64-041 Übung Rechnerstrukturen

Aufgabenblatt 5 Ausgabe: 08.11., Abgabe: 15.11. 12:00

Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 5.1 (Punkte 10)

Multiplikation im Dual-/Hexadezimalsystem: Viele Prozessoren liefern bei der Multiplikation von ganzen Zahlen ein Ergebnis, das doppelt so viele Bits hat wie die beiden zu multiplizierenden Zahlen. Wenn a und b also 16/32 Bit breit sind, erhält man auf Machinen-/Assemblerebene ein 32/64 Bit breites Ergebnis $c = a \cdot b$, mit dem man weiterarbeiten könnte, wenn man es möchte. Wenn man in einer höheren Programmiersprache wie z.B. C oder JAVA arbeitet, ist das aber wenig sinnvoll; wenn a, b, c vom Typ *int* sind, nützt ein Ergebnis vom Typ *long* wenig. Man weist darum einfach c die unteren 16/32 Bit des Multplikationsergebnisses zu und rechnet damit weiter. Falls das Multiplikationsergebnis zu groß wird, hat man Pech gehabt.
Außerdem verwendeten die frühen Compiler statt des Multiplikationsbefehls für Zahlen im Zweierkomplement (es gibt sowohl positive wie negative Zahlen) den für Zahlen vom Typ *unsigned* (alle Zahlen werden als positiv interpretiert), weil dieser Befehl damals bis zu 15% schneller war.

Für positive Zahlen ist der oben beschriebene Ansatz auf jeden Fall sinnvoll, solange das Ergebnis nicht zu groß wird. Begründen Sie, warum diese Aussage auch für negative, im Zweierkomplement dargestellte, Zahlen gilt, auch wenn man bei der Multiplikation so tut, als wären sie positiv.
Zur Erinnerung: Wenn wir im Zweierkomplement mit 16 Bit arbeiten und a eine positive Zahl ist, dann liefert $2^{16} - a$ die Zweierkomplementdarstellung von $-a$.

Aufgabe 5.2 (Punkte 5+5+5)

Gleitkomma-Addition (I): Wir betrachten ein Gleitkommaformat im Dezimalsystem mit zwei Stellen für den Exponenten und vier Nachkommastellen für die normalisierte Mantisse. Addieren Sie die beiden Gleitkommazahlen und geben Sie sowohl die Zwischenrechnungen, als auch das normalisierte und gerundete Ergebnis an.

$8,626 \cdot 10^3 + 9,9442 \cdot 10^5$

Führen Sie diese Berechnung zweimal, mit unterschiedlichen Rundungsstrategien, durch:

(a) Mit einmaliger Rundung am Ende nach der Normalisierung.

(b) Bei der Berechnung werden alle Zahlen, auch die Zwischenergebnisse, auf vier Nachkommastellen gerundet.

(c) Welches Verfahren ist vorzuziehen? Beziehungsweise, was wäre ein sinnvolles Rundungsverfahren? Begründen Sie Ihre Antwort.

Aufgabe 5.3 (Punkte 10)

Gleitkomma-Addition (II): Addieren Sie die beiden folgenden mit einfacher Genauigkeit gemäß IEEE 754 dargestellten Gleitkommazahlen. Von der Mantisse sind dabei nur die oberen 8 Bit angegeben, die anderen sind 0. Geben Sie dabei die Zwischenrechnungen und das normalisierte Ergebnis an.

$(0 \mid 10000000 \mid 1000\,0000) + (0 \mid 10000011 \mid 1111\,0000)$

Aufgabe 5.4 (Punkte 5)

ASCII-Code (ISO-8859-1): Entschlüsseln Sie mit Hilfe der Tabellen aus dem Vorlesungsskript (Zeichensatz nach ISO-8859-1) die folgende Zeichenkette. Schreiben Sie dabei auch alle Steuerzeichen mit auf. Die einzelnen Zeichen sind als Hexwerte angegeben.

52 53 20 49 53 54 20 43 4F 4F 4C 21 0D 0A 4F 44 45
52 20 45 54 57 41 20 4E 49 43 48 54 3F 3F 0D 0A

Aufgabe 5.5 (Punkte 8+2+5)

UTF-8: Die ISO-8859-1 Kodierung benutzt 8 Bit für jedes enthaltene Zeichen. Die direkte Kodierung der basic-multilingual Plane von Unicode (Java Datentyp `char`) verwendet pro Zeichen 16 Bit, während die UTF-8 Kodierung vielfache von 8 Bit benutzt.

(a) Wir betrachten einen deutschsprachigen Text mit insgesamt 800 000 Zeichen. Wir nehmen die folgenden Wahrscheinlichkeiten für die Umlaute an, andere Sonderzeichen kommen nicht vor:

Ä/ä	Ö/ö	Ü/ü	ß
0,56%	0,29%	0,62%	0,31%

Wie viele Bytes belegt dieser Text bei Kodierung nach ISO-8859-1, in direkter Unicode Darstellung und in UTF-8?

(b) Wir betrachten einen chinesischen Text mit insgesamt 800 000 Schriftzeichen. Im Unicode-Standard sind für die CJK-Symbole (chinesisch, japanisch, koreanisch) die Bereiche von `U+3400` bis `U+4DBF` und `U+4E00` bis `U+9FCF` reserviert.

Wie viele Symbole sind das?

(c) Wie viele Bytes belegt der chinesische Text bei direkter Unicode Darstellung und bei Kodierung als UTF-8?

Aufgabe 5.6 (Punkte 5+5+5)

Shift-Operationen statt Multiplikation: Ersetzen Sie die folgenden Berechnungen *möglichst effizient* durch eine Folge von Operationen: <<, +, -. Nehmen Sie für die Variablen x und y den Datentyp `int` (32-bit Zweierkomplementzahl) an.

(a) $y = 10 * x$

(b) $y = 30 * x$

(c) $y = -48 * x$

Aufgabe 5.7 (Punkte 5+5+10+10)

Logische- und Shift-Operationen: Realisieren Sie, die folgenden Funktionen als *straightline*-Code in Java, das heißt ohne Schleifen, If-Else Abfragen oder den ternären Operator `.. ? .. : ...` Außerdem dürfen nur einige der logischen und arithmetischen Operatoren benutzt werden:

`! ~ & ^ | + << >> >>>`

Alle Eingabeparameter und Rückgabewerte sind jeweils (32-bit) Integerwerte.

(a) `bitNand(x,y)` Diese Funktion soll das bitweise NAND liefern: $\overline{x_i \wedge y_i}$. Als Operatoren dürfen nur | und ~ (OR, Negation) benutzt werden.

(b) `bitXor(x,y)` Diese Funktion soll die XOR-Verknüpfung (Antivalenz) realisieren: $x_i \oplus y_i$. Als Operatoren dürfen nur & und ~ (AND, Negation) benutzt werden.

(c) `rotateLeft(x,n)` Die Funktion soll den in Java nicht vorhandenen Rotate-Left Operator für `x` nachbilden. Für das zweite Argument `n` gilt: $0 \leq$ `n` ≤ 31.

(d) `abs(x)` Der Absolutwert (Betrag) von `x`. Welchen Wert liefert ihre Funktion für den Eingabewert -2^{31}? Beschreiben Sie, wie Ihre Lösung funktioniert.