierten Befehle (2)

			V	N	Z	C
INCD	dst	dst + 2 → dst	*	*	*	*
INV	dst	Invertiere dst	*	*	*	@Z
NOP		Keine Aktivität (no operation)	-	-	-	-
POP	dst	Hole Wort vom Stack und speichere es in dst	-	-	-	-
RET		Rückkehr vom Unterprogramm (return)	-	-	-	-
RLA	dst	Arithm. Linksschieben der dst	*	*	*	*
RLC	dst	Logisches Linksschieben der dst durch den carry	*	*	*	*
SBC	dst	Subtrahiere carry von dst	*	*	*	*
SETC		Setze Carry-Bit	-	-	-	1
SETN		Setze Negativ-Bit	-	1	-	-
SETZ		Setze Zero-Bit	-	-	1	-
TST	dst	Teste dst	0	*	*	1