Halblogarithmische Zahlendarstellung (Z3-Modell)

Timm Grams, Fulda, 19. März 2012 (aktualisiert: 04.07.13)

Rechnen binär – Ein-/Ausgabe dezimal

Mit binären Zahlen wird die Rechnung besonders einfach. Alle Rechenoperationen lassen sich auf Additionen, Komplementbildung und Schiebeoperationen zurückführen. Leibniz hat bereits 1697 auf die Vorteile des Rechnens im Dualsystem hingewiesen.

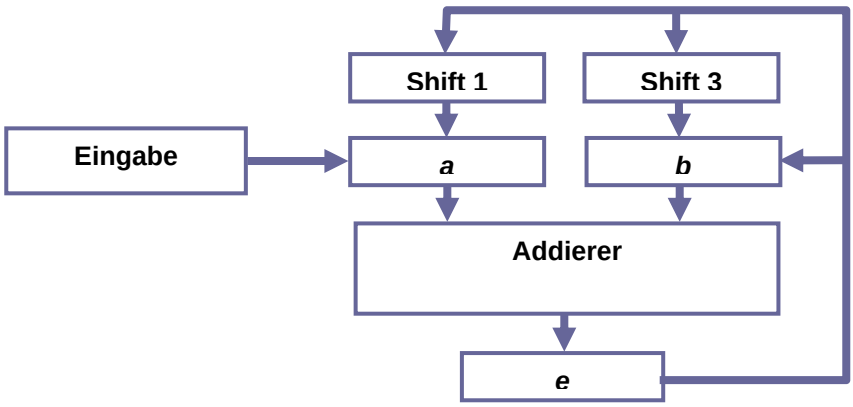
Und warum hat man dann nicht schon längst solche Maschinen gebaut? Nun ja: Der Mensch hat sich an das Dezimalsystem gewöhnt. Es ist schwer, ihm den Umgang mit einer Maschine, die nur Nullen und Einsen kennt, schmackhaft zu machen.

Dieses Problem hat Konrad Zuse gelöst. Die interne Zahlendarstellung muss keinerlei Rücksicht auf den Menschen nehmen, denn in der Maschine „sind die Zahlen unter sich“, wie Zuse sagte. Einen Nutzen hat eine solche Maschine aber nur, wenn sie sich vom Menschen mit seinen Gewohnheiten möglichst einfach bedienen lässt. Aus diesem Grund nahm Konrad Zuse einen sehr hohen Aufwand für die Zahlenwandlung vom Dezimal- ins Dualsystem und umgekehrt in Kauf.

Gerade an der Zahlendarstellung sieht man, wie sich Zuse von den Ingenieurstugenden leiten ließ. Er strebte nach Korrektheit, Bedienfreundlichkeit und Zuverlässigkeit der Funktion. Dabei war ihm die Eleganz der apparativen Ausführung wichtig: Zweckerfüllung und Effizienz bei möglichst geringem Aufwand.

Eingabe einer ganzen Dezimalzahl

Mit dem Eingabetastenfeld des Z3-Modells werden von links nach rechts die vier Ziffern 1, 9, 4 und 1 eingegeben. Den Exponenten lassen wir erst einmal auf dem Wert null stehen. Dann entspricht die Ziffernfolge genau der Zahl 1941. Jede Ziffer ist rechnerintern so verdrahtet bzw. verschaltet, dass ihr eine vierstellige binäre Zahlendarstellung entspricht, in unserem Beispiel: 0001, 1001, 0100 und 0001.



Das Einlesen der ersten Ziffer geht so vor sich:

(1) Die Zahl 1 wird in das *a*-Register gebracht. Dort steht dann *a* = 00…0001.

(2) Diese Zahl wird mit dem Inhalt des *b*-Registers (momentan noch null) addiert. Das Ergebnis erscheint im Register *e*. Wegen *b* = 0 ist nun *e* = *a* = 00…0001.

(3) Darauf erfolgt eine Multiplikation des Ergebnisses mit zehn, in Binärschreibweise: *e*×1010. Das lässt sich zerlegen in eine Multiplikation mit 2 und eine Multiplikation mit 8 und die Addition der beiden Ergebnisse: *e*×1010 = *e*×(0010 + 1000) = *e*×0010 + *e*×1000. Die Multiplikationen lassen sich einfach durch Verschieben der Zahl *e* um eine bzw. um drei Stellen nach links bewerkstelligen: Shift 1 und Shift 3. Die Summanden erscheinen in den Registern *a* und *b:* *a* = 00…00010, *b* = 00…01000.

(4) Das Additionsergebnis kommt nach *e:* *e* = 00…01010 (dezimal: 10)

Für die folgenden Ziffern werden im Wesentlichen diese vier Schritte wiederholt:

(1') Das Ergebnis *e* wird nach *b* und gleichzeitig wird die nächste Ziffer, nämlich die 9, von der Tastatur nach *a* gebracht. Die beiden Register sind jetzt folgendermaßen belegt: *a* = 00…01001, *b* = 00…01010.

(2') Ergebnis der Addition *a* + *b:* *e* = 00…010011 (dezimal: 19).

(3') Ergebnis der Shiftoperationen liefern *a* = 00…0100110, *b* = 00…010011000

(4') Ergebnis der Addition: *e* = 00…010111110 (dezimal: 190).

(1'') *a* = 0100, *b* = 00…010111110.

(2'') *e* = 00...011000010 (dezimal: 194).

(3``) *a* = 00...0110000100, *b* = 00...011000010000.

(4'') *e* = 00…011110010100 (dezimal : 1940).

(1''') *a* = 0001, *b* = 00…011110010100.

(2''') *e* = 00…011110010101 (dezimal: 1941).

Nun steht im Ergebnisregister *e* die Ganzzahldarstellung der eingegebenen Dezimalzahl.

Interne Zahlendarstellung und Normalisierung

Im Z3-Rechner werden Zahlen als 22-stellige Binärzeichen abgespeichert. Damit lassen sich 2^{22} = 4194304 ganze Zahlen darstellen. Ingenieure und Naturwissenschaftler, und inzwischen sogar Finanzleute, hantieren mit sehr großen Zahlen. Beispielsweise überschreitet bereits die Lichtgeschwindigkeit, ausgedrückt in Meter je Sekunde, den durch 22 Bits gegebenen Bereich ganzer Zahlen: 299 792 458.

Um mit nur wenigen Bits auch in den Bereich sehr großer und sehr kleiner Zahlen vorstoßen zu können, nutzte Konrad Zuse die halblogarithmischen Zahlendarstellung, auch Gleitkommadarstellung oder wissenschaftliche Notation genannt: $2{,}99792458\times10^{8}$. Für viele Anwendungen kommt man mit reduzierter Genauigkeit aus und rundet die Mantisse – das ist die Zahl mit dem Komma – auf beispielsweise vier Stellen: $2{,}998\times10^{8}$.

Wir haben gesehen, wie eine vierstellige Dezimalzahl vom Rechner in eine Folge von Binärzeichen gewandelt wird. Vierzehn Bits reichen dazu aus: 00011110010101. Diese vierzehn Bits werden als Mantisse interpretiert, mit dem Komma nach der ersten Stelle. Da das Komma zu diesem Zweck um dreizehn Stellen nach links verschoben werden muss, was einer Division durch 10^{1101} (dezimal 2^{13}) entspricht, muss diese Zahl dementsprechend mit diesem Faktor multipliziert werden: $00011110010101 = 0{,}0011110010101\times10^{1101}$.

Allerdings ist die Darstellung noch zu normalisieren: Das Komma muss hinter der ersten Eins stehen. Diese Normalisierung wird mit Hilfe eines Shifters („große Weiche“) durchgeführt, der zwischen *e*- und *b*-Register liegt. Der Shifter wird von einer Fünfbitvariablen gesteuert. Sie gibt die Zahl der Stellen an, um die der Inhalt des *e*-Registers nach links oder nach rechts zu verschieben ist. Das Ergebnis der Schiebeoperation steht im *b*-Register.

Die Anzahl der Verschiebungen nach links hängt von der Zahl der führenden Nullen im *e*-Register ab. Zur Ermittlung dieser Zahl nutzt Zuse wieder die Relaiskettenschaltungen, die ihm schon bei der Addition den Übertrag in einem einzigen Schritt ermöglichten. Auch das Verschieben gelingt in nur einem Taktschritt und nicht wie bei einem Schieberegister durch sequentielles „Herausschieben“ der Nullen. Für unser Zahlenbeispiel erhalten wir nach der Schiebeaktion

$$1{,}1110010101000\times 10^{1010}.$$

Zahlendarstellung für die Speicherung

Da die führende Eins der Mantisse immer da ist und daher keinerlei Information enthält, wird sie bei der Speicherung von Zahlen weggelassen. Abgespeichert werden

1. das Vorzeichen (1 Bit),
2. der Exponent (7 Bits) und
3. die Mantisse ohne führende Eins (14 Bits).

Das sind insgesamt 22 Bits. Unsere Beispielzahl 1941 führt zu dieser Speicherbelegung im Rechner:

1	0001010	11100101010000

Die erste Eins steht für das positive Vorzeichen der Zahl. (Heute ist es üblich, das positive Vorzeichen durch die Null und das negative durch die Eins zu kennzeichnen.)

Bisher war zu sehen, wie sich Zahlen mit einem Betrag größer oder gleich eins im Rechner darstellen lassen. Für Zahlen, die betragsmäßig kleiner als eins sind, muss es möglich sein, auch negative Exponenten im Rechner darzustellen. Für die Darstellung ganzer Zahlen, positiver wie negativer, verwendet Konrad Zuse konsequent die *Komplementdarstellung* bezüglich des gerade nicht mehr darstellbaren Wertes: Bei einer 7-Bit-Darstellung ist die Zahl 128 gerade nicht mehr darstellbar. Die Zahl -33 wird dann repräsentiert durch die Zahl 128-33 = 95.

Anstatt bei einer negativen Zahl erst den Wert der höchsten Stelle zu addieren (+64) und dann den Wert der nächsthöheren, gerade nicht mehr vorhandenen, Stelle zu subtrahieren (-128), kann man den Wert der höchsten Stelle auch abändern in -64. Positive und negative Zahlen werden bei dieser Darstellung einheitlich dadurch verwirklicht, dass man die Wertigkeit der höchsten Stellen mit negativem Vorzeichen versieht. Die Wertigkeiten der sieben Bits des Exponenten sind demnach die folgenden: -64, 32, 16, 8, 4, 2, 1. Auf diese Weise lassen sich mit den sieben Bits des Exponenten alle ganzen Zahlen von -64 bis +63 darstellen.

Die interne Zahl 0 1111110 10000000000000 ist, übersetzt in Dezimaldarstellung, gleich $-1\frac{1}{2} \times 2^{-2} = -3/8 = -0{,}375$. Man beachte, dass die führende Eins der Mantisse nicht mit abgespeichert ist.

Übungen für Fortgeschrittene und Ehrgeizige

1 Zeigen Sie, dass man das Komplement einer negativen Zahl *-z* dadurch bilden kann, dass man jedes Bit der positiven Zahl *z* durch sein *Einerkomplement* ersetzt ($0 \rightarrow 1$, $1 \rightarrow 0$) und

zum Ergebnis die Zahl 1, in 7-Bit-Darstellung also 0000001, addiert. Tipp: 128-*z* = 127+1-*z* = ($1111111_{\text{binär}}$ – z) + 1.

2 Zeigen Sie, dass sich die Subtraktion zweier Zahlen im Rechner auf Komplementbildung und Additionen zurückführen lässt.

3 Bei der Addition zweier Zahlen mit gleichem Vorzeichen kann es zu Bereichsüberschreitungen kommen. Dann ist das Ergebnis mit 7-Bits nicht mehr darstellbar. Das kann im Bereich der positiven Zahlen passieren (*Überlauf*, Overflow), oder auch im negativen Zahlenbereich (*Unterlauf*, Underflow). Wann kann das passieren?

Eingabe einer Dezimalbruches

Die Eingabe von Dezimalzahlen mit positivem (Dezimal-)Exponenten macht keine Schwierigkeiten. In diesem Fall muss die eingegebene vierstellige Dezimalzahl nur noch so oft mit dem Faktor 10 (dezimal) multipliziert werden, wie der Zehnerexponent angibt. Und wie das geht, haben wir oben bereits gesehen. Schwieriger wird die Sache für negative Exponenten, wenn also die eingegebene Zahl durch 10 (dezimal) zu teilen ist, eventuell mit Wiederholungen.

Die folgende Tabelle enthält die wichtigsten Schritte der Eingabe für das Beispiel 299,8. In den Rechenregistern für die Mantisse erscheint auch die führende Eins. Das Ergebnisregister *e* am Addiererausgang für die Mantisse hat eine weitere Vorkommastelle für einen bei Additionen und Multiplikationen möglichen Übertrag. Um Rundungsfehler zu reduzieren umfasst das Register 16 Nachkommastellen. Insgesamt werden also 18 Bits für die Darstellung der Mantisse aufgewendet. Erst beim Abspeichern wird die Zahl der Stellen auf 14 reduziert.

In der vorletzten Tabellenzeile steht das Ergebnis der Division durch 10 (dezimal). Diese Division wird mittels Multiplikation mit dem Faktor 0,1 (dezimal) erledigt. Dieser Faktor lässt sich im Rechner nicht exakt darstellen, denn er läuft auf einen periodischen Dualbruch hinaus. Näherungsweise wird der Binärwert 0,0001100110011001 genommen. Die Multiplikation geschieht in Teilschritten. Die Zahl wird nacheinander mit 1,1, 1,0001, 1,00000001 und 1,0000000000000001 multipliziert. Außerdem ist noch der Zweierexponent um 4 zu vermindern. Jede der Multiplikationen lässt sich durch Addition der Zahl selbst mit der um 4, 8 oder 16 Stellen nach rechts verschobenen Zahl bewerkstelligen.

In der Tabelle sind die Zeilen, in denen eine Normalisierung vorgenommen wird, grau unterlegt.

Jede Zeile der Tabelle erfüllt die folgende Bedingung: Die Summe aus interner Zahl (gegeben durch Mantisse und Zweierexponent) und externer Mantisse multipliziert mit der durch den Zehnerexponenten gegebenen Zehnerpotenz ist (bis auf Rundungsfehler) gleich dem Eingabewert 299,8. Eine Bedingung, die in jedem Berechnungsschritt erfüllt ist, nennt der Informatiker *Invariante*. In Kurzform lautet unsere Invariante so:

$$(\text{Interne Zahl} + \text{Dezimalmantisse}) \times 10^{\text{Zehnerexponent}} = 299{,}8.$$

Im Grunde wird sukzessive Ziffer für Ziffer der externen Darstellung in die interne Darstellung transportiert und schließlich wird noch der Zehnerexponent auf 0 gebracht, so dass die interne Zahlendarstellung komplett wird.

Invarianten sind dem Informatiker, was dem Naturwissenschaftler die Naturgesetze sind: Sie beschreiben die Gesetzmäßigkeiten von (Programm-)Abläufen.

Eingabe der Zahl 2998×10-1

Intern (binär)		Extern (dezimal)	
Mantisse	Zweierexponent	Mantisse	Zehnerexponent
+00,0000000000000000	0001101	2998	-1
+00,0000000000000000	0001101	2,998	2
+00,0000000000010000	0001101	0,998	2
+00,0000000010100000	0001101	9,98	1
+00,0000000011101000	0001101	0,98	1
+00,0000100100010000	0001101	9,8	0
+00,0000100101011000	0001101	0,8	0
+00,0101110101110000	0001101	8	-1
+00,0101110110110000	0001101	0	-1
+01,0111011011000000	0001011	0	-1
+10,0101011110011001	0000111	0	0
+01,0010101111001100	0001000	0	0

Ausgabe

Die folgende Tabelle zeigt das Prinzip der Ausgabe. Auch hier erfüllt jede Zeile die bereits beschriebene *Invariante:* Die Summe aus interner Zahl (Mantisse mal Zweierpotenz gemäß Zweierexponent) und externer Mantisse multipliziert mit der durch den Zehnerexponenten gegebenen Zehnerpotenz ist stets (ungefähr) gleich der auszugebenden Zahl.

Die Wandlung beginnt damit, dass die interne Zahl (gegebenenfalls mehrfach) mit zehn multipliziert oder durch zehn dividiert wird, bis sie einen Wert von wenigstens eins aber kleiner als 16 annimmt. Das ist dann der Fall, wenn der interne Exponent nichtnegativ und höchstens gleich 3 ist. Danach wird der interne Zweierexponent auf den Wert zwei gesetzt und die Mantisse dementsprechend verschoben, so dass der interne Wert gleich bleibt.

Intern (binär)		Extern (dezimal)		Anmerkungen
Mantisse	Zweier-exponent	Man-tisse	Zehner-exponent	
+01,0010101111001100	0001000		0	
+01,1101111110101011	0000100		1	
+01,0111111110111011	0000001		2	Bedingung für Wandlung erfüllt
+**00,10**11111111011101	0000010		2	Zweierexponent auf +2 gesetzt
+00,0011111111011101	0000010	2	2	
+**10,01**11111010100010	0000010	20	1	Fett markiert: Ganzzahlanteil
+00,0011111010100010	0000010	29	1	
+**10,01**11001001010100	0000010	290	0	
+00,0011001001010100	0000010	299	0	
+**01,11**11011101001000	0000010	2990	-1	
+00,0011011101001000	0000010	2997	-1	Interne Zahl ≥ ½.
		2998	-1	Deshalb: aufrunden

Nun ist der ganzzahlige Anteil der internen Zahl durch eine der Zahlen von 1 bis 15 darstellbar. Diese Ziffern werden auf die ersten beiden Spalten der Anzeige gebracht. Das ist übrigens der Grund dafür, dass den vier Anzeigepositionen mit den Ziffern 0 bis 9 noch eine fünfte Anzeige mit nur einer einzigen Ziffer, nämlich der 1, vorangestellt ist. Der interne Wert wird um den (externen) Anzeigewert vermindert, so dass die Invariante erhalten bleibt.

Nun werden die folgenden beiden Schritte dreimal wiederholt: Erstens wird die interne Mantisse mit zehn multipliziert und der externe Exponent entsprechend erhöht. Zweitens wird der Ganzzahlanteil der internen Zahl (er ist jetzt mit nur einer Dezimalziffer darstellbar) in das Anzeigeregister gebracht. Dementsprechend wird die interne Zahl vermindert.

Das fünfte Bit in der vorletzten Zeile zeigt an, ob die Zahl noch aufzurunden ist oder nicht. Zu diesem Zweck wird dieses Bit ebenfalls in das Ausgaberegister gebracht. Alle Bits, die nacheinander in dem Ausgaberegister erscheinen, sind in der Tabelle fett markiert.

Die Architektur der Z3 im Detail

Die letzte Seite zeigt ein Bild der vollständigen Architektur der Z3. Es ist dem von Raúl Rojas herausgegebenen Buch „Die Rechenmaschinen von Konrad Zuse“, Springer-Verlag, Berlin, Heidelberg 1998, entnommen (S. 54).

Die für den Programmierer relevanten Register des Rechenwerks sind die *f*- und die *b*-Register. Sie werden auch als R_1 und R_2 bezeichnet. Die *a*- und *e*-Register sind für das Aufbewahren von Zwischenergebnissen da. Im Schaltbild sind auch die Relais eingezeichnet, die die einzelnen Datenübertragungspfade wirksam machen.

Z3-Simulation

Ein Z3-Simulationsmodell ist im Internet unter der Adresse

http://www.zib.de/zuse/Inhalt/Programme/Simulationen/Z3_Sim/index.html

frei zugänglich. Autor des Programms und der Dokumentation ist Georg-Alexander Thurm. Das Simulationsprogramm erlaubt eine Beobachtung der einzelnen Rechenschritte bis in die Details.

Spezielle Zahlen und Ausnahmebehandlung

Es gibt spezielle Darstellungen für die Werte 0 und ∞. (Dafür sind der kleinste und der größte Exponent reserviert, -64 und 63.) Mit diesen Werten wird nach den in der reellen Analysis üblichen Regeln gerechnet. In diesem Punkt waren Zuses Rechner schon weiter fortgeschritten als manches Programmiersystem unserer Tage. (Noch heute gibt es Programme, die bei einer Division durch null nicht etwa den korrekten Wert ∞ liefern, sondern dieses Ereignis mit einem Programmabbruch quittieren.)

Durch die Rechnung kann es vorkommen, dass das Resultat den zur Verfügung stehenden Zahlenbereich verlässt. Zuse-Rechner fangen solche Über- oder Unterläufe ab und behandeln sie angemessen: Bei Überlauf oder Unterlauf des Zweierexponenten werden die Zahlenwerte auf ∞ oder 0 gesetzt (Exponent gleich 63 bzw. gleich -64). Außerdem gibt es Fälle, die kein Rechner zufriedenstellend lösen kann. Auch darauf haben Zuses Rechner die passende Antwort: Undefinierte Ausdrücke wie ∞-∞, 0×∞, 0/0 werden als Fehler gemeldet. Wir wollen uns ansehen, wie in den Computern gestern und heute ein Über- oder ein Unterlauf erkannt wird.

Für das Studium dieser Situationen ist es sinnvoll, im Exponenten das Bit mit der Wertigkeit --64 gesondert zu betrachten. Wir nennen es *v*. Es gibt uns an, ob die Zahl positiv ($v = 0$) oder ob sie negativ ist ($v = 1$). Die restlichen 6 Bits fassen wir zusammen und bezeichnen sie als *r*. Für das Register *a* sieht die Aufteilung beispielsweise so aus: $a = (v_a, r_a)$. Beispiele: Steht die Zahl 13_{dezimal} bzw. $0001101_{\text{binär}}$ im Register *a*, so ist $v_a = 0$ und $r_a = 13$. (Fortan wird vorzugsweise die Dezimalschreibweise benutzt und nicht weiter gekennzeichnet.) Steht die Zahl -11 im Register *b*, so ist $v_b = 1$ und $r_b = 64\text{-}11 = 53$.

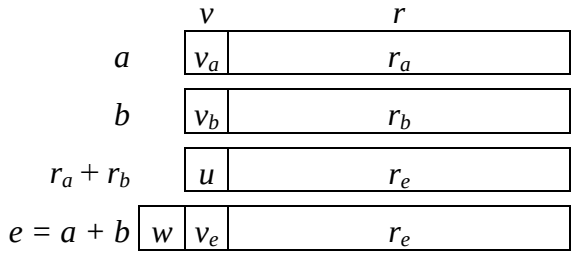
Das Komplement einer positiven Zahl *z* des zulässigen Zahlenbereichs ($0<z<64$) zu 64, lässt sich rechentechnisch leicht erzeugen: Man bildet zur Zahl *z* in Binärschreibweise das stellenweise Einerkomplement mit der Stellenzahl 6 und addiert eine Eins. Letztere nennt Konrad Zuse die „flüchtige Eins“. Für das Einerkomplement einer Zahl *z* schreiben wir ~*z*. Die folgenden Gleichungen machen die Zusammenhänge deutlich:

$$64 - z = (63 - z) + 1 = (111111_{\text{binär}} - z) + 1 = \sim z + 1.$$

Beispiel: $64-11 = \sim 11+1 = \sim 001011_{\text{binär}} + 1 = 110100_{\text{binär}} + 1 = 110101_{\text{binär}} = 53$.

Nun zur Addition zweier Zahlen. Die Summanden stehen in den Registern *a* und *b*, jeweils aufgeteilt auf das „Vorzeichen“ *v* und den „Rest“ *r*. Die Summenbildung geschieht über alle Stellen, einschließlich der „Vorzeichen“-Stelle, gerade so, als hätte die „Vorzeichen“-Stelle nicht die Wertigkeit -64 sondern +64. Da jeder der Summanden auch negativ sein darf, wird die Subtraktion in naheliegender Weise gleich mit erledigt. Das Ergebnis der Addition steht im Register *e*. Es hat eine Stelle mehr, da es bei der stellenweisen Addition zum Übertrag auf eine achte Stelle, hier als *w* bezeichnet, kommen kann.

Neben dem *w* ist noch das *u*-Bit von Bedeutung. Es ist das Übertragsbit, das sich ergibt, wenn man nur die Addition der *r*-Teile der Register *a* und *b* betrachtet: $(u, r_e) = r_a + r_b$.



Unter welchen Bedingungen der Addierer auch negative Zahlen richtig behandelt und wie man Überläufe erkennen kann, erfordert ein genaues Studium der Übertragsbits *u* und *w*. Aus der Tatsache, dass der Addierer über alle Bits des Registers richtig addiert ($e=a+b$), folgt:

$$(w, v_e) = v_a + v_b + u.$$

Anders ausgedrückt: v_e ist das Summenbit ohne Übertrag und *w* ist das Übertragsbit der Summe der Werte v_a, v_b und *u*. Die folgenden Tabelle zeigt die möglichen Resultate der Addition.

Reg. *a*	**Reg. *b***	**Bedingung**	*w*	v_e	*u*	**Auswirkung**
$0 \le x < 64$	$0 \le y < 64$	$0 \le x+y < 64$	0	0	0	Korrektes nichtnegatives Ergebnis
$0 \le x < 64$	$0 \le y < 64$	$64 \le x+y$	0	1	1	Überlauf
$0 \le x < 64$	$-64 \le y < 0$	$-64 \le x+y < 0$	0	1	0	Korrektes negatives Ergebnis
$0 \le x < 64$	$-64 \le y < 0$	$0 \le x+y < 64$	1	0	1	Korrektes nichtnegatives Ergebnis
$-64 \le x < 0$	$0 \le y < 64$	$-64 \le x+y < 0$	0	1	0	Korrektes negatives Ergebnis
$-64 \le x < 0$	$0 \le y < 64$	$0 \le x+y < 64$	1	0	1	Korrektes nichtnegatives Ergebnis
$-64 \le x < 0$	$-64 \le y < 0$	$-64 \le x+y < 0$	1	1	1	Korrektes negatives Ergebnis
$-64 \le x < 0$	$-64 \le y < 0$	$x+y < -64$	1	0	0	Unterlauf

Fazit: Bereichsüberschreitungen (Überlauf oder Unterlauf) sind eindeutig daran zu erkennen, dass die Bits *u* und *w* verschieden voneinander sind.

Der Minusoperator: Sei *x* eine in unserem System darstellbare positive oder negative Zahl: $x = (v, r)$. Die Zahl $-x$ wird durch $(\sim v, \sim r+1) = (\sim v, \sim r) +1$ repräsentiert, falls dieser Wert in unserem System darstellbar ist. Der Minusoperator wird also dadurch realisiert, dass man das Einerkomplement über sämtliche Stellen einschließlich des „Vorzeichens“ bildet und dann eine Eins addiert. Für positive Zahlen *x*, also für Werte von 1 bis 63, folgt dieser Zusammenhang unmittelbar aus der Zahlendarstellung positiver und negativer Zahlen. Die Null nehmen wir heraus, da ja -0=0 gilt. Sei nun *x* eine negative in unserem System darstellbare Zahl: $x = (v, r) = (1, 64+x)$, dann liefert die Anwendung des Minusoperators die Darstellung $(0, \sim(64+x)+1) = (0, 64-(64+x)) = (0, -x)$. Das ist eine korrekte Darstellung des positiven Wertes $-x$, wenn *x* größer als -64 ist. Da -(-64) = 64 in unserem System nicht existiert, ist nicht zu erwarten, dass die Anwendung des Minusoperators auf die Zahl -64 den gewünschten Effekt hat.

Spektakuläre Unfälle aufgrund der Computerarithmetik

Im THEMA-Heft 2/2006 der Hochschule Fulda berichtete ich unter dem Stichwort Oberflächenkompetenz von der Forderung, jedem Schüler ein Notebook zur Verfügung zu stellen. Klaus Haefner von der Universität Bremen begründete dies damit, dass Lesen, Sprechen, Kreativ-Sein, Innovationsfähigkeit, Organisieren-Können, Solidarisch-Sein typisch menschliche Qualifikationen seien. Für den Rest, nämlich die „kognitive Sklavenarbeit", habe man das „Denkzeug". Sein Fazit: „Schreiben und Rechnen wird nicht mehr gebraucht. Sprechen und Lesen reicht."

Dagegen wandte ich ein: „Schüler, die nicht mehr schriftlich mit Zahlen umgehen, verlieren den Begriff der Zahl. Diese Menschen werden später dem Computer hilflos ausgeliefert sein und ihn nicht oder falsch verstehen. Der Rechner kann nämlich aus prinzipiellen Gründen die Welt nie eins zu eins abbilden. Die Computerarithmetik weicht von unseren mathematischen Vorstellungen ab. Programmierfehler sind allgegenwärtig. Modellierungsfehler auch. Die Bedienoberflächen stecken voller Fallen."

Einige spektakuläre Unfälle lassen sich auf die Computerarithmetik zurückführen. Manchen Reinfall habe ich selbst erlebt. Die folgenden Beispiele sollen eindringlich vor Augen führen, wie wichtig es auch für die Anwender ist, sich mit Computerarithmetik zu befassen. Die Z3-Modelle sind bestens geeignete Studienobjekte.

Jungfernflug der Ariane 5 scheiterte aufgrund eines Überlauffehlers

Am 4. Juni 1996 scheiterte der Jungfernflug der europäischen Trägerrakete Ariane 5. Auslösendes Ereignis war die Bereichsüberschreitung einer Variablen innerhalb der Software an Bord der Rakete etwa 30 Sekunden nach dem Start. Das fehlerauslösende Software-Modul war praktisch unverändert von der Ariane 4 übernommen worden. Die anstelle der Flugdaten interpretierten Diagnosedaten bewirkten eine Maximalauslenkung der Steuerdüsen. Durch die hohe Belastung zerbrach die Rakete.

Unvollständige Situationserfassung führte die Voyager 2 fast in die Irre

Die Raumsonde Voyager 2 verirrte sich fast, als sie im Dezember 1985 am Planeten Uranus vorbeiflog. Das Programm hatte unter anderem auch fortlaufend den Schätzwert der Uranusmasse zu korrigieren. Die Anfangsschätzung der Planetenmasse lag um 0,3 % neben dem wahren Wert; und das war für das Berechnungsverfahren zu viel. Es konvergierte gegen ein lokales Optimum und nicht gegen das globale. Das Problem konnte rechtzeitig gelöst werden.

Harmlose Populationsgleichungen lassen Rundungsfehler explodieren

Im Praktikum zum Thema Umweltsimulation fällt einer Arbeitsgruppe ein Numerikproblem auf. Das Arbeitsblatt zur ökologischen Simulation des Falke-Taube-Spiels liefert bei bestimmten Parameterkonstellationen offensichtlich falsche Ergebnisse. Die Populationsanteile entwickeln sich zunächst erwartungsgemäß. Später aber bricht die Sache zusammen; die Populationsanteile addieren sich nicht mehr auf 100%. Die Analyse des Problems ergibt, dass aufgrund der Rundung ein exponentiell wachsender Fehler entsteht.

Die vollständige Architektur der Z3