BTI7061 CSBas
Grundlagen der Informatik

1. Modulinformationen

Dozent: Ueli Schrag
Webseite des Kurses: http://www.benoist.ch/CSbasics/

1.1 Moodle

Kursunterlagen auf Moodle: https://moodle.bfh.ch/enrol/instances.php?id=15295
Access Key für Moodle: 1234

Bücher für den Kurs sind hier im PDF-Format zu finden.
Übungen für die jeweiligen Wochen sind auch hier zu finden.

2. Zahlensysteme

Verschiedene Zahlensysteme kann man anhand der Basis unterscheiden:

- Binär (Basis 2)
- Oktal (Basis 8)
- Dezimal (Basis 10)
- Hexadezimal (Basis 16)

Die Basis zeigt dabei an wie viele verschiedene Zustände ein Zahlensystem hat. Die Zustände werden dann auch jeweils mit verschiedenen Zeichen dargestellt. Die Basis zeigt somit auch an, wie viele Zeichen ein Zahlen-System benötigt. Die 0 als Zeichen stellt nichts dar und die 1 stellt die Einheit (Unit) dar.

Z. B. hat das Dezimal-System zehn Zustände und zwar die Zahlen von 0 - 9. Hingegen hat das Binär-System mit der Basis 2 nur zwei Zustände und zwar die Zahlen 0 und 1.

Eine Zahl im Dezimal-System wird folgendermassen dargestellt:

$$327.56_{10} = 3 * 10^2 + 2 * 10^1 + 7 * 10^0 + 5 * 10^{-1} + 6 * 10^{-2} \quad (1)$$

Nach dem selben Schema kann auch für andere Zahlensysteme vorgegangen werden. Hier ein Beispiel für ein Zahlensystem mit der Basis 3:

$$1022.102_3 = 1 * 3^3 + 0 * 3^2 + 2 * 3^1 + 2 * 3^0 + 1 * 3^{-1} + 0 * 3^{-2} + 2 * 3^{-3} = 33.407_{10} \quad (2)$$

Diese Schreibweise hat auch den Vorteil, dass man die Zahl einen beliebigen Zahlensystemes in eine Dezimalzahl umrechnen kann. Die obige Zahl 1022.102_3 wird somit, wenn man den Ausdruck ausrechnet zu 33.407_{10} im Dezimal-System.

Auf diese Weise kann so Zahlen in einem beliebigen Zahlensystem darstellen und man erhält jeweils direkt den Wert im Dezimal-System. Die Zahl 56.7_x mit der Basis x wird folgendermassen dargestellt.

$$56.7_x = 5 * x^1 + 6 * x^0 + 7 * x^{-1} \tag{3}$$

2.1 Andere Zahlensysteme

Neben dem oben erwähnten "Prinzip" gibt es auch noch andere Zahlensysteme. Diese sind aber in der heutien Zeit nicht mehr wichtig und relevant:

- Römische Zahlen

2.2 Oktal

Das Oktal-System ist das Zahlensystem mit der Basis 8. Also werden die Zeichen 0 - 7 verwendet. Nachfolgend ein Beispiel für eine oktale Zahl.

$$76225_8 = 7 * 8^4 + 6 * 8^3 + 2 * 8^2 + 2 * 8^1 + 5 * 8^0 = 31893_{10} \tag{4}$$

2.3 Hexadezimal

Das Hexadezimal-System ist das Zahlensystem mit der Basis 16. Es werden die Zeichen 0 -9 verwendet und zusätzlich auch noch die Zeichen A, B, C, D, E und F. Die Zeichen A - F stehen dabei für die Zahlen 10 - 15. Nachfolgend ein Beispiel für eine Zahl im Hexadezimal-System.

$$3C0A9_{16} = 3 * 16^4 + 12 * 16^3 + 0 * 16^2 + 10 * 16^1 + 9 * 10^0 = 245929_{10} \tag{5}$$

2.4 Binär

Das Binär-System ist das Zahlensystem mit der Basis 2. Dadurch werden nur die Zeichen 0 und 1 benötigt.

$$110_2 = 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 6_{10} \tag{6}$$

Vom Binär-System kann man einfach in ein anders Zahlensystem mit einer Basis von einer zweier Potenz (4, 8, 16) umrechnen. Dazu nimmt man je nach Basis "Pakete" von der Binär-Zahl und rechnet diese in die Basis um. Für die 4er-Basis nimmt man 2er-Pakete, für die 8er-Basis nimmt man 3er-Pakete und für die 16er-Basis nimmt man 4er-Pakete.

$$10101111.10101101_2 = 2233.2231_4 = 257.532_8 = AF.AD_{16} \tag{7}$$

3. Computer

3.1 Speicher

3.1.1 Stack

Ein "Stapel" im Speicher. Er funktioniert nach dem last-in-first-out Prinzip. Der Stackpointer (SP) zeigt auf die Adresse des Spitzes des Stacks.

Die zwei Funktionen, die man mit dem Stack benutzen kann sind PUSH und POP. Mit PUSH kann man einen Wert auf den Stack "legen" und mit POP entfernt man den obersten Wert des Stacks.

3.1.2 Protected Mode Flat Model

Das heute verbreite Model für den Speicher. Das Betriebssystem hat hier eine grössere Kontroller über den Speicher als früher. Das Betriebssystem stellt jedem Programm einen Speicherbereich zur Verfügung. Dadurch kann ein Programm nicht auf eine beliebige Speicheradresse zugreifen und unter Umständen das System zerstören.

3.2 CPU

Die CPU kann nur Befehle ausführen, die in ihrem Instruktionssatz beschrieben ist. Instruktionen sind dabei einfach eine Folge von Binären Daten mit denen die CPU umgehen kann. Diese Instruktionen nennt man auch OpCodes.

Die CPU führt jeweils den Befehl an der Adresse aus zu der der IP (Instruction Pointer) hinzeigt. Nach dem Ausführen des Befehls wird der Instruction Pointer an die nächste Adresse mit einem Befehl verschoben. Den IP kann man auch manuell anpassen um zu einer bestimmten Stelle im Programm zu springen.

3.3 Register

Register sind der Datenspeicher mit dem die CPU arbeitet. Der Zugriff zu Register ist sehr schnell.

Register haben keine Adresse sondern einen Namen (EAX, EDI, R9). Die Register können entweder 8, 16, 32 oder 64 Bit sein auf einem 64-Bit CPU. Ein paar Register haben eine spezielle Funktion, für die sie zuständig sind.

3.4 Betriebssystem

Das Betriebssystem ist für basis Funktionen wie Input und Output zuständig. Heutzutage sind sie auch für das Verwalten des Speichers und für die Multitask-Fähigkeit zuständig. Im Grunde ist das Betriebssystem auch nur ein Programm welches im Speicher hinterlegt ist.

3.5 Zahlendarstellung im Computer

3.5.1 Negative Zahlen

Man kann negative Zahlen auf mehrere Arten darstellen die beste Methode ist dabei das Zweierkomplement.

Dazu geht man folgendermassen vor um eine positive Zahl in die passende negative Zahl umzuwandeln:

1. Den Wert der positiven Zahl nehmen
2. Das Einerkomplement bilden (Invertieren, NOT-Operation)
3. Mit eins addieren

Beispiel:
Der Binärwert von 7 ist 00000111. Man invertiert diesen Wert um das Einerkomplement zu erhalten: 11111000. Schlussendlich addiert man noch eins zu diesem Wert um das Zweierkomplement zu erhalten: 11111001

Eine Addition kann man im Zweierkomplement ohne Problem durchführen. Für eine Substraktion führt man auch eine Addition aus, einfach mit dem Zweierkomplement des Subtrahenden. Das funktioniert weil $10 - 3 = 10 + (-3)$.

3.5.2 Kommazahlen

Nach dem IEEE 754 Standard werden die Kommazahlen binär kodiert. Das erste Bit ist dafür für das Vorzeichen zuständig. Die nächsten 11 Bits sind für den Exponenten und die letzten 52 Bits sind für die Mantisse zuständig. Je nach Standard können die Bit-Anzahl variieren aber im Prinzip funktioniert es immer gleich.

TODO: Finish explanation

Der Exponent ist hier dabei nicht im Zweierkomplement hinterlegt sondern in der sogenannten "Biased Form". Für diese Form muss man zuerst einen Biased Faktor berechnen. Dieser berechnet sich durch die Formel $2^{n-1} - 1 = 127$ wobei n die Anzahl Bits für den Exponenten ist. Im Falle, wenn der Exponent aus 8 Bits besteht ist der Biased Faktor $2^7 - 1 = 127$.

4. Assembler

Im nachfolgendem Kapitel wird die Programmierung mit Assembler behandelt. Für dieses Modul programmieren wir unter Linux mit dem NASM-Assembler.

4.1 Make

Make ist ein Build-System mit dem man für den Build-Prozess von Programmen Abhängigkeiten definieren kann. Diese Abhängigkeiten werden in einem sogenannten "makefile" definieren.

Folgendes ist ein Beispiel für ein makefile, welches aus einer "hello.asm" Datei ein lauffähiges Programm erstellt:

```
1  all: hello
2
3  clean:
4    rm -f *.o > /dev/null
5
6  hello: hello.o
7    ld -o hello hello.o
8
9  hello.o: hello.asm
10   nasm -f elf64 -g -F dwarf hello.asm
```

Innerhalb dieser Datei kann man nun "Targets" wie hello.o definieren. Nach dem Doppelpunkt gibt man dann die Abhängigkeiten dafür an. In diesem Fall ist es hello.asm. Wenn diese Abhängigkeit erfüllt ist, wird dann der angegebene Befehl ausgeführt. So wird dann das ganze makefile ausgeführt bis alle Targets erfüllt sind.

Im obigen makefile wird zusätzlich auch noch ein Befehl namens clean definiert. Dieser führt einfach direkt den angebenenen Befehl aus.

Das makefile führt man Schlussendlich mit ganz einfach mit "make" aus. Definierte Befehle kann mit mit z. B. "make clean" ausführen.

4.2 Aufbau eines Programmes

Ein Assembler Programm besteht aus verschiedenen Sektionen. In der ".data" Sektion kann man Daten definieren auf die man später im Code zugreifen kann. Die ".text" Sektion ist dann für den eigentlich Programmcode reserviert.

Der untenstehende Code gibt einfach "Hello, World!" gefolgt von einem Zeilenumbruch aus.

```
1  section .data
2    Message:        db "Hello, World!", 0xA
3    MessageLength:  equ $ - Message
4
5  section .text
6    global        _start
7
8  _start:
```

```
9     mov   rax, 4
10    mov   rbx, 1
11    mov   rcx, Message
12    mov   rdx, MessageLength
13    int   0x80
14
15  exit:
16    mov   rax, 1
17    mov   rbx, 0
18    int   0x80
```

4.3 Instruktionen

4.3.1 MOV

Mit der MOV-Instruktion kann man Werte in Register und Speicherorte verschieben.

Der untenstehende Code schiebt den Wert 10 in das Register rax.

```
1   mov   rax, 10
```

4.3.2 INC

Die INC-Instruktion inkrementiert das angegebene Register um 1.

```
1   mov   rax, 5
2   inc   rax      ; rax ist nun 6
```

4.3.3 DEC

Die DEC-Instruktion dekrementiert das angegebene Register um 1.

```
1   mov   rax, 5
2   dec   rax      ; rax ist nun 4
```

4.3.4 ADD

Die ADD-Instruktion addiert den Operanden zu angegebenen Register.

```
1   mov   rax, 5
2   add   rax, 7   ; rax ist nun 12
```

4.3.5 SUB

Die SUB-Instruktion subtrahiert den Operanden vom angegebenen Register.

```
1   mov   rax, 5
2   sub   rax, 3   ; rax ist nun 2
```

5. Terminologie

Paragraph
Eine Gruppe von 16 Bytes.

Nibble
Eine Gruppe von 4 Bit.

Byte
Eine Gruppe von 8 Bit.

Word
Eine Gruppe von 16 Bit oder 2 Bytes.

Double word
Eine Gruppe von 4 Bytes.

Quad word
Eine Gruppe von 8 Bytes.

6. Übungen

6.1 Basen umrechnen

a.) $AB2.EF_{16} \rightarrow X_{10}$ $\approx 2738.933_{10}$

b.) $AB.E1_{16} \rightarrow X_8$ $= 1253.702_8$

c.) $AB.E1_{16} \rightarrow X_4$ $= 2223.2301_4$

d.) $253.51_{10} \rightarrow X_{16}$ $\approx FD.828F_{16}$

e.) $233.81_{10} \rightarrow X_8$ $\approx 351.63_8$

f.) $123.51_{16} \rightarrow X_2$ $= 000100100011.01010001_2$

g.) $237.715_8 \rightarrow X_{10}$ $\approx 159.9004_{10}$

h.) $291.85_{10} \rightarrow X_8$ $\approx 442.663_8$

i.) $1010111.1101_2 \rightarrow X_{10}$ $= 87.8125_{10}$

j.) $18.7_{10} \rightarrow X_2$ $\approx 10010.1011001_2$

k.) $4321_5 \rightarrow X_{10}$ $= 1186_{10}$

l.) $AB5_{16} \rightarrow X_8$ $= 5265_8$

m.) $1011001_2 \rightarrow X_8$ $= 131_8$

n.) $431_5 \rightarrow X_8$ $= 164_8$

o.) $378_{10} \rightarrow X_2$ $= 101111010_2$

p.) $5732_8 \rightarrow X_2$ $= 101111011010_2$

q.) $AB3_{16} \rightarrow X_2$ $= 101010110011_2$

r.) $432_5 \rightarrow X_2$ $= 1110101_2§§§$

6.2 Oktales Rechnen

a.) $237_8 + 531_8$ $= 770_8$

b.) $371_8 * 7_8$ $= 3317$

c.) $2667_8 : 7_8$ $= 321$

d.) $256_8 - 71_8$ $= 165_8$

6.3 Binäres Rechnen

a.) $1010 + 1011 + 111$ $= 11100$

b.) $110111 - 10110$ $= 10110$

c.) $1101 * 1110$ $= 10110110$

d.) $10100011011 : 1011$ ≈ 1110110

6.4 Arithmetic in Hexadecimal II

6.4.1 Multiplication

- $A123_{16} * 50_{16} = 325AF0_{16}$
- $1E3E4E_{16} * EEE_{16} = 1C3863084_{16}$
- $FFF_{16} * 3_{16} = 2FFD_{16}$
- $C123C_{16} * CCC_{16} = 9A7957D0_{16}$

6.4.2 Logical Operations

- $10011110_2 \wedge 00111001_2 = 00011000_2$
- $01111101_2 \wedge 11110000_2 = 01110000_2$
- $11001001_2 \wedge 11110010_2 = 11000000_2$
- $11111001_2 \wedge 10110100_2 = 10110000_2$
- $00001000_2 \wedge 11011000_2 = 00001000_2$
- $10011110_2 \vee 00111001_2 = 10111111_2$
- $01111101_2 \vee 11110000_2 = 11111101_2$
- $11001001_2 \vee 11110010_2 = 11111011_2$
- $11111001_2 \vee 10110100_2 = 11111101_2$
- $00001000_2 \vee 11011000_2 = 11011000_2$
- $10011110_2 \oplus 00111001_2 = 10100111_2$
- $01111101_2 \oplus 11110000_2 = 10001101_2$
- $11001001_2 \oplus 11110010_2 = 00111011_2$
- $11111001_2 \oplus 10110100_2 = 01001101_2$
- $00001000_2 \oplus 11011000_2 = 11010000_2$
- $\neg 00001010_2 = 11110101_2$
- $\neg 10101110_2 = 01010001_2$
- $\neg 00011110_2 = 11100001_2$
- $\neg 11110000_2 = 00001111_2$

6.5 Exercise 4.1.1: Little Endian

Compute in decimal numbers the following 32-bit integers in little endian convention.

- $00000100_{16} = 00010000_{16} = 65536_{10}$
- $000000F0_{16} = F0000000_{16} = 4026531840_{10}$

- 32050000_{16} $= 00000532_{16} = 1330_{10}$
- $000FF0F0_{16}$ $= F0F00F00_{16} = 4042264320_{10}$
- $A0C400DD_{16}$ $= DD00C4A0_{16} = 218154144_{10}$

Write the value in memory of the following numbers in little endian on a 32-bit computer.

- 35 $= 23000000_{16}$
- 90 $= 5A000000_{16}$
- 150 $= 96000000_{16}$
- 1003 $= EB030000_{16}$

6.6 Exercise 4.1.2: Signed integers

Using the two's complement notation write the representation of the following numbers of two bytes. Write the result in hexadecimal.

- 100 $= 64_{16}$
- 67 $= 43_{16}$
- -10 $= F6_{16}$
- -5 $= FB_{16}$
- -67 $= BD_{16}$
- -130 $= F7E_{16}$
- -89 $= A7_{16}$
- -255 $= F01_{16}$

Execute the following additions in hexadecimal (using two's complement notation):

- $100 - 67$ $= 21_{16}$
- $67 - 5$ $= 3E_{16}$
- $-67 - 5$ $= B7_{16}$

6.7 Exercise 4.1.3: Unsigned Integers with Bias

Compute the value of the following unsigned integers with the bias 127 (e.g -10 is represented by the number 117 and 20 is represented by the number 147). Write the number in hexadecimal form.

1. 0 $= 127 = 7F_{16}$
2. 10 $= 137 = 89_{16}$
3. 120 $= 247 = F7_{16}$
4. -20 $= 107 = 6B_{16}$
5. -15 $= 112 = 70_{16}$

6. -109 $= 18 = 12_{16}$
7. -76 $= 51 = 33_{16}$
8. -38 $= 89 = 59_{16}$

6.8 Exercise 4.1.4: Floating Point Numbers

Compute the representation of the following numbers on 32 bits (write it in bits, then in hexadecimal notation):

- 0.5 $= 3F000000_{16}$
- 12 $= 41400000_{16}$
- 34.25 $= 42090000_{16}$
- 0.1 $= 3DCCCCCC_{16}$
- 10.98 $= 412FAE14_{16}$