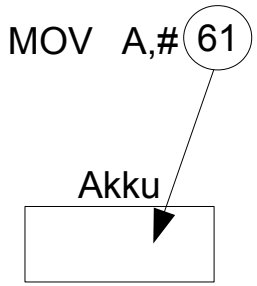
Kapitel 9

Basisbefehle des 8051 Prozessors

Operandenadressierung

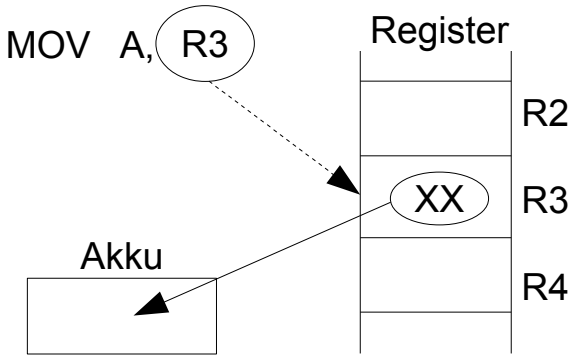
- **Immediate-Werte** (Unmittelbare Werte)
  - Der Immediate-Wert ist ein konstanter Bytewert, der direkt im Befehl steht.
  - Dass es sich um einen Immediate-Wert handelt, wird über das Rautezeichen (**#**) angegeben.
  - Der Immediate-Wert kann angegeben werden als
    - Zahl (z.B.: 61, 0x3D, 10110010B) oder Zeichen (z.B.: 'a').
    - Name, dem in einer Deklaration (z.B.: EQU) ein konstanter Wert zugewiesen wird.
  - Ein Immedate-Wert kann nur als Quelloperand, nie als Zieloperand fungieren.



(A) ← 61

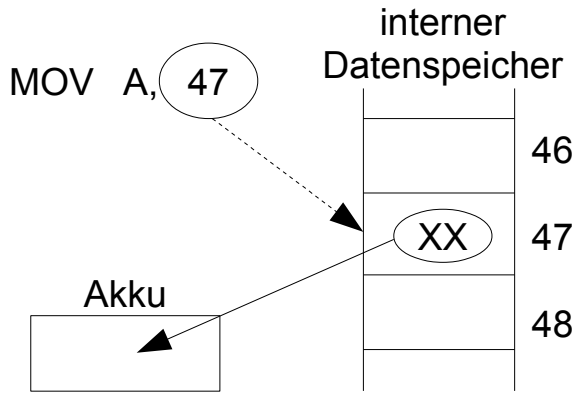
Schreibweise : (x) = Inhalt von x

- **Direkte Registeradressierung**
  - Die Registeradresse Rn im Assemblerbefehl adressiert eines der 8 Register R0, R1, … R7.
  - Der Inhalt des Registers (im Beispiel R3) kann als Quell- oder Zieloperand verwendet werden
  - Registeradresse ist Teil des OpCodes (Bits 2..0).
    - MOV A,Rr : 1110 1rrr – z.B. MOV A,R3 : 1110 1011



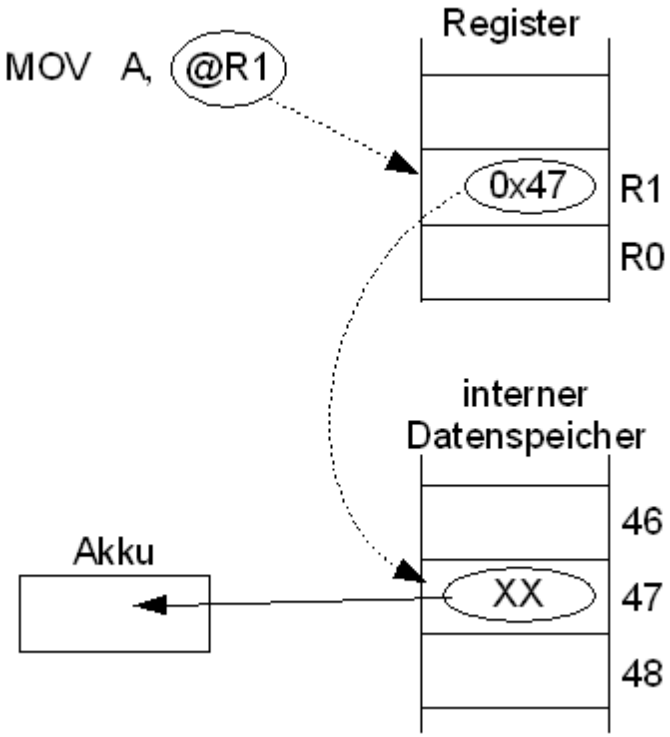
(A) ← (R3)

- **Direkte Datenspeicheradressierung**
  - Die direkte Datenspeicheradresse im Assemblerbefehl adressiert ein Byte im internen Datenspeicher.
  - Die Datenspeicheradresse kann angegeben werden als
    - Zahl (z.B.: 61, 0x3D),
    - Name,für den in einer Deklaration (via Befehl DS) ein Byte im Datenspeicher reserviert wird.
  - Der Inhalt dieses Bytes wird als Quell- oder Zieloperand verwendet.



(A) ← (47)

- **Register-indirekte Datenspeicheradressierung**
  - Das Zeichen @ weist auf die indirekte Adressierung hin.
  - Zunächst adressiert die Registeradresse im Assemblerbefehl das Register R0 oder R1.
  - Der Wert in R0 bzw. R1 ist dann eine Adresse, die ein Byte im internen Datenspeicher adressiert .
  - Der Inhalt dieses Bytes wird als Quell- oder Zieloperand verwendet.
  - Registeradresse ist Teil des OpCodes (Bits 0).
    - MOV A,@Rr : 1110 011i – z.B. MOV A,@R1 : 1110 0111



(A) ← ((R1))

Transport-Befehl : **MOV**

Skript Kap. 3.3, S.41

- MOV Zieloperand,Quelloperand
  - Der Wert des Quelloperanden wird in den Zieloperanden kopiert
  - Alter Wert des Zieloperanden geht verloren

    (A) ← c8
    (A) ← (dadr)
    (A) ← ((Ri))
    (A) ← (Rn)
- Nur maximal eine Registerspezifikation in einem Befehl
  - weil im OpCode Bits 2..0 für Rr bzw. Bit 0 für Ri verwendet werden.

OpCode	Ziel-Op.	Quell-Op.	Beispiel
74	A	#c8	MOV A,#77
E5		dadr	MOV A,32
E6+i		@Ri	MOV A,@R1
E8+r		Rr	MOV A,R4
78+r	Rr	#c8	MOV R3,#0x4A
F8+r		A	MOV R7,A
A8+r		dadr	MOV R5,Var1
75	dadr	#c8	MOV Var2,#Konst1
F5		A	MOV Var3,A
85		dadr	MOV 73,Var4
86+i		@Ri	MOV Var5,@R0
88+r		Rr	MOV 73,R5
76+i	@Ri	#c8	MOV @R0,#Konst2
F6+i		A	MOV @R1,A
A6+i		dadr	MOV @R0,Var6

Löschen-Befehl : **CLR**

- ***CLR A*** (Opcode E4)
  - Setzt den Akkumulator auf 0x00 $(A) \leftarrow 0$

Vertausch-Befehle : **XCH, XCHD, SWAP**

OpC.	OpC.	Ziel-Op.	Quell-Op.
XCH	**XCHD**		
C5	--	A	dadr
C6	D6		@Ri
C8	--		Rn

- ***XCH, XCHD Zieloperand,Quelloperand***
  - XCH : Quelloperand und Zieloperand werden vertauscht.
    - z.B.: XCH A,R5
  - XCHD : Nur die Bits 3 ... 0 des Quell- und Zieloperanden werden vertauscht.

C5	*C6*	*C8*	*D6*
$(HV) \leftarrow (A)$	$(HV) \leftarrow (A)$	$(HV) \leftarrow (A)$	$(HV) \leftarrow (A)$
$(A) \leftarrow (dadr)$	$(A) \leftarrow ((Ri))$	$(A) \leftarrow (Rn)$	$(A)_{3...0} \leftarrow ((Ri))_{3...0}$
$(dadr) \leftarrow (HV)$	$((Ri)) \leftarrow (HV)$	$(Rn) \leftarrow (HV)$	$((Ri))_{3...0} \leftarrow (HV)_{3...0}$

- **SWAP A** (Opcode C4)
  - Bits 7 ... 4 und Bits 3 ... 0 des Akkumulators werden vertauscht.

$(HV) \leftarrow (A)$

$(A)_{3...0} \leftarrow (HV)_{7...4}$

$(A)_{7...4} \leftarrow (HV)_{3...0}$

Übungsaufgaben : 1.1.1 … 1.2.1

Arithmetische Befehle : **ADD, ADDC, SUBB, INC, DEC, MUL, DIV, DA**

Skript Kap. 3.4, S.46

- ***ADD, ADDC Zieloperand,Quelloperand***
  - Der Quelloperand und der Zieloperand werden addiert, und das Ergebnis wird im Zieloperanden gespeichert.
  - Der Akkumulator ist der erste Quelloperand und gleichzeitig der Zieloperand.
  - Der alte Wert des Zieloperanden geht verloren.
  - Bei dem Befehl ADD wird **kein** Carry in Bitposition 0 addiert, auch wenn das Carry Flag vorher gesetzt war.

OpC.	OpC.	OpC.	Ziel-Op.	Quell-Op.
ADD	**ADDC**	**SUBB**		
24	34	94	A	#c8
25	35	95		dadr
26+i	36+i	96+i		@Ri
28+r	38+r	98+r		Rr

  - Bei dem Befehl ADDC wird der alte Wert des Carry Flags in die Addition mit einbezogen.
  - Das Carry Flag (CY) und das Overflow Flag (OV) im PSW werden ggf. gesetzt.

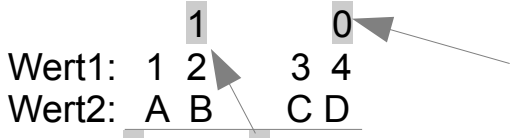
ADD
(A) ← (A) + c8 + (CY = 0) *(A) ← (A) + (dadr) + 0* *(A) ← (A) + ((Ri)) + 0* *(A) ← (A) + (Rn) + 0*

ADDC
(A) ← (A) + c8 + (CY) *(A) ← (A) + (dadr) + (CY)* *(A) ← (A) + ((Ri)) + (CY)* *(A) ← (A) + (Rn) + (CY)*

SUBB
(A) ← (A) – c8 - (CY) *(A) ← (A) – (dadr) - (CY)* *(A) ← (A) - ((Ri)) - (CY)* *(A) ← (A) – (Rn) - (CY)*

(CY) wird in allen Fällen auf 0 oder 1 gesetzt.

Beispiel: 0x1234 + 0xABCD = 0xBE01




```
MOV   A, Wert1 + 1
ADD   A, Wert2 + 1   ;ADD Befehl setzt Carry-In = 0
MOV   Wert1 + 1,A    ;Ergebnis retten
MOV   A, Wert1
ADDC  A, Wert2       ;ADDC schließt Carry in Berechnung ein
MOV   Wert1,A
```

- **SUBB Zieloperand,Quelloperand**
  - Der Quelloperand wird vom Zieloperanden subtrahiert, und das Ergebnis wird im Zieloperanden gespeichert.
  - Der Akkumulator ist der erste Quelloperand und der Zieloperand.
  - Der alte Wert des Zieloperanden geht verloren.
  - Das Carry Flag (CY) stellt bei einem SUBB Befehl den Borrow dar.
  - Das Carry Flag (CY) und das Overflow Flag (OV) im PSW werden ggf. gesetzt.
  - Der alte Wert des Carry Flags wird in die Subtraktion einbezogen.
    - Deshalb sollte vor Beginn einer Subtraktion das Carry Flag auf 0 gesetzt werden.
- **INC, DEC Zieloperand**
  - Der Wert des Zieloperanden wird um 1 erhöht bzw. erniedrigt.
  - Das Carry Flag (CY) und das Overflow Flag (OV) im PSW werden **nicht** verändert.

INC : (A) ← (A) + 1 (dadr) ← (dadr) + 1 ((Ri)) ← ((Ri)) + 1 (Rn) ← (Rn) + 1

DEC : (A) ← (A) - 1 (dadr) ← (dadr) - 1 ((Ri)) ← ((Ri)) - 1 (Rn) ← (Rn) - 1

OpC. INC	OpC. DEC	Ziel-Op.
04	14	A
05	15	dadr
06+i	16+i	@Ri
08+r	18+r	Rr

- **MUL AB** (Opcode A4)
  - Die Werte im Akkumulator und im B-Register werden multipliziert. Das MSB des Ergebnisses wird im B-Register abgespeichert, das LSB im Akkumulator.
  - Z.B.:

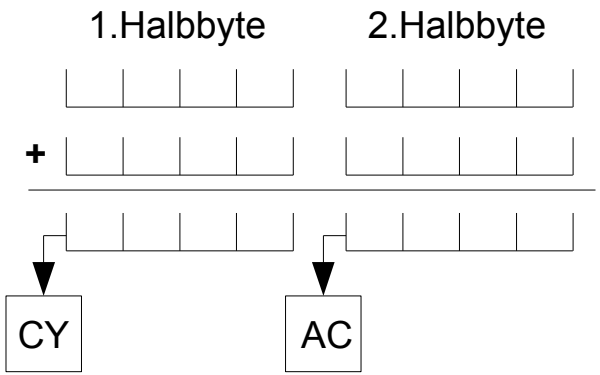
0x81,	0x02	MUL AB		0x01	0x02
A	B	→		B	A

- **DIV AB** (Opcode 84)
  - Der Wert im Akkumulator wird durch den Wert im B-Register dividiert. Der ganzzahlige Quotient wird im Akkumulator abgespeichert, der ganzzahlige Rest im B-Register.
    - Input : Dividend → A; Divisor → B; Output : Quotient → A; Rest → B
  - Z.B.:

0x43	0x02	DIV AB		0x21	0x01
A	B	→		A	B

Addieren von packed BCD-Zahlen

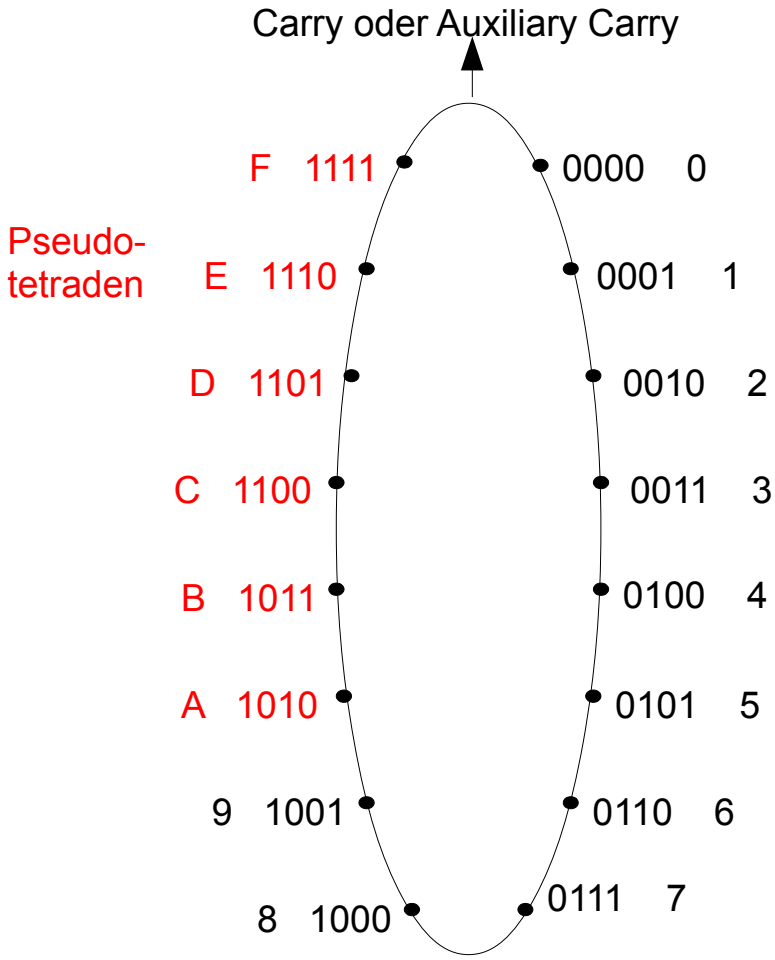
- Zusätzlich zu Carry und Overflow Flag, gibt es noch das **Auxiliary Carry Flag (AC)**.
  - Es wird gesetzt, bei einem Carry aus Bitposition 3.



- Drei Fälle :
  - Ist das Ergebnis **keine Pseudotetrade** und ist **kein Carry/Auxiliary Carry** aufgetreten, ist keine Korrektur erforderlich.z.B.: 4 + 3 → 7 + 0
  - Ist das Ergebnis eine **Pseudotetrade**, um 6 Positionen im Uhrzeigersinn vorrücken. Dadurch ergibt sich ein Carry, der im nächst höheren Halbbyte berücksichtigt werden muss.z.B.: 4 + 7 → B + 6 → 1 + Carry/Auxiliary Carry
  - Ist bei der Addition ein **Carry** aufgetreten, um 6 Positionen im Uhrzeigersinn vorrücken. Der Carry aus der Addition muss im nächst höheren Halbbyte berücksichtigt werden. z.B.: 9 + 8 → 1 + Carry + 6 → 7 + Carry/Auxiliary Carry

Zahlenkreis

Addieren : im Uhrzeigersinn vorrücken



- **DA A** (OpCode D4)
  - Der Befehl **Decimal Adjust** unterstützt die Addition von **gepackten Dezimalzahlen**
    - Bei der Addition von zwei gepackten Dezimalzahlen mit einem regulären ADD-Befehl kann wegen der Pseudotetraden das Ergebnis falsch sein.
    - Zur Addition von gepackten Dezimalzahlen verwendet man deshalb die Kombination aus ADD-Befehl und **direkt darauf folgendem** DA-Befehl.
      - Der DA-Befehl korrigiert den eventuell beim ADD-Befehl entstehenden 'Fehler'.
  - IF $(A)_{3...0} > 9$ THEN (A) ← (A) + 6
    ELSE IF (AC) = 1 THEN (A) ← (A) + 6;
    IF $(A)_{7...4} > 9$ THEN (A) ← (A) + 0x60
    ELSE IF (CY) = 1 THEN (A) ← (A) + 0x60;

- **Beispiel : 3962 + 4753 (gepackte Dezimalzahlen)**

```
MOV   A,#0x62          3  9   6  2
ADD   A,#0x53        + 4  7   5  3     ADD
DA    A              --------------------
                       B  5
MOV   A,#0x39             0   0
ADDC  A,0x47         +            0     DA
DA    A              --------------------
                       B  5
                        0
                     +  6        DA
                     --------------------
                       1  5
                        1
                     3  9
                     + 4  7        ADDC
                     --------------------
                     8  1   1  5
                      0   1
                     +  6          DA
                     --------------------
                     8  7   1  5
                      0
                     + 0           DA
                     --------------------
                     8  7   1  5
```

Logische Befehle : **ANL, ORL, XRL, CPL**

Skript Kap. 3.4, S.46

- ***ANL, ORL, XRL Zieloperand,Quelloperand***
  - Der Quelloperand und der Zieloperand werden bitweise AND-, OR- oder XOR-verknüpft, und das Ergebnis wird im Zieloperanden gespeichert.
  - Der alte Wert des Zieloperanden geht verloren.
  - Das Carry Flag (CY) und das Overflow Flag (OV) im PSW werden nicht verändert.
  - ANL geeignet zum Löschen einzelner Bits in einem Byte
    - Maske : zu löschende Bits = 0
  - ORL geeignet zum Setzen einzelner Bits in einem Byte
    - Maske : zu setzende Bits = 1
- CPL A (Opcode F4)
  - Der Akkumulator wird bitweise invertiert
  - Das Carry Flag (CY) und das Overflow Flag (OV) im PSW werden **nicht verändert**.

OpC. ANL	OpC. ORL	OpC. XRL	Ziel-Op.	Quell-Op.
54	44	64	A	#c8
55	45	65		dadr
56+i	46+i	66+i		@Ri
58+r	48+r	68+r		Rr
52	42	62	dadr	A
53	43	63		#c8

z.B: ANL A, #c8

(A) ← (A) AND c8

Übungsaufgaben :

2.1.1 … 2.2.3, 3.1.1

Sprungbefehle

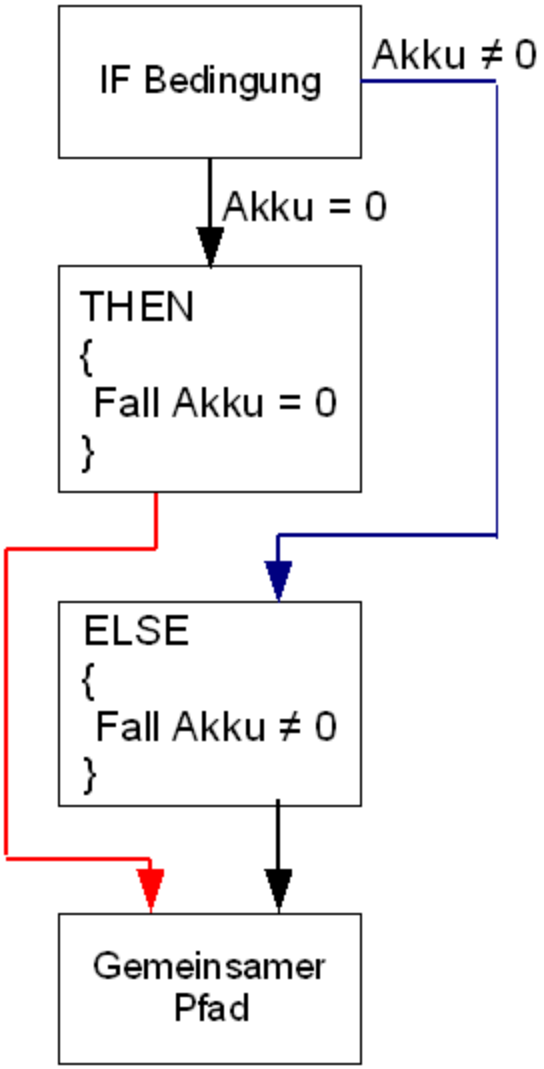
Befehlszeiger (PC)

- Bei der Ausführung der Befehle eines Programms zeigt das Hardwareregister **Befehlszeiger = Program Counter (PC)** im Steuerwerk immer auf den OpCode des als nächstes auszuführenden Befehls.
- Um einen Befehl auszuführen
  1. liest das Steuerwerk den Befehl an der im PC vorhandenen Adresse aus dem Programmspeicher aus,
  2. setzt dann den Befehlszeiger (PC) auf die Adresse des OpCodes des nächsten Befehls
     - d.h. das Steuerwerk addiert die Länge des gerade gelesenen Befehls zum PC,
  3. und beginnt erst dann mit der eigentlichen Ausführung des Befehls.
- Bei Befehlen, die keine Sprungbefehle sind, erfolgt auf diese Weise eine kontinuierliche Inkrementierung des PC.
- Bei Sprungbefehlen wird während der Ausführungsphase der PC noch einmal modifiziert.
  - Im einfachsten Fall wird der Wert im PC durch die Adresse im Sprungbefehl **ersetzt**.

Programm-speicher-adresse	**Maschinen befehl**	**Befehls-zeiger (PC)**	**Assemblerprogramm**		
2103	Befehl a	auf 2105		Befehl a	
2105	Befehl b	auf 2106		Befehl b	
2106	Springe nach 0x210B	auf 210B		JMP	Label
2109	Befehl c			Befehl c	
210B	Befehl d	auf 210D	Label:	Befehl d	
210D	Befehl e	auf 210E		Befehl e	

Vergleich : C-Code - Assembler

C-Programmfluss



Assemblercode

```
....                    ;Berechnung der Bedingung liefert
....                    ;entweder Akku = 0 oder Akku ≠ 0

JNZ  AkkuNE0            ;Sprung, wenn Akku ≠ 0
                        ;kein Sprung, wenn Akku = 0

....                    ;Befehle des Zweiges für
....                    ;die Bedingung Akku = 0
....

JMP  Gemeinsam          ;unbedingter Sprung

AkkuNE0:

....                    ;Befehle des Zweiges für
....                    ;die Bedingung Akku ≠ 0
....

Gemeinsam:
....                    ;Gemeinsamer Pfad
....
```

Nicht-sequentieller Kontrollfluss in C

- Die Verzweigungs- und Schleifenkonstrukte in C verursachen eine nicht-sequentielle Verarbeitung von Anweisungen.
- Es gibt jeweils eine **Bedingung**, die entscheidet, ob bei einer Verzweigung der eine oder der andere Pfad eingeschlagen wird, bzw. ob das Durchlaufen einer Schleife beendet werden soll oder nicht.
  - Eine solche Bedingung kann in C eine komplexe Form haben.
  - Die Auswertung der Bedingung liefert den boolschen Wert TRUE oder FALSE, der dann für die Entscheidung herangezogen wird.

Nicht-sequentieller Kontrollfluss in Assembler

- Die nicht-sequentielle Ausführung von Befehlen wird in Assemblersprache durch **Sprungbefehle** realisiert.
- Sprungbefehle beinhalten ebenfalls eine Bedingung, die entscheidet, ob der Sprung zu einem Befehl an einer anderen Stelle erfolgen soll oder ob die Befehlsausführung mit dem nächsten sequentiellen Befehl erfolgen soll.
  - Die Bedingungen bei Sprungbefehlen sind jedoch viel einfacher als in C.
    - Typische Bedingungen sind z.B.: ist ein Wert = 0 oder nicht, ist ein Bit gesetzt oder nicht.
    - Somit stellt die Bedingung von Sprungbefehlen ebenfalls einen boolschen Wert dar.
- Die meisten Prozessoren erzeugen bei arithmetischen und logischen Befehlen neben dem Ergebnis noch binäre Indikatoren, die für die Entscheidung in nachfolgenden Sprungbefehlen geeignet sind.
  - So wird z.B. ein Flag auf '1' gesetzt, wenn eine arithmetische oder logische Operation den Wert 0 als Ergebnis hat, und auf '0', wenn dies nicht der Fall ist. → das sog. **ALU Zero** Flag.
  - Ein nachfolgender Sprungbefehl kann dann einen solchen Wert benutzen, um zu entscheiden, ob tatsächlich ein Sprung durchzuführen ist oder nicht (z.B.: JZ Sprungziel1, aber auch JNZ Sprungziel2)

Nicht-sequentieller Kontrollfluss im 8051 Prozessor

- Die 8051 Implementierung unterscheidet sich von der Implementierung der meisten Prozessoren (s.o.).
- Der 8051 Prozessor berechnet den boolschen Wert innerhalb des Sprungbefehl selbst.
  - Es gibt also keine Indikatoren, die von anderen Befehlen gesetzt, und dann im Sprungbefehl verwendet werden.
  - Betrachten wir den typischen 8051 Sprungbefehl **JZ Sprungziel1**.
    - Er hat die Bedeutung : Springe, wenn der Akkumulator den Wert 0 enthält.
    - Im Sprungbefehl selbst überprüft das Steuerwerk den Akkumulator, bestimmt so den boolschen Wert für die Entscheidung und führt den Sprung aus oder nicht.
- Wenn der Compiler eine nicht-triviale Bedingung in einem C-Programm realisieren will, muss er häufig zusätzliche Assemblerbefehle einfügen, die aus der ursprünglichen Bedingung eine einfache Bedingung wie z.B ALU=0 ableiten, die dann in einem Sprungbefehl verwendet werden können.
  - z.B.: C-Bedingung aa == bb → MOV A,aa; XRL A,bb; JZ Sprungziel
  - In der gleichen Lage ist auch der Assemblerprogrammierer.

Der unbedingte Sprungbefehl

- Bei einem **unbedingten** Sprungbefehl (z.B. **JMP Sprungziel3**) wird **immer** der Sprung zur Zieladresse durchgeführt.
  - Man kann den unbedingten Sprungbefehl als Sonderfall betrachtet, bei dem die Bedingung lautet : **immer**.

Relokation

- Wenn ein Programm bei mehrfacher Ausführung immer an anderen Stellen im Programmspeicher geladen werden kann, ist der Assembler nicht in der Lage, das Problem der unterschiedlichen Anfangsadressen zu lösen. Er benutzt deshalb beim Assemblieren einfach die Anfangsadresse 0.
- Wenn nun das Programm zu unterschiedlichen Zeiten an verschiedene Adressen im Programmspeicher geladen wird, stimmen die vom Assembler/Compiler vergebenen Zieladressen von Sprungbefehlen nicht mehr.
- Die Zieladressen müssen also angepasst werden → **Relokation**
  - (Andere Gründe für Relokation werden später noch behandelt)
- Um den Aufwand für Relokation zu vermindern, wurde im 8051 das Konzept der **relativen Sprungadressen** eingeführt.

8051 Sprungziele

- Die 8051 Familie stellt drei Adresstypen für Sprungbefehle zur Verfügung:
  1. Absolute Sprungadressen
  2. Relative Sprungadressen
  3. Absolute Sprungadressen in der selben Page (wenig verwendet)
- **Absolute Adressen**
  - Die Zieladresse im Sprungbefehl **ersetzt** den Wert im PC
    - ***LJMP adr16*** *(Befehlslänge 3 By)* (Opcode 02)

      z.B.: Assemblerbefehl LJMP 0x4321 Springe zum Befehl bei Adresse 0x4321
      Maschinenbefehl 0x02 4321

  - Die absoluten Zieladressen müssen bei der Relokation angepasst werden.
  - In 8051 sind absolute Zieladressen nur in unbedingten Sprungbefehlen vorgesehen.

- **Relative Adressen**
  - Die relative Adressangabe im Sprungbefehl wird zu dem Wert im PC **addiert** und das Ergebnis in den PC zurückgeschrieben.
  - Die 8 Bits große relative Adressangabe wird als **vorzeichenbehafteter** Wert betrachtet und erlaubt damit Sprünge um maximal 127 Bytes vorwärts und maximal 128 Bytes rückwärts relativ zum momentanen Wert des PC.
  - Da der PC am Anfang der Befehlsausführung bereits so modifiziert wird, dass er auf den nächsten Befehl zeigt, wird die Zieladresse relativ zur Adresse des auf den Sprungbefehl folgenden Befehls berechnet.

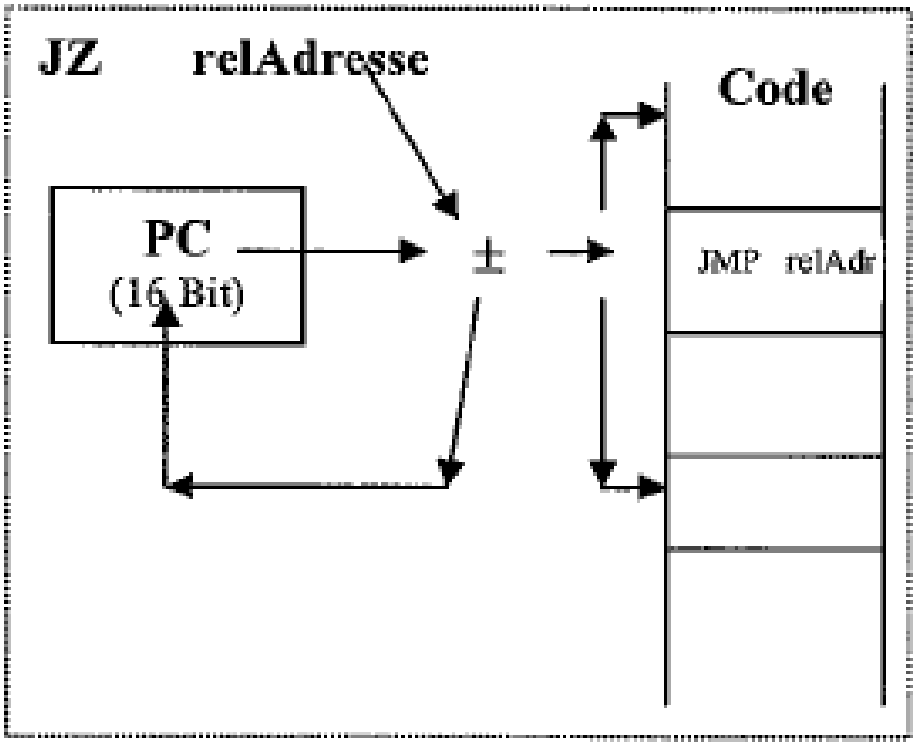


Bild 44: Relative Adressierung

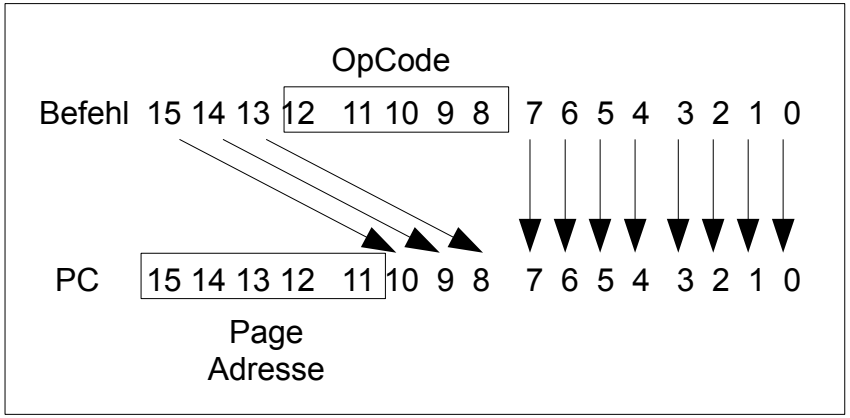
- ***SJMP rel*** *(Befehlslänge 2 By)* (Opcode 80)
  siehe *Skript 3.3.7*

  Der Sprung wird unbedingt durchgeführt.

  z.B.: SJMP 0x21 → 0x80 21 : Inkrementiere den Befehlszähler um 0x0021.
  z.B.: SJMP 0xF8 → 0x80 F8 : Inkrementiere den Befehlszähler um 0xFFF8
  → dekrementiere den Befehlszähler um 8.

- Vorteil : Bei der Relokation brauchen die relativen Adressangaben nicht angepasst zu werden.
- Nachteil : Die Addition des Wertes im PC und der relativen Adressangabe ist allerdings zeitraubender als eine einfache Ersetzung des PC-Wertes.

- **Absolute Adressierung in der selben Page**
  - Die Adresse im PC kann man sich zusammengesetzt vorstellen aus :
    - der Page Adresse : PC Bits 15 …11
    - dem Displacement : PC Bits 10 … 0
    - Das Displacement eines Bytes ist eine Art Offset vom Page-Anfang zu dem betrachteten Byte
    - Wegen der Displacement-Länge von 11 Bits ist jede Page 2 kB groß.
  - 8051 Befehl : **AJMP label** (Befehlslänge 2 Bytes)
    - Ersetzt die 11 Displacement Bits im PC durch 11 Adressbits aus dem Sprungbefehl.
    - Verursacht einen Sprung innerhalb der Page zum 11 Bit Offset, der im Sprungbefehl steht.
    - Allerdings stehen die 11 Adressbits im Sprungbefehl nicht direkt hintereinander.
    - Die PC Bits 10 … 8 werden ersetzt durch die Bits 15 … 13 des Sprungbefehls.
    - Die PC Bits 7 … 0 werden ersetzt durch die Bits 7 … 0 des Sprungbefehls.



    - Die Bits 12 … 8 des Sprungbefehls sind sozusagen die OpCode-Spezifikation
      - OpCode : xxx0 0001B (0x01, 0x21, 0x41, 0x61, … , 0xE1)
  - Vorteil : Sprungadressen brauchen nicht angepasst werden, wenn das Programm auf **irgend eine 2 kB Grenze** geladen wird.

Weitere Sprungbefehle mit relativer Adresse als Sprungziel

- ***Jump if Akku is Zero; Jump if Akku is Not Zero***

  JZ rel (Opcode 60)***; JNZ rel*** (Opcode 70)

  Wenn der Wert im Akkumulator = 0 (JZ) bzw. ≠ 0 (JNZ) ist, wird ein Sprung zu dem über rel spezifizierten Befehl durchgeführt.Trifft die Bedingung nicht zu, wird der nächste sequentielle Befehl ausgeführt.

  JZ rel

  (PC) ← (PC) + 2;
  IF (A) = 0 THEN (PC) ← (PC) + rel;

  JNZ rel

  (PC) ← (PC) + 2;
  IF (A) ≠ 0 THEN (PC) ← (PC) + rel;

- ***Compare and Jump if Result is Not Equal***

  CJNE A,#c8,rel (Opcode B4)***, CJNE Rr,#c8,rel*** (Opcode B8)***, CJNE @Ri,#c8,rel*** (Opcode B6)

  Der Wert im Akkumulator, im Register Rr, oder an der durch Ri indirekt adressierten Datenspeicheradresse wird verglichen mit dem Immediatewert #c8. Wenn die beiden Werte nicht gleich sind, erfolgt der Sprung zu dem über rel spezifizierten Befehl. Wenn die beiden Werte gleich sind, wird der nächste sequentielle Befehl ausgeführt.
  Wenn der 1.Operand (A, Rn, @Ri) kleiner ist als der 2. Operand (#c8), wird das Carry Flag gesetzt. Andernfalls wird das Carry Flag gelöscht.

  CJNE A,#c8,rel

  (PC) ← (PC) + 3;
  IF (A) ≠ c8 THEN (PC) ← (PC) + rel;
  IF (A) < c8 THEN (CY) ← 1
  ELSE (CY) ← 0;

  CJNE Rn,#c8,rel

  (PC) ← (PC) + 3;
  IF (Rn) ≠ c8 THEN (PC) ← (PC) + rel;
  IF (Rn) < c8 THEN (CY) ← 1
  ELSE (CY) ← 0;

  CJNE @Ri,#c8,rel

  (PC) ← (PC) + 3;
  IF ((Ri)) ≠ c8 THEN (PC) ← (PC) + rel;
  IF ((Ri)) < c8 THEN (CY)← 1
  ELSE (CY) ← 0;

Weitere Sprungbefehle mit relativer Adresse als Sprungziel (Forts.)

- ***Compare and Jump if Result is Not Equal***

  CJNE A,dadr,rel (Opcode B5)

  Der Wert im Akkumulator wird verglichen mit dem Inhalt der Datenspeicheradresse dadr. Wenn die beiden Werte nicht gleich sind, erfolgt der Sprung zu dem über rel spezifizierten Befehl. Wenn die beiden Werte gleich sind, wird der nächste sequentielle Befehl ausgeführt.
  Wenn der 1.Operand (A) kleiner ist als der 2. Operand (dadr), wird das Carry Flag gesetzt. Andernfalls wird das Carry Flag gelöscht.

  CJNE A,dadr,rel

  (PC) ← (PC) + 3;
  IF (A) ≠ (dadr) THEN (PC) ← (PC) + rel;
  IF (A) < c8 THEN (CY) ← 1
  ELSE (CY) ← 0;

- ***Decrement and Jump if Result is Not Zero***

  DJNZ Rn,rel (Opcode D8)***; DJNZ dadr,rel*** (Opcode D5)

  Der Wert im im Register, bzw. in der Datenspeicheradresse dadr wird zunächst dekrementiert. Hat dieser Wert nicht den Wert 0 erreicht, erfolgt der Sprung zu dem über rel spezifizierten Befehl. Wenn dieser Wert gleich 0 ist, wird der nächste sequentielle Befehl ausgeführt.

  DJNZ Rn,rel

  (PC) ← (PC) + 2;
  (Rn) ← (Rn) – 1
  IF (Rn) ≠ 0 THEN (PC) ← (PC) + rel;

  DJNZ dadr,rel

  (PC) ← (PC) + 2;
  (dadr) ← (dadr) - 1
  IF (dadr) ≠ 0 THEN (PC) ← (PC) + rel;

Weitere Sprungbefehle mit relativer Adresse als Sprungziel (Forts.)

- **Jump if Carry Flag on; Jump if Not Carry Flag on**

  JC rel (Opcode 40)**; JNC rel** (Opcode 50)

  Wenn der Wert des Carry Flags den Wert '1' (JC) bzw. den Wert '0' (JNC) aufweist, erfolgt der Sprung zu dem über rel spezifizierten Befehl. Andernfalls wird der nächste sequentielle Befehl ausgeführt.

JC rel	***JNC rel***
(PC) ← (PC) + 2; *IF (CY) = 1 THEN (PC) ← (PC) + rel;*	*(PC) ← (PC) + 2;* *IF (CY) = 0 THEN (PC) ← (PC) + rel;*

Weitere Sprungbefehle

- Sprungbefehle, die die Entscheidung zum Sprung in Abhängigkeit von Bitwerten treffen, werden in der Sektion **Bitbefehle** behandelt.
- Der Sprungbefehl **JMP @+DPTR**, der neben einer absoluten Adresse noch einen Indexwert einschließt, wird in der Sektion **Adressräume** behandelt.
- Im 8051 Assembler steht auch der Befehl **JMP** (z.B.: JMP label) zur Verfügung. Der Assembler erzeugt daraus den 2 Byte Befehl SJMP, wenn möglich, in den anderen Fällen erzeugt er den 3 Byte Befehl LJMP.
  - Im Fall eines Sprungs nach rückwärts um weniger als 128 Bytes ist der SJMP Befehl mit relativem Sprungziel möglich.
  - Bei Sprüngen nach vorwärts kennt der Assembler den Abstand noch nicht. Er benutzt in diesen Fällen immer den LJMP Befehl.

Abfragen

Beispiele von häufig vorkommenden Bedingungen und ihre Realisierung durch 8051 Assemblerbefehle :

- Wert = (≠) konstanter Vergleichswert
  - *CJNE (z.B.: CJNE A,#c8,rel)*
  - DJNZ (z.B.: DJNZ dadr,rel) ;Vergleichswert ist '0'
  - XRL A,#c8 ;vergleiche bitweise
    JZ rel ;Sprung, wenn alle Bits gleich sind
  - ANL A,#c8 ;vergleiche Bits, die in der Konstante = 1 sind
    JZ rel ;Sprung, wenn all diese Bits = 0
- *Wert im Akku = (≠) variabler Vergleichswert z.B. im R7*
  - *CJNE (z.B.: CJNE A,dadr,rel)*
  - XRL A,R7 ;vergleiche bitweise
    JZ rel ;Sprung, wenn alle Bits gleich
- *Wert im Akku > (<, ≥, ≤)* konstanter oder *variabler Vergleichswert*
  - SUBB A,R7 ;Borrow wird gesetzt, wenn A < R7
    JC SPRUNGZIEL ;Sprung, wenn A < R7
  - *SUBB A,#100 ;Borrow wird gesetzt wenn A < 100*
    JNC SPRUNGZIEL ;Sprung, wenn A ≥ 100
  - *SUBB A,R7 ;Borrow wird gesetzt, wenn* A < R7
    JC SPRUNGZIEL ;Sprung, wenn A < R7
    JZ SPRUNGZIEL ;Sprung, wenn A = R7
    ;also Sprung, wenn A ≤ R7

Übungsaufgaben :

2.1.1 … 2.2.3, 3.1.1