I. GRUNDLAGEN

Aufgaben der Hardware:

Ein- und Ausgabe von Daten
Verarbeiten von Daten
Speichern von Daten

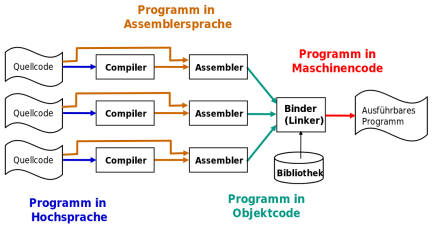
Klassische Hardwarekomponenten:

Ein- und Ausgabe
Hauptspeicher
Rechenwerk
Leitwerk

II. ANFORDERUNGEN HÖHERER PROGRAMMIERSPRACHEN

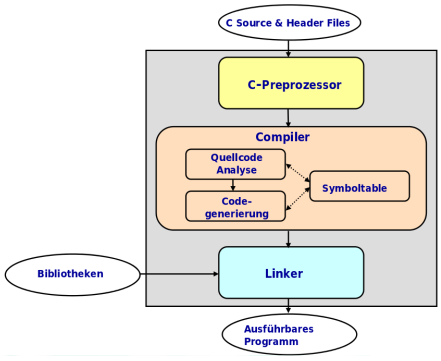
Begriffe:

Maschinensprache: Für Prozessor verständliche Anweisungsrepräsentation, z.B. 00101101001110101
Assemblersprache: Für Menschen verständliche Maaschinensprache, z.B. `add $s2, $s1, $s0`
Assembler: Übersetzt Assemblersprache eindeutig in Maschinensprache
Objektcode: Maschinenprogramm mit ungelösten externen Referenzen
Binder/Linker: Löst ungelöste Referenzen auf, verbindet alles zu einem ausführbaren Maschinenprogramm



Programmiersprache C:

Zwischenstellung zwischen Assembler und Hochsprache
hohe Portabilität trotz guter Architekturanpassung
einfache Programmierung
Datentypen: `char, int, float, double`
Kontrollstrukturen: Entscheidungen, Schleifen, Blöcke, Unterprogramme
Zeiger als Parameter möglich



C - Datentypen:

`char`: Ein Zeichen, meist 1 Byte
`int`: Integerzahl, 2 oder 4 Byte
`float`: Gleitkommazahl, meist 4 Byte
`double`: Gleitkommazahl, meist 8 Byte

C - Operatoren:

`*`: Multiplikation (`x*y`)
`/`: Division (`x/y`)
`%`: Modulo (`x%y`)
`+`: Addition (`x+y`)
`-`: Subtraktion (`x-y`)
`+` und `-` auch als Prä- und Postfix, alle auch als assign (`=` anhängen)

C - Bit-Operatoren:

`~`: Bitwise NOT (`~x`)
`<<`: links schieben (`x<<y`)
`>>`: rechts schieben (`x>>y`)
`&`: bitweise AND (`x&y`)
`^`: bitweise XOR (`x^y`)
`|`: bitweise OR (`xy|`)
alle auch als Assign (`=` anhängen)

C - Vergleichsoperatoren:

`>,<`: größer, kleiner als (`x>y, x<y`)
`>=,<=`: größergleich, kleinergleich als (`x>=y, x<=y`)
`==,!=`: gleich, ungleich (`x==y, x!=y`)

C - Spezialoperatoren:

Auswahloperator: `z = (a < b) ? a : b` (`z=a`, falls `a<b`, sonst `z=b<`)

C - Operatoren-Priorität

Operator Type	Operator	Associativity
Primary Expression Operators	`() [] . -> expr++ expr--`	left-to-right
Unary Operators	`* & + - ! ~ ++expr --expr (typecast) sizeof`	right-to-left
Binary Operators	`* / %`	left-to-right
	`+ -`	
	`>> <<`	
	`< > <= >=`	
	`== !=`	
	`&`	
	`^`	
	`|`	
	`&&`	
	`||`	
Ternary Operator	`?:`	right-to-left
Assignment Operators	`= += -= *= /= %= >>= <<= &= ^= |=`	right-to-left
Comma	`,`	left-to-right

C - Kontrollstrukturen

```
if (Bedigung) { Aktionen_if } else { Aktionen_else }
switch (var) { case a: ... break; ... default: ... break; }
while (Bedigung) { ... }
for (init; Bedingung; reinit) { ... }
do { ... } while (Bedingung)
```

C - Programmaufbau

1. Präprozessor-Anweisungen:
   (a) `#include <stdio.h>` (Bibliotheken einbinden)
   (b) `#include "modul.h"` (Module einbinden)
   (c) `#define COLOR blau` (Globale Textersetzung)
2. Globale Deklarationen/Definitionen:
   (a) `int i;` (Deklaration)
   (b) `int j = 13;` (Definition)
   (c) `int fakultaet (int n);` (Funktionsprototyp)
3. Funktionen/Programmstruktur

   `int fakultaet (int n) { ... }`
   jedes Programm enthält Funktion `void main(...) { ... }`
   Unterprogramm = Funktion
   Programmstart: `main` wird aufgerufen
   Rekursion ist zulässig

C - Parameterübergabe

1. Call by Value: Normalfall, Kopie des Parameters wird an Funktion übergeben, bei Änderung keine Auswirkung beim Aufrufer
2. Call by Reference: Mit Zeigern umsetzbar, selbe Speicheradresse wie Aufrufer

C - globale und lokale Variablen

Global: Sind gesamtem Programm bekannt (zu vermeiden)
Lokal: Nur in Block deklariert

C - Speicherklassen

`auto`: lokale Variablen
`register`: wird in CPU-Register gespeichert, nur für zeitkritische Variablen zu verwenden
`static`: statischer Speicherplatz
`extern`: globale Variable

C - Zeiger und Vektoren

Pointer: Enthält Adresse, die auf Daten verweist
`int* p` (`p` ist Zeiger auf `int`)
`a = 3; p = &a` (`p` enthält Adresse von `a`)
`int b = *p + 1` (=4)

```
int *p;
int *q;
int a = 3;
int b;

p = &a;
b = *p + 1;
q = (int*) 0x8010
```

	Adresse	Inhalt
p	...	0x8004
a	0x8004	3
b	...	4
q	...	0x8010
	0x8010	

III. ZAHLENDARSTELLUNG

Zahlensysteme – Stellenwertsystem

Darstellung einer Zahl durch Ziffern z_i – Stellenwert $i$te Position: $i$te Potenz der Basis b
Wert $X_b = \sum_{i=-m}^{n} z_i b^i$
Wichtige Zahlensysteme: Dual-, Oktal-, Dezimal-, Hexadezimalsystem

	Dual	Oktal	Dezimal	Sedezimal
0	0	0	0	0
1	1	1	1	1
2		2	2	2
3		3	3	3
4		4	4	4
5		5	5	5
6		6	6	6
7		7	7	7
8			8	8
9			9	9
10				A
11				B
12				C
13				D
14				E
15				F

Umwandlung von Dezimal zu Basis b

1. euklidischer Algorithmus:
   (a) Berechne p mit $b^p \leq Z < b^{p+1}$, setze $i = p$
   (b) Berechne $y_i = Z_i \text{ div } b^i$, $R_i = Z_i \bmod b^i$
   (c) Wiederhole (b)H für $i = p-1, \ldots,$ ersetze dabei Z durch R_i, bis $R_i = 0$ oder b^i klein genug ist

$$2^3 \leq 13 < 2^4$$
$$13 : 2^3 = 1 \text{ Rest } 5$$
$$5 : 2^2 = 1 \text{ Rest } 1$$
$$1 : 2^1 = 0 \text{ Rest } 1$$
$$1 : 2^0 = 1 \text{ Rest } 0$$
$$\rightsquigarrow Z = 13_{10} = 1101_2$$

2. Horner-Schema:
   (a) ganzzahliger Teil: 15741_{10} in Hexadezimal:

$$15741_{10} : 16 = 983 \text{ Rest } 13 \ (= D_{16})$$
$$983_{10} : 16 = 61 \text{ Rest } 7 \ (= 7_{16})$$
$$61_{10} : 16 = 3 \text{ Rest } 13 \ (= D_{16})$$
$$3_{10} : 16 = 0 \text{ Rest } 3 \ (= 3_{16})$$
$$\rightsquigarrow Z = 15741_{10} = 3D7D_{16}$$

   (b) Nachkommateil: $0,233_{10}$ in Hexadezimal:

$$0,233_{10} * 16 = \underline{3},728$$
$$0,728_{10} * 16 = \underline{11},648$$
$$0,648_{10} * 16 = \underline{10},368$$
$$0,368_{10} * 16 = \underline{5},888$$
$$\rightsquigarrow Z = 0,233_{10} \approx 0,3BA5_{16}$$

Umwandlung Basis b zu Dezimal

Einzelne Stellen nach Stellenwertgleichung addieren

$$101101,1101_2 = 2^{-4} + 2^{-2} + 2^{-1} + 2^0 + 2^2 + 2^3 + 2^5 = 45,8125_{10}$$

Umwandlung Basis b_1 zu Basis b_2

1. Umwandlung über Dezimalsystem
2. Ist eine Basis Potenz der anderen, so können mehrere Stellen zu einer Ziffer zusammengefasst werden

$$0110100,110101_2 = 0011\ 0100,1101\ 0100 = 34,D4_{16}$$

Darstellung negativer Zahlen

1. Betrag und Vorzeichen: Erstes Bit von Links ist Vorzeichen, Rest ist Betrag (`0001 0010 = 18, 1001 0010 = -18`)

   Vorteile: Symmetrischer Zahlenbereich
   Nachteile: Darstellungsänderung bei Bereichserweiterung, gesonderte Vorzeichenbehandlung bei Addition und Subtraktion, doppelte Darstellung der Null

2. Einerkomplement: Negative Zahl = NOT(positive Zahl)

   ```
   0000 = 0    1111 = -0
   0001 = 1    1110 = -1
   0010 = 2    1101 = -2
   0011 = 3    1100 = -3
              ...
   ```

   Vorteile: Symmetrischer Zahlenbereich, keine gesonderte Betrachtung des ersten Bits
   Nachteile: doppelte Darstellung der Null

3. Zweierkomplement: = Einerkomplement + 1

   ```
   0000 = 0    1111 = -1
   0001 = 1    1110 = -2
   0010 = 2    1101 = -3
   0011 = 3    1100 = -4
              ...
   ```

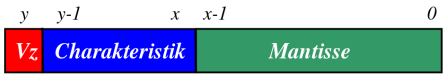
   Vorteile: Wie Einerkomplement, eindeutige Null
   Nachteile: Asymmetrischer Zahlenbereich (eine negative Zahl mehr)

4. Exzess-Darstellung: Verschiebung nach oben derart, dass kleinste negative Zahl die Darstellung `0...0` hat

Darstellung von Kommazahlen

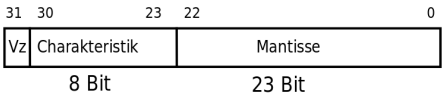
1. Festkommazahlen: Komma sitzt an einer festen Stelle
2. Gleitkommazahlen: $X = \pm\text{Mantisse} * b^{\text{Exponent}}$ (b fest)

   $X = (-1)^{\text{Vorzeichen}} * (0, \text{Mantisse}) * b^{\text{Exponent}}$
   $\text{Exponent} = \text{Charakteristik} - b^{(y-1)-x}$

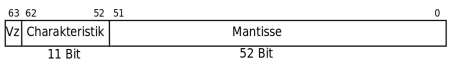


3. IEEE-Standard:

   (a) 32-Bit:



   (b) 64-Bit:



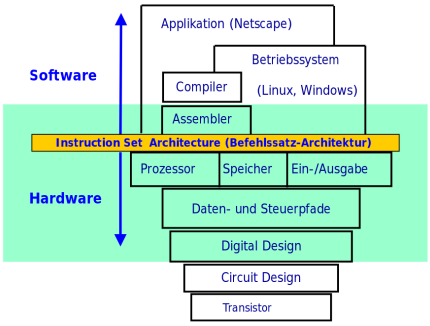
Codierungen

1. BCD: Dezimalzahl ziffernweise als Binärzahl (= *Tetrade*) codieren:

   ```
   8127 = 1000 0001 0010 0111
   ```

   Nachteil: Verbraucht viel Speicher, ungeschickt zum Rechnen
2. ASCII: 7-Bit-Codierung zur Textdarstellung
3. Unicode: Weltweit genormte Codierung aller Zeichen (wegen der vielen inkompatiblen ASCII-Derivaten)

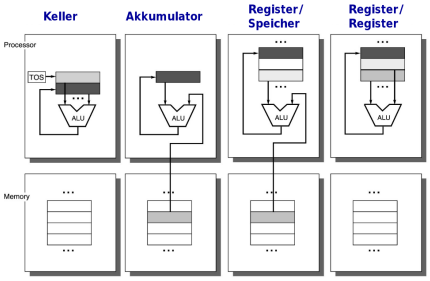
IV. BEFEHLSSATZARCHITEKTUR



ISA – Aufgaben

Wie werden Daten repräsentiert?
Wo werden Daten gespeichert?
Welche Operationen können auf den Daten ausgeführt werden?
Wie werden die Befehle codiert?
Wie wird auf die Daten zugegriffen?
⇝ abstrahiert Hardware für den Maschinenprogrammierer

Ausführungsmodelle



Ausführungsmodell – Register-Register

Alle Operanden und Ergebnis stehen in Allzweckregistern
Load/Store: Bestimmte Befehle holen Operanden aus Hauptspeicher, schreiben Inhalte von Registern in Speicher
Dreiadressformat

```
load R2,A   R2<-mem[A]
load R3,B   R3<-mem[B]
add R1,R2,R3   R1<-R2+R3
store C,R1   mem[C]<-R1
```

Vorteile: Einfaches und festes Befehlsformat, einfaches Code-Generierungsmodell, etwa gleiche Ausführungszeit der Befehle
Nachteile: Höhere Anzahl von Befehlen im Vergleich zu Architekturen mit Speicherreferenzen, längere Programme

Ausführungsmodell – Register-Speicher

Ein Operand im Speicher, ein Operand im Register, Ergebnis in Speicher oder Register
Explizite Adressierung mit/ohne Überdeckung
Zweiadressformat

```
add A,R1   mem[A]<-mem[A]+R1
add R1,A   R1<-R1+mem[A]
```

Vorteile: Zugriff auf Daten ohne vorherige Ladeoperationen, Befehlsformat-Kodierung ⇝ höhere Code-Dichte
Nachteile: Keine gleiche Operanden-Behandlung bei Überdeckungen, Taktzyklen pro Instruktion von Adressrechnung abhängig

Ausführungsmodell – Akkumulator-Register

- Akkumulator: Ausgezeichnetes Register, dient als Quelle eines Operanden und als Ziel für das Resultat (zweistellige Operationen)
- Implizite und überdeckte Adressierung
- Spezielle Befehle ermöglichen Operanden-Transport
- Einadressformat

```
 add A   acc<-acc+mem[A]
addx A   acc<-acc+mem[A+x]
add R1   acc<-acc+R1
```

Ausführungsmodell – Keller

- Operanden einer zweistelligen Operation stehen auf den obersten zwei Kellerelementen
- Ergebnis wird auf Keller abgelegt
- Implizite Adressierung über Kellerzeiger (`tos`)
- Überdeckung
- Nulladressformat

```
add  tos<-tos+next
```

Ausführungsmodelle – Übersicht

```
C=A+B; D=C-B
```

Register-Register	Register-Speicher	Akkumulator	Keller

load Reg1,A
load Reg2,B
Add Reg3,Reg1,Reg2
store C,Reg3
load Reg1,C
load Reg2,B
sub Reg3,Reg1,Reg2
store D,Reg3
	load Reg1,A
	add Reg1,B
	store C,Reg1
	load Reg1,C
	sub Reg1,B
	store D,Reg1
		load A
		add B
		store C
		load C
		sub B
		store D
			push B
			push A
			add
			pop C
			push B
			push C
			sub
			pop D