Assembler 68HC08

Einführung

Der 68HC08 hat den binärkompatibel erweiterten Befehlsatz des 68HC05. Das erklärt einige Unübersichtlichkeiten, z.B. bezüglich des X-Registers.

Diese Beschreibung dient erstens dazu die Unterschiede der hier verwendeten Assemblersyntax gegenüber dem Format im Motorola „CPU08 Reference Manual“ zu erläutern.

Zweitens sind hier die Opcodes nach Funktionsgruppen geordnet, was fürs Erlernen des Befehlssatzes hilfreich ist. Es fehlt die Darstellung vieler Details die sich im Motorola Manual finden.

Schreibweise

Gegenüber Motorola wurden alle Opcodes einheitlich auf 3 Buchstaben gekürzt um den Disassembler zu vereinfachen. Ferner wurde ein „`,`“ angehängt. Sonst wäre z.B. `ADC` mit einer Hexzahl verwechselbar. Der Motorola-Name wird kursiv angegeben wenn er unterschiedlich ist.

Die Operanden und Adressierungsarten stehen vor dem Opcode, wie bei Postfix-Assemblern üblich. Beim Compilieren geht der Operand direkt auf den Stack, die Adressierungsart speichert einen Token in der

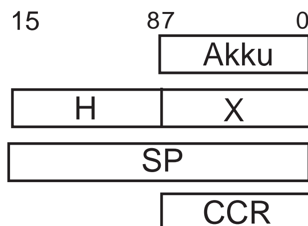


Bild1: Register

Tabelle 1: Adressierungsarten

	Motorola	FORTH
Inherent		
Immediate	#hh	hh #.
	#hhhh	hhhh #.
Direct (8 Bit Adr)	hh	hh
Extended (16 Bit Adr)	hhhh	hhhh
Indexed no Offset	,X	0,X
Indexed 8 Bit Offset	hh,X	hh ,X
Indexed 16 Bit Offset	hhhh,X	hhhh ,X
Stack Pointer 8 Bit Offset	hh,SP	hh ,SP
Stack Pointer 16 Bit Offset	hhhh,SP	hhhh ,SP
Relative	label	hh
Memory to Memory		
Immediate to Direct		
Direct to Direct		
Indexed to Direct with PostInc		
Direct to Indexed with PostInc		
Indexed with PostInc		
Indexed 8 Bit Offset with PostInc		

8 Bit Variable `A&`. Der Opcode schluckt beide Werte. Also:

```
45 ,X ADC,
```

`A&` muß für korrekte Funktion auf 00 initialisiert sein, was mit `0A&!` in `:CODE` und `[CODE` geschieht. Bei den Parametern und Operanden werden nicht alle denkbaren fehlerhaften Eingaben mit Fehlermeldungen quittiert, sondern nur die üblicheren Programmierfehler. Die Schreibweise der Adressierungen unterscheidet sich leicht von der Motorolas (Tabelle 1).

„Inherent“ bedeutet offensichtlich, daß kein Operand benötigt wird. Bei „Immediate“ wird eine Konstante geladen die dem Opcode folgt. „Extended“ bezieht sich auf eine 16 Bit Adresse im Speicher. „Direct“ ist eine Variante davon für die Adressen 00 ... FF , der ZeroPage des Prozessors. Der Assembler entscheidet selbstätig anhand der Adressen ob mit 8 Bit compiliert werden kann oder 16 Bit verwendet werden muß. Bei den „Indexed“ Adressierungen wird zur angegeben Adresse 0, hh oder hhhh der zur Ausführungszeit im Register HX vorhandene Inhalt addiert. Die Summe ergibt die effektive Adresse. Wenn man nur die unteren 8 Bit X als Offset verwenden will wie beim 68HC05, muß man sicherstellen, daß oberen 8 Bit H den Wert 00 enthalten.

Das Problem haben die Befehle nicht, die den Stackpointer SP addieren, denn der ist immer 16 Bit. Dafür haben die einen 16 Bit Opcode und sind langsam. Alle übrigen aufgeführten Adressierungen sind so unsystematisch, daß sie anhand der Opcodes die sie verwenden erläutert werden.

Die Eingabe eines Assemblerbefehls ist formatfrei. Bei mehreren numerischen Operanden ist jedoch deren Reihenfolge zu beachten. Es können beliebig viele Opcodes in einer Zeile stehen. Üblich ist ein Opcode pro Zeile aus Gründen der Übersichtlichkeit.

Einbindung in FORTH

Befehle die in Assembler realisiert wurden verhalten sich genauso, als ob sie aus FORTH-Sourcecode erzeugt worden wären.

```
:CODE TEST
78 #. LDA,  0200 STA,
RTS, CODE;
```

Die Rückkehr zu FORTH erfolgt durch den Opcode `RTS,`. Aufgerufen wird ein Befehl in FORTH durch `JSR,` . Praktisch sind FORTH-Befehle also Assemblerunterprogramme.

Als Arbeitsspeicher steht in der ZeroPage der N-Bereich mit 8 Bytes zur Verfügung.

```
N      LDA,
N 7 + STA,
```

Er wird auch von FORTH-Befehlen benutzt und kann deshalb nicht zur Parameterübergabe in FORTH-Programmen oder in Interrupts benutzt werden.

Der FORTH-Stack wird mit dem X-Register adressiert und wächst abwärts. Das obere H-Byte enthält in FORTH immer den Wert 00. Beim Einsprung in ein Assemblerprogramm zeigt das X-Register auf das obere Byte des obersten Stackwertes. Direkt darüber liegt das untere Byte („Big Endian“). Beispiel:

```
( CDAB 3412 ---     )
```

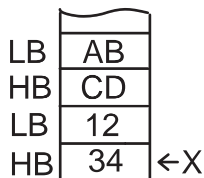


Bild 2: FORTH-Stack

Man muß das X-Register also retten, wenn man es innerhalb eines Assemblerprogramms benutzen will. Mit dem Namen `XSAVE` ist dafür eine 16 Bit Variable reserviert:

```
:CODE TEST
XSAVE SHX,
...
XSAVE LHX,
RTS, CODE;
```

Man kann auch nur das untere Byte retten, wenn man das H-Register nicht ändert:

```
:CODE TEST
XSAVE 1+ STX,
...
XSAVE 1+ LDX,
RTS, CODE;
```

Der Akku und die Bits im Condition Code Regsiter CCR dürfen verändert werden.

Der Zugriff auf Variablen (wie `N`) und Konstanten im Assembler unterscheidet sich nicht vom Zugriff in FORTH.

`TEST1` würde den Wert 1256 auf den FORTH-Stack legen, `TEST2` einen Wert vom Stack nehmen und ihn in 0230 abspeichern.

```
:CODE TEST1 \ ( — 1256 )
DEX, DEX,
56 #. LDA,    \ LB
 1 ,X STA,
12 #. LDA,    \ HB
 0,X  STA,
RTS, CODE;

:CODE TEST2 \ ( UN1 —   )
0,X  LDA,
0230 STA,  \ HB
1  ,X LDA,
0231 STA,  \ LB
INX, INX, RTS, CODE;
```

Innerhalb von `:CODE ... CODE;` verbleibt der Compiler im Interpretermodus. Will man Assembler direkt in FORTH-Befehlen einbinden muß man mit `[CODE` und `CODE]` aus dem Compiliermodus zurückschalten.

```
: TEST
...
[CODE
78 #. LDA, 0200 STA,
CODE]
... ;
```

Die CPU-Flags

Die Flags sind im „Condition Code Register“ CCR zusammengefaßt. Der Programmierer hat jedoch selten mit diesem direkt zu tun. Überwiegend werden die Flags mit den bedingten Sprungbefehlen getestet. Das CCR ist 8 Bit breit und enthält folgende Bits:

V Overflow-Flag
H Half-Carry
I Interrupt-Flag
N Negative-Flag
Z Zero-Flag
C Carry-Flag

Bild 3: CCR

Ein- und Auslesen des kompletten Bytes geschieht durch diese Befehle:

```
TAP, \ Transfer Akku to CCR
TPA, \ Transfer CCR to Akku
```

`TAP,` ist nützlich um so auf einen Schlag alle Flags nach Reset in eine definierte Ausgangsposition bringen.

Zero-Flag

Wird von den meisten Befehlen verändert. Wenn das Ergebnis einer Operation 00 war, wird es gesetzt. Man kann es sich deshalb als 8-fach NOR-Funktion vorstellen. Sehr häufig benutzt. Direkt testbar mit `BEQ,` und `BNE,`.

Negative-Flag

„Negative“ bedeuted nur, daß das Ergebnis einer Operation im Bit 7 eine 1 hatte. Deshalb beinhaltet es eigentlich nur Bit 7 des letzten verarbeiteten Bytes. Direkt testbar mit `BMI,` und `BPL,`.

Carry-Flag

Wird von den arithmetischen Befehlen verändert, wie `ADC,` oder `SBC,` . Dort hat es die Funktion eines 9. Bits für den Akku in dem ein möglicher Überlauf gespeichert werden kann. Auch benutzt von den Schiebebefehlen `ROL, ROR, LSR, ASL,` . Oder man manipuliert es direkt:

```
SEC,  \  Carry setzen
CLC,  \  Carry löschen
```

Mit `BCS,` und `BCC,` direkt testbar. Eine sinnvolle Anwendung ist deshalb der Transfer einer Ja/Nein-Entscheidung aus oder in ein Unterprogramm.

Overflow-Flag

Für Arithmetik mit 2er-Komplement Zahlen. Die Shiftbefehle ändern das Flag auch, aber bei ihnen kein sinnvolles Ergebnis.

IRQ-Flag

Globale Freigabe und Sperren von Interrupts.

```
SEI, \ IRQ sperren Flag = 1
CLI, \ IRQ freigeben    = 0
```

Half-Carry Flag

Für BCD-Arithmetik gedacht. In nanoFORTH nicht verwendet.

X-Register

Das Indexregister X des 68HC05 war nur 8 Bit breit und wurde für den 68HC08 auf das 16 Bit Register HX erweitert. Gleichzeitig sollte aber Kompatibilität erhalten bleiben. Als X wird deshalb weiter die untere Hälfte bezeichnet, mit H die obere Hälfte. Der ursprüngliche Befehlssatz des 68HC05 wurde um Befehle ergänzt die statt X den Teil H verwenden. Dabei muß man beachten, daß es sich manchmal nur um die oberen 8 Bit handelt, in anderen Fällen aber um das komplette HX Register bei dem beide Teile zusammengefaßt sind.

Tabelle 2: Stackpointer schreiben & lesen

```
TSX, \ Transfer Stackpointer +1 to HX Register 16 Bit
TXS, \ Transfer HX-Register -1 to Stackpointer 16 Bit
```

Tabelle 3: Daten auf Stack zwischenspeichern

```
PHA, \ PSHA Push Akku to Stack
PHX, \ PSHX Push X Register to Stack
PHH, \ PSHH Push H Register to Stack

PLA, \ PULA Pull Akku from Stack
PLX, \ PULX Pull X Register from Stack
PLH, \ PULH Pull H Register from Stack
```

CPU-Stack

Der 68HC05 hatte einen 8 Bit Stackpointer der durch den Befehl

```
RSP, \ Reset Stack Pointer
```

auf FF initialisiert wurde. Aus Kompatibilität ist dieser Befehl immer noch vorhanden. Da der 68HC08 aber einen 16 Bit Stackpointer hat hier nutzlos. Laden und Auslesen erfolgt über das 16 Bit Register HX (Tabelle 2). Man beachte die eigentümlichen Inkrements und Dekrements. Initialisierung des Stackpointers auf 01FFh deshalb so:

```
H% 01FF 1+ #. LHX, TXS,
```

Der Stackpointer zeigt auf das nächste freie Byte und wächst im Speicher abwärts. Während jedes Unterprogrammaufrufs sind zwei weitere Bytes belegt, ein Interrupt schlägt mit 5 Bytes zu Buche.

Zwischenspeichern von Daten erfolgt mit den Befehlen in Tabelle 3.

Bei abgeschaltetem IRQ kann man den Stackpointer als zweites Index-Register verwenden. Selbstmodifizierender Code in der ZeroPage dürfte aber oft effizienter sein.

Labels für Sprungbefehle

Der Assembler unterscheidet lokale und globale Labels. Die Labels selbst sind Zahlen von 00 - 1F.

Lokale Labels können vorwärts oder rückwärts zeigen. Sie werden in `CODE;` bzw. `CODE]` aufgelöst und sind auf einen `:CODE`-Befehl bzw. den `[CODE`-Abschnitt beschränkt. Die Quelle des Sprungs wird mit `$` markiert, das Ziel mit `$:` . Vor diesen beiden Worten muß jeweils die Nummer stehen. Mehrere Quellen können auf ein Ziel zeigen.

Globale Labels können nur rückwärts zeigen. Quelle wird mit `G$` und Ziel mit `G$:` markiert. Der Sprung wird von `G$` aufgelöst. Ein Ziel darf nur auf eine Quelle verweisen.

Zwischen den von globalen und lokalen Labels verwendete Zahlen besteht kein Zusammenhang. Jede Zahl kann also doppelt verwendet werden.

```
:CODE TEST1
4 $:  NOP,
...
4 G$: NOP,
...
4 $   JMP,
3 $   BEQ,
...
4 $   BNE,
...
3 $:  RTS,
CODE;
```

```
:CODE TEST2
...
4 G$ JMP,
CODE;
```

Soll ein Assemblerprogramm `TEST` das mit `:CODE ... CODE;` erzeugt wurde von einem anderen Assemblerprogramm mit z.B. `JMP,` angesprungen werden, so erhält man die Adresse des ersten Opcodes so:

```
´ TEST JMP,
```

`:CODE ... CODE;` ermöglicht den Aufruf von FORTH aus, erzeugt aber Speicheroverhead. Oft braucht man diesen Kopf nicht, dann sollten `G$` und `G$:` verwendet werden. Die Zwischenspeicherung der Labels erfolgt im RAM-Bereich `$HEAP`. Er faßt um Grundzustand 11d Einträge. Wenn beim Compilieren ein Überlauffehler eintritt, sollte man das Programm in geschlossene Abschnitte unterteilen, in denen alle lokalen Sprünge mit `SOLVE$` lösbar sind. Dieser Befehl ist in `CODE;` bzw. `CODE]` enthalten und löst dort die Zuweisungen auf.

```
:CODE TEST
1 $:  NOP,
...
1 $   BNE,
SOLVE$
1 $   BEQ,
...
1 $:  RTS,
CODE;
```

Sprungbefehle

```
hh        JMP, \ Jump
hhhh      JMP,
0,X       JMP,
hh ,X     JMP,
hhhh ,X JMP,
```

Der am häufigst verwendeten Sprungbefehl ist `hhhh JMP,` der mit seinem 16 Bit Operanden jedes Ziel im Speicher der CPU erreicht.

```
0200 JMP, \Sprung nach 0200
```

Praktisch gibt man jedoch nie direkt 0200 als absolute Hexadresse im Programm an, sondern verwendet ein Label oder eine andere symbolische Bezeichnung.

Adressierung die auf die ZeroPage 00 ... FF zeigen, wo sich RAM aber kein FLASH, also Programmspeicher befinden, haben praktische Bedeutung. Man kann dort kurze Unterprogramme mit selbstmodifizierendem Code anlegen um komplexe Adressierungsarten zu simulieren, die im Befehlssatz nicht vorhanden sind.

Die ,X Varianten sind nützlich für Sprungzieltabellen:

```
TABLE QUAD-TAB
TEST1 JMP, NOP,   \ 0
TEST2 JMP, NOP,   \ 4
TEST3 JMP, NOP,   \ 8
TEST4 JMP,        \ C

:CODE QUAD \ in: Akku =
           \ 0,1,2,3
A. LSL, A. LSL, TAX,
QUAD-TAB ,X JMP,
CODE;
```

Der `NOP,` in der Tabelle ist nötig, damit man den Offset einfach durch Shifts erzeugen kann. Wenn die Unterprogramme auf die man verzweigt so kurz sind, daß sie alle zusammen in etwa 100d Byte passen, kann man die Tabelle auch mit 2 Byte Schritten und dem relativen Sprung `BRA,` implementieren.

```
radr BRA, \ Branch always
```

Sprungbefehle mit relativer Adressierung benutzen als Bezugspunkt den Opcode. Ein Programm das nur relative Sprünge verwenden würde wäre also im Speicher verschiebbar. Die CPU verfügt aber nur über relative Sprungbefehle mit Byte-Offset. Man kann so nur 127 Bytes vorwärts oder 128 Bytes rückwärts springen. Die Berechnung des Offsets erfolgt durch den Compiler. Deshalb besteht für den Programmierer kein offensichtlicher Unterschied zur absoluten Adressierung. Wird die Sprungweite jedoch überschritten erfolgt Fehlermeldung. Soweit das Ziel erreichbar ist spart dieser Befehl gegenüber `JMP,` ein Byte Speicher.

Unterprogramme

Auch zum Aufruf für Unterprogramm `JSR,` gibt es eine Variante `BSR,` mit relativer Adressierung.

```
hh        JSR, \ Jump
               \ Subroutine
hhhh      JSR,
0,X       JSR,
hh ,X     JSR,
hhhh ,X JSR,
radr      BSR, \ Branch
               \ Subroutine
```

Rücksprung:

```
          RTS,  \ Return from
                \ Subroutine
```

Tabelle 4: bedingte Sprungbefehle für CCR

```
radr  BCC, \ C=0 BCC Branch if Carry Clear
           \     BHS Branch on Higher or Same
radr  BCS, \ C=1 BCS Branch on Carry set
           \     BLO Branch if Lower
radr  BNE, \ Z=0 BNE branch if not Equal
radr  BEQ, \ Z=1 BEQ Branch if Equal
radr  BPL, \ N=0 BPL Branch if Plus
radr  BMI, \ N=1 BMI Branch if Minus
radr  BMC, \ I=0 BMC Branch if Interrupt Mask Clear
radr  BMS, \ I=1 BMS Branch if Interrupt Mask Set
radr  BHC, \ H=0 BHCC Branch if Half Carry Clear
radr  BHS, \ H=1 BHCS Branch if Half Carry Set
```

Tabelle 5: bedingte Sprungbefehle für Bits in ZeroPage

```
radr hh h BBC, \ BRCLR Branch if Bit in ZeroPage Clear
radr hh h BBS, \ BRSET Branch if Bit in ZeroPage Set
```

Tabelle 6: bedingte Sprungbefehle für IRQ-Pin

```
radr  BIL, \ Branch if IRQ Pin Low
radr  BIH, \ Branch if IRQ Pin High
```

3.4

Weder `JSR,` noch `RTS,` beeinflussen die Flags oder den Akku.

Während des Unterprogramms liegen zwei Bytes auf dem Stack. Sie enthalten die Rücksprungadresse Obenauf das HB, darunter das LB.

Anhand dieser Stackwerte kann das Unterprogramm seine Position im Speicher bestimmen. Manchmal ist es auch nützlich die Rücksprungadresse im Unterprogramm zu verändern.

Bedingte Sprungbefehle

Sie sind nur mit 8 Bit relativer Adressierung verfügbar und springen abhängig vom Zustand von Bits. Tabelle 4 zeigt die Varianten die die Bits im CCR auswerten.

Daneben kann man auch Bits im Speicher, aber nur in der ZeroPage testen (Tabelle 5). Der Operand h = 0 - 7 gibt das Bit im Byte an. Damit sind alle I/O-Register zugänglich.

Der Pegel am IRQ-Pin des Controllers kann mit den Befehlen in Tabelle 6 abgefragt werden.

Für den Test auf Gleichheit von Speicher und Register gibt es einen Befehl der Vergleich und bedingten Sprung vereinigt:

```
\ Compare & Branch if Equal
radr hh        CBQ, \ CBEQ
radr hh #. A. CBQ,
radr hh #. X. CBQ,
radr   0,X     CBQ,
radr hh ,X     CBQ,
radr hh ,SP    CBQ,
```

Vergleichsbefehle

Eigentlich eine Subtraktion Register minus Memory: r - m . Wobei das Register Akku oder X ist. Anders als beim Subtraktionsbefehl wird das Register nicht verändert, sondern nur die Flags im CCR-Register werden angepaßt.

Für vorzeichenlose Zahlen sind Flags Z und C verwertbar:

r = m : Z = 0
r < m : C = 1

Für Zahlen mit Vorzeichen spielt zusätzlich das Overflow-Flag eine Rolle. Da man aber die Befehle aus Tabelle 7 verwenden kann, muß man die Details nicht wissen.

```
hh #. CMP, \ Compare Akku
hh      CMP,
hhhh    CMP,
0,X     CMP,
hh ,X CMP,
hhhh ,X CMP,
hh    ,SP CMP,
hhhh ,SP CMP,
```

Compare X Register 8 Bit mit Speicher 8 Bit:

```
hh #. CPX,
hh      CPX,
hhhh    CPX,
0,X     CPX,
hh ,X CPX,
hhhh ,X CPX,
hh    ,SP CPX,
hhhh ,SP CPX,
```

Compare HX-Register 16 Bit mit Speicher 16 Bit

```
hhhh #. CHX, \ CPHX
  hh     CHX,
```

Der Befehl TEST setzt nur die Flags N und Z im CCR entsprechend Inhalt von Register oder Speicher. Register und Speicher werden nicht verändert.

```
A.      TST, \ Test
X.      TST,
hh      TST,
0,X     TST,
hh ,X TST,
hh ,SP TST,
```

Mit dem BIT-Befehl werden Akku und Speicher durch AND verknüpft. Dabei wird jedoch der Akku nicht verändert, nur die CCR-Flags werden angepaßt.

```
hh #. BIT, \ Bit Test
hh      BIT,
hhhh    BIT,
0,X     BIT,
hh ,X BIT,
hhhh ,X BIT,
hh    ,SP BIT,
hhhh ,SP BIT,
```

Tabelle 7: bedingte Sprungbefehle nach `CMP,`

Vorzeichenlose Zahlen:

```
radr BHI, \ r>m    Branch if Higher
radr BHS, \ r>=m   Branch if Higher Same
radr BEQ, \ r=m    Branch if Equal
radr BNE, \ r><m   Branch if Not Equal
radr BLS, \ r=<m   Branch if Less Same
radr BLO, \ r<m    Branch if Lower
```

2er-Komplement Zahlen:

```
radr BGT, \ r>m    Branch if Greater
radr BGE, \ r>=m   Branch if Greater Same
radr BEQ, \ r=m    Branch if Equal
radr BNE, \ r><m   Branch if Not Equal
radr BLE, \ r=<m   Branch if Less Equal
radr BLT, \ r<m    Branch if Less Than
```

Daten bewegen

Da der 68HC05 eine Akkumulatormaschine war sind `LDA,` und `STA,` die meistgebrauchten Befehle.

Store Akku in Speicher:

```
hh      STA,
hhhh    STA,
0,X     STA,
hh ,X STA,
hhhh ,X STA,
hh    ,SP STA,
hhhh ,SP STA,
```

Load Akku aus Speicher:

```
hh #. LDA,
hh      LDA,
hhhh    LDA,
0,X     LDA,
hh ,X LDA,
hhhh ,X LDA,
hh    ,SP LDA,
hhhh ,SP LDA,
```

Um den Wert 00 zu speichern gibt es einen eigene Befehle. Akku wird nicht verändert:

```
A.    CLR,  \ Clear
X.    CLR,
H.    CLR,
hh    CLR,
0,X   CLR,
hh ,X CLR,
hh ,SP CLR,
```

Man beachte, daß es keinen `hhhh CLR,` gibt.

Zwischen X und Akku Daten bewegen:

```
TAX, \ Transfer Akku to X
TXA, \ Transfer X to Akku
```

Im 68HC08 hat man noch einen MOVE-Befehl a la Registermaschinen angepappt (Tabelle 8). Spart Speicher. Bewegt wird jeweils ein Byte. Die beiden ersten Varianten ersetzen:

```
hh2 #. LDA, hh1 STA, bzw.
   hh2 LDA, hh1 STA,
```

Die beiden letzten verwenden das volle 16 Bit HX Register als Quelle bzw. Ziel. Dieses wird nach der Datenbewegung zusätzlich noch inkrementiert. Funktion also ca.:

```
0,HX LDA, hh1 STA, HX. INC,
```
bzw.
```
hh2 LDA, 0,HX STA, HX. INC,
```

Um den Assembler zu vereinfachen wurde aber nicht versucht das in der Adressierung darzustellen.

Äquivalent zu den Befehlen für den Akku gibt es entsprechende Befehle für das X-Register.

Store X in Speicher:

```
hh    STX,
hhhh  STX,
0,X   STX,
hh ,X STX,
hhhh ,X STX,
hh   ,SP STX,
hhhh ,SP STX,
```

Tabelle 8: MOVE

```
hh1 hh2 #. MOV,    \ ZPadr1 <- #.data2
hh1 hh2    MOV,    \ ZPadr1 <- ZPadr2
hh1 0,X    MOV,    \ ZPadr1 <- 0,HX   ; IncHX
,X  hh2    MOV,    \ 0,HX   <- ZPadr2 ; incHX
```

Load X aus Speicher:

```
hh #. LDX,
hh    LDX,
hhhh  LDX,
0,X   LDX,
hh ,X LDX,
hhhh ,X LDX,
hh   ,SP LDX,
hhhh ,SP LDX,
```

Store HX-Register 16 Bit in Speicher:

```
hh  SHX, \  STHX
```

Load HX-Register 16 Bit aus Speicher:

```
hhhh #. LHX, \ LDHX
hh      LHX,
```

Arithmetik

Addition

`ADD,` addiert Akku und Operand. Alle Flags werden verändert. Carry-Flag = 1, wenn das Ergebnis größer FF ist. Also overflow für vorzeichenlose Zahl. Das V-Flag = 1, wenn die Summe von zwei 2er-Komplementzahlen mit gleichem Vorzeichen außerhalb +7F oder -7F liegt.

```
hh #. ADD,
hh    ADD,
hhhh  ADD,
0,X   ADD,
hh ,X ADD,
hhhh ,X ADD,
hh   ,SP ADD,
hhhh ,SP ADD,

hh #. ADC,
hh    ADC,
hhhh  ADC,
0,X   ADC,
hh ,X ADC,
hhhh ,X ADC,
hh   ,SP ADC,
hhhh ,SP ADC,
```

`ADC,` addiert Carryflag, Akku und Operand. Das ist besonders für Addition von 16 und 32 Bit Datenworten nützlich. Z.B. 2-Byte-Addition:

```
LB1 LDA, \ unteres Byte
LB2 ADD,
LB3 STA,
HB1 LDA, \ oberes Byte
HB2 ADC,
HB3 STA,
```

Für die Addition von +1 zu einem Register oder zu Speicher gibt es den Increment-Befehl:

```
A.    INC,
X.    INC,
hh    INC,
0,X   INC,
hh ,X INC,
hh ,SP INC,
```

16 Bit Inkrement:

```
    LB  INC,
1 $     BNE,
     HB INC,
1 $:
```

Subtraktion

`SUB,` subtrahiert vom Akku den Operanden. Reihenfolge beachten: Akku und Operand dürfen nicht vertauscht sein. Für vorzeichenlose Zahlen ist CarryFlag = 1, wenn das Ergebnis <0 ist, d.h wenn Operand grösser als Akku war. V-Flag = 1, wenn Operanden nicht das gleiche Vorzeichen hatten, und das Ergebnis außerhalb +7F und -7F liegt.

```
hh #. SUB,
hh    SUB,
hhhh  SUB,
0,X   SUB,
hh ,X SUB,
hhhh ,X SUB,
hh   ,SP SUB,
hhhh ,SP SUB,
```

```
hh #. SBC,
hh    SBC,
hhhh  SBC,
0,X   SBC,
hh ,X SBC,
hhhh ,X SBC,
hh   ,SP SBC,
hhhh ,SP SBC,
```

`SBC,` subtrahiert vom Akku den Operanden und das invertierte Carryflag. Benötigt für lange Datenworte.
2 Byte Subtraktion:

```
LB1 LDA, \ unteres Byte
LB2 SUB,
LB3 STA,
HB1 LDA, \ oberes Byte
HB2 SBC,
HB3 STA,
```

Der Befehl NEGATE berechnet für ein Byte das 2er-Komplement, d.h. er invertiert das Vorzeichen einer 2er-Komplementzahl. Die Berechnung ist 00 - opr , also eine Variante des Subtraktionsbefehls. Der Wert 80h = -128d bleibt unverändert.

```
A.    NEG, \ Negate
X.    NEG,
hh    NEG,
0,X   NEG,
hh ,X NEG,
hh   ,SP NEG,
```

Anwendung kann z.B. Absolutwertbildung eines Bytes sein.

```
1 $ LB 7 BBC, \ test sign
    LB   NEG,
1 $:
```

Bei Anwendungen in der Signalverarbeitung kann es aber sinnvoll sein den Wert -128d auf +127d zu ändern.

Für die Subtraktion -1 gibt es den Dekrement-Befehl:

```
A.    DEC,
X.    DEC,
hh    DEC,
0,X   DEC,
hh ,X DEC,
hh ,SP DEC,
```

16 Bit Dekrement mit Überlauf:

```
     LB  LDA,
1 $      BNE,
     HB  DEC,
1 $: LB  DEC,
```

Speziell zur Programmierung von Schleifen:
„Decrement and Branch if Not Zero“

```
radr    hh DBN, \ DBNZ,
radr    A. DBN,
radr    X. DBN,
radr  0,X  DBN,
radr hh ,X DBN,
radr hh ,SP DBN,
```

Es gibt auch zwei Varianten der Addition die speziell für Stackframes gedacht sind:
Add immediate signed byte to Stack Pointer

```
hh #. AIS,
```

Add immediate signed byte to HX-Register

```
hh #. AIX,
```

Man beachte, daß das Byte hier explizit als -128 ... +127 interpretiert wird.

Multiplikation & Division

```
MUL,
```

Der vorzeichenlose Inhalt von Akku und X-Register werden zu einen 16 Bit Wert zusammenmultipliziert. Danach liegt das obere Byte im X-Register, das untere im Akku.

```
DIV,
```

Die vorzeichenlose 16 Bit Zahl in den Registern H (oberes Byte) und Akku (unteres Byte) werden durch das X-Register geteilt. Quotient landet im Akku, der Rest im H-Register. Das X-Register wird nicht verändert. Wenn eine Division durch Null versucht wurde oder ein Überlauf erfolgte, weil das Resultat nicht in 8 Bit paßt, wird das Carryflag gesetzt.

Boolsche Befehle

Es wird immer ein Byte bearbeitet. Die boolsche Funktion erfolgt bitweise. Einer der Operanden muß sich im Akku befinden. Das Ergebnis ist danach im Akku.

Logisches AND

```
hh #. AND,
hh    AND,
hhhh  AND,
0,X   AND,
hh ,X AND,
hhhh ,X AND,
hh   ,SP AND,
hhhh ,SP AND,
```

Inklusives OR

```
hh #. ORA,
hh    ORA,
hhhh  ORA,
0,X   ORA,
hh ,X ORA,
hhhh ,X ORA,
hh   ,SP ORA,
hhhh ,SP ORA,
```

Exclusives OR, XOR

```
hh #. EOR,
hh    EOR,
hhhh  EOR,
0,X   EOR,
hh ,X EOR,
hhhh ,X EOR,
hh   ,SP EOR,
hhhh ,SP EOR,
```

Der Complement-Befehl berechnet das 1er-Komplement, also die bitweise Invertierung. Effizienter als `FF #. EOR,`

```
A.    COM, \ Complement
X.    COM,
hh    COM,
0,X   COM,
hh ,X COM,
hh ,SP COM,
```

Boolsche Befehle kann man verwenden um Bits zu setzen:

```
hhhh  LDA,
0F #. ORA, \ Bit 0 .. 3 = 1
hhhh  STA,
```

Bits zu löschen:

```
hhhh  LDA,
F0 #. AND, \ Bit 0 .. 3 = 0
hhhh  STA,
```

Bits abzufragen:

```
hhhh  LDA,
03 #. AND,
hhhh  BEQ, \ Branch wenn
           \ Bit 0 & 1 = 0
```

Für die Verarbeitung von einzelnen Bits sind aber die Bit-Befehle günstiger. Die sind aber nur für Adressen in der ZeroPage verwendbar

```
hh h MBS, \ BSET Bit setzen
          \      h = 0 -7
hh h MBC, \ BCLR löschen
```

Es gibt sie auch mit integriertem relativem Sprung (Tabelle 5).

Schiebebefehle

Der Inhalt des Akkus bleibt unverändert, wenn der Operand sich im Speicher oder X-Register befindet.
Arithmetic Shift Left
Alias: LSL,

```
A.     ASL,
X.     ASL,
hh     ASL,
0,X    ASL,
hh ,X  ASL,
hh ,SP ASL,
```

Arithmetic Shift Right

```
A.     ASR,
X.     ASR,
hh     ASR,
0,X    ASR,
hh ,X  ASR,
hh ,SP ASR,
```

Logical Shift Right

```
A.     LSR,
X.     LSR,
hh     LSR,
0,X    LSR,
hh ,X  LSR,
hh ,SP LSR,
```

Rotate Left through Carry

```
A.     ROL,
X.     ROL,
hh     ROL,
0,X    ROL,
hh ,X  ROL,
hh ,SP ROL,
```

Rotate Right through Carry

```
A.     ROR,
X.     ROR,
hh     ROR,
0,X    ROR,
hh ,X  ROR,
hh ,SP ROR,
```

Was mit `ROR,` und `ROL,` reingeschoben wird, läßt sich mit `SEC,` und `CLC,` steuern. Abfragen des C-Flags erfolgt mit `BCC,` und `BCS,` . Mit `ASL,` und `LSR,` lassen sich Multiplikation `*2` und vorzeichenlose Division `/2` für 8 Bit Zahlen durchführen. Erweitertung z.B. auf 16 Bit:

```
LB LSL, \ *2
HB ROL,

HB LSR, \ /2 unsigned
LB ROR,
```

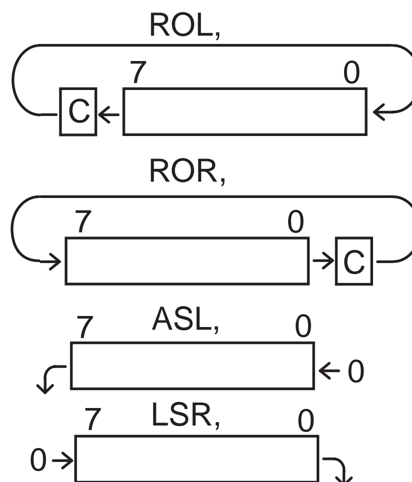


Bild 4: Schiebebefehle

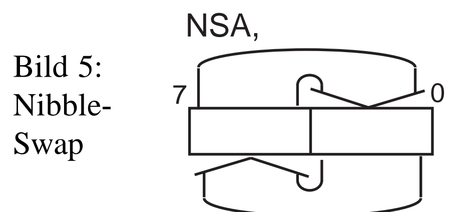
`/2` für 2erKomplementzahl 16 Bit z.B. so:

```
           SEC, \ C = 1
1 $   HB 7 BBS, \ Bit 7 ?
           CLC, \ C = 0
1 $: HB    ROR, \ ersetzt:
                \ HB LSR,
     LB    ROR,
```

Mit etwas Phantasie kann man den Befehl Nibble-Swap als 4fach Rotationsbefehl ohne Carry ansehen. Er tauscht im Akku B0-B3 mit B4-B7. Man beachte: Flags werden nicht verändert.

```
NSA, \ Nibble Swap Accu
```

Bild 5: Nibble-Swap



Interrupts

Die IRQs kann man global per Befehl sperren oder freigeben:

```
CLI, \ Clear Interrupt
     \ Mask Bit : enable
SEI, \ Set Interrupt
     \ Mask Bit : disable
```

Einzelne Interruptquellen müssen jedoch in der I/O separat freigeschaltet werden.

Tritt ein IRQ auf, wird ersteinmal der gerade bearbeitete Befehl zuende ausgeführt. Dann werden Programmzeiger, X-Register, Akkumulator, CCR auf den Stack gelegt:

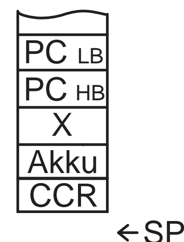


Bild 6: Stack während IRQ

Dann wird der jeweilige IRQ-Vektor geholt und als neuer Programmzeiger geladen. Er enthält also die erste Adresse des IRQ-Unterprogramms.

Aus Kompatibilität zum 68HC05 wird das H-Register nicht automatisch auf den Stack gelegt. Entweder man tut im Interrupt-programm nichts was es verändert oder man sichert es auf dem Stack:

```
:CODE MY-INTERRUPT
PHH,
...
PLH,
RTI, CODE;
```

Der `RTI,`-Befehl beendet den IRQ und holt die 5 automatisch gespeicherten Werte wieder vom Stack.

```
RTI,  \ Return from
      \ Interrupt
```

Es gibt kompatibel zum 68HC05 einen Opcode der einen Interrupt auslöst und einen eigenen Interupt-vektor hat:

```
SWI, \ Software Interrupt
```

Das war früher für Testsoftware wie Breaks und Singlestepper nützlich, wenn sich das Programm im RAM befindet. Im 68HC908 ist es typisch im FLASH und solche Funktionen werden durch I/O unterstützt.

Sonderbefehle

Den `NOP,` hat jede CPU und der 68HC08 hat sogar zwei:

```
NOP,        \ No Operation
            \ 1 cyc 1 byte
radr  BRN,  \ Branch Never
            \ 3 cyc 2 byte
```

Nur für BCD-Arithmetik, deshalb nicht beschrieben:

```
DAA,    \ Decimal Adjust
        \ Akkumlator
```

Nur in Verbindung mit der I/O zu erklären:

```
WIT,   \ WAIT
       \ Enable Interrupts,
       \ Stop Prozessor
STP,   \ STOP
       \ Enable IRQ-Pin
       \ Stop Oscillator
```

Liste der Opcodes

Angegeben sind Name und Seitenzahl wo näher beschrieben. Sowie Speicherverbrauch in Bytes, Rechenzeit in Taktzyklen und veränderte Flags.

Byt.Zyk. Fl.

Add with Carry 3.6

```
hh #. ADC,       2 2 VHNZC
hh    ADC,       2 3
hhhh  ADC,       3 4
0,X   ADC,       1 2
hh ,X ADC,       2 3
hhhh ,X ADC,     3 4
hh   ,SP ADC,    3 4
hhhh ,SP ADC,    4 5
```

Add without Carry 3.6

```
hh #. ADD,       2 2 VHNZC
hh    ADD,       2 3
hhhh  ADD,       3 4
0,X   ADD,       1 2
hh ,X ADD,       2 3
hhhh ,X ADD,     3 4
hh   ,SP ADD,    3 4
hhhh ,SP ADD,    4 5
```

Add immediate value signed to Stack Pointer 3.7

```
hh #. AIS,       1 2
```

Add immediate Signed to Index Register 3.7

```
hh #. AIX,       2 2
```

Logical AND 3.7

```
hh #. AND,       2 2 VNZ
hh    AND,       2 3
hhhh  AND,       3 4
0,X   AND,       1 2
hh ,X AND,       2 3
hhhh ,X AND,     3 4
hh   ,SP AND,    3 4
hhhh ,SP AND,    4 5
```

Byt.Zyk. Fl.

Arithmetic Shift Left 3.8

```
A.     ASL,       1 1 VNZC
X.     ASL,       1 1
hh     ASL,       2 4
0,X    ASL,       1 3
hh ,X  ASL,       2 4
hh ,SP ASL,       3 5
```

Arithmetic Shift Right 3.8

```
A.     ASR,       1 1 VNZC
X.     ASR,       1 1
hh     ASR,       2 4
0,X    ASR,       1 3
hh ,X  ASR,       2 4
hh ,SP ASR,       3 5
```

Relative Branch 3.4

```
radr   Bxx,       2 3
```

Branch if Bit in ZeroPage Clear/Set 3.4

```
radr hh h BBC,    3 5  C
radr hh h BBS,    3 5
```

Bit Test 3.5

```
hh #.  BIT,       2 2  VNZ
hh     BIT,       2 3
hhhh   BIT,       3 4
0,X    BIT,       1 2
hh ,X  BIT,       2 3
hhhh ,X BIT,      3 4
hh   ,SP BIT,     3 4
hhhh ,SP BIT,     4 5
```

Branch to Subroutine 3.4

```
radr BSR,         2 4
```

Compare and Branch if Equal 3.5

```
radr hh        CBQ,  3 5
radr hh #. A.  CBQ,  3 4
radr hh #. X.  CBQ,  3 4
radr  0,X      CBQ,  2 4
radr hh ,X     CBQ,  3 5
radr hh ,SP    CBQ,  4 6
```

Byt.Zyk. Fl.

Clear Carry Bit 3.3

```
CLC,               1 1 C
```

Clear Interrupt Mask Bit 3.8

```
CLI,               1 2 I
```

Clear 3.6

```
A.     CLR,        1 1
X.     CLR,        1 1
H.     CLR,        1 1
hh     CLR,        2 3
0,X    CLR,        1 2
hh ,X  CLR,        2 3
hh ,SP CLR,        3 4
```

Compare 3.5

```
hh #. CMP,         2 2 VNZC
hh    CMP,         2 3
hhhh  CMP,         3 4
0,X   CMP,         1 2
hh ,X CMP,         2 3
hhhh ,X CMP,       3 4
hh   ,SP CMP,      3 4
hhhh ,SP CMP,      4 5
```

Complement 3.7

```
A.     COM,        1 1 VNZC
X.     COM,        1 1
hh     COM,        2 4
0,X    COM,        1 3
hh ,X  COM,        2 4
hh ,SP COM,        3 5
```

Compare HX with Memory 3.5

```
hhhh #. CHX,       3 3 VNZC
  hh    CHX,       2 4
```

Compare X with Memory 3.5

```
hh #. CPX,       2 2 VNZC
hh    CPX,       2 3
hhhh  CPX,       3 4
0,X   CPX,       1 2
hh ,X CPX,       2 3
hhhh ,X CPX,     3 4
hh   ,SP CPX,    3 4
hhhh ,SP CPX,    4 5
```

Byt.Zyk. Fl.

Decimal Adjust Akku 3.9

```
DAA,                1 2 VNZC
```

Dec and Branch if Not Zero 3.7

```
radr    hh DBN,   3 5
radr    A. DBN,   2 3
radr    X. DBN,   2 3
radr  0,X  DBN,   2 4
radr hh ,X DBN,   3 5
radr hh ,SP DBN,  4 6
```

Dekrement 3.7

```
A.     DEC,         3 2 VNZ
X.     DEC,         2 1
hh     DEC,         2 3
0,X    DEC,         2 4
hh ,X  DEC,         3 5
hh ,SP DEC,         4 6
```

Divide 3.7

```
DIV,                1 7 ZC
```

EOR Memory with Akku 3.7

```
hh #. EOR,          2 2 VNZ
hh    EOR,          2 3
hhhh  EOR,          3 4
0,X   EOR,          1 2
hh ,X EOR,          2 3
hhhh ,X EOR,        3 4
hh   ,SP EOR,       3 4
hhhh ,SP EOR,       4 5
```

Increment 3.6

```
A.     INC,         1 1 VNZ
X.     INC,         1 1
hh     INC,         2 4
0,X    INC,         1 3
hh ,X  INC,         2 4
hh ,SP INC,         3 5
```

Jump 3.4

```
hh       JMP,       2 2
hhhh     JMP,       3 3
0,X      JMP,       1 2
hh ,X    JMP,       2 3
hhhh ,X  JMP,       3 4
```

Byt.Zyk. Fl.

Jump Subroutine 3.4

```
hh       JSR,         2 4
hhhh     JSR,         3 5
0,X      JSR,         1 4
hh ,X    JSR,         2 5
hhhh ,X  JSR,         3 6
```

Load Accumulator from Memory 3.5

```
hh #. LDA,            2 2 VNZ
hh    LDA,            2 3
hhhh  LDA,            3 4
0,X   LDA,            1 2
hh ,X LDA,            2 3
hhhh ,X LDA,          3 4
hh   ,SP LDA,         3 4
hhhh ,SP LDA,         4 5
```

Load HX from Memory 3.6

```
hhhh #. LHX,          3 3 VNZ
hh      LHX,          2 4
```

Load X from Memory 3.6

```
hh #. LDX,            2 2 VNZ
hh    LDX,            2 3
hhhh  LDX,            3 4
0,X   LDX,            1 2
hh ,X LDX,            2 3
hhhh ,X LDX,          3 4
hh   ,SP LDX,         3 4
hhhh ,SP LDX,         4 5
```

Logic Shift Left 3.8

```
vgl ASL,
```

Logical Shift Right 3.8

```
A.     LSR,           1 1 VNZC
X.     LSR,           1 1
hh     LSR,           2 4
0,X    LSR,           1 3
hh ,X  LSR,           2 4
hh ,SP LSR,           3 5
```

Set/Clear Bit in Memory 3.8

```
hh h MBC,             2 4
hh h MBS,             2 4
```

Byt.Zyk. Fl.

Move 3.6

```
hh hh #. MOV,   3 4 VNZ
hh hh    MOV,   3 5
hh 0,X   MOV,   2 4
,X hh    MOV,   2 4
```

Unsigned Multiply 3.7

```
MUL,            1 5  HC
```

Negate Zweierkomplement 3.7

```
A.    NEG,      1 1 VNZC
X.    NEG,      1 1
hh    NEG,      2 4
0,X   NEG,      1 3
hh ,X NEG,      2 4
hh ,SP NEG,     3 5
```

No Operation 3.9

```
NOP,            1 1
```

Nibble Swap Akku 3.8

```
NSA,            1 3
```

OR Akkur and Memory 3.7

```
hh #. ORA,      2 2 VNZ
hh    ORA,      2 3
hhhh  ORA,      3 4
0,X   ORA,      1 2
hh ,X ORA,      2 3
hhhh ,X ORA,    3 4
hh   ,SP ORA,   3 4
hhhh ,SP ORA,   4 5
```

Push Accu to Stack 3.3

```
PHA,            1 2
```

Push H to Stack 3.3

```
PHH,            1 2
```

Push X to Stack 3.3

```
PHX,            1 2
```

Byt.Zyk. Fl.

Pull Akku from Stack 3.3

```
PLA,            1 2
```

Pull H from Stack 3.3

```
PLH,            1 2
```

Pull X from Stack 3.3

```
PLX,            1 2
```

Rotate Left through Carry 3.8

```
A.    ROL,      1 1 VNZC
X.    ROL,      1 1
hh    ROL,      2 4
0,X   ROL,      1 3
hh ,X ROL,      2 4
hh   ,SP ROL,   3 5
```

Rotate Right through Carry 3.8

```
A.    ROR,      1 1 VNZC
X.    ROR,      1 1
hh    ROR,      2 4
0,X   ROR,      1 3
hh ,X ROR,      2 4
hh   ,SP ROR,   3 5
```

Reset Stack Pointer 3.3

```
RSP,            1 1
```

Return from Interrupt 3.9

```
RTI,            1 7
```

Return from Subroutine 3.4

```
RTS,            1 4
```

Subtract with Carry 3.7

```
hh #. SBC,      2 2 VNZC
hh    SBC,      2 3
hhhh  SBC,      3 4
0,X   SBC,      1 2
hh ,X SBC,      2 3
hhhh ,X SBC,    3 4
hh   ,SP SBC,   3 4
hhhh ,SP SBC,   4 5
```

Byt.Zyk. Fl.

Set Carry 3.3

```
SEC,            1 1 C
```

Set IRQ-Bit 3.8

```
SEI,            1 2
```

Store Akkumulator in Memory 3.5

```
hh    STA,      2 3 VNZ
hhhh  STA,      3 4
0,X   STA,      1 2
hh ,X STA,      2 3
hhhh ,X STA,    3 4
hh   ,SP STA,   3 4
hhhh ,SP STA,   4 5
```

Store HX 3.6

```
hh  SHX,        2 4 VNZ
```

Stop 3.9

```
STP,            1 1
```

Store Index Register 8 Bit 3.6

```
hh    STX,      2 3 VNZ
hhhh  STX,      3 4
0,X   STX,      1 3
hh ,X STX,      2 2
hhhh ,X STX,    3 4
hh   ,SP STX,   3 4
hhhh ,SP STX,   4 5
```

Subtract 3.6

```
hh #. SUB,      2 2 VNZC
hh    SUB,      2 3
hhhh  SUB,      3 4
0,X   SUB,      1 2
hh ,X SUB,      2 3
hhhh ,X SUB,    3 4
hh   ,SP SUB,   3 4
hhhh ,SP SUB,   4 5
```

Software Interrupt 3.9

```
SWI,            1 9
```

Byt.Zyk. Fl.

Transfer Akku to CCR 3.2

```
TAP,                     1 2
```

Transfer Akkumulator to X 3.6

```
TAX,                     1 1
```

Transfer CCR to Akku 3.2

```
TPA,                     1 1
```

Test for Negative or Zero 3.5

```
A.     TST,              1 1   N Z
X.     TST,              1 1
hh     TST,              2 3
0,X    TST,              1 2
hh  ,X TST,              2 3
hh  ,SP TST,             3 4
```

Byt.Zyk. Fl.

Transfer SP to XH 3.3

```
TSX,                     1 2
```

Transfer X to Akku 3.6

```
TXA,                     1 1
```

Transfer XH to SP 3.3

```
TXS,                     1 2
```

Wait 3.9

```
WIT,                     1 1
```

Wandlung Hex/Dezimal/Binär 4 Bit

Hex	Dez	Bin
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Wandlung Hex / Dezimal 8 Bit

Low Nibble

High Nibble	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255