

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben – 2. November 2023

Aufgabe 1: Zahlensysteme

Wandeln Sie ohne Hilfsmittel

- | | | |
|--------------------|----------------------------|-----------------|
| • nach Dezimal: | • nach Hexadezimal: | • nach Binär: |
| (a) $0010\ 0000_2$ | (d) $0010\ 0000_2$ | (g) 750_8 |
| (b) 42_{16} | (e) 42_{10} | (h) 42_{10} |
| (c) 17_8 | (f) $192.168.20.254_{256}$ | (i) $AFFE_{16}$ |

Berechnen Sie ohne Hilfsmittel:

- (j) $750_8 \& 666_8$
(k) $A380_{16} + B747_{16}$
(l) $AFFE_{16} >> 1$

Die tiefgestellte Zahl steht für die Basis des Zahlensystems. Jede Teilaufgabe zählt 1 Punkt.
(In der Klausur sind Hilfsmittel zugelassen, daher ist dies *keine* typische Klausuraufgabe.)

Lösung

Wandeln Sie ohne Hilfsmittel

- nach Dezimal:
 - (a) $0010\ 0000_2 = 32_{10}$
Eine Eins mit fünf Nullen dahinter steht binär für $2^5 = 32$:
mit 1 anfangen und fünfmal verdoppeln.
 - (b) $42_{16} = 4 \cdot 16 + 2 \cdot 1 = 64 + 2 = 66$
 - (c) $17_8 = 1 \cdot 8 + 7 \cdot 1 = 8 + 7 = 15$

Umwandlung von und nach Dezimal ist immer rechenaufwendig. Umwandlungen zwischen Binär, Oktal und Hexadezimal gehen ziffernweise und sind daher wesentlich einfacher.

- nach Hexadezimal:
 - (d) $0010\ 0000_2 = 20_{16}$
Umwandlung von Binär nach Hexadezimal geht ziffernweise:
Vier Binärziffern werden zu einer Hex-Ziffer.
 - (e) $42_{10} = 32_{10} + 10_{10} = 20_{16} + A_{16} = 2A_{16}$
 - (f) $192.168.20.254_{256} = C0\ A8\ 14\ FE_{16}$
Umwandlung von der Basis 256 nach Hexadezimal geht ziffernweise:
Eine 256er-Ziffer wird zu zwei Hex-Ziffern.
Da die 256er-Ziffern dezimal angegeben sind, müssen wir viermal Dezimal nach Hexadezimal umwandeln. Hierfür bieten sich unterschiedliche Wege an.
 $192_{10} = 128_{10} + 64_{10} = 1100\ 0000_2 = C0_{16}$
 $168_{10} = 10_{10} \cdot 16_{10} + 8_{10} = A_{16} \cdot 10_{16} + 8_{16} = A8_{16}$
 $20_{10} = 16_{10} + 4_{10} = 10_{16} + 4_{16} = 14$
 $254_{10} = 255_{10} - 1_{10} = FF_{16} - 1_{16} = FE_{16}$
- nach Binär:
 - (g) $750_8 = 111\ 101\ 000_2$
Umwandlung von Oktal nach Binär geht ziffernweise:
Eine Oktalziffer wird zu drei Binärziffern.

- (h) $42_{10} = 2A_{16}$ (siehe oben) = 0010 1010₁₆
 Umwandlung von Hexadezimal nach Binär geht ziffernweise:
 Eine Hex-Ziffer wird zu vier Binärziffern.
- (i) $AFFE_{16} = 1010\ 1111\ 1111\ 1110_2$
 Umwandlung von Hexadezimal nach Binär geht ziffernweise:
 Eine Hex-Ziffer wird zu vier Binärziffern.

Berechnen Sie ohne Hilfsmittel:

(j) $750_8 \& 666_8 = 111\ 101\ 000_2 \& 110\ 110\ 110_2 = 110\ 100\ 000_2 = 640_8$

Binäre Und-Operationen lassen sich am leichtesten in binärer Schreibweise durchführen. Umwandlung zwischen Oktal und Binär geht ziffernweise: Eine Oktalziffer wird zu drei Binärziffern und umgekehrt. Mit etwas Übung funktionieren diese Operationen auch direkt mit Oktalzahlen im Kopf.

(k)
$$\begin{array}{r} A380_{16} \\ + B747_{16} \\ \hline 15AC7_{16} \end{array}$$

Mit Hexadezimalzahlen (und Binär- und Oktal- und sonstigen Zahlen) kann man genau wie mit Dezimalzahlen schriftlich rechnen. Man muß nur daran denken, daß der „Zehner“-Überlauf nicht bei 10_{10} stattfindet, sondern erst bei $10_{16} = 16_{10}$ (hier: $8_{16} + 4_{16} = C_{16}$ und $3_{16} + 7_{16} = A_{16}$, aber $A_{16} + B_{16} = 10_{10} + 11_{10} = 21_{10} = 16_{10} + 5_{10} = 10_{16} + 5_{16} = 15_{16}$).

(l) $AFFE_{16} >> 1 = 1010\ 1111\ 1111\ 1110_2 >> 1 = 0101\ 0111\ 1111\ 1111_2 = 57FF_{16}$

Bit-Verschiebungen lassen sich am leichtesten in binärer Schreibweise durchführen. Umwandlung zwischen Hexadezimal und Binär geht ziffernweise: Eine Hex-Ziffer wird zu vier Binärziffern und umgekehrt.

Mit etwas Übung funktionieren diese Operationen auch direkt mit Hexadezimalzahlen im Kopf.

Aufgabe 2: Ausgabe von Hexadezimalzahlen

Schreiben Sie eine Funktion `void print_hex (uint32_t x)`, die eine gegebene vorzeichenlose 32-Bit-Ganzzahl `x` als Hexadezimalzahl ausgibt. (Der Datentyp `uint32_t` ist mit `#include <stdint.h>` verfügbar.)

Verwenden Sie dafür *nicht* `printf()` mit der Formatspezifikation `%x` als fertige Lösung, sondern programmieren Sie die nötige Ausgabe selbst. (Für Tests ist `%x` hingegen erlaubt und sicherlich nützlich.)

Die Verwendung von `printf()` mit anderen Formatspezifikationen wie z. B. `%d` oder `%c` oder `%s` ist hingegen zulässig.

(8 Punkte)

(Hinweis für die Klausur: Abgabe auf Datenträger ist erlaubt und erwünscht, aber nicht zwingend.)

Lösung

Um die Ziffern von `x` zur Basis 16 zu isolieren, berechnen wir `x % 16` (modulo 16 = Rest bei Division durch 16) und dividieren anschließend `x` durch 16, solange bis `x` den Wert 0 erreicht.

Wenn wir die auf diese Weise ermittelten Ziffern direkt ausgeben, sind sie *Little-Endian*, erscheinen also in umgekehrter Reihenfolge. Die Datei `loesung-2-1.c` setzt diesen Zwischenschritt um.

Die Ausgabe der Ziffern erfolgt in `loesung-2-1.c` über `printf ("%d")` für die Ziffern 0 bis 9. Für die darüberliegenden Ziffern wird der Buchstabe `a` um die Ziffer abzüglich 10 inkrementiert und der erhaltene Wert mit `printf ("%c")` als Zeichen ausgegeben.

Um die umgekehrte Reihenfolge zu beheben, speichern wir die Ziffern von `x` in einem Array `digits[]` zwischen und geben sie anschließend in einer zweiten Schleife in umgekehrter Reihenfolge aus (siehe `loesung-2-2.c`). Da wir wissen, daß `x` eine 32-Bit-Zahl ist und daher höchstens 8 Hexadezimalziffern haben kann, ist 8 eine sinnvolle Länge für das Ziffern-Array `digits[8]`.

Nun sind die Ziffern in der richtigen Reihenfolge, aber wir erhalten zusätzlich zu den eigentlichen Ziffern führende Nullen. Da in der Aufgabenstellung nicht von führenden Nullen die Rede war, sind diese nicht verboten; [loesung-2-2.c](#) ist daher eine richtige Lösung der Aufgabe.

Wenn wir die führenden Nullen vermeiden wollen, können wir die **for**-Schleifen durch **while**-Schleifen ersetzen. Die erste Schleife zählt hoch, solange *x* ungleich 0 ist; die zweite zählt von dem erreichten Wert aus wieder herunter – siehe [loesung-2-3.c](#). Da wir wissen, daß die Zahl *x* höchstens 32 Bit, also höchstens 8 Hexadezimalziffern hat, wissen wir, daß *i* höchstens den Wert 8 erreichen kann, das Array also nicht überlaufen wird.

Man beachte, daß der Array-Index nach der ersten Schleife „um einen zu hoch“ ist. In der zweiten Schleife muß daher *zuerst* der Index dekrementiert werden. Erst danach darf ein Zugriff auf [digit\[i\]](#) erfolgen.

Alternativ können wir auch mitschreiben, ob bereits eine Ziffer ungleich Null ausgegeben wurde, und andernfalls die Ausgabe von Null-Ziffern unterdrücken – siehe [loesung-2-4.c](#).

Weitere Möglichkeiten ergeben sich, wenn man bedenkt, daß eine Hexadezimalziffer genau einer Gruppe von vier Binärziffern entspricht. Eine Bitverschiebung um 4 nach rechts ist daher dasselbe wie eine Division durch 16, und eine Und-Verknüpfung mit $15_{10} = f_{16} = 1111_2$ ist dasselbe wie die Operation Modulo 16. Die Datei [loesung-2-5.c](#) ist eine in dieser Weise abgewandelte Variante von [loesung-2-3.c](#).

Mit dieser Methode kann man nicht nur auf die jeweils unterste Ziffer, sondern auf alle Ziffern direkt zugreifen. Damit ist kein Array als zusätzlicher Speicher mehr nötig. Die Datei [loesung-2-6.c](#) setzt dies auf einfache Weise um. Sie gibt wieder führende Nullen mit aus, ist aber trotzdem eine weitere richtige Lösung der Aufgabe.

Die führenden Nullen ließen sich auf die gleiche Weise vermeiden wie in [loesung-2-4.c](#).

Die Bitverschiebungsmethode hat den Vorteil, daß kein zusätzliches Array benötigt wird. Auch wird die als Parameter übergebene Zahl *x* nicht verändert, was bei größeren Zahlen, die über Zeiger übergeben werden, von Vorteil sein kann. Demgegenüber steht der Nachteil, daß diese Methode nur für eine ganze Anzahl von Bits funktioniert, also für Basen, die Zweierpotenzen sind (z. B. 2, 8, 16, 256). Für alle anderen Basen (z. B. 10) eignet sich nur die Methode mit Division und Modulo-Operation.

Aufgabe 3: Einfügen in Strings

Wir betrachten das folgende Programm ([aufgabe-3.c](#)):

```
#include <stdio.h>
#include <string.h>

void insert_into_string (char src, char *target, int pos)
{
    int len = strlen (target);
    for (int i = pos; i < len; i++)
        target[i+1] = target[i];
    target[pos] = src;
}

int main (void)
{
    char test[100] = "Hochshule_Bochum";
    insert_into_string ('c', test, 5);
    printf ("%s\n", test);
    return 0;
}
```

Die Ausgabe des Programms lautet: `Hochschhhhhhhhhhh`

(a) Erklären Sie, wie die Ausgabe zustandekommt. (3 Punkte)

- (b) Schreiben Sie die Funktion `insert_into_string()` so um, daß sie den Buchstaben `src` an der Stelle `pos` in den String `target` einfügt.
Die Ausgabe des Programms müßte dann `Hochschule Bochum` lauten. (2 Punkte)
- (c) Was kann passieren, wenn Sie die Zeile `char test[100] = "Hochshule_Bochum";` durch `char test[] = "Hochshule_Bochum";` ersetzen? Begründen Sie Ihre Antwort. (2 Punkte)
- (d) Was kann passieren, wenn Sie die Zeile `char test[100] = "Hochshule_Bochum";` durch `char *test = "Hochshule_Bochum";` ersetzen? Begründen Sie Ihre Antwort. (2 Punkte)

Lösung

- (a) **Erklären Sie, wie die Ausgabe zustandekommt.**

In der Schleife wird *zuerst* der nächste Buchstabe `target[i + 1]` gleich dem aktuellen gesetzt und *danach* der Zähler `i` erhöht. Dadurch wird im nächsten Schleifendurchlauf der bereits verschobene Buchstabe noch weiter geschoben und letztlich alle Buchstaben in `target[]` durch den an der Stelle `pos` ersetzt.

- (b) **Schreiben Sie die Funktion `insert_into_string()` so um, daß sie den Buchstben `src` an der Stelle `pos` in den String `target` einfügt.**

Die Ausgabe des Programms müßte dann `Hochschule Bochum` lauten.

Um im String „Platz zu schaffen“, muß man von hinten beginnen, also die Schleife umdrehen (siehe: `loesung-3.c`):

```
for (int i = len; i >= pos; i--)
    target[i + 1] = target[i];
```

- (c) **Was kann passieren, wenn Sie die Zeile `char test[100] = "Hochshule_Bochum";` durch `char test[] = "Hochshule_Bochum";` ersetzen und warum?**

Die Schreibweise `test[]` bedeutet, daß der Compiler selbst zählt, wieviel Speicherplatz der String benötigt, un dann genau die richtige Menge Speicher reserviert (anstatt, wie wir es manuell getan haben, pauschal Platz für 100 Zeichen).

Wenn wir nun in den String ein zusätzliches Zeichen einfügen, ist dafür kein Speicherplatz reserviert worden, und wir **überschreiben** dann Speicher, an dem sich andere Variable befinden, was zu einem **Absturz** führen kann.

Da wir hier nur ein einziges Zeichen schreiben, wird dieser Fehler nicht sofort auffallen. Dies ist schlimmer, als wenn das Programm direkt beim ersten Test abstürzt, denn dadurch entsteht bei uns der Eindruck, es sei in Ordnung. Wenn danach der Fehler in einer Produktivumgebung auftritt, kann dadurch Schaden entstehen – je nach Einsatzgebiet der Software u. U. erheblicher Vermögens-, Sach- und/oder Personenschaden (z. B. Absturz eines Raumflugkörpers).

- (d) **Was kann passieren, wenn Sie `char test[100] = "Hochshule_Bochum";` durch `char *test = "Hochshule_Bochum";` ersetzen und warum?**

In diesem Fall wird der Speicher für den eigentlichen String in einem unbenannten, **nicht schreibbaren** Teil des Speichers reserviert. Unser Versuch, dorthin ein zusätzliches Zeichen zu schreiben, führt dann normalerweise zu einem **Absturz**.

In manchen Systemen (Betriebssystem, Compiler, ...) ist der Speicherbereich tatsächlich sehr wohl schreibbar. In diesem Fall tritt der Absturz nicht immer und nicht immer sofort auf – genau wie in Aufgabenteil (c).