Architektur der Intel 8051 Familie und Grundlegende Programmierung

Reto Gurtner

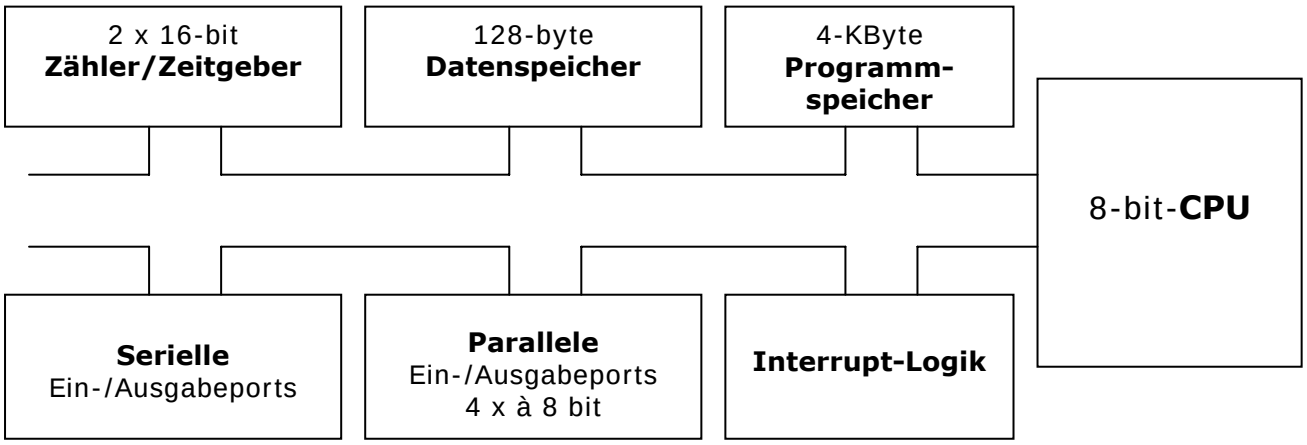
2005

1. Die Geschichte der 8051-Mikrocontrollerfamile

- Anfang **80er Jahre** brachte Intel die 8051 Generation auf den Mark
- Der Prozessor war auf die **Lösung von Steueraufgaben** zugeschnitten
- Zu Beginn gab es **3 Prozessorvarianten**:
    - 8051
        - Integrierten Programmspeicher (ROM)
        - Anwendungen mit hohen Stückzahlen
    - 8031
        - Kein interner Programmspeicher
    - 8751
        - Integriertes EPROM
- Zum **ersten Mal** die **Special Function Register** im internen RAM → dadurch wurde der **Rechner sehr flexibel**
- **Mehrere Hersteller** haben den **Kern des Prozessors abgeändert** (z.B. Siemens, Philips) → führte zu vielen verschiedenen Varianten des Prozessors

2. Grundlegende Hardwarestruktur und Funktionsmodell

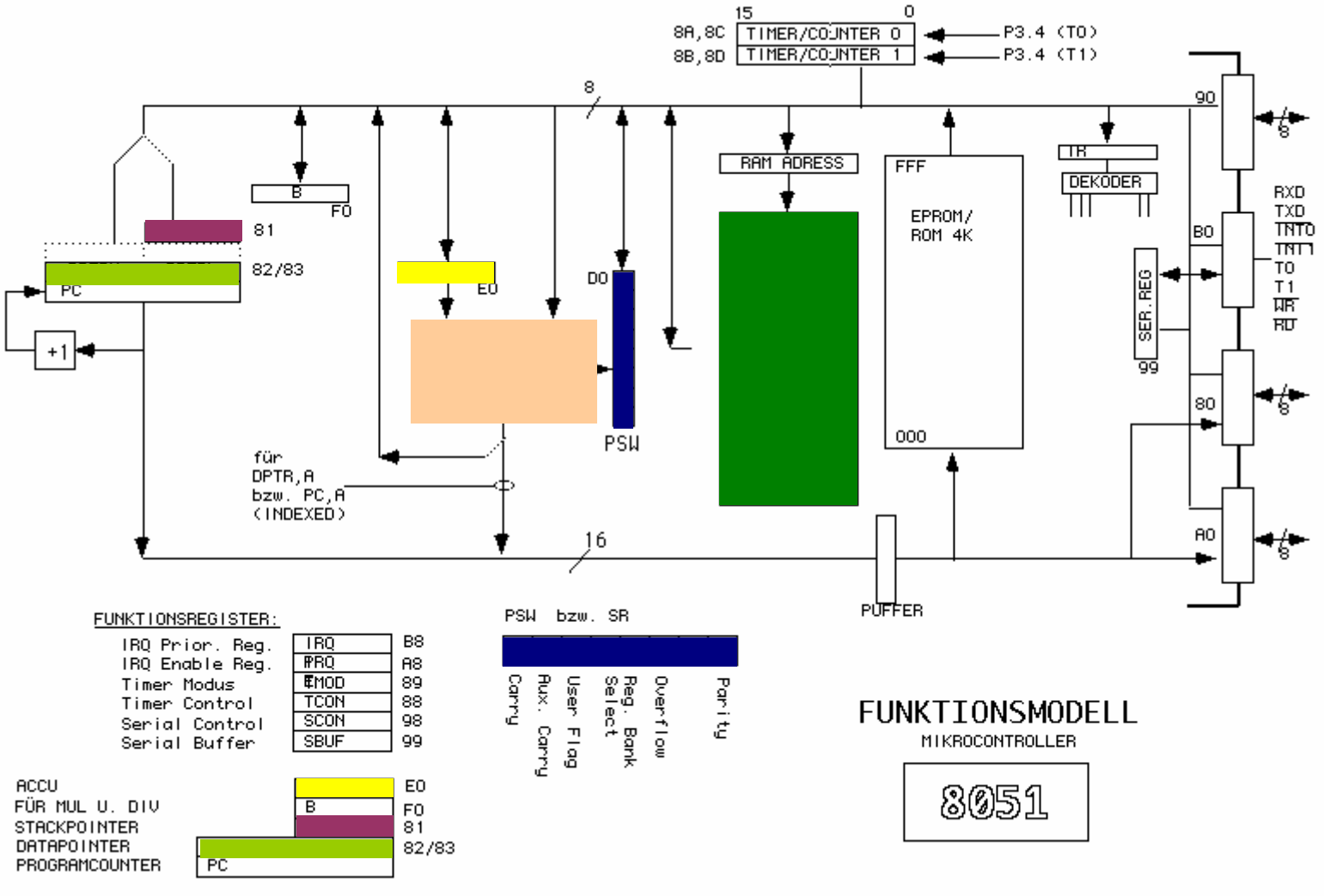
2.1 Hardwarestruktur



Die CPU hat 3 Arbeitsregister:

- **Accumulator (A):** Für arithemtische und logische Operationen
- **B-Register:** Für 8-Bit Multiplikation, Divison sowie als allgemein nutzbares Register
- **Programmstatuswort PSW**

2.2 Funktionsmodell 8051



2.3 Kurzübersicht

ALU – Arithmetic Logical Unit

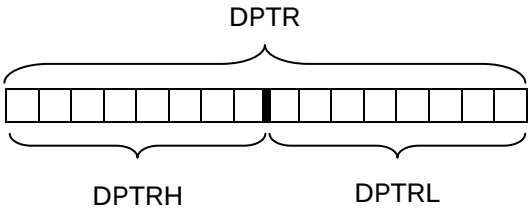
- Wird auch als **Rechenwerk** bezeichnet
- Führt **Rechenoperationen** aus (Addieren, Subtrahieren, Dividieren und Multiplizieren)

Accumulator

- als **A** in Assembler definiert
- **8-Bit** gross

DPTRL

- Repräsentiert den **Datapointer**
- **16–Bit** gross
- Kann getrennt Adressiert werden (DPTRH, DPTRL)
- **DPTRH** und **DPTRL** sind **je 8-Bit gross**



Internes RAM

- **256 MB** gross
- Adressenbereich: **0x00 – 0xFF**
- Siehe Kapitel 2!

PSW – Status Register

- **8-Bit** gross
- enthält **verschiedene wichtige Bits** wie z.B. das Carry!

CY	AC	F0	RS1	RS0	OV	--	P

CY- Übertragsbit

Beeinflusst von: Addition- und Subtraktion, logischen und Schiebeoperationen

Wird gesetzt:
Wenn das Resultat grösser als 8 Bit ist

AC- Hilfsübertragsbit (Auxiliary Carry).

Beeinflusst von: Addition und Subtraktion
Muss bei BCD-Arithmetik verwendet werden

F0- Ein vom Benutzer frei verfügbares Bit

RS1, RS2 Wählen die gerate aktive Registerbank aus
Beim Einschalten des Prozessors wird automatisch RegisterBank 0 gesetzt.

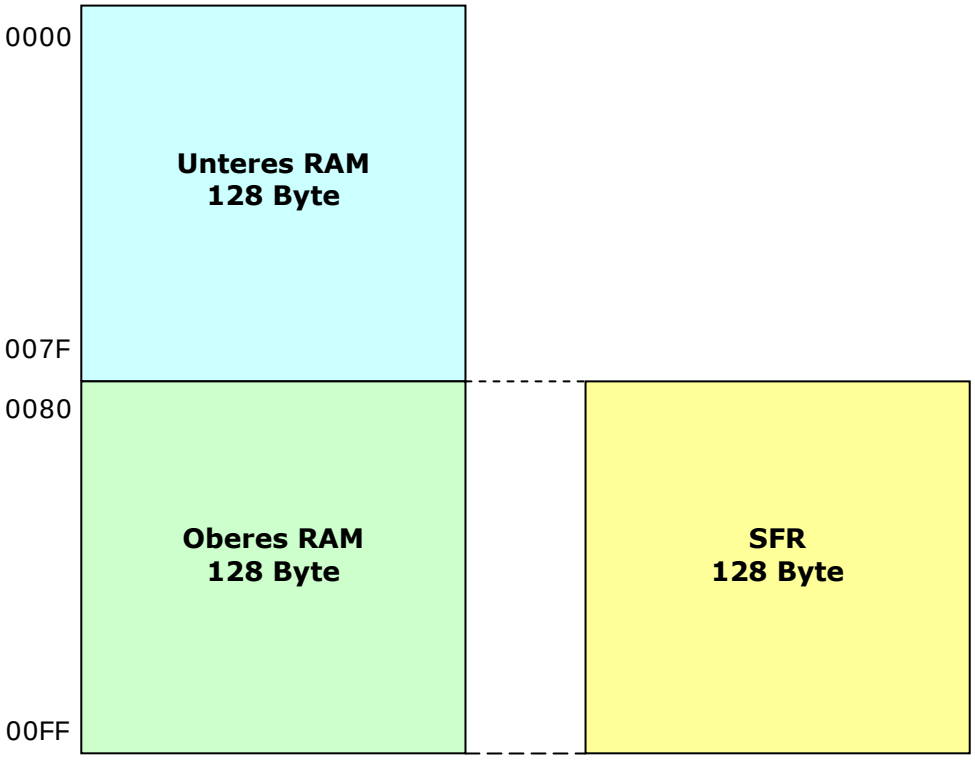
OV Überlaufbit bei Rechnung mit zwei Zahlen im 2 Komplement

P Parität. Wird gesetzt, wenn Anzahl Einsen im Accumulator ungerade ist

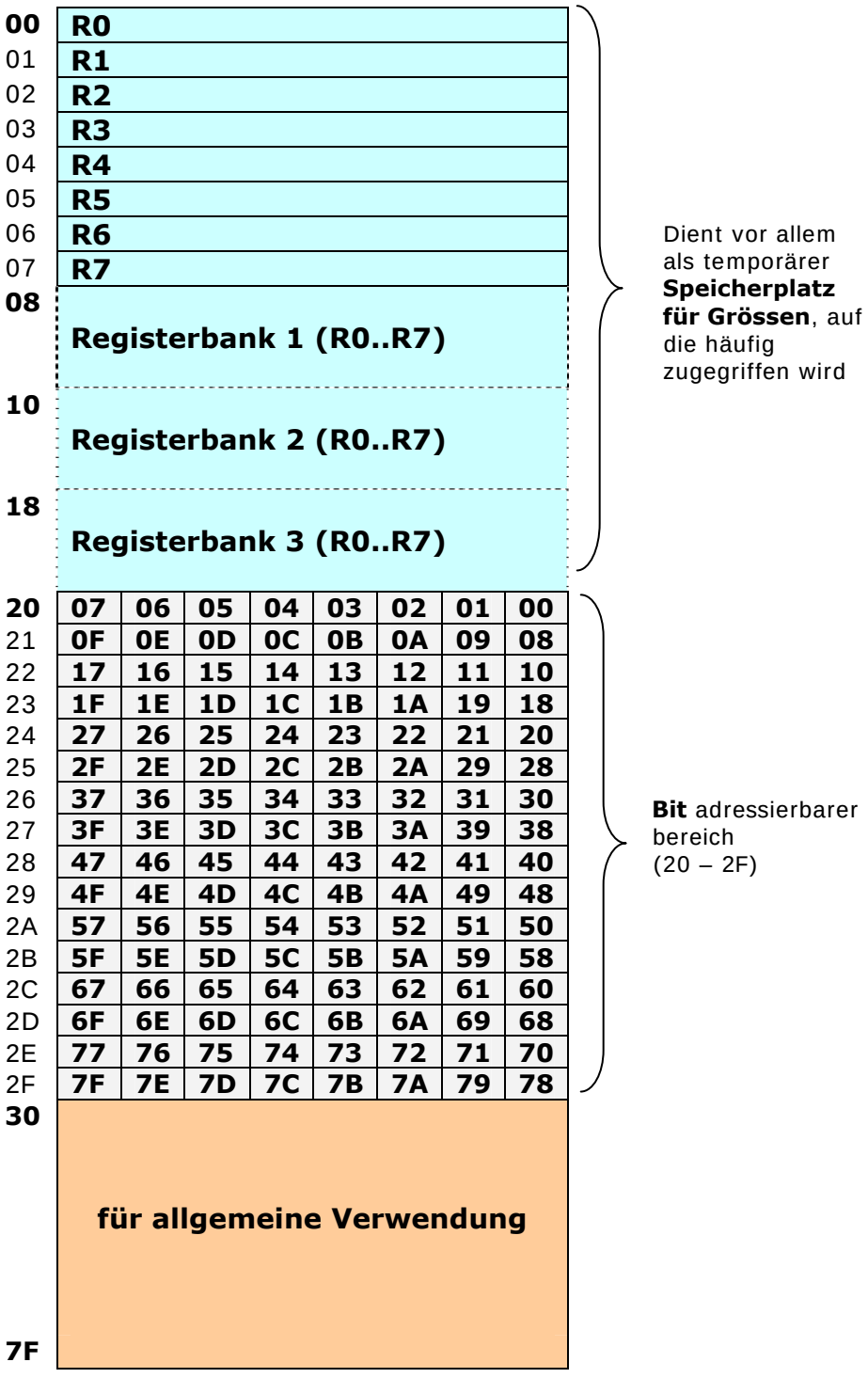
---

3. Internes RAM

3.1 Aufbau des Internen RAM



3.2 Unteres RAM



Allgemein

- **128** Byte Gross
- **Alle Speicherplätze** sind **byteweise les- und schreibbar**
- **In jeder Intel 8051 Familie** enthalten
- Alle Speicherplätze **indirekt oder direkt adressierbar**

unterer Bereich

- **32 Byte Gross**
- Es ist immer **nur eine Registerbank aktiv**!
- **Welche** dies ist, wird über die Bits **RS0 und RS1** im Programmstatuswort festgelegt
- **Zwischen** den **Bänken kann umgeschaltet** werden → daher verlieren wir nicht immer wieder die gespeicherten Werte

mittlerer Bereich

- **16 Byte Gross**
- Zusätzlich **Bit Adressierbar**
- **Jedes Bit** kann **direkt angesprochen/ausgewertet** werden
- **Bedingte Sprünge** abhängig von einem Bit sind **möglich!**
- **Bits** können **mit Symbolischen Namen** übersehen werden

Assembler:

z.B. Bit 8 Setzen: **SETB** 8	z.B Bit 8 zurücksetzen: **CLR** 8
Bedingter Sprung für Bit 8 **JB** 8, Fehler	Symbolischer Name **JB** name, Fehler

Oberer Bereich

- **80 Byte Gross**
- Für die **allgemeine Verwendung**
- Im **8751** auch **noch** der **Stack** enthalten

3.3 Oberes RAM

Allgemein

- **128 Byte** Speicherzellen
- **Keine Sonderfunktionen**
- Nur **indirekt adressierbar** → geschieht durch die **Benutzung der Register R0 oder R1** der aktiven Registerbank.

Adressieren dieses Bereichs:

Um einen Speicherplatz in den oberen 128 Byte anzusprechen, muss dieser zuerst in das Register R0 oder R1 geladen werden:

Assembler:

```
MOV R0, #80H      // Adresse ins Zeigerregister laden
MOV A, @R0        // Inhalt in den Accumulator bringen
```

- Diese Adressierungsart ist **langsamer als direkt**! → desshalb **oft verwendete Konstanten** usw. im **unteren Bereich** des **Internen RAM's speichern**

3.4 SFR – Bereich

Allgemein

- Jeder Controller der Familie besitzt je nach Ausstattung **unterschiedliche Mengen von SFR-Registern.**
- Können **nur direkt adressiert** werden
- Enthält diverse bekannte Register (z.B Accumulator)

Abbild des 8051 Standart SFR-Bereich

80	P0 *
81	SP
82	DPL
83	DPH
87	PCON
88	TCON *
89	TMOD
8A	TL0
8B	TL1
8C	TH0
8D	TH1
90	P1 *
98	SCON *
99	SBUF
A0	P2*
A8	IE *
B0	P3 *
B8	IP *
D0	PSW (Status Register) *
E0	A (Accumulator) *
F0	B *

* Bitadressierbar

Adressieren dieses Bereichs

Assembler:

```
//Inhalt des Speicherplatzes 80H in den Accumulator laden
MOV A #80H
```

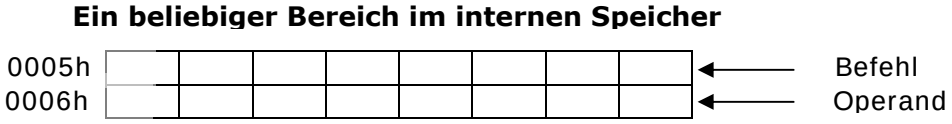
3.5 Zusammenfassung: Zugriff auf das Interne RAM

Adressierungen

Bereich	Adressierungsart	Adressenbereich
Unteres RAM	Direkt / Indirekt	0000 – 007F
Oberes RAM	Indirekt	0080 – 00FF
SFR	Direkt	0080 – 00FF

Befehle mit Operanden:

- Hat ein Befehl Operanden, so sind die **Operanden unmittelbar nach dem Befehl im Programmspeicher** gespeichert:



Definition Operand:
Numerischer Wert oder String , der durch einen Operator zu einem anderen Operanden (oder mehreren) in Beziehung gesetzt wird.

Bei 2 * 3 sind die Zahlen die Operanden und " * " der Operator.

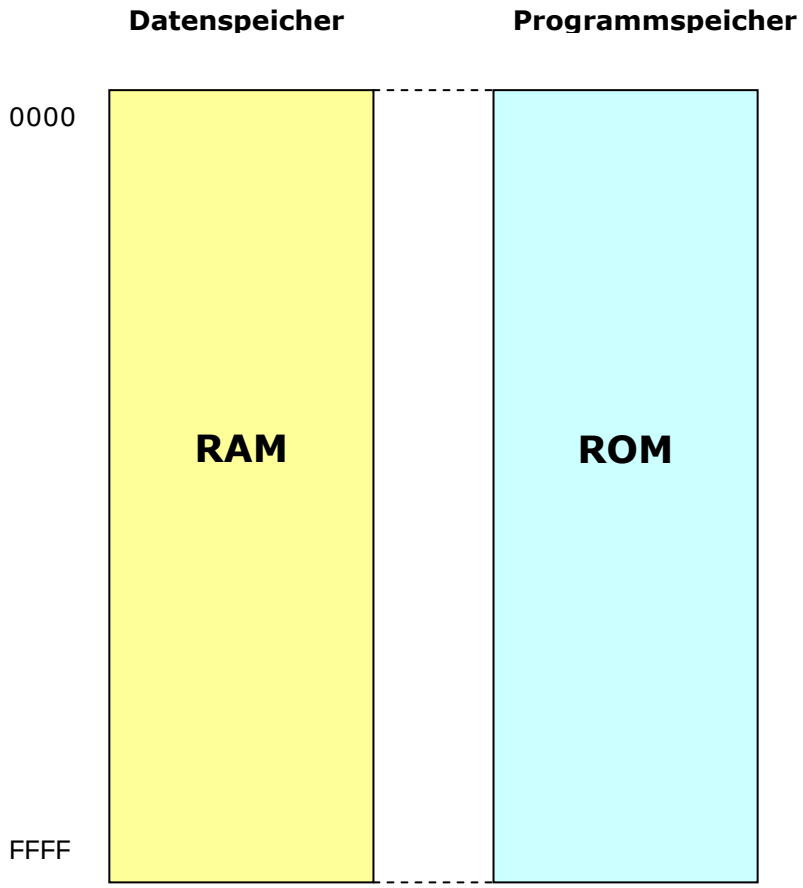
Befehle auf Registern

- Befehle die nur auf Registern geschehen, haben keine Operanden:

Beispiele:
- Inkrementier A
- AND A mit einem Wert, dessen Datenspeicheradresse im Register R0 liegt
- XOR A mit dem Inhalt von Register R3

4. Externer Speicher

4.1 Aufbau des externen Speicher



4.2 Datenspeicher

- Hier kann man lesen und schreiben
- Max. 64 KB Gross
- Interner Datenspeicher und Externer Datenspeicher → Adressparallel. Unterscheidung mit Hilfe von: MOV (intern) und MOVX (extern)

Lesen:	**MOVX** A, @R0 **MOVX** A, @R1	**MOVX** A, @DPTR
Schreiben:	**MOVX** @R0, A **MOVX** @R1, A	**MOVX** @DPTR, A

4.3 Programmspeicher

- Hier kann man nur lesen!
- Er ist bei gewissen Prozessoren auch Intern!
- Max. 64 KB gross

Lesen:	**MOVC** A, @A + PC	**MOVC** A, @A + DPTR

5. Befehlssatz und Adressierungsarten

5.1 Transportbefehle

Register ←→ Register

Der Befehlssatz kennt für die Registeradressierung:

- Die **Register R0, R1, R2... R7** der ausgewählten Registerbank
- **Accumulator** A
- Register **B**
- 16-Bit-Datapointer **DPTR** (besteht aus 2 x 8-Bit Register (DPH, DPL)

Beispiele:

```
MOV R0, A   // Inhalt von A nach R0 bringen
MOV A, R7   // Inhalt von R7 nach A bringen
XCH A, R1   // Inhalte von A und R1 vertauschen
```

Register ←→ Speicher

Beispiele:

```
MOV A, 40h          // Inhalt des internen Speicherplatzes 40h
                    // in den Akkumulator bringen

MOV @R0, A          // Inhalt des Accumulator in den durch R0 adressierten
                    // Speicherplatz in oberen Bereich des internen RAM
                    // bringen

XCH A, 40h          // Inhalt des Speicherplatzes 40h mit dem Accumulator
                    // in den vertauschen

MOVX A, @R0         // Inhalt des R0 adressierten externen RAM in den
                    // Accumulator bringen

MOVC A, @A+DPTR     // Inhalt des Speicherplatzes, dessen Adresse die
                    // Summe des Accumulators und des Datapointers ist,
                    // in den Accumulator bringen
```

Register ←→ Ein / Ausgabe

- Ein/Ausgabe erfolgt grundsätzlich über die SFR-Register
- Für Parrallele Ein/Ausgabe sind dies die 8-Bit-Ports P0..P3
- Für Serielle Ein/Ausgabe das SBUF Register im SFR-Bereich

Beispiele:

```
MOV P3, A        // Den im Accumulator stehenden Wert über Port 3 ausgeben

MOV SBUF, R3     // Den Inhalt von R3 seriell über SBUF ausgeben
```

Speicher ←→ Ein / Ausgabe

- Es ist grundsätzlich möglich den Inhalt von Speicherplätzen direkt auszugeben/einzulesen

Beispiele:

```
MOV P3, 40h      // Inhalt des internen Speicherplatzes 40h über Port 3
                 // ausgeben

MOV @R0, P0      // Port 0 einlesen und unter dem internen Speicherplatz
                 // abspeichern, dessen Adresse in R0 steht
```

Bit ←→ Bit

- Max 256 Bit sind direkt adressierbar
- Der eine Bereich internes RAM
- Anderer Bereich im SFR

Beispiele:

```
MOV C, 3         // Bring das direkt adressierte Bit 2 in das Carry-Bit

MOV 20.2, C      // Lade in der Adresse 20 in das 2 Bit das was im Carry steht

MOV C, P3.4      // Die an Port 3 Bit 4 anliegende Information ins Carry lesen
```

5.2 Logische Befehle

Bedingte Sprünge

- Bei bedingten Sprüngen wird das **Sprungziel immer relativ zum aktuellen Stand des Befehlszählers** in einer 8-Bit 2-Komplementzahl angegeben

  - **JZ** Sprung, wenn A = 0
  - **JNZ** Sprung, wenn A != 0

  - **JC** Springe, wenn das Carry-Flag gesetzt ist
  - **JNC** Springe, wenn das Carry-Flag nicht gesetzt ist

  - **JB** Springe, ein direkt adressiertes Bit gesetzt ist
  - **JNB** Springe, wenn ein direkt adressiertes Bit nicht gesetzt ist

  - **CJNE** Vergleiche zwei Operanden, springe wenn sie ungleich sind

  - **DJNZ** Dekrementiere einen Schleifenzähler,springe solange bis dieser !=0 ist.

Unbedingte Unterprogrammaufrufe

- Zwei verschiedene Befehle:

- **ACALL:**
  - Wenn Startadresse innerhalb der ersten 2 k-byte des Programmspeichers liegt
  - Befehlslänge 2 K-Bite, 11 Bit Adresse

- **LCALL:**
  - Kann jedes Unterprogramm an beliebiger Stelle im Programmspeicher aufrufen
  - 3 Byte Länge, 16 Bit Adresse

Bedingungen von Unterprogrammen

- Unterprogramme müssen mit Befehl **RET** abgeschlossen werden

- Es wird **zurück** zu der **aufrufenden Adresse** gesprungen und der **nächste Befehl** ausgeführt

6. Adressierungsarten

6.1 Unmittelbare Adressierung

```
MOV A, #daten
```

- **Konstanten** können direkt **als Teile von Befehlen** angegeben werden
- Die Zahl #daten soll unmittelbar in den Akkumulator geladen werden
- **Umkehrung nicht möglich!**

Beispiel:
MOV A, #23h

6.2 Direkte Adressierung

```
MOV A, direkte Adresse
```

- **Inhalt** der **Speicherzelle** wird **direkt in A geladen** → Anhand der Speicheradresse finden wir direkt zu dem gespeicherten Zahlenwert.
- **Umkehrung** des Befehls ist **möglich**

Beispiele:
MOV A, R0
MOV R0, A

6.3 Indirekte Adressierung

```
MOV direkt, #12
MOV A, @direkt
```

- @direkt: es soll **nicht** der **Inhalt** von „direkt“ in A geladen werden, **sondern** der **Inhalt** der sich **hinter** – der in „direkt“ gespeicherten – **Adresse befindet.**
- **Umkehrung** des Befehls ist **möglich!**
- **Vorteil:** Während der Laufzeit kann mit immer dem gleichen Maschinenbefehl auf unterschiedliche Speicherzellen zugegriffen werden.

6.4 Indirekte, indizierte Adressierung

- **Gleich** wie **indirekte Adressierung**, aber es wird zuerst noch Inhalt des **Akkumulators hinzugefügt**.

6.5 Adressierung über Basis- plus Indexregister

- Einzige Adressierungsart um auf Programmspeicher zu zugreifen.
- Indexregister = Accumulator
- Basisregister = DPTR oder PC

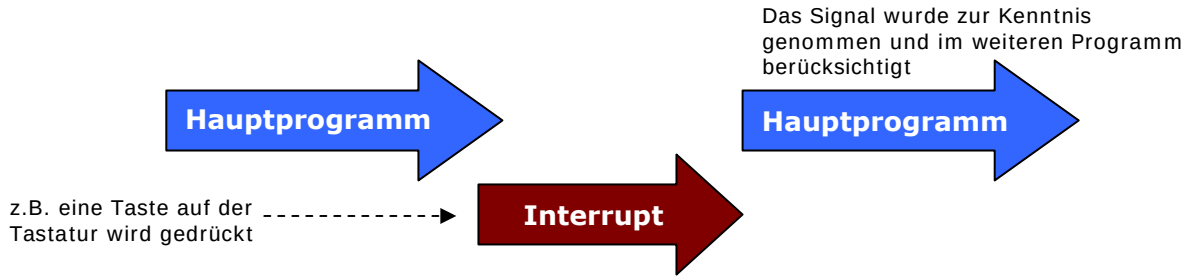
Beispiele:

```
MOVC A, @A + DPTR        // Inhalt A + Inhalt DPTR werden zur Adresse
MOVC A, @A + PC          // Inhalt A + Inhalt PC werden zur Adresse
JMP @A + DPTR            // Inhalt A + Inhalt DPTR werden zur Sprungadresse
```

7. Interrupts

7.1 Allgemein

- Interrupts sind **durch Hardwaresignale gesteuerte Unterprogrammsprünge.**
- Werden **durch externe Ereignisse ausgelöst** wie z.B (überlauf eines Zählers, empfang eines Zeichens über serielle Schnittstelle…)
- Unter Interrupt versteht man eine **kurzfristige Unterbrechung** eines Programmflusses um einen anderen Prozess auszuführen. Anschliessend wird die Ausführung des Programms an der Unterbrechungsstelle fortgesetzt.
- Allgemein kann man sagen, dass Interrupts nötig sind, **um auf unvorhergesehene interne und externe Ereignisse reagieren** zu können.



7.2 die Interrupt Typen

- Es gibt 2 Interrupt-Typen:

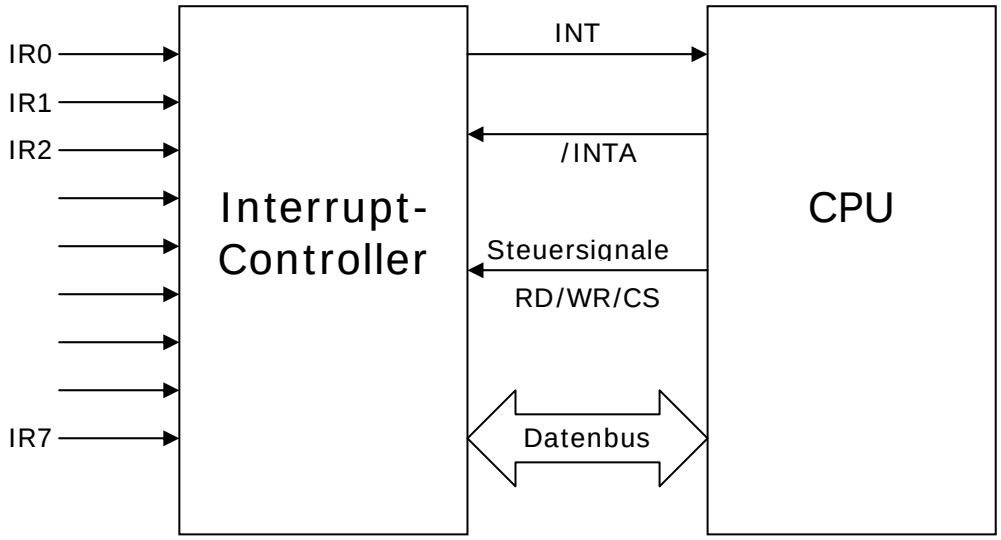
IRQ (Interrupt with Request)

- Sind **maskierbar** → **können** vom Programm **gesperrt / freigegeben werden**
- Ein **gesperrter Interrupt** verursacht **kein Programm unterbruch**, auch wenn eigentlich der Interrupt ausgelöst hätte werden sollen

NMI (Not Maskable Interrupt)

- Sind nicht **maskierbar!**
- Werden **immer** einen Programm **Unterbruch verursachen**, wenn sein assoziierte Indikator aktiv wird und dieser Interrupt die höchste Priorität unter den aktiven Interrupts hat.

7.3 Ablauf einer Programmunterbrechung



1. Wenn neu angefordertes Interrupt höchste Priorität hat → wird über INT eine Programmunterbrechung angefordert.

2. Ist **Interrupt-System** der CPU **aktiv** → der **aktuell** in Bearbeitung befindlicher **Befehl** wird **zu Ende geführt**

3. **über /INTA** wird die **Bestätigung** der **Annahme des Interrupts** ausgegeben.

4. der **Interruptcontroller überträgt** via den Datenbus die **Nummer des Interrupts oder die Startadresse** des zugehörigen **Interruptbedienprogramms**

5. Die **CPU unterbricht** das augenblicklich ausgeführte **Programm** und **speichert** die später noch benötigten **Informationen auf den Stack**.

6. **Interrupt Service Routine (ISR)** wird abgearbeitet

7. **vor** der **Beendigung** des **ISR** wird dem Interruptcontroller über einen **speziellen Befehl mitgeteilt**, dass die Bearbeitung des angeforderten **Interrupts abgeschlossen** ist.

8. Der **aktuelle Interrupt wird gelöscht**.

9. **Wenn** es **weitere** gleich oder niedriger priorisierte **Interrupts hat**, werden **diese nun ausgeführt.**

10. **Sobald** es **keinen weiteren Interrupt** mehr hat, wird **zum** unterbrochenen **Programm zurückgekehrt und fortgesetzt.**

7.4 Interrupt Service Routine (ISR)

- Spezielle Sorte von Unterprogrammen
- **Programm das bei einem Interrupt ausgeführt wird.**
- Enthalten Code um Operationen im Auftrag der CPU auszuführen
- Werden aufgerufen beim auftretten eines Interrupts, kein CALL
- Keine Aufrufkonvention – ISR muss alle Register absichern

7.5 Ansteuerung ISR

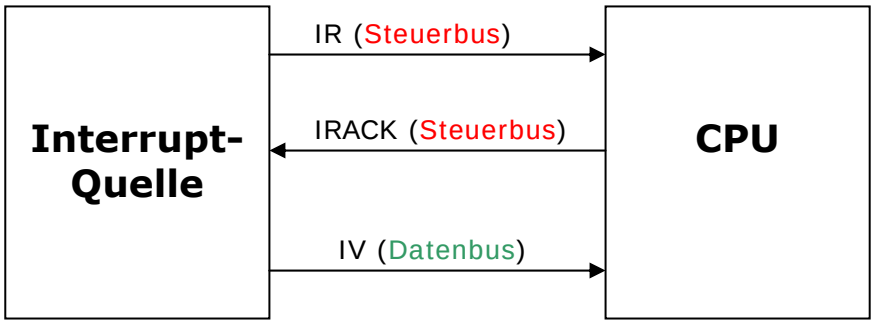
Es gibt **zwei möglichkeiten** diese ISR anzusprechen:

Leitergebunden

- CPU hat **für jede** mögliche **Interrupt-Quelle** einen **entsprechenden Anschluss**
- **Jeder** dieser **Anschlüsse** hat eine **spezifische Adresse** im Programmspeicher
- Ist der entsprechende - nicht maskierte - Interrupt erkannt, wird zu dieser Adresse gesprungen.
- **Zwischen** den **hartgebundenen Adressen** gibt es auch **hartgebundenen Speicherplatz** → reicht gewöhnlich nicht aus um den ganzen code für die ISR zu speichern
- Daher ist der **erste Befehl der ISR** gewöhnlich ein **unbedingter Sprung zum effektiven Anfang der Routine**
- **Vorteil:** kurze Reaktionszeiten
- **Nachteile:**
    - Es können **nur so viele Interrupts** verarbeitet werden, **wie Leiter** existieren
    - Der Programmierer ist selber verantwortlich die richtige Sprungadresse als ersten Befehl im Programm an die richtigen „hartgebundenen“ Adressen zu schreiben.

Vektorisiert

- CPU hat **nur einen Anschluss für alle Interrupt-Quellen**
- Dialog zwischen CPU und Interrupt-Quelle über dem **Steuerbus und** dem **Datenbus** ist notwendig damit der Prozessor die Interrupt-Quelle identifizieren und der Ablauf dann zur entsprechenden ISR springen kann:



7.6 Prioritäten unter Interrupts

- Die Interrupts können mit **verschiedenen Prioritäten** verseht werden.
- Wir **während der Abarbeitung** eines **Interrupt**s mit **niedriger Priorität**, ein **Interrupt mit höherer Priorität ausgelöst**, so wird die Behandlung des Interrupts mit der **niedrigen Priorität unterbrochen** und der mit der **höheren ausgeführt**. Anschliessend wird die Behandlung des niedrigeren Interrupts fortgesetzt.
- In **einigen Prozessoren** sind die **Prioritäten Hardware gebunden**
- In den **meisten** sind sie jedoch **Programmierbar**!
- Oft sind sie jedoch **nicht beliebig individuel**l, da die **Anzahl der möglichen Prioritäten < anzahl Interrupts**

7.7 Hardware vers. Software Interrupt

Hardware:

- **Asynchron**
- Werden von einem **Hardwareelement ausgelöst**
- Sind schlecht berechenbar

Software:

- **Synchron**
- Ein Software-Interrupt wird über den **INT-Befehl** ausgeführt, auf den die Nummer des Interrupts folgen muss.
- Streng genommen **Software Interrupt ≠ richtiger Interrupt**, da er vom Programmierer explizit gesetzt wird.
- INT-Befehl: Elegante Methode für Routinen aufzurufen die sich irgendwo im Speicher befinden

7.8 Interrupts der 8051 Familie

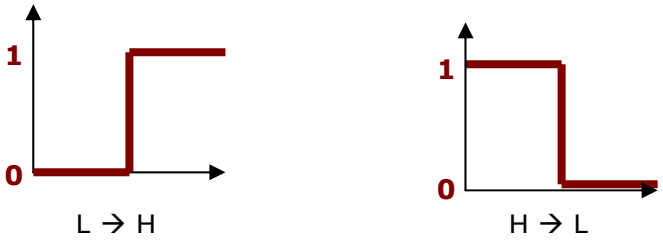
- Alle **14 Interrupts** sind **Leitergebunden und Maskierbar**
- **Prioritäten** können durch die Software **nur für Gruppen** von Interrupts **festgelegt** werden. (Priorität **0 – 3**)
- Wenn 2 oder **mehrere Interrupts** die **gleiche Priorität** haben, dann wird **an Hand** der **relativen Hardware** Gebundenen Positionen der Interrupts **entschieden**, welcher bedient wird.

Zustandgesteuerte Interrupts:

- **INT0** und **INT1** sind zustandgesteuert
- Wurde einer dieser beiden Interrupts aufgerufen und ist ihr **Indikator auch nach der Abarbeitung der ISR noch aktiv**, so wird er gleich **wieder aufgerufen**
- Wenn der **Indikator** eines zustandsgesteuerten Interrupts **inaktiv** wird, **bevor** die **ISR aufgerufen** wurde → **Interrupt** wird **nicht bedient**

Flankengesteuert Interrupts:

- Ihre IR-FlipFlops werden durch eine **L → H** oder **H → L** Flanke gesetzt.



- Die **meisten** werden **automatisch** durch die Hardware (am Anfang der ISR) wieder **gelöscht**
- **Einige** müssen **durch** die **Software** als Teil der ISR wieder **gelöscht werden**.

Management

- Um Unkonsistenzen bei der Steuerung der Interrups zu vermeiden, wird im nächsten Maschinenzyklus nur zu einer „aktiven" ISR gesprungen, wenn der aktuelle Programmbefehl

  1. kein Rücksprung von einer ISR ist
  2. kein Schreibbefehl in die Interrupt-Steuerregister IEN0, IEN1, IEN2, IP0 und IP1 geschrieben wurde

  Treffen weder 1 noch 2 zu, kann der Sprung um einen Maschinenzyklus verzögert werden.

- Die **Erkennung** eines **Interrupts** braucht:
  - **1** Maschinenzyklus für **zustandgesteuerte** Interrupts
  - **2** Maschinenzyklen für ein **flankengesteuertes** Interrupt
- **Der Sprung zum ersten Befehl der ISR braucht 2 Maschinenzyklen → minimale Reaktionszeit = 3 Maschinenzyklen**
- Ist der aktuelle Befehl eine Multiplikation oder Division → der Sprung muss weitere 3 Maschinenzyklen warten
- Durchschnittlich rechnet man zwischen 3 – 9 Maschinenzyklen