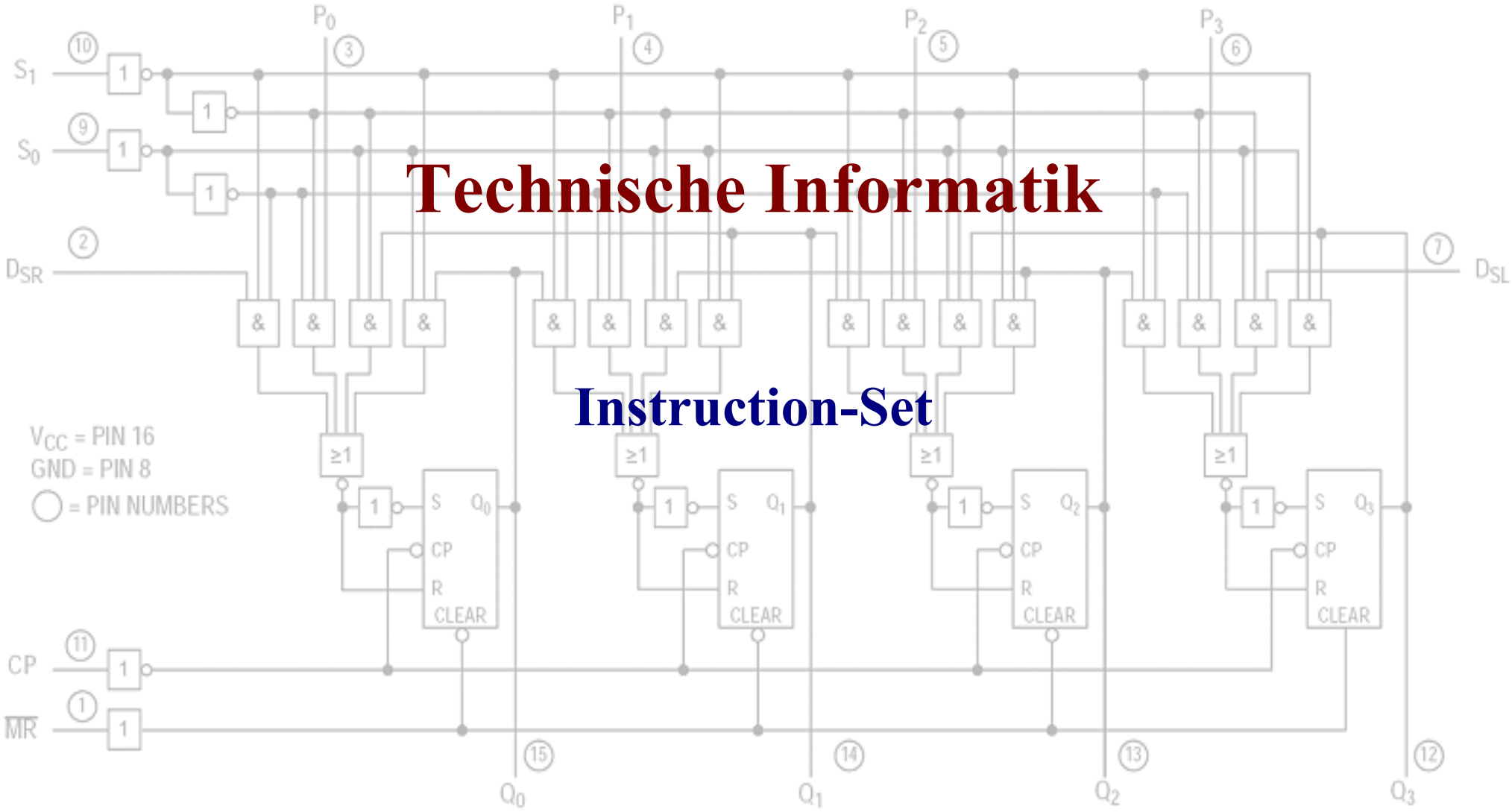
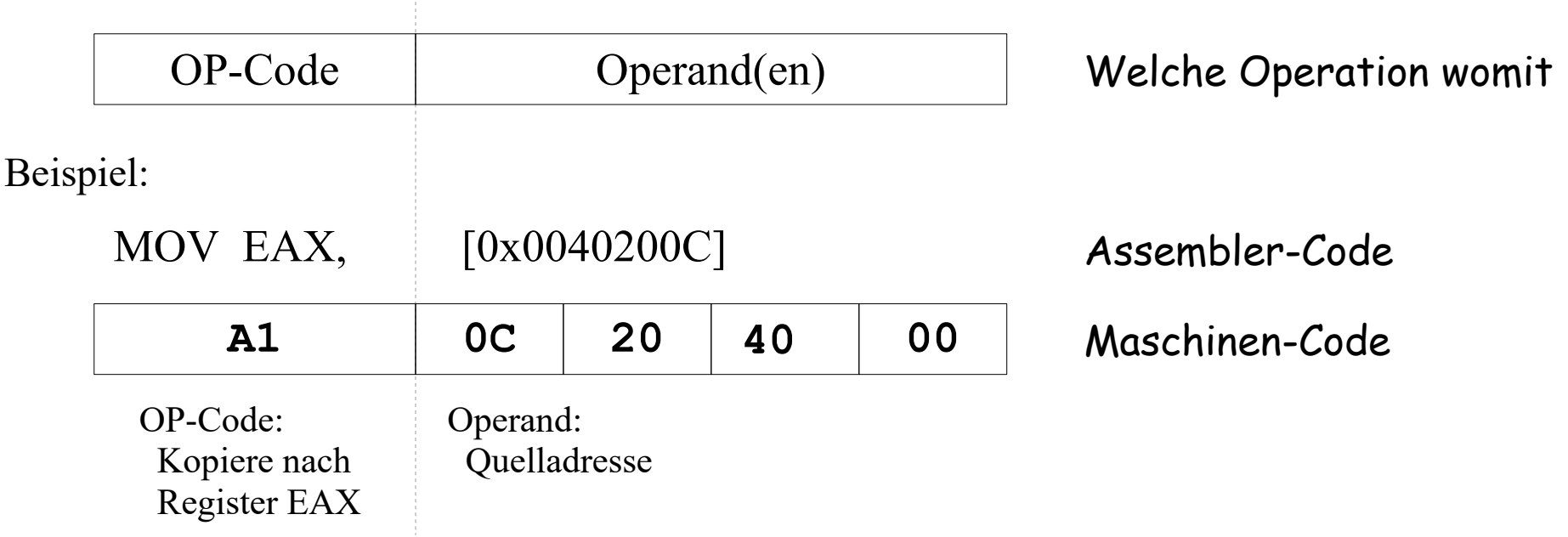
Technische Informatik

Instruction-Set



Maschinenbefehl

- Für alle Operationsarten gilt:
  - Maschinenbefehl benötigt mehrere Zyklen (Mikrocode)
  - Ablauf ist abhängig vom OP-Code und den Flags der Verarbeitungseinheit
  - In jedem Zyklus werden Steuerleitungen gesetzt
  - Innerhalb der Steuereinheit werden Maschinenbefehle durch Mikroprogramme abgearbeitet
  - Ein Maschinenbefehl kann (je nach Architektur) einen oder mehrere Operanden beinhalten



Makro- und Mikrocode

- Maschinenbefehl (Makrocode)
  - Befehle, aus denen sich ein Programm zusammensetzt
  - Assembler
    - „Lesbare“ Notation der Maschinenbefehle
    - Ein Assembler-Programm bildet die auf einer Maschine ausführbaren Befehle ab
    - Werkzeug, mit dem ein Assembler-Programm in ein Maschinenprogramm übersetzt wird

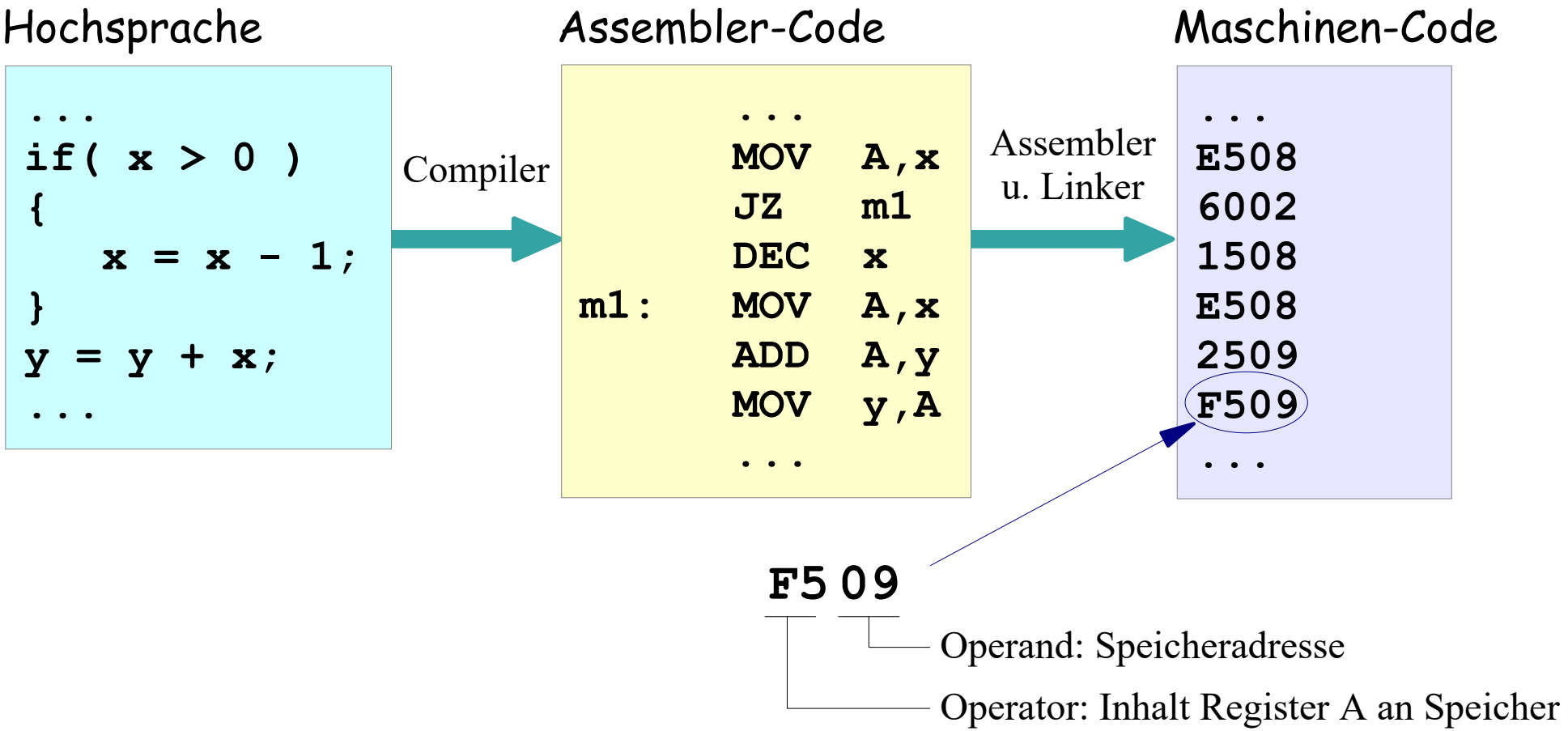
Auszug aus einem Assembler-Code:

```
push   acc
mov    b,#100
div    a,b
add    a,#30h
lcall  fktOut
inc    r0
```

- Mikrocode
  - Bestandteil des Steuerwerkes
  - Bildet einen Maschinenbefehl auf eine Folge von Steuersignalen ab
  - Realisierung durch EPROM (Festwertspeicher) oder fest verdrahtetes Schaltwerk
  - Wird bei einigen Architekturen auf den *Nanocode* heruntergebrochen

Maschinenbefehl

- Beispiel für die Umsetzung eines Hochsprachen-Codes in Maschinen-Code
  - 8-Bit-Controller (C51)
  - Ein-Adress-Befehle



Instruction Set Architecture (ISA)

- Instruction Set Architecture: Vollständiger Befehlssatz einer CPU
  - Arithmetische und logische Befehle
  - Datentransfer- und Kontrollfluss-Befehle
- Die ISA ist der software-relevante Aspekt der Rechnerarchitektur
  - Register
  - Adressierungsarten (direkt, indirekt)
  - Datentypen (8-, 16,-... Bit, Integer, Floating-Point)
  - Interrupt, Stack
  - Verarbeitungseinheit(en)
- SW-Entwicklungswerkzeuge (Compiler, Assembler, Linker etc.) müssen die ISA abbilden

Instruction Set Architecture (ISA)

- Transfer - Operationen *Register ↔ Register oder Speicher*
  - Datenaustausch zwischen Speicher und Register oder zwischen zwei Registern
  - **`MOV ziel, quelle`**
    - „Move“ - Kopiert Daten, *ziel* = *quelle*
  - **`PUSH quelle`**
    - Kopiert *quelle* auf den Stack
  - **`POP ziel`**
    - Holt Daten vom Stack und speichert diese in *ziel*
  - **`LDR, STR, ...`**

Instruction Set Architecture (ISA)

- Verknüpfungs – Operationen *Register/Register → Register*
  - Arithmetische oder logische Verknüpfung von Register- oder Speicherinhalten. Das Ergebnis wird in ein Register geschrieben, ggf. werden Zusatzinformationen über das Ergebnis in das Flag-Register eingetragen
  - **ADD *ziel, quelle***
    - Addiert *ziel* und *quelle*, das Ergebnis wird in ziel abgelegt: *ziel* = *ziel* + *quelle*
  - **SUB, MUL, DIV**
  - **AND *ziel, quelle***
    - Führt bitweise eine logische UND-Verknüfung von *ziel* und *quelle* durch, das Ergebnis wird in *ziel* abgelegt
  - **OR, XOR, NOT**
  - **SHL *ziel, operand***
    - Verschiebt das *ziel* um *operand* Stellen nach links: $ziel = ziel \cdot 2^{operand}$
  - **SHR, ROL, ROR**
  - **CMP *operand1, operand2***
    - Vergleicht *operand1* mit *operand2*. Das Ergebnis wird im Flag-Register eingetragen
  - *SUB, MUL, DIV, SHL, CMP, ...*

Instruction Set Architecture (ISA)

- Verzweigungs – Operationen *Register/Speicher → Befehlsadress-Register (PC)*
  - Nicht-sequenzielle Fortsetzung der Programmausführung (durch Eintrag in den Program Counter)
  - Bedingte Verzweigungen prüfen das Flag-Register
  - **`JMP`** ***`zieladresse`***
    - Programmausführung wird unbedingt ab der Adresse *zieladresse* fortgesetzt
  - **JZ** *zieladresse*
    - Programmausführung wird ab *zieladresse* fortgesetzt, falls Zero Flag gesetzt
  - **`JNZ, JE, JNE, ...`**
  - **`CALL`** ***`zieladresse`***
    - Um einen Rücksprung an die aufrufende Stelle zu ermöglichen, wird zunächst der Program Counter auf den Stack kopiert. Anschließend wird die Programmausführung ab *zieladresse* fortgesetzt
  - **`RET`**
    - Rücksprung, indem Program Counter vom Stack geholt wird
  - ***`JMP, JNE, CALL, RET, IRET,...`***

Instruction Set Architecture (ISA)

- Sonstige Operationen
  - **`NOP`**
    - Ein Dummy-Befehl, der keine Operation ausführt, wird z.B. als Wartezyklus verwendet
  - **`STI`**
    - Schaltet Interrupts frei
  - ***`CLI, STC, CLC, ...`***

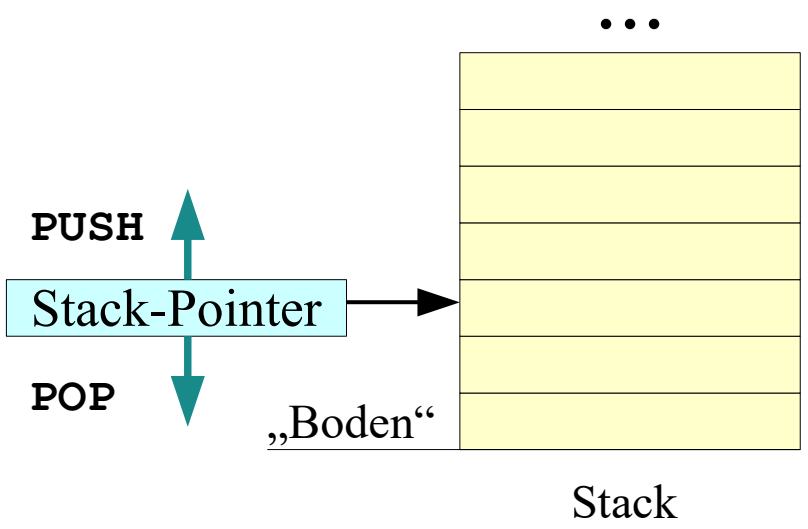
Adressierung

- Adressierungsarten
  - Für den Zugriff auf Operanden stehen verschieden Adressierungsarten zur Verfügung
  - Die ISA definiert, welche Adressierungsart in welcher Kombination verwendet werden kann

Modus	Speicherort	Operand	Beispiel	Erläuterung
immidiate	Konstante (Programm)	Der Operand ist eine Konstante (read only)	`MOV EAX,14`	Die Konstante '14' wird in das Register EAX kopiert
register	Register	Inhalt eines Registers	`MOV EAX,EBX`	Der Inhalt des Registers EBX wird in das Register EAX geschrieben
direct	Speicher	Inhalt einer Speicherstelle, die Adresse ist bereits zur Compilezeit bekannt	`MOV EAX,[100]`	Der Inhalt der Speicherstelle mit der Adresse 100 wird in EAX kopiert
indirect	Speicher	Inhalt einer Speicherstelle, die Adresse ist erst zur Laufzeit bekannt	`MOV EAX,[EBX]`	Das Register EBX enthält eine Adresse. Der Inhalt der Speicherstelle mit dieser Adresse wird in EAX kopiert
indexed	Speicher	Wie „indirect“, jedoch mit Berechnung eines Offsets	`MOV EAX,[EBX+4]`	Das Register EBX enthält eine Adresse, die um 4 inkrementiert wird. Der Inhalt der Speicherstelle mit der resultierenden Adresse wird in EXA kopiert
impilcit	Register Speicher	Operand bzw. Operandenadresse sind fest vorgegeben	`POP EAX`	Der Inhalt der Speicherstelle, die mit dem Stack-Pointer impilzit adressiert ist, wird in EAX kopiert

Stack (Stapelspeicher)

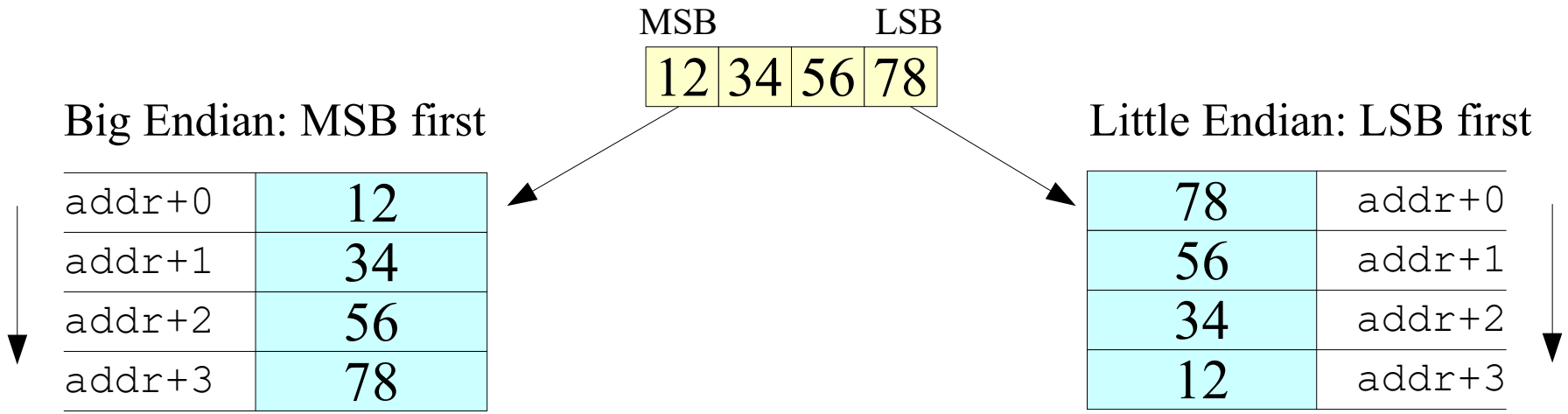
- Der **Stack** ist ein Hilfsmittel zur Verarbeitung von Unterprogrammen und Unterbrechungen (= *Interrupt*)
- Keine komplexe Adressierung, statt dessen:
  - LIFO-Prinzip (Last In First Out)
  - Zugriff über die Operationen **`PUSH`** (= Schreiben) und **`POP`** (= Lesen)
  - Aktuelle Schreib-/Leseposition wird im Stackpointer (Prozessorregister) verwaltet



- Verwendung:
  - Zwischenspeichern von Rücksprungadressen und Registerinhalten
- Realisierung:
  - Speicher im Prozessor (schneller Zugriff)
  - Teil des (externen) Hauptspeichers (Speichertiefe beliebig vergrößerbar)

Byte-Order

- Byte-Order
  - Gibt an, in welcher Reihenfolge die Bytes einer Ganzzahl im Speicher organisiert sind
  - Big Endian
    - Das höchstwertigste Byte wird zuerst (an kleinster Adresse) gespeichert
    - Beispiele: Motorola, PowerPC, MIPS, SPARC...
  - Little Endian
    - Das kleinstwertigste Byte wird zuerst (an kleinster Adresse) gespeichert
    - Beispiele: x86, AVR, ARM,...



Byte-Order

- Byte-Order
  - Die Byte-Order spielt bei Datenübertragungen zwischen unterschiedlichen Architekturen eine Rolle
  - Network Byte Order
    - Im Protokoll definierte Byte Order (meist Big Endian)
  - Host Byte Order
    - Byte Order des jeweiligen Systems
  - Konvertierung mit C-Funktionen `htonl()`, `ntohl()` etc.
    - Die jeweiligen Implementierungen sind natürlich von der Byte Order des Rechners abhängig

Rechner A

```
int x = 0x12345678;

transmit( htonl(x) );
```

Rechner B

```
int x;

x = ntohl( receive() );
```

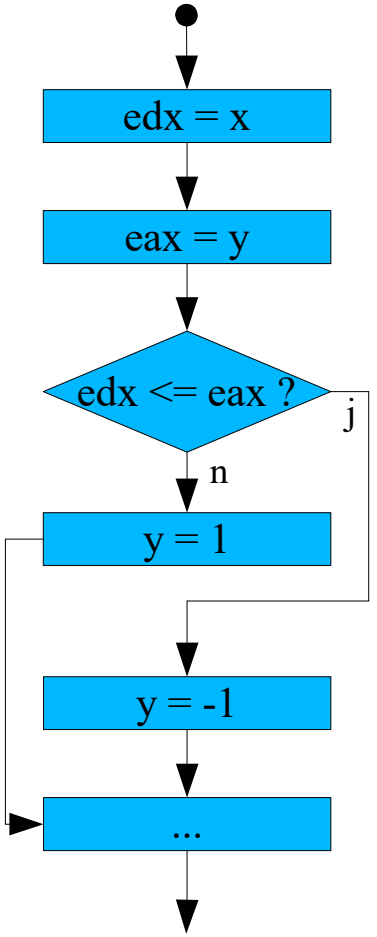
Programmfluss

- Verzweigung

C-Programm

```
int x,y;

main()
{
  if( x > y )
  {
    y = 1;
  }
  else
  {
    y = -1;
  }
}
```

```
                                  Speicheradresse 'x'
0x1BFB   mov   edx,[0x402018]
0x1C02   mov   eax,[0x40201c]
                                  Speicheradresse 'y'
0x1C08   cmp   eax,edx
0x1C0B   jle   0x1c18

0x1C0D   mov   [0x40201c],0x1
0x1C16   jmp   0x1c21

0x1C18   mov   [0x40201c],0xffff

0x1C21   ...
```



Programmfluss

- Schleife

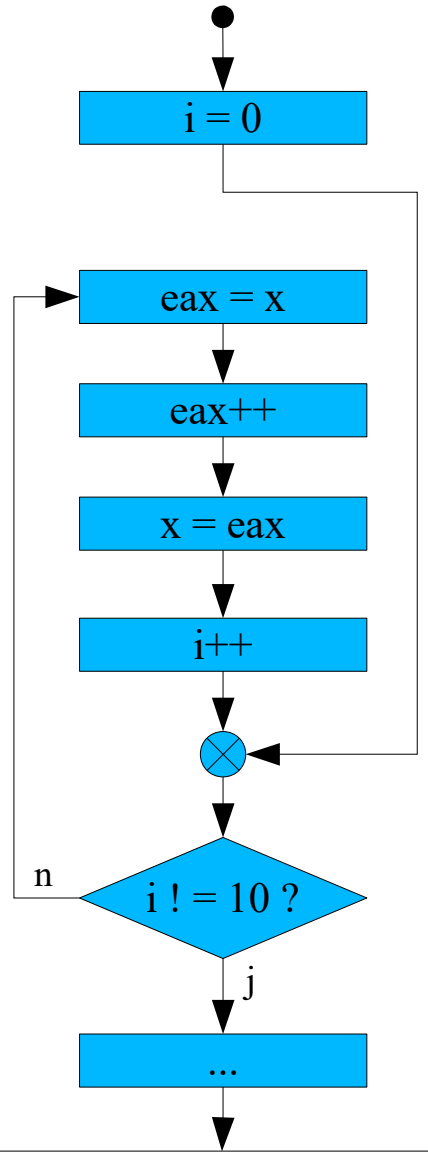
C-Programm

```
int x;

main()
{
  for(int i=0;i!=10;i++)
  {
    x++;
  }
}
```

'i' liegt auf dem Stack

```
0x1C5A   mov   [esp+28],0x0
0x1C62   jmp   0x1c75

0x1C64   mov   eax,[0x40201c]

0x1C6A   inc   eax

0x1C6B   mov   [0x40201c],eax

0x1C71   inc   [esp+28]



0x1C75   cmp   [esp+28],0xa
0x1C7F   jne   0x1c64

...
```

Speicheradresse 'x'



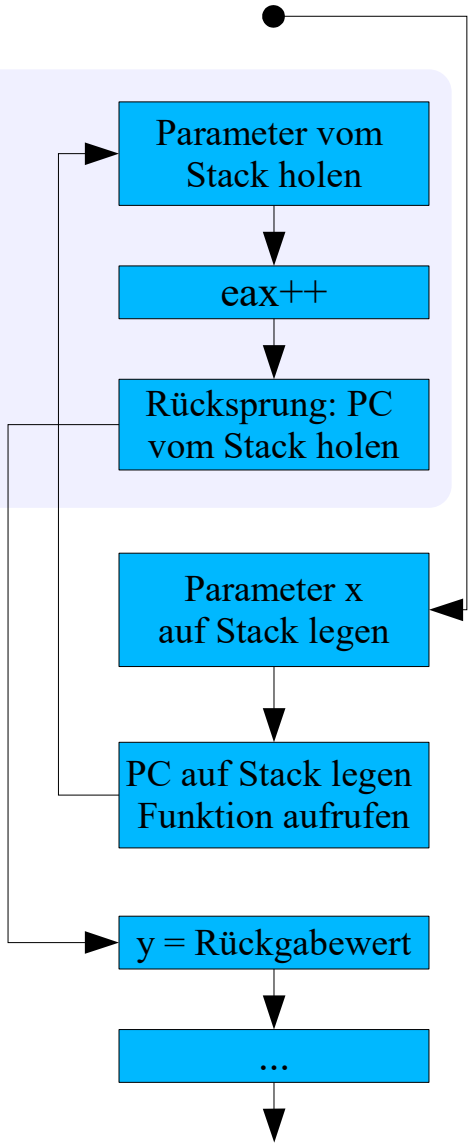
Programmfluss

- Funktionsaufruf

C-Programm

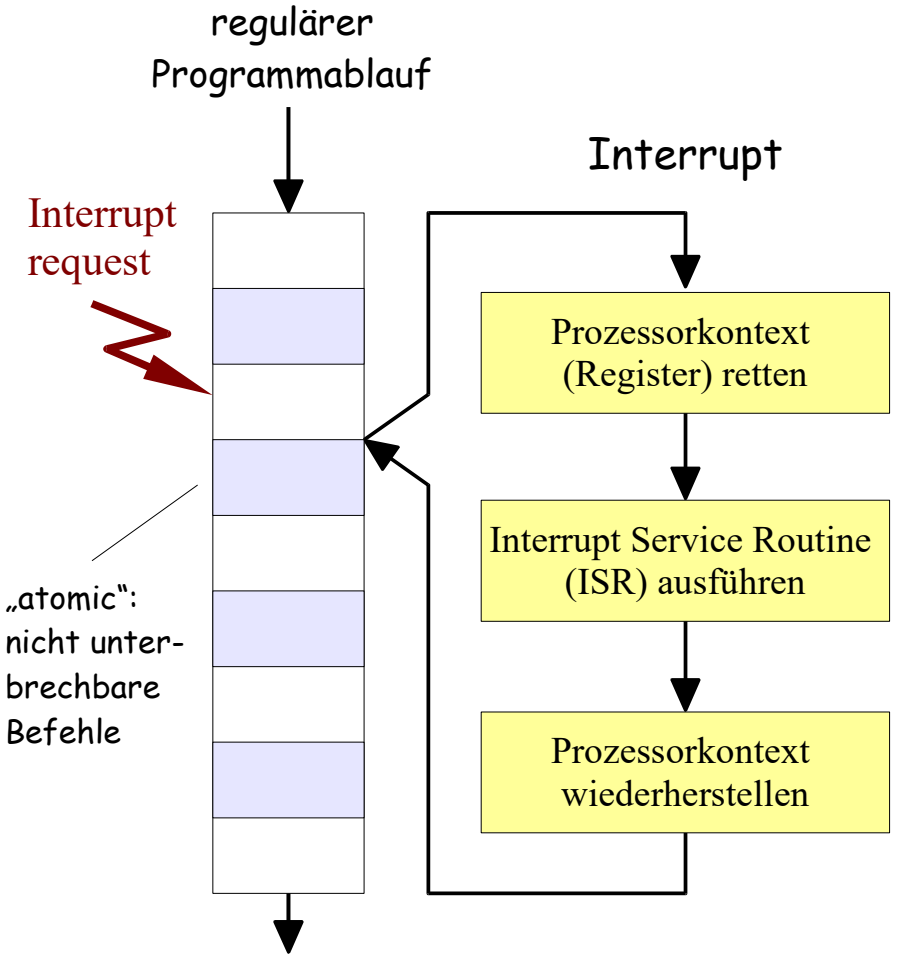
```
int function( int a )
{
  return( a+1 );
}

int x,y;

main()
{
  y = function( x );
}
```

```
; function
0x1BFD  mov   eax,[esp+4]

0x1C00  inc   eax

0x1C02  ret
; end of function

0x1C51  mov   eax,[0x402018]
0x1C58  mov   [esp],eax

0x1C5B  call  0x1BFD


0x1C60  mov   [0x40201c],eax

...
```



Interrupt

- **Interrupt**
  - Unterbrechung des regulären Programmablaufs durch Interrupt-Anforderung und Ausführung einer Interrupt-Service-Routine (ISR)
  - Der Aufruf der quellenspezifischen ISR erfolgt durch Setzten des PC (Befehlsadresse) auf die in der Interrupt-Vektortabelle abgelegten Adresse



Interrupt

- Interrupt-Quellen
  - Interne Ereignisse
    - Ausnahmebehandlung während des Programmablaufs,
      z.B. Division durch 0, "Traps", Supervisor call
  - Externe Ereignisse
    - Signale oder Daten von/zu externen Einheiten (meist über Interrupt Controller an den Prozessor-Interrupt-Eingang)
    - Alarmsignale (z.B. Hardwaredefekt, Timer, ...)
- Eigenschaften:
  - Priorisierung: Prioritätsskala für die verschiedenen Interrupts
    - Legt Ausführungsreihenfolge bei konkurrierenden Interrupt-Anforderungen fest
  - Maskierbarkeit: Sperrung einiger oder aller Interrupts
- Atomare Befehle sind nicht unterbrechbar