Multiplikation mittels Bitoperatoren I

von-Neumann-Addition

- In jedem Schritt:
  1. Ziffern *ohne* Übertrag bestimmen (Halbaddition)
     Logische Operation: EXKLUSIVES ODER
  2. Überträge bestimmen
     Logische Operationen: UND, gefolgt von LINKSSHIFT
- Fortsetzung solange, bis keine Überträge mehr auftreten
- Liefert Summe, weil Summe = Halbaddition + Überträge

Bsp.:

$$
\begin{array}{rr}
 & 1011001 \\
+ & 1010011 \\
\hline
 & 0001010 \\
+ & 1010001 \\
\hline
 & 10101000 \\
+ & 0000001 \\
\hline
 & 10101100 \\
+ & 000000 \\
\hline
 & 10101100
\end{array}
$$

Multiplikation mittels Bitoperatoren II

Multiplikation

- Im Dualsystem: Multiplikation mit 2 durch Linksshift
- Multiplikation wie üblich, jedoch am besten von rechts nach links durchführen!

Bsp.:

$$\begin{array}{r} 1010 \cdot 1011 \\ \hline 1010 \\ 1010 \\ 0000 \\ 1010 \\ \hline 1101110 \end{array} \qquad a = (1010)_2 \quad b = (1011)_2$$

Pseudocode $b =: (b_{n-1} \ldots b_0)_2$

1. $s = 0;$
2. `for` $(j = 0;\ j < n;\ ++j)$ `{`
   `if` $(b_j != 0)\ s = s + (a << j);$
   `}`

Multiplikation mittels Bitoperatoren III

Programmiermethode I: bottom-up

- Direkte Lösung von Teilproblemen, schrittweise Zusammenfassung zu jeweils größeren Einheiten bis zur Lösung des Gesamtproblems

Vorgehensweise hier - Skizze

1. Programmierung der Multiplikation mit herkömmlicher Addition (als Funktion), evtl. zunächst ohne Überlaufbehandlung
2. Programmierung der von-Neumann-Addition (sinnvollerweise als Funktion), evtl. zunächst ohne Überlaufbehandlung
3. Ersetzen der herkömmlichen Addition in 1. durch die von-Neumann-Addition
4. Einbauen der restlichen Anforderungen, z.B. der Überlaufbehandlung

Multiplikation mittels Bitoperatoren IV

Programmiermethode II: top-down

- Lösung des Gesamtproblems durch Zerlegung in Teilprobleme, die entweder elementar gelöst oder weiter zerlegt werden („schrittweise Verfeinerung“, „strukturierte Programmierung“)
- Zu den elementaren Anweisungen werden auch bedingte Anweisungen, Auswahl-und Wiederholungsanweisungen gerechnet, *nicht* aber direkte Sprünge
- Hilfsmittel: Pseudocode, Nassi-Shneiderman-Diagramme
- Ausgangspunkt oft: Eingabe, Verarbeitung, Ausgabe

Multiplikation mittels Bitoperatoren V

Exemplarische Vorgehensweise

Daten: $a = \sum_{i=0}^{m-1} 2^i a_i, \quad b = \sum_{j=0}^{n-1} 2^j b_j \quad (a_i, b_i \in \{0,1\})$

$$a \cdot b = \sum_{j=0}^{n-1} 2^j a \cdot b_j$$

0. Problem

1. a, b (hexadezimal) eingeben
2. `/*` Multiplikation auf Multiplikation mit Zweierpotenzen und Addition zurückführen `*/`

   $s = 0$;
   `for` $(j = 0;\ j < n;\ ++j)$
   `if` $(b_j$ `!=0)` `{` $s = s + 2^j a$; `}`
3. a, b, $a \cdot b$ (herkömml. Mult.) und s ausgeben

Multiplikation mittels Bitoperatoren VI

1. Verfeinerungsschritt (zu 2.)

2. $s = 0; \tilde{a} = a;$ `/*` $\tilde{a}^{(j)} = 2^j a$ `*/`
   `for` `(`$j = 0$`;` $j < n$`;` `++`j`)`
   `{`
   `if` `(`b_j`!=0)` `{`
   $s = s + \tilde{a}$ durch Von-Neumann-Addition
   `}`
   $\tilde{a} = 2 \cdot \tilde{a}$ durch Linksshift mit Überlaufbehandlung
   `}`

Problem: Bestimmung von b_j, n ?

Idee: b_j schrittweise durch Rechtsshift von b (Bez. $\tilde{b}$) und Abfrage von $\tilde{b}$`&1` bestimmen.
Beendigung, falls $\tilde{b} = 0$

Multiplikation mittels Bitoperatoren VII

1. Verfeinerungsschritt (zu 2.) - Modifikation

2.1 $s = 0; \tilde{a} = a; \tilde{b} = b;$ /* $\tilde{a}^{(j)} = 2^j a;$ $\tilde{b}^{(j)} = b >> j$ */

2.2
```
while (b̃!=0)
{
```

2.2.1 $s = s + \tilde{a}$ durch Von-Neumann-Addition

```
}
```

2.3 `b̃>>=1;`

2.4 Falls höchstes Bit in $\tilde{a}$ gesetzt und $\tilde{b} \neq 0$, Überlauf melden

2.5 `ã<<=1;`

```
}
```

Multiplikation mittels Bitoperatoren VIII

2. Verfeinerungsschritt (zu 2.2.1)

2.2.1 `/*` $s + \tilde{a}$ durch von-Neumann-Addition berechnen, Ergebnis auf g `*/`

$g = s$; $h = \tilde{a}$;

```
do
{
```

übertrag = *g*`&`*h*;

Falls höchstes Bit in *übertrag* gesetzt, Überlauf melden

übertrag `<<=1;`

halbaddition = *g*`^`*h*;

g =*halbaddition*; *h* =*übertrag*;

```
}
while (übertrag!=0);
```

`/*` $s = s + \tilde{a}$ `*/`

$s = g$;

Multiplikation mittels Bitoperatoren IX

Anmerkungen zur Top-Down-Methode

- Schrittweise Verfeinerung erfolgt oft direkt in der Programmquelle.
  Empfehlung: Noch nicht ausgeführte Programmteile als Kommentare formulieren!
- Die Top-Down-Methode zerlegt den *zeitlichen* Ablauf des Programms in elementarere Schritte.
  Zur Organisation der Daten gibt es weitere Methoden.
- Ziel ist es auch, den Datenfluss auf den unbedingt notwendigen Teil zu beschränken.
- Direkte Sprünge (`goto`) sind oft ein Verstoß gegen den Geist der Methode.

Multiplikation mittels Bitoperatoren X

Allgemeine Anmerkungen zu den Programmiermethoden

- Beide Methoden sind Hilfsmittel zur Programmierung, können diese unterstützen, ersetzen eigene Überlegungen natürlich nicht.
- Beide Methoden werden nur selten durchgehend auf ein Gesamtproblem angewandt. Häufiger ist der Einsatz bei geeigneten Teilproblemen wie in dieser Übungsaufgabe.
- Die Top-down-Methode kann zu tiefen Verschachtelungen führen, die aus Gründen der Übersichtlichkeit nicht immer erwünscht sind.
- Viele Programmiersprachen unterstützen auf der Funktionsebene keine Unterfunktionen, sondern nur Blockschachtelungen.
- Bei der Bottom-up-Methode kann das Zusammenspiel der Funktionen und der Zugriff auf gemeinsame Daten ein Problem darstellen. Der Ersatz von Funktionsparametern durch globale Variable kann zu schwer lokalisierbaren Fehlern führen.

Multiplikation mittels Bitoperatoren XI

Weitere Hinweise zur Aufgabe

- Zweck der Aufgabe ist das Vertrautwerden mit den Bitoperationen und ihr genaues Verständnis.
- Daher ist es hilfreich sein, zuerst eine Funktion

  ```
  void printbits(unsigned u)
  ```

  zu erstellen, die alle (32) Bits von u ausgibt, und mit dieser die Rechenschritte im einzelnen nachzuvollziehen.
- Möglicherweise ist es nützlich, auch die hexadezimale Eingabe mit dezimaler Ausgabe und umgekehrt zuvor auszuprobieren.
- Sinnvollerweise sollten erste Tests mit kleinen natürlichen Zahlen und algorithmischen Spezialfällen (z.B. Addition bzw. Multiplikation mit Zweierpotenzen) vorgenommen werden.