Labor 1 - Theoretische Unterstützung

Umrechnung und Operationen in verschiedenen Nummerierung Basen

Es wird untersucht:

- die Umrechnung von Ganzzahlen und Dezimalzahlen von der Basis 10 in eine Basis, insbesondere die Basis 16 und 2
- die umgekehrte Umrechnung von verschiedene Basen zu Base 10, insbesondere von Base 16 und 2 zu Base 10.
- Die Umrechnung von Basis 16 direkt zu Basis 2 und umgekehrt.

Theoretische Überlegungen

Ein Nummerierungssystem besteht aus allen Regeln der Darstellung von Zahlen mit Hilfe bestimmter Symbole, die als Zahlen bezeichnet werden. Für jedes Nummerierungssystem ist die Anzahl der unterschiedlichen Vorzeichen für die Systemnummern gleich der Basis (b).

Für die Basis b = 2 (binär geschriebene Zahlen) sind die Vorzeichen also die Ziffern 0 und 1.

Für die Basis b = 16 (hexadezimal) lauten die Vorzeichen 0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F. Beachten Sie, dass für Zahlen, die in einer Basis geschrieben sind, die größer als die Basis 10 (Dezimalzahl) ist, zusätzlich zu den üblichen Zahlen in der Basis 10 andere Symbole (Buchstaben) verwendet werden.

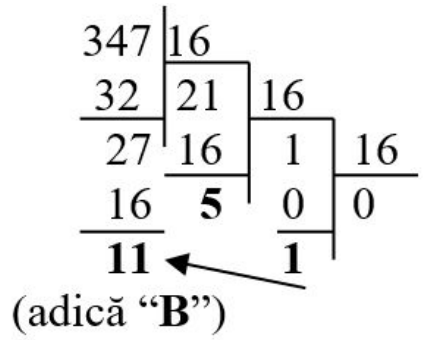
Somit haben im Fall von hexadezimal geschriebenen Zahlen die Buchstaben A, B, C, D, E, F als zugehörige Werte 10, 11, 12, 13, 14, 15. Notation Modi: Schreiben am Ende der Zahl in Klammern der Basis, zum Beispiel: $100101001_{(2)}$ oder $17A6B_{(16)}$.

Die Umrechnung der Zahlen von der Basis 10 in eine andere Basis

Der einfachste Algorithmus besteht darin, die in die Basis 10 geschriebene Zahl nacheinander durch die Basis zu teilen, zu der die Umwandlung gewünscht wird (die Zahl durch die Basis teilen und dann die erhaltene Zahl durch die Basis teilen und so weiter, bis die Zahl 0 wird). Dann werden die in umgekehrter Reihenfolge erhaltenen Reste genommen, bei denen es sich um den Wert der Zahl in der gewünschten Basis handelt.

Beispiele:

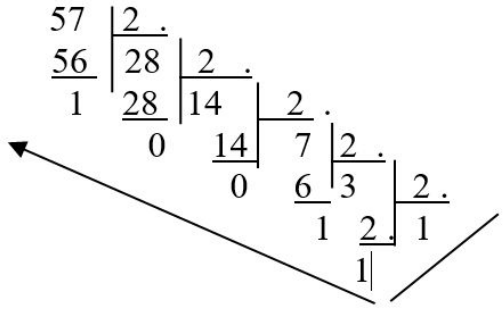
1. Was ist die Umrechnung von 347 von Basis 10 zu Basis $16_{(H)}$. Die Umrechnung sollte zuerst zur Basis 16 gemacht werden, da dies durch weniger Divisionen als die Konvertierung zu Basis 2 erfolgt.



Nehmen wir also der Rest in umgekehrter Reihenfolge, so erhalten wir $15B_{(H)}$.

$347_{(D)}=15B_{(H)}$

2. Was ist die Umrechnung von 57 von Basis 10 zu Basis $2_{(B)}$.



$57_{(D)} = 111001_{(B)}$

Es gibt auch eine schnellere Methode für die Umrechnung von Zahlen zwischen den Basen 2 und 16, wobei berücksichtigt wird, dass für jede Hex Zahl vier entsprechende Binärziffern:

Der Dezimalwert	**Der Wert in Hexadezimal**	**Die Binärzahl, die der Hex-Zahl entspricht**
0	0	0000
1	1	0001

2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Bei der Weitergabe einer Zahl durch die Basen 2 und 16 muss auch berücksichtigt werden, dass die Gruppierung der Zahlen aus der Basis 2 von rechts nach links erfolgt (indem links von der Zahl Nullen eingefügt werden, der ihren Wert nicht beeinflusst).

Beispiel:

1. Was ist die Umrechnung von 347 von Basis 10 zu Basis $2_{(B)}$ und $16_{(H)}$.

   $347_{(D)} = 15B_{(H)} = 0001\ 0101\ 1011_{(B)}$

Andere Beispiele:

$2 = 2_{(10)} = 10_{(2)}$
$62_{(10)} = 111110_{(2)}$

$1995_{(10)} = 11111001011_{(2)}$
$1024_{(10)} = 10000000000_{(2)}$

Die Umrechnung einer Zahl von einer Basis zur Basis 10

Um eine Zahl von einer Basis zur Basis 10 umzuwandeln, kann die im ersten Teil dieses Dokument definierte Formel verwendet werden:

$$\mathbf{Nr(b) = C_n\, C_{n-1}\, C_{n-2} \ldots C_2 C_1\, C_0}$$

dann ist sein Wert unter Basis 10:

$$\mathbf{Nr(10) = C_n * b^n + C_{n-1} * b^{n-1} + \ldots + C_2 * b^2 + C_1 * b^1 + C_0}$$

Beispiele:

1. Wir haben die Ganzzahl in Hexadezimal $3A8_{(H)}$ und fragen nach ihrem Wert in Dezimal:

$$N = 3*16^2 + 10*16^1 + 8 = 3*256 + 160 + 8 = 936_{(10}$$

2. Wir haben die Ganzzahl in Hexadezimal $86C_{(H)}$ und fragen nach ihrem Wert in Dezimal:

$$86C_{(16)} = 8 * 16^2 + 6 * 16 + 12 = 2156_{(10)}$$

3. Wir haben die Ganzzahl in Binär $1101101_{(H)}$ und fragen nach ihrem Wert in Dezimal:

$$1101101_{(2)} = 1*2^6+1*2^5+0*2^4+1*2^3+1*2^2+0*2+1=109_{(10)}$$

Andere Beispiele:

$$1010011_{(2)} = 83_{(10)}$$
$$11100011_{(2)} = 227_{(10)}$$
$$1000000000_{(2)} = 512_{(10)}$$
$$11001_{(2)} = 25_{(10)}$$

Das Bit. Das Vorzeichenbit. Der Komplementär Code. Darstellungsregel für ganze Zahlen mit Vorzeichen

Das Bit

- Durch ihre Besonderheiten ist die binäre Arithmetik besser in der Automatisierung als die Arithmetik in jeder anderen Numerierungsbasis. Dies ist der Grund, warum die

Arithmetik in der Basis 2 in Computern verwendet wird, und wir werden den Begriff Bit alternativ als Synonym für Binärziffern verwenden.

- **Das Bit = elementare Informationseinheit**
- In einem Bit können Sie Informationen darstellen, die nur zwei mögliche Werte haben können: 0 und 1.
- Je nach Kontext kann ein Bit 0 oder 1 sein, wahr oder falsch, gut oder schlecht usw. Es hängt alles von der Interpretation ab, die dem Bit gegeben wird!
- Ein Oktett / Byte ist eine 8-Bit-Sequenz, die wie in der folgende Abbildung von 0 bis 7 nummeriert ist:

7 Das “high” Bit	6	5	4	3	2	1	0 Das “low” Bit

Das Vorzeichenbit. Der Komplementär Code.

Wenn wir eine bestimmte Bit-Anordnung als Ganzzahl mit einem Vorzeichen interpretieren, wird gemäß der Konvention ein einzelnes Bit verwendet, um das Vorzeichen einer Zahl darzustellen. Dies ist das High-Bit (Bit 7) des High-Bytes der Stelle, an der die Nummer dargestellt wird. Wenn der Wert dieses Bits Null ist, ist die Zahl positiv. Wenn das Bit eins ist, ist die Zahl negativ.

Wie sind die Ganzzahlen in der Vorzeichenkonvention dargestellt?

Im Laufe der Zeit wurden verschiedene Richtungen vorgeschlagen:

- **Direktcode**: Darstellung des Absolutwerts der Zahl auf den n-1 Bits des n der Aufstellungsraum und im höchstwertigen Bit das Vorzeichen zu setzen. Obwohl diese Lösung der natürlichen sehr nahe kommt, hat sie sich als weniger effektiv erwiesen als andere. Ein Problem wäre, dass in dieser Darstellung: $-7 + 7 \neq 0$
- **Gegensatzcode** (das **Einerkomplement**): Darstellung des Absolutwerts der Zahl auf n-1 Bits des n der Aufstellungsraum, und wenn die Zahl negativ ist, invertieren Sie alle n Bits der Darstellung. Auch auf diese Darstellung wurde verzichtet, da sie ineffizient war (zB das Problem erscheint wieder $-7 + 7 \neq 0$)
- **Komplementär Code** (das **Zweierkomplement**): Um eine durch n Bits dargestellte Ganzzahl zu komplementieren, werden zuerst die Werte aller Bits (Wert 0 wird 1 und Wert 1 wird 0) von der Aufstellungsraum aus abgesetzt, wonach 1 zu dem erhaltenen Wert addiert wird .

Die dritte Richtung ist auch diejenige, die für die Darstellung von ganzen Zahlen mit Vorzeichen auferlegt wurde.

Alternative Regeln für die Komplementierung der Zahlen:

- Zulassen, dass die Bits von rechts von der Binärdarstellung auf das erste Bit 1 einschließlich geändert werden; Die restlichen Bits werden bis einschließlich Bit n-1 negiert.
- Subtrahieren Sie den Inhalt (offensichtlich binär) der Aufstellungsraum das komplementiert wird von 100 ... 00, wobei die Zahl nach der Binärziffer 1 so viele Nullen hat, wie Bits der Aufstellungsraum hat.
- Subtrahieren Sie den hexadezimalen Inhalt (offensichtlich hexadezimal) der Aufstellungsraum das komplementiert wird von 100 ... 00, wobei nach der hexadezimalen Zahl 1 so viele Nullen erscheinen wie die hexadezimalen Ziffern, die der Aufstellungsraum hat.

Die drei alternativen Regeln fur den Komplement entsprechen der oben per Definition angegebenen Komplement Regel.

Wenn wir beispielsweise ein 1 Byte Aufstellungsraum komplementieren möchten, die die Nummer $(18)_{10}$ enthält:

Anfangsstandort:	00010010
Nach die Negierung des Bits	11101101
Addiere 1	11101101+
	00000001
Komplement:	11101110

Also hat die Zahl $(18)_{10}$, das heißt $(12)_{16}$, das heißt $(00010010)_2$,also als Komplement die Zahl $(11101110)_2$, das heißt $(EE)_{16}$, das heißt $(238)_{10}$.

Unter Anwendung der Regel der binären Subtraktion haben wir:

	100000000-
Anfangsstandort:	00010010
Komplement:	11101110

Unter Anwendung der Hexadezimalregel haben wir:

	100-
Anfangsstandort:	12
Komplement:	EE

Darstellungsregel für Ganzzahlen mit Vorzeichen

Eine Ganzzahl zwischen $\mathbf{-2^{n-1}}$ und $\mathbf{2^{n-1}-1}$ wird an einer **n**-Bit-Aufstellungsraum wie folgt dargestellt:

- ist die Zahl positiv, so wird an der Stelle die jeweilige in Basis 2 geschriebene Zahl dargestellt;
- Wenn die Zahl negativ ist, wird das Komplement der Darstellung in Basis 2 der Zahl an der Stelle eingeschrieben.
-

Bevor wir zu den Beispielen übergehen, müssen wir die Situation der Darstellung der Zahl -2n-1 klären. Sein absoluter Wert kann nicht auf **n-1** Bits dargestellt werden, so dass Platz für das Vorzeichenbit vorhanden ist, aber er wird auf **n** Bits dargestellt und ist **100 ... 0.**

Diese Darstellung zeigt zum einen eine negative Zahl an! Andererseits habe ich bereits gezeigt, dass diese Nummer seine eigenes Komplement ist. Aus diesen Gründen wurde vereinbarungsgemäß festgelegt, dass die Zahl $\mathbf{-2^{n-1}}$ in komplementären Code auf n Bits durch **100 ...0** dargestellt wird. Dieselbe Konfiguration, die ohne Vorzeichen interpretiert wird, repräsentiert die Zahl $\mathbf{2^{n-1}}$.

Die folgende Tabelle zeigt die Darstellungen mehrerer Zahlen in Aufstellungsraum mit 8 Bits - 1 Byte, 16 Bits - 2 Bytes und 32 Bits - 4 Bytes.

Standortgröße (oktet)	Dezimalzahl	Darstellung in komplementären Code (hexadezimal)	Darstellung in komplementären Code (binär)
1	0	00	00000000
2	0	0000	0000000000000000
1	1	01	00000001
2	1	0001	0000000000000001
1	-1	FF	11111111
2	-1	FFFF	1111111111111111
1	127	7F	01111111
2	127	007F	0000000001111111
1	-128	80	10000000
2	-128	FF80	1111111110000000
2	128	0080	0000000010000000
2	32767	7FFF	0111111111111111

Warum komplementärer Code?

Die Implementierungen der Operationen über ganze Zahlen müssen einerseits effizient sein und andererseits, unabhängig von der Darstellungskonvention, so weit wie möglich gemeinsame Algorithmen zur Bewertung der grundlegenden Operationen über ganze Zahlen verwenden.

Bisher erfüllt die Darstellung in komplementären Code die obigen Anforderungen am besten. Die Hauptgründe sind die folgenden zwei:

- Die Additionsoperation wird gleich ausgeführt, unabhängig davon, ob es sich um die Konvention der Darstellung ohne Vorzeichen oder die der Darstellung mit Vorzeichen

handelt. Die ausgeführte Operation ist eine einfache Addition von n Bits (n - die Größe der Position), wobei der letzte Transport ignoriert wird.

- Die Subtraktionsoperation wird auf die Operation des Addierens des Subtrahierens mit dem Komplement des Minuend reduziert.

Multiplikations- und Divisionsoperationen werden mit verschiedene Algorithmen für die Darstellungen mit und ohne Vorzeichen durchgeführt, aber der Prozentsatz der Additions- und Subtraktionsoperationen ist in Anwendungen viel höher als der der multiplikativen Operationen. Daher die Präferenz der Designer für die Annahme des komplementär Codes für die Darstellung aller Zeichen. Ein weiterer Vorteil ist, dass Sie bei der Arbeit mit positiven Zahlen (und das wissen Sie von Anfang an) kein bisschen für das Vorzeichen verlieren. Der Darstellungsbereich ist viel größer.

Die Größe der Darstellung

In Bezug auf Berechnungen, die mit einer Maschine durchgeführt werden, gibt es natürlich eine Reihe von Einschränkungen für die Darstellung aller Ganzzahlen (und nicht nur).

Das wichtigste davon ist die Dimension der Darstellung, dh die maximale Anzahl von Binärziffern (die Anzahl von Bits) in der Darstellung einer ganzen Zahl. Bezeichnen wir diese Dimension der Darstellung mit "**n**".
Die Werte von **n** für aktuelle Computer können sein: 8, 16, 32 und 64.

Aufgabe: Wenn eine Zahl durch 7 Bits dargestellt wird, wie stellen wir sie durch 8 Bits dar?
Antwort: Es kommt auf die Interpretation an

- In der vorzeichenlosen Darstellung vervollständigen wir die verbleibenden hohen Bits mit Nullen
- In der Vorzeichendarstellung vervollständigen wir die verbleibenden Bits mit dem Vorzeichen

Beispiel: $(10011)_2$ kann man als ein oktett so darstellen:

- $(00010011)_2$, in der vorzeichenlosen Darstellung
- $(11110011)_2$. In der Zeichendarstellung

In der folgenden Tabelle sind die Nummernbereiche aufgeführt, die je nach Größe sowohl in der Konvention ohne Vorzeichen als auch in der Konvention mit Vorzeichen an einem Speicherort dargestellt werden können.

Nr. Octeți	Convenția fără semn	Convenția cu semn
1	$[0, 2^8-1] = [0 , 255]$	$[-2^7 , 2^7-1] = [-128 , 127]$
2	$[0, 2^{16}-1] = [0 , 65535]$	$[-2^{15} , 2^{15}-1] = [-32\,768 , 32\,767]$
4	$[0, 2^{32}-1] = [0 , 4\,294\,967\,295]$	$[-2^{31} , 2^{31}-1] = [-2\,147\,483\,648 , 2\,147\,483\,647]$

8	$[0, 2^{64}-1]$ = [0 , 18 446 824 753 389 551 615]	$[-2^{63} , 2^{63}-1]$ = [-9 223 412 376 694 775 808 , 9 223 412 376 694 775 807]

Die Hilfsprogramme für den Labor

- Editor: Notepad++
- Asamblor: NASM
- Linker: ALINK
- Debugger: Olly DBG

Das Toolkit kann hier heruntergeladen werden: 🔗 ASM-Tools. Schüler, die andere Betriebssysteme als Windows verwenden, können entweder einen Emulator verwenden (die Tools wurden erfolgreich mit Wine getestet) oder die Tools auf einer virtuellen Windows-Maschine ausführen. Um das Schreiben von ASM-Programmen zu erleichtern, enthält das Toolkit einen visuellen Editor (Notepad ++) mit einem integrierten Plugin, mit dem die meisten Vorgänge in mehreren Tastenkombinationen ausgeführt werden können.

Gebrauchsanweisung:

- Laden Sie das Toolkit herunter und extrahieren Sie den Inhalt des Archivs
- Starten Sie den Editor Notepad ++ im Verzeichnis npp (verwenden Sie nicht die eigene Version des Herausgebers)
- Fahren Sie mit dem Schreiben der Programme fort, die die Probleme lösen, die Ihnen mitgeteilt werden. Sie können die folgenden Tastenkombinationen im Menü Plugins -> ASM Plugin verwenden:

ASM Code Template	Ctrl+Shift+N	Vervollständigen Sie im Editor ein Minimalprogramm in ASM
Build ASM	Ctrl+F7	Stellen Sie das aktuelle Programm zusammen
Run program	Ctrl+F6	Führen Sie das aktuelle Programm aus
Debug program	F6	Problembehandlung für das aktuelle Programm mit Olly Debugger (die Dokumentation für den Debugger finden Sie im Verzeichnis ollydbg im Archiv).

Ein minimales Beispiel für ein Assemblersprachenprogramm

Beispiel:

```
bits 32 ;Assemblierung und Zusammenstellung für 32-Bit-Architektur
; Wir definieren den Einstiegspunkt im Hauptprogramm
global start

; Wir deklarieren die für unser Programm notwendigen externen Funktionen
extern exit ; wir zeigen der Assembly an, dass exit existiert, obwohl wir es
nicht definieren werden
import exit msvcrt.dll  ; exit ist eine Funktion, die den Vorgang abschließt
und in msvcrt.dll definiert ist
             ; msvcrt.dll enthält exit, printf und alle anderen wichtigen
C-Laufzeitfunktionen

; das Datensegment, in dem die Variablen definiert werden
segment data use32 class=data
; ...

; Codesegment
segment code use32 class=code
start:
; ...

    ; exit(0)
    push dword 0 ; Der Parameter der Exit-Funktion wird auf den Stack gelegt
     call [exit] ; den Funktionsaufruf exit, um die Programmausführung zu
beenden
```

Labor 1 - Vorgeschlagenen Übungen

Löse auf Papier und erkläre die folgenden Übungen:

1. Wandle von Basis 10 in 2 und dann in 16 die folgenden Zahlen um:

- 4
- 10
- 15
- 32

2. Konvertieren Sie von Basis 10 zu 16 und dann zu 2 die folgenden Zahlen:

- 3
- 11
- 16
- 17

3. Konvertieren Sie von Basis 2 zu Basis 16 die folgenden Zahlen:

- 1010
- 0111
- 1111
- 10001010
- 110101111

4. Konvertieren Sie von Basis 16 zu Basis 2 die folgenden Zahlen:

- 3
- A
- F
- 2B
- 2F8

5. Führen Sie die folgenden Operationen in Basis 2 aus (ohne zu Basis 10 zu konvertieren):

- 1+1
- 10+10
- 111+1
- 1010-1
- 1000-10

6. Führen Sie die folgenden Vorgänge auf Basis 16 aus (ohne Konvertierung auf Basis 10):

- 9+1
- B+2
- F+1
- 10+A
- 10-2
- B-3

7. Prüfen Sie anhand von mindestens zwei der Komplementaritätsregeln, ob:

- In einer 2-Byte-Position sind die Zahlen $(9A7D)_{16}$ und $(7583)_{16}$ komplementär.
- In einer 4-Byte-Position sind die Zahlen $(000F095D)_{16}$ und $(FFF0F6A3)_{16}$ komplementär
- In einer 2-Byte-Position sind die Zahlen $(4BA1)_{16}$ und $(5C93)_{16}$ komplementär
- an einer 1-Byte-Stelle sind die Zahlen $(7F)_{16}$ und $(81)_{16}$ komplementär
- In einer 2-Byte-Position sind die Nummern $(732A)_{16}$ und $(4E58)_{16}$ komplementär

8. Stellen Sie die folgenden Zahlen in einem Byte dar, in Zeichendarstellung und ohne Zeichendarstellung:

$(11001)_2$
$(1010)_2$
$(11001)_2$
$(1011)_2$
$(1110111)_2$
$(10111)_2$
$(110)_2$
$(11)_2$

9. Stellen Sie die folgenden Zahlen in zwei Bytes dar, in Zeichendarstellung und ohne Zeichendarstellung:

$(11001010)_2$
$(1110010)_2$
$(11001010)_2$
$(1010110)_2$
$(11010)_2$
$(10010)_2$
$(0010111)_2$
$(010101)_2$