

10. Der Befehlssatz des MSP 430

10.1 Befehlsformate

10.2 Zweiadressbefehle

10.3 Einadressbefehle

10.4 Sprungbefehle

10.5 Emulierte Befehle

10.1 Befehlsformate

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.

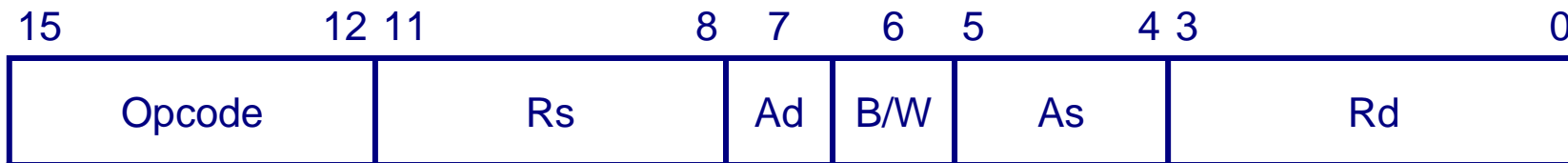
Die verwendeten Zeichen bei der Behandlung der Status-Bits haben auf den nächsten Folien folgende Bedeutungen:

- * 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.

10.2 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 relativ 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 \rightarrow dst | 0 | * | * | @Z |
| BIC | src,dst | .not. src .and. dst \rightarrow dst | - | - | - | - |
| BIS | src,dst | src .or. dst \rightarrow dst | - | - | - | - |
| BIT | src,dst | src .and. dst \rightarrow SR | 0 | * | * | @Z |
| CMP | src,dst | Vergleiche src und dst: (dst-src) \rightarrow 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 \rightarrow dst) | * | * | * | * |
| SUBC | src,dst | Subtrahiere src mit carry von dst (dst + .not. src + C \rightarrow dst) | * | * | * | * |
| XOR | src,dst | src .xor. dst \rightarrow 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

Beispiel: Eine Wort-Tabelle, auf die R13 zeigt, soll in einen RAM-Teil beginnend bei Label RAMT kopiert 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 zurück gesetzt, falls das Ergebnis positiv ist (MSB=0)
 - Z Wird gesetzt, falls das Ergebnis Null ist, wird zurück gesetzt, falls das Ergebnis nicht null ist
 - C Wird gesetzt, wenn bei der Operation ein Übertrag entsteht, wird zurück gesetzt, falls nicht.
 - V Wird gesetzt, wenn das Ergebnis zweier positiver Zahlen negativ ist oder falls das Ergebnis zweier negativer Zahlen positiv ist. Zeigt also an, ob wegen des Zweierkomplements ein Rechenfehler entstanden ist. Wird andernfalls zurück gesetzt.

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 source with destination |
| 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**
- 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 zurück gesetzt, falls das Ergebnis nicht null ist
 - C Wird gesetzt, wenn das Ergebnis nicht null ist, wird zurück gesetzt, falls das Ergebnis null ist. Das Carry-Bit entspricht damit dem invertierten Zero-Bit! $C = \text{not } Z$
 - V Wird immer zurück gesetzt (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 source and destination |
| Syntax | CMP src,dst oder CMP.W src, dst CMP.B src, dst |
| Operation | (dst-src) (intern: dst + .not. src +1) |
| Beschreibung | <p>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!</p> <p>Der Byte-Befehl CMP.B löscht das obere Byte eines als Destination verwendeten Registers nicht, da kein Schreibbefehl auf das Register erfolgt (nur lesen).</p> |

Der Befehl CMP (2)

- Statusbits**
- N Wird gesetzt, falls das Ergebnis negativ ist ($\text{src} > \text{dst}$), wird zurück gesetzt, falls das Ergebnis positiv oder null ist ($\text{src} \leq \text{dst}$)
 - Z Wird gesetzt, falls das Ergebnis null ist ($\text{src} = \text{dst}$). Wird zurück gesetzt, falls das Ergebnis nicht null ist ($\text{src} \neq \text{dst}$)
 - 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 positives 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:

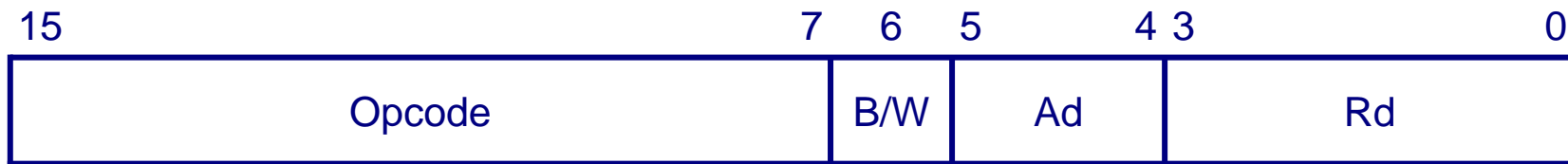
- Falls Taste KEY1 gedrückt ist, soll zur Marke MENU1 gesprungen werden.
- Falls Taste KEY2 gedrückt ist, soll zur Marke MENU2 gesprungen werden.
- Falls beide oder keine Tasten gedrückt sind, erfolgt keine Aktion

```
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
```

10.3 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 word on stack push byte on stack |
| 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
```

```
;lege den Inhalt von R8 auf  
den Stack ab
```

Beispiel 2:

```
PUSH.B  TONI
```

```
;oberes Byte vom TOS (top  
of stack) wird nicht ver-  
ändert!
```

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 geschrieben. Details siehe weiter unten! tmp → PC Destination wird in den PC geschrieben: Das Programm fährt an der Startadresse des Unterprogramms fort. |

Der Befehl CALL (2)

Beschreibung Es wird ein Unterprogramm sprung 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 ist zu beachten, dass im MSP 430 der Befehlszähler (PC) immer sofort nach dem Lesen eines Befehswortes (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 Autoincrement Mode benutzt, oder aber auf das 2. 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 um 2 erhöht, um das zweite Wort des CALL-Befehls zu überspringen.

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 Hardware-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 | Return 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 Anwendungsprogramm
```

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

10.4 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 Offset wird im Zweierkomplement angegeben.

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 $dst \geq src$ (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 $dst \geq src$ (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 $dst \neq src$ (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 Less than“ Jump if $N==1 \text{ xor } C==1$ (if $\text{dest} < \text{src}$)

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, z. B. nach dem Befehl TST.

JEQ wird gerne nach dem Vergleich zweier arithmetischer Operanden verwendet, z. B. 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
```

10.5 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 |