FH-Report

Marcel Beuler

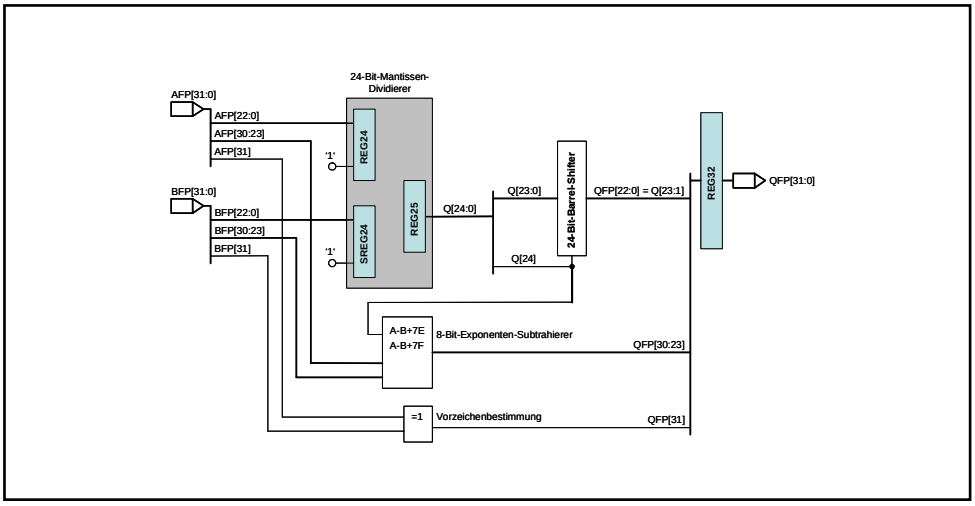
Realisierung von Arithmetik-Baugruppen für das 32-Bit-Gleitkommaformat der Norm ANSI/IEEE 754 mittels VHDL

FACHHOCHSCHULE
GIESSEN
FRIEDBERG

Dipl.-Ing.(FH) Marcel Beuler
Fachhochschule Gießen-Friedberg
FB Elektro- und Informationstechnik
Wiesenstraße 14

35390 Gießen

Realisierung von Arithmetik-Baugruppen für das 32-Bit-Gleitkommaformat der Norm ANSI/IEEE 754 mittels VHDL



Version 1.1

Vorwort

Die in Vorlesungen zur Digitaltechnik üblicherweise behandelten Arithmetik-Schaltungen beschränken sich meist auf den Halb- und Volladdierer sowie der Kaskadierung der Volladdierer zum Paralleladdierer, wobei der Übertrags-Vorausberechnung (Carry-Look-Ahead) zur Minimierung der Additionszeit eine besondere Bedeutung zukommt. Seltener wird auf Berechnungsmethoden für die Multiplikation und Division eingegangen. All diese arithmetischen Basisverknüpfungen werden nicht nur für den Sonderfall der ganzen Zahlen, sondern in der Praxis vorwiegend für reelle Zahlen benötigt (diese entstehen ohnehin meist bei der Division). Hierfür gibt es zwei grundlegende Zahlendarstellungen: Festkomma- und Gleitkommazahlen.

Diese Zusammenfassung beinhaltet die Darstellung von 32-Bit-Gleitkommazahlen nach dem ANSI/IEEE-Standard 754 und die daraus ableitbaren Konstruktionsprinzipien (Steuerwerk, Rechenwerk) für jede Gleitkomma-operation, wobei Addition und Subtraktion gemeinsam als vorzeichenbehaftete Addition betrachtet werden. Eine Besonderheit stellt die Umsetzung der Zustandsgraphen und Funktionsschaltungen in VHDL-Quellcode dar, wodurch sich eine logische Verifikation mit dem frei verfügbaren Simulator ModelSim XE Starter der Fa. Mentor Graphics durchführen lässt. Zwar ist in [Hoff93] im Rahmen der “Mikroalgorithmen und Rechenwerke für die Grundrechenarten“ entsprechender Quellcode in der Sprache HDL aufgeführt, jedoch hat sich VHDL mittlerweile im deutschsprachigen Raum durchgesetzt.

Auch bei der Gleitkomma-Multiplikation und –Division werden Algorithmen zur Multiplikation und Division ganzer Zahlen benötigt. Um das Hauptaugenmerk auf der Gleitkommaarithmetik zu halten, kommen nur serielle Verfahren zum Einsatz. Der Hardwareaufwand ist hier wesentlich geringer, allerdings auf Kosten einer höheren Berechnungszeit.

Somit kann mit der vorliegenden Zusammenfassung die grundlegende Arbeitsweise einer Gleitkomma-Arithmetikeinheit (Floating-Point Unit) relativ einfach verstanden werden. Um eine in der Praxis einsetzbare Floating-Point Unit zu erhalten, müssen noch Optimierungen hinsichtlich der Berechnungszeit durch parallele Algorithmen sowie Pipelining vorgenommen und die hier nicht behandelten Themen Bereichsunter- und –überschreitung sowie Rundung berücksichtigt werden. Zur Erzielung einer höheren Genauigkeit kann man auf das 64-Bit-Format wechseln. Das hier bereitgestellte Grundlagenwissen bildet eine gute Basis für diese Ergänzungen.

Giessen, im April 2008 M. Beuler

Inhaltsverzeichnis

1 Einführung

1.1 Vorzeichen-Betrag-Darstellung

Eine intuitive Darstellung für positive und negative Zahlen ist die Vorzeichen-Betrag-Darstellung. Dabei wird die Zahl x durch ein Vorzeichenbit S_x und den Betrag |x| dargestellt. Nach [Hoff93] wird S_x=0 für positive x-Werte und S_x=1 für negative x-Werte vereinbart. Eine positive n-stellige Dualzahl ergibt sich aus der Summe der gewichteten Ziffern a_i:

$$X = \sum_{i=0}^{n-1} a_i \cdot 2^i \tag{1.1}$$

Eine positive Dualzahl X mit Vorzeichenbit soll Vorzeichenzahl (S_x, X) heißen:

$$X = |x| \qquad S_x = \begin{cases} 0 & \text{für } x \geq 0 \\ 1 & \text{für } x \leq 0 \text{ bzw. } x < 0 \end{cases} \tag{1.2}$$

$$x = \left(1 - 2 \cdot S_x\right) \cdot X \tag{1.3}$$

Bei n Stellen für den Betrag lassen sich alle Werte $|x| \leq 2^{n-1}$ darstellen. Die Null kann dabei als “positive Null“ (0,0) und als “negative Null“ (1,0) dargestellt werden. Wenn man die negative Null verbietet, dann gestalten sich die Addition und Subtraktion schwieriger.

1.2 Zweierkomplement-Darstellung

Bei der Zweierkomplement-Darstellung für positive und negative Zahlen wird das höchstwertige Bit als Vorzeichen interpretiert. Bei positiven Zahlen ist es 0, bei negativen Zahlen 1. Der Wert einer n-stelligen Zweierkomplementzahl X ergibt sich aus der Summe der gewichteten Ziffern a_i, wobei das Vorzeichenbit a_{n-1} mit negativem Gewicht einbezogen wird [Flik98]:

$$X = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i \tag{1.4}$$

Beispiel: $10111101b = -1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

Das heißt, bei positivem Vorzeichen a_{n-1}=0 ist der Zahlenwert gleich dem Wert, den die n-1 verbleibenden Bits als vorzeichenlose Zahl haben. Bei negativem Vorzeichen a_{n-1}=1 ist er gleich dem Wert, den die n-1 verbleibenden Bits als vorzeichenlose Zahl haben, jedoch um die Größe des halben Wertebereichs 2^{n-1} in den negativen Zahlenraum verschoben. Die Umrechnung einer positiven in die entsprechende negative Zahl oder umgekehrt (Komplementbildung) erfolgt durch Invertierung aller n Bits und anschließende Addition von Eins.

Beispiel 1.1:

```
+67d = 01000011b     Invertierung:   10111100
                     Addition:     +        1
                                    10111101b = -67d
```

Tabelle 1.1 zeigt positive und negative Dualzahlen mit jeweils n=3 Betragsbits und einem Vorzeichenbit in Vorzeichen-Betrag- sowie Zweierkomplement-Darstellung.

Dezimalzahl	Vorzeichen und Betrag	Zweierkomplement
7	0111	0111
6	0110	0110
5	0101	0101
4	0100	0100
3	0011	0011
2	0010	0010
1	0001	0001
+0	0000	0000
-0	1000	(0000)
-1	1001	1111
-2	1010	1110
-3	1011	1101
-4	1100	1100
-5	1101	1011
-6	1110	1010
-7	1111	1001
-8		1000

Tabelle1.1: Darstellungsarten positiver und negativer Dualzahlen

Eine bemerkenswerte Eigenschaft der Zweierkomplement-Darstellung ist, dass nur ein Ausdruck zur Darstellung der Zahl Null existiert, nämlich 000…0 [SeBe98].

1.3 Gleitkommazahl

Man unterscheidet zwei grundlegende Zahlendarstellungen: Festkommazahlen (fixed-point numbers) und Gleitkommazahlen (floating-point numbers). Festkommazahlen werden durch eine binäre Zahl mit Maschinenwortbreite n dargestellt, bei der das Komma immer an der gleichen Stelle d angenommen wird [ScSc99]. Mit dieser Zahlendarstellung können insgesamt 2^n verschiedene Werte dargestellt werden, die je nach Position des Kommas einen mehr oder weniger großen Bereich auf der Zahlenachse erfassen. Bei der Zweierkomplement-Darstellung enthält der Wertebereich sowohl positive als auch negative Zahlen.

Beispiel 1.2:
Ist n=8 und d=0 (Komma rechts daneben), so werden folgende Wertebereiche dargestellt:

```
Dual-Darstellung:                0...255
Zweierkomplement-Darstellung:  -128...127
```

Der wertmäßige Abstand a zweier benachbarter Binär-Darstellungen beträgt 1. Dieser Abstand halbiert sich, wenn wir d=1 wählen:

```
Dual-Darstellung:                0,0.5,...,127,127.5
Zweierkomplement-Darstellung:  -64,-63.5,...,63,63.5
```

Man erkennt, dass durch Verschiebung des Kommas der darstellbare Wertebereich verringert wird und die Auflösung a steigt. Die Verschiebung des Kommas nach links bedeutet eine Division durch 2^d. Allgemein gilt $a=2^{-d}$ und der Wert x der dargestellten Zahl liegt in folgenden Bereichen:

Dual-Darstellung: $$0 \le x \le 2^{n-d} - a \quad (1.5a)$$

Zweierkomplement-Darstellung: $$-2^{n-d-1} \le x \le 2^{n-d-1} - a \quad (1.5b)$$

Festkommazahlen decken einen festen und relativ kleinen Wertebereich ab. Arithmetische Operationen können wie mit ganzen Zahlen ausgeführt werden. Rechenwerke für Festkommazahlen sind dementsprechend wenig aufwendig. Jedoch kann es aufgrund des eingeschränkten Wertebereichs schnell zu Bereichsüberschreitungen kommen, weshalb eine Festkommaarithmetik nur in wenigen Prozessoren vorhanden ist [Flik93].

Bei einer Gleitkommazahl werden zusätzliche Bits verwendet, die die Stellung des Kommas angeben. Eine Gleitkommazahl, die den Wert x darstellt, besteht aus der Mantisse mx und dem Exponenten ex zur Basis b. Man spricht in diesem Zusammenhang auch von einer halblogarithmischen Darstellung. Wir setzen hier stets die duale Basis b=2 voraus. Nach [Hoff93] gilt somit:

$$x = mx \cdot b^{ex} = mx \cdot 2^{ex} \tag{1.6}$$

Der Vorteil der Gleitkommazahl gegenüber der Festkommazahl besteht darin, dass mit der gleichen Wortbreite n ein wesentlich größerer Wertebereich dargestellt werden kann. Somit werden bei arithmetischen Operationen nur selten die Bereichsgrenzen (Überlauf, Unterlauf) überschritten. Ein Nachteil der Gleitkommazahlen ist der wesentlich höhere Verarbeitungsaufwand bei arithmetischen Operationen durch die getrennte Verarbeitung von Mantisse und Exponent.

Gleitkommazahlen müssen stets normalisiert werden, da es für einen bestimmten Wert keine eindeutige Darstellung gibt. So kann z.B. 0.1 dargestellt werden als $0.1=0.01 \cdot 10^{1}=100 \cdot 10^{-3}$. Eine einheitliche Form der Gleitkomma-Darstellung wird erst durch die Normalisierung erreicht, die die Komma-Position der Mantisse so fixiert, dass das Komma ganz links steht und die erste Ziffer der Mantisse ungleich 0 ist. Alle Darstellungen, die mit 0 hinter dem Komma beginnen, sind verboten. Die normalisierte Darstellung mit Berücksichtigung des Vorzeichens der Gleitkommazahl lautet:

$$x = S_x \cdot mx_{norm} \cdot 2^{ex} \tag{1.7}$$

Hierbei ist mx_{norm} die positive normalisierte Mantisse und S_x das Vorzeichen (+1,-1). Damit die Mantisse keine führende Null hat, muss bei ihrer Darstellung durch eine r-stellige binäre Zahl folgende Normalisierungsbedingung erfüllt werden:

$$0.5 \le mx_{norm} \le 1 - 2^{-r} < 1 \tag{1.8}$$

In [Hoff93] wird zusätzlich noch eine Konstante 2^k definiert, die die normalisierte Mantisse bewertet und damit eine andere Kommastellung für die Mantisse $\overline{mx}_{norm}$ definiert:

$$x = S_x \cdot mx_{norm} \cdot 2^{k} \cdot 2^{ex-k} = S_x \cdot \overline{mx}_{norm} \cdot 2^{ex-k} \tag{1.9}$$

Für k=0 ist $\overline{mx}_{norm} = mx_{norm}$. Mit k=1 erhält man die Kommastellung des IEEE-Gleitkommaformats (siehe nächsten Abschnitt), d.h. das Komma steht nicht vor, sondern hinter der höchstwertigen 1. In diesem Falle gilt für den Wert der normalisierten, r-stelligen Mantisse:

$$1 \le \overline{mx}_{norm} \le 2 - 2^{-r} < 2 \tag{1.10}$$

1.3.1 ANSI/IEEE-Standard 754

Zur Vereinheitlichung der Gleitkommaverarbeitung wurde der Standard ANSI/IEEE 754 erarbeitet und 1985 veröffentlicht. Er definiert u.a. mehrere Datenformate, von denen hier exemplarisch die 32-Bit-Gleitkomma-Darstellung herausgegriffen werden soll. Weitere Datenformate sind die 64-Bit- und die 80-Bit-Gleitkomma-Darstellung. Das 32-Bit-Wort wird in ein Vorzeichenbit S_x, eine 8-Bit Charakteristik EX und eine 23-Bit Mantisse aufgeteilt, siehe Bild 1.1. Die 32-Bit-Gleitkomma-Darstellung hat folgende Form:

$$x = (-1)^{S_x} \cdot (1.FX) \cdot 2^{EX-127} \tag{1.11}$$

Das Vorzeichen S_x ist 0 für positive Zahlen und 1 für negative Zahlen.

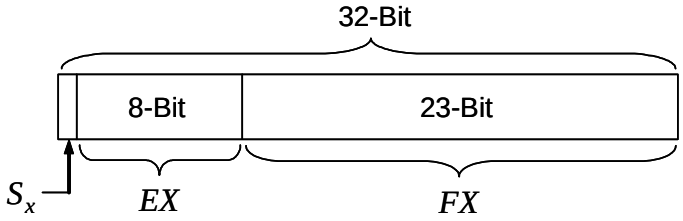


Bild 1.1: Darstellung des ANSI/IEEE-32-Bit-Gleitkommaformats

Die Mantisse MX=(1.FX) wird durch Erhöhen/Vermindern des Exponenten auf Werte zwischen 1 und 2 normalisiert, genauer $1 \leq MX \leq 2-2^{-23}$ (siehe Gl. (1.10)). Da die Stelle vor dem Komma immer gleich 1 ist (sog. hidden Bit), werden nur die 23 Binärstellen FX hinter dem Komma (fractional part) im Datenformat gespeichert. Dieser veränderliche Teil der Mantisse kann die Werte $0 \leq FX \leq 1-2^{-23}$ annehmen. Die um das hidden Bit erweiterte 24-Bit Mantisse wird nach [ScSc99] auch als Signifikand bezeichnet. Gl. (1.11) lässt sich auch wie folgt darstellen:

$$x = (-1)^{S_x} \cdot 2^{EX-127} \cdot \left(1.0 + \sum_{i=1}^{23} FX_{-i} \cdot 2^{-i} \right) \tag{1.12}$$

Hierbei bezeichnet FX_{-i} die Belegung der einzelnen Bits der Mantisse nach dem führenden Komma, d.h. dem niederwertigsten Bit im 32-Bit Wort entspricht FX_{-23}.

Darstellung der Null:

Bild 1.2 zeigt den insgesamt zur Verfügung stehenden Zahlenbereich für eine 32-Bit-Gleitkomma-Darstellung. Er besteht aus zwei Teilbereichen, einem für negative und einem für positive normalisierte Zahlen. Aufgrund der Normalisierungsbedingung gibt es jedoch keine normalisierte Darstellung für die Null. Die kleinste normalisierte Zahl ergibt sich dann, wenn der Exponent ex seinen größten negativen Wert annimmt, die führende Ziffer der Mantisse (hidden Bit) 1 ist und der veränderliche Teil der Mantisse den Wert 0 hat. Die dadurch dargestellten Werte können positiv oder negativ sein und liegen symmetrisch um den "wahren" Nullpunkt [ScSc99].

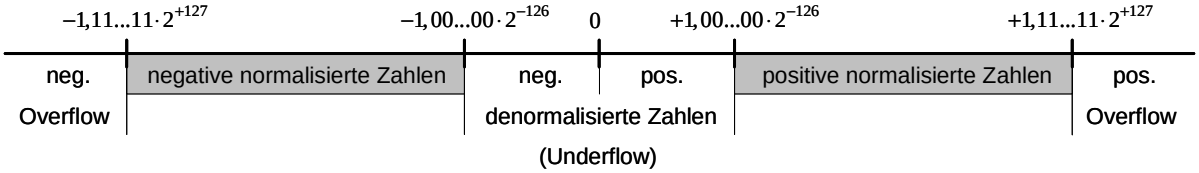


Bild 1.2: Gesamter Zahlenbereich der 32-Bit-Gleitkomma-Darstellung einschließlich Underflow und Overflow

Die Null wird daher in Datenformat als eine Zahl dargestellt, die kleiner ist als die kleinste Zahl aus dem zulässigen Wertebereich. Die Darstellung des zunächst vorzeichenbehafteten Exponenten ex erfolgt als sog. Charakteristik EX (biased exponent), d.h. zu ihm wird, ausgehend von seinem Zweierkomplement, ein Basiswert (bias=127) addiert, so dass sich eine vorzeichenlose Zahl EX ergibt:

$$\begin{aligned} EX &= ex + 127 \qquad \text{für } ex = -126,...,+127 \\ ex &= EX - 127 \qquad \text{für } EX = 1,...,254 \end{aligned} \tag{1.13}$$

Werte, die als Ergebnis einer Gleitkommaoperation zwischen der Null und der kleinsten positiven oder größten negativen normalisierten Zahl liegen, gelten als Bereichsunterschreitung (Underflow). Sie werden durch das Rechenwerk entweder näherungsweise zu Null gesetzt (EX=0) oder als denormalisierte Zahlen ohne führende 1 angegeben. Ergebnisse, die größer als die größte positive oder kleiner als die kleinste negative Zahl sind, gelten als Bereichsüberschreitung (Overflow). Sie werden im Datenformat durch EX=255 als $\pm\infty$ ausgewiesen. Diese vorzeichenlose Darstellung sowie die in Bild 1.1 gezeigte Unterbringung der Charakteristik im linken und der normalisierten Mantisse im rechten Teil des Datenwortes ermöglicht es auch, positive Werte direkt miteinander zu vergleichen

(Größer/Kleiner-Relation). Tabelle 1.2 fasst die gültigen und ungültigen Exponenten EX noch einmal zusammen.

ex	EX	Interpretation
128	255 = 1111 1111	kein gültiger Exponent; zur Darstellung von ∞ und Kontrollcodes
127	254 = 1111 1110	
126	253 = 1111 1101	
…		gültige Exponenten
0	127 = 0111 1111	für normalisierte
-1	126 = 0111 1110	Gleitkommazahlen
…		
-126	1 = 0000 0001	
-127	0 = 0000 0000	kein gültiger Exponent; zur Darstellung von Null bzw. kleiner Zahl

Tabelle 1.2: Gültige und ungültige Exponenten EX

Zur Darstellung von Null und Unendlich, von denormalisierten Zahlen und sog. Not-a-Numbers (NaNs) über den größten bzw. kleinsten Wert der Charakteristik wird in Zusammenhang mit der Mantisse folgende Konvention getroffen:

- `Null` $(-1)^{S_x} \cdot 0$ $EX = 0, FX = 0$
- `denormalisiert` $(-1)^{S_x} \cdot (0.FX) \cdot 2^{-126}$ $EX = 0, FX \neq 0$
- `normalisiert` $(-1)^{S_x} \cdot (1.FX) \cdot 2^{EX-127}$ $0 < EX < 255$
- `unendlich` $(-1)^{S_x} \cdot \infty$ $EX = 255, FX = 0$
- `Not-a-Number` `---` $EX = 255, FX \neq 0$

Not-a-Numbers erlauben keine mathematische Interpretation. Mit ihnen lassen sich z.B. Variablen als nichtinitialisiert kennzeichnen [Flik98]. Das 32-Bit-Gleitkommaformat weist folgenden normalisierten Wertebereich für den Betrag von x auf (dual und dezimal, letztere gerundet):

$$1.0 \cdot 2^{-126} \leq |x| \leq (1.0 + 1.0 - 2^{-23}) \cdot 2^{127} \approx 2^{128}$$
$$1.175 \cdot 10^{-38} \leq |x| \leq 3.4 \cdot 10^{38} \qquad (1.14)$$

Bei einer 32-Bit Zweierkomplement-Darstellung mit d=0 erhält man dagegen nur folgenden Wertebereich für den Betrag von x (ebenfalls gerundet):

$$0 \leq |x| \leq 2.15 \cdot 10^{9} \qquad (1.15)$$

Der darstellbare Wertebereich ist also bei Gleitkomma-Darstellung erheblich größer. Bei der Festkomma-Darstellung wird jedoch eine höhere Auflösung (absolute Genauigkeit) erreicht, da mit 32 Bit $2^{32} \approx 4.3 \cdot 10^{9}$ verschiedene Einzelwerte innerhalb eines festen Wertebereichs darstellbar sind. Dagegen kann bei der Gleitkomma-Darstellung mit der Charakteristik zwischen verschiedenen 2er-Potenz-Bereichen umgeschaltet werden. Innerhalb eines 2er-Potenz-Bereichs lassen sich jedoch wegen der kleineren 23-Bit Mantisse nur $2^{23} \approx 8.4 \cdot 10^{6}$ Einzelwerte unterscheiden. Diese relative Genauigkeit ist über den gesamten Zahlenbereich hinweg konstant. Da sich die 2er-Potenz-Bereiche mit steigendem Wert der Charakteristik vergrößern, vergrößern sich auch die Abstände der Zahlen in ihnen, weshalb die absolute Genauigkeit der Zahlen mit steigendem Wert der Charakteristik abnimmt.

Bei der 32-Bit-Gleitkomma-Darstellung unterscheidet sich die größte Zahl (alle Stellen der Mantisse sind 1) von der zweitgrößten Zahl (letzte Stelle der Mantisse ist eine 0) um folgenden Wert [ScSc99]:

$$2^{127} \cdot (2 - 2^{-23}) - 2^{127} \cdot (2 - 2^{-22}) = 2^{128} - 2^{104} - 2^{128} + 2^{105} = 2^{105} - 2^{104} \approx 2 \cdot 10^{31}$$

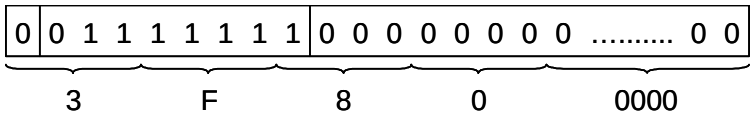
Im kleinsten 2er-Potenz-Bereich beträgt der minimale Unterschied zwischen zwei Zahlen dagegen:

$$(1.0 + 2^{-23}) \cdot 2^{-126} - 1.0 \cdot 2^{-126} = 2^{-149} \approx 1.4 \cdot 10^{-45}$$

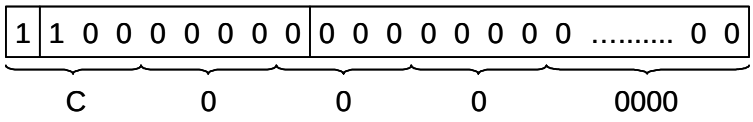
Beispiel 1.3:
Folgende Dezimalzahlen sollen in das 32-Bit-Gleitkommaformat konvertiert werden:

$x_1 = 1.0_{10}$; $x_2 = -2.0_{10}$; $x_3 = 0.21484375_{10}$

a) $x_1 = 1.0_{10} \Rightarrow x_1 = 1.0_2 = 1.0 \cdot 2^0$; $ex = 0 \Rightarrow EX = 127$; $S_x = 0$

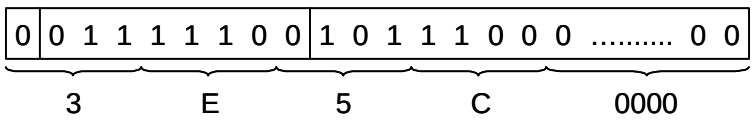


b) $x_2 = -2.0_{10} \Rightarrow x_2 = -10.0_2 = -1.00 \cdot 2^1$; $ex = 1 \Rightarrow EX = 128$; $S_x = 1$



c) $x_3 = 0.21484375_{10} = 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} \Rightarrow x_3 = 0.00110111_2 = 1.10111 \cdot 2^{-3}$

$ex = -3 \Rightarrow EX = 124$; $S_x = 0$



2 Gleitkommaarithmetik-Baugruppen

2.1 Addition von Gleitkommazahlen

Im Gegensatz zur Addition von Festkommazahlen ist die Addition von Gleitkommazahlen wesentlich komplizierter, weil die Exponenten und Mantissen getrennt behandelt werden müssen [Hoff93]. Gesucht ist die normalisierte Mantisse ms und der Exponent es folgender Summe:

$$s = a + b = ms \cdot 2^{es} = ma \cdot 2^{ea} + mb \cdot 2^{eb} \tag{2.1}$$

Bevor die Addition der mit dem hidden Bit auf 24 Stellen vervollständigten Mantissen vorgenommen werden kann, muss der kleinere Exponent an den größeren angepasst werden. Dazu muss die Mantisse, die zu dem kleineren Exponenten gehört, um so viele Stellen nach rechts geschoben werden, wie die Differenz der Exponenten beträgt:

$$ms \cdot 2^{es} = \begin{cases} \left(ma + mb \cdot 2^{-(ea-eb)}\right) \cdot 2^{ea} & eb \le ea \\ \left(mb + ma \cdot 2^{-(eb-ea)}\right) \cdot 2^{eb} & ea \le eb \end{cases} \tag{2.2}$$

Aufgrund dieser Rechtsverschiebung wird ein 24-Bit-Barrelshifter benötigt, der eine 24-Bit-Mantisse von 0 bis zu 23 Bitstellen nach rechts verschieben kann. Das VHDL-Modul SHFT24R des Barrelshifters als kombinatorische Lösung ist nachfolgend dargestellt [Jork04].

```
entity SHFT24R is
  port(X: in std_logic_vector(23 downto 0);
       COUNT: in std_logic_vector(7 downto 0);
       SB_1,SB_2: in std_logic;   -- signed bits
       Y: out std_logic_vector (23 downto 0));
end SHFT24R;

architecture SHFT24R_A of SHFT24R is
signal M: std_logic_vector(23 downto 0);
signal SELECTOR: std_logic_vector(1 downto 0);
begin
  SELECTOR <= SB_1 & SB_2;
  with COUNT select
    M <=      X               when X"00",
         '0' & X(23 downto  1) when X"01",
         "00" & X(23 downto  2) when X"02",
         "000" & X(23 downto  3) when X"03",
         "0000" & X(23 downto  4) when X"04",
         "00000" & X(23 downto  5) when X"05",
         "000000" & X(23 downto  6) when X"06",
         "0000000" & X(23 downto  7) when X"07",
         "00000000" & X(23 downto  8) when X"08",
         "000000000" & X(23 downto  9) when X"09",
         "0000000000" & X(23 downto 10) when X"0A",
         "00000000000" & X(23 downto 11) when X"0B",
         "000000000000" & X(23 downto 12) when X"0C",
         "0000000000000" & X(23 downto 13) when X"0D",
         "00000000000000" & X(23 downto 14) when X"0E",
         "000000000000000" & X(23 downto 15) when X"0F",
         "0000000000000000" & X(23 downto 16) when X"10",
         "00000000000000000" & X(23 downto 17) when X"11",
         "000000000000000000" & X(23 downto 18) when X"12",
         "0000000000000000000" & X(23 downto 19) when X"13",
         "00000000000000000000" & X(23 downto 20) when X"14",
         "000000000000000000000" & X(23 downto 21) when X"15",
         "0000000000000000000000" & X(23 downto 22) when X"16",
         "00000000000000000000000" & X(23)           when X"17",
         "000000000000000000000000"                   when others;

  with SELECTOR select
    Y <= ((not M) + 1) when "10",
         M when others;
end SHFT24R_A;
```

Die Anzahl der Rechtsverschiebungen wird mit dem Signalvektor COUNT[7:0] vorgegeben. Ist COUNT[7:0] > 23, dann soll die 24-Bit-Mantisse den Wert X“000000“ annehmen. Analog zu [Jork04] sollen beide Operanden über einen eigenen Barrelshifter verfügen. Die Verschiebung wird für den größeren Operanden auf “00000000“ und für den kleineren Operanden auf die Differenz der Exponenten eingestellt. Zusätzlich müssen die Vorzeichen der beiden Operanden berücksichtigt werden. Sind beide Operanden gleich (positiv oder negativ), dann können die Beträge der Mantissen einfach addiert werden. Ist einer der beiden Operanden negativ, so erfolgt für diesen nach der eigentlichen Verschiebeoperation noch eine Komplementbildung.

Bild 2.1 zeigt die Struktur der kombinatorischen Logik für die Gleitpunkt-Addition. Die 23-Bit-Mantissen der Operanden werden an der werthöchsten Stelle mit dem Wert ’1’ ergänzt und an je einen 24-Bit-Barrelshifter gelegt. Eine Auswerte-Schaltung für die Exponenten bestimmt für jeden Operanden die Zahl der Verschiebungen. Abhängig von den Vorzeichen wird noch eine Komplementbildung durchgeführt. Für ungültige Exponenten erfolgt über die Signale Azero und Bzero lediglich eine Abfrage auf Null, d.h. denormalisierte Mantissen und Unendlich werden von der Kombinatorik nicht abgefangen!

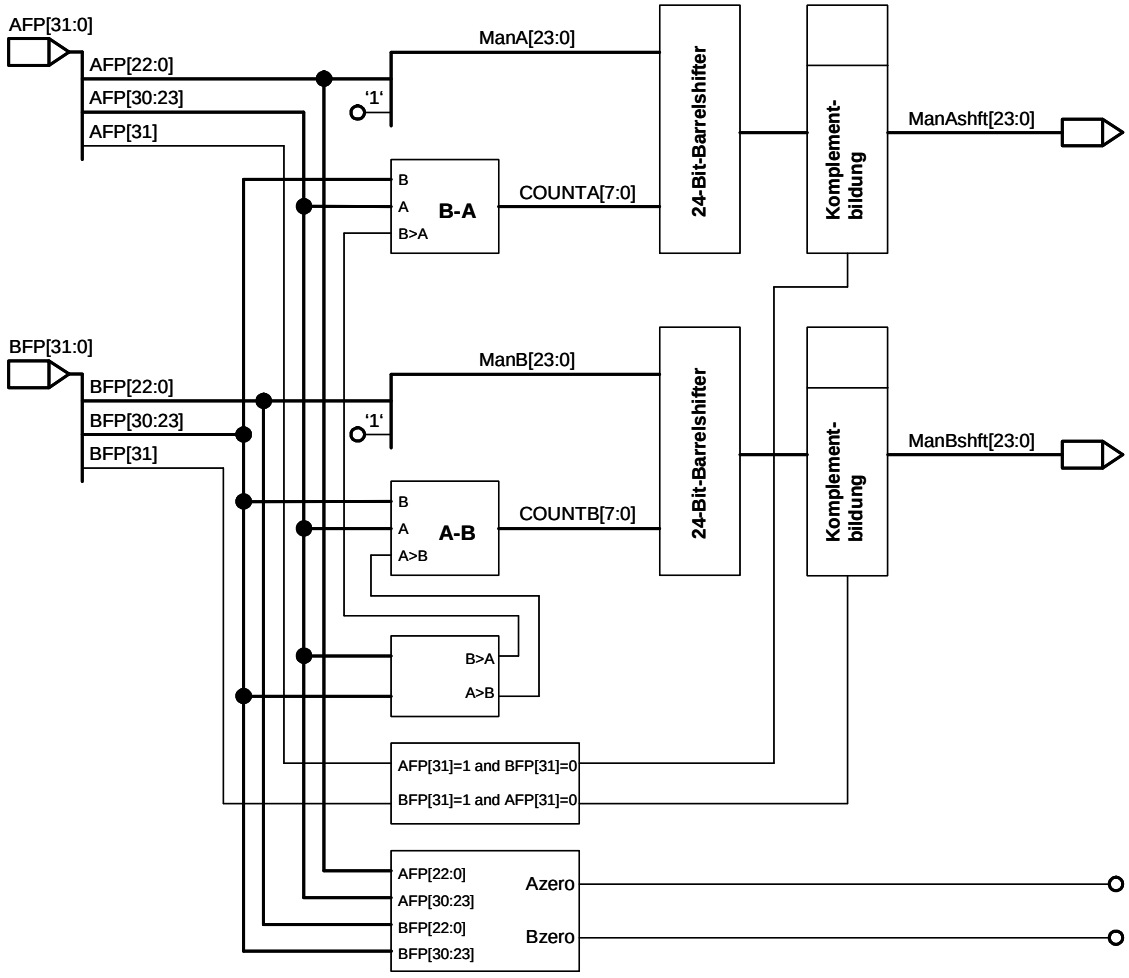


Bild 2.1: Kombinatorische Logik für die Gleitpunkt-Addition

Das für die Gleitpunkt-Addition entwickelte Steuerwerk in Form eines Zustandsgraphen zeigt Bild 2.2. Nach einem /RESET befindet sich das Steuerwerk im Zustand Z0, der nur über das Signal /START_ADD verlassen werden kann. Im Zustand Z1 erfolgt die Abfrage der beiden Operanden AFP[31:0] und BFP[31:0] auf Null. Ist einer dieser Operanden Null, so wird in den Zustand Z5 gewechselt, in welchem der jeweils andere Operand in QSFP gespeichert wird. Sind beide Operanden Null, so ist auch QSFP Null. Zusätzlich werden im Zustand Z1 die über die kombinatorische Logik nach Bild 2.1 angepassten Mantissen ManAshft[23:0] und ManBshft[23:0] addiert und der größere der beiden Exponenten in QExpS[7:0] gespeichert.

Nun werden die Vorzeichenbits analysiert. Sind sie identisch (positiv oder negativ), so geht das Steuerwerk in den Zustand Z2 über, in welchem das Vorzeichen gespeichert und bei einem evtl. aufgetretenen Mantissenübertrag (QManS(24) = '1') eine Korrektur durchgeführt wird. Eine solche Korrektur beinhaltet eine zusätzliche Rechtsverschiebung der Ergebnis-Mantisse um eine Bitstelle und die Erhöhung des Ergebnis-Exponenten um den Wert 1. In VHDL wird dies im Prozess A_Best wie folgt ausgedrückt (siehe Anhang A1):

```
if QManS(24) = '1' then       -- Mantissenübertrag
   QSFP(22 downto 0) <= QManS(23 downto 1);
   QSFP(30 downto 23) <= QExpS + 1;
else
   QSFP(22 downto 0) <= QManS(22 downto 0);
   QSFP(30 downto 23) <= QExpS;
end if;
```

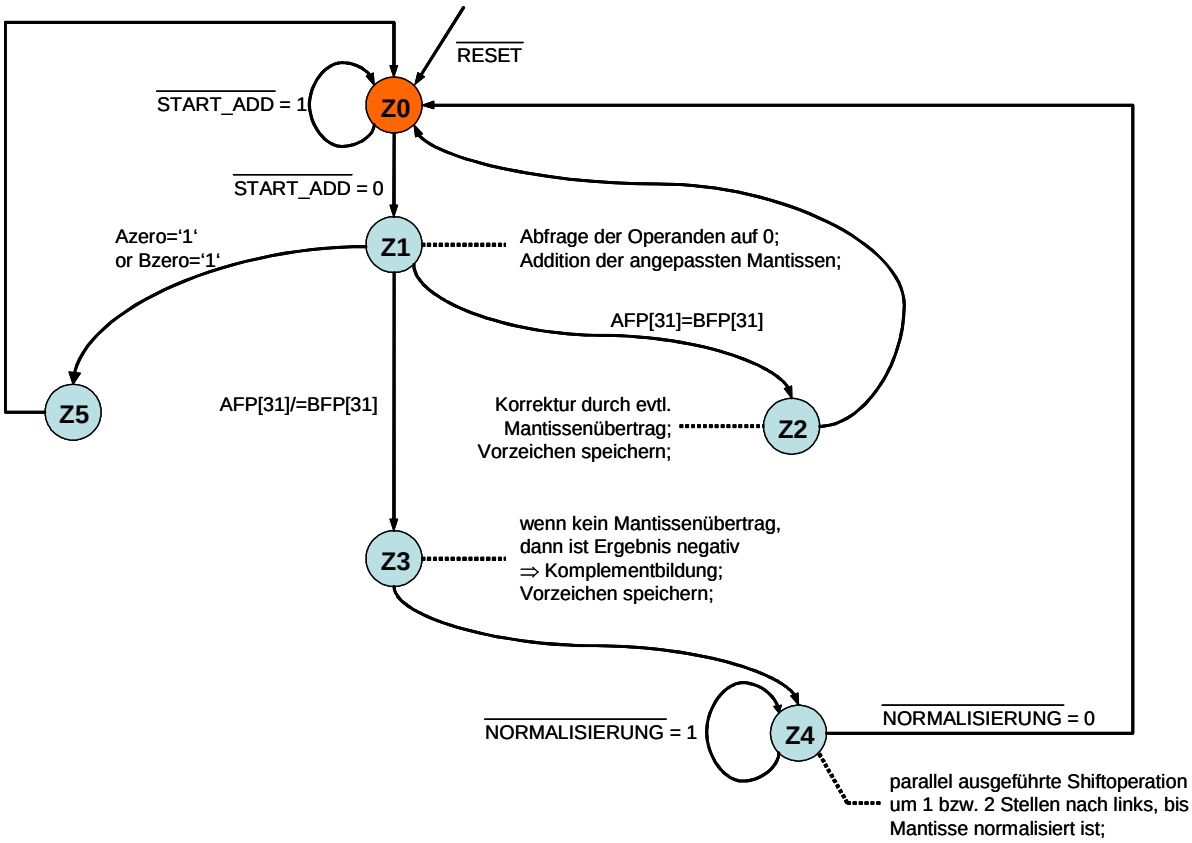


Bild 2.2: Zustandsgraph des Steuerwerks zur Gleitpunkt-Addition

Sind die Vorzeichenbits ungleich, dann wechselt das Steuerwerk von Zustand Z1 in den Zustand Z3. Ist kein Mantissenübertrag aufgetreten (QmanS(24) = '0'), so ist das Ergebnis negativ und es muss eine Komplementbildung erfolgen. Um dies zu verdeutlichen, betrachten wir folgendes Zahlenbeispiel:

Beispiel 2.1:
Es sollen die Dezimalzahlen +17.25 und -10.5 addiert werden:

```
+17.25 = +10001.01*2^127 = +1.000101*2^131
-10.5  = -1010.100*2^127 = -1.010100*2^130 = -0.101010*2^131
```

Komplementbildung von -10.5 durch Kombinatorik nach Bild 2.1 und anschließende Addition:

```
                    1.010101                      1.000101
Komplementbildung:  +       1      Addition:     +1.010110
                    1.010110                      10.011011
```

Es tritt ein Mantissenübertrag auf, d.h. das Ergebnis ist positiv (der Übertrag bleibt für die weitere Berechnung unberücksichtigt). Eine Normalisierung der positiven Mantisse liefert:

```
Normalisierung:   0.011011*2^131 = 1.1011*2^129 = 6.75d
```

Nun sollen die Dezimalzahlen +17.25 und -22.25 addiert werden:

```
   +17.25 = +10001.01*2^127 = +1.000101*2^131
   -22.25 = -10110.01*2^127 = -1.011001*2^131
```

Komplementbildung von -10.5 durch Kombinatorik nach Bild 2.1 und anschließende Addition:

```
                        0.100110                      1.000101
Komplementbildung:    +       1       Addition:     +0.100111
                        0.100111                    01.101100
```

Da kein Mantissenübertrag auftritt, ist das Ergebnis negativ und es muss eine Komplementbildung mit anschließender Normalisierung der dann positiven Mantisse erfolgen (der Übertrag bleibt wieder unberücksichtigt):

```
                        0.010011
Komplementbildung:    +       1
                        0.010100

Normalisierung:   -0.010100*2^131 = -1.01*2^129 = -5d
```

Mittels Analyse des Mantissenüberlaufs kann das Vorzeichen der berechneten Gleitkommazahl im Zustand Z3 bestimmt werden. Im nun folgenden Zustand Z4 werden zur Normalisierung der Mantisse Shiftoperationen um 1 bzw. 2 Stellen nach links bei gleichzeitiger Dekrementierung des Exponenten QExpS ausgeführt. Konnte die Mantisse normalisiert werden, dann wird das Signal /NORMALISIERUNG auf 0 gesetzt und das Steuerwerk geht in den Zustand Z0 über. Die Shiftoperationen werden in VHDL im Prozess A_Best wie folgt beschrieben (siehe Anhang A1):

```
if QManS(23) = '1' then
   QSFP(22 downto 0) <= QManS(22 downto 0);
   QSFP(30 downto 23) <= QExpS;
   NORMALISIERUNG <= '0';
elsif QManS(22) = '1' then
   QSFP(22 downto 0) <= QManS(21 downto 0) & '0';
   QSFP(30 downto 23) <= QExpS - 1;
   NORMALISIERUNG <= '0';
else
   QManS(23 downto 0) <= QManS(21 downto 0) & "00";
   QExpS <= QExpS - 2;
end if;
```

Aus dem VHDL-Code erkennt man, dass bei Linksshift der (positiven) Mantisse von rechts Nullen nachgezogen werden. Besitzt die Mantisse ms k führende Nullen, dann wird mathematisch betrachtet folgende Operation durchgeführt [Hoff93]:

$$\begin{aligned} ms &= ms \cdot 2^k \quad \text{"Linksshift um k Stellen" und} \\ es &= es - k \quad \text{"Exponent erniedrigen"} \end{aligned} \tag{2.3}$$

Weiterhin ist dem Code-Ausschnitt zu entnehmen, dass für k≥2 der Zustand Z4 mindestens noch einmal durchlaufen werden muss, da ja nur die Positionen QManS(23) und QManS(22) auf eine führende 1 abgefragt werden bzw. im else-Zweig ein Linksshift um 2 Stellen ohne Prüfung des Resultats ausgeführt wird. Das gesamte Steuerwerk (Automat) ist in drei nebenläufige Prozesse zur Zustandsaktualisierung, zur Folgezustands- und zur Ausgangsberechnung gegliedert [ReSc03]. Die Interprozesskommunikation, also die Kommunikation zwischen nebenläufigen Prozessen, erfolgt über die Signale ZUSTAND und FOLGE_Z, für die ein Aufzählungstyp ZUSTAENDE deklariert ist. Da der gezeigte Code-Ausschnitt im Zustand Z4 im Prozess A_Best(ZUSTAND) zur Ausgangsberechnung steht, wird er gemäß der Empfindlichkeitsliste nur bei einer Zustandsänderung ausgeführt. Deshalb ist hier in der Empfindlichkeitsliste zusätzlich noch das Signal QS aufgeführt, welches im Prozess

Z_SPEICHER im Zustand Z4 bei jeder steigenden Flanke des Clocksignals inkrementiert wird. Der vollständige VHDL-Quellcode zur 32-Bit-Gleitkomma-Addition ist im Anhang A1 dargestellt. Er teilt sich auf in die beiden VHDL-Files Top_Level_Modul.vhd zur Beschreibung des Zustandsgraphen nach Bild 2.2 sowie 24_Bit_Barrelshifter.vhd zur Beschreibung der Kombinatorik nach Bild 2.1.

2.1.1 Simulation

Die rein logische Verifikation (keine Synthese!) des VHDL-Codes zur Gleitkomma-Addition erfolgt mit dem Simulator ModelSim XE Starter der Fa. Mentor Graphics. Eine kurze Einführung in die Bedienung des Software-Tools ist in [ReSc03] zu finden. Nachfolgend wollen wir die Gleitkomma-Addition an mehreren Simulationsbeispielen betrachten. Um die Dezimalzahlen möglichst einfach in das 32-Bit-Gleitkommaformat umrechnen zu können, soll ein Java-Applet verwendet werden. Es findet sich unter der URL http://www.h-schmidt.net/FloatApplet/IEEE754de.html. Die Bedienung ist selbsterklärend.

Die Stimulidaten werden in einer do-Datei gespeichert und sind bis auf die beiden Operanden afp und bfp für alle Simulationen in diesem Abschnitt identisch:

```
#Kommandodatei für Gleitkomma-Addition
restart -f
radix hex
force reset 0 0, 1 70ns
force start_add 1 0, 0 175ns, 1 320ns
force clk 0 0, 1 50ns -repeat 100ns
force afp x"XXXXXXXX"  # Operand einsetzen
force bfp x"XXXXXXXX"  # Operand einsetzen
run 800ns
```

Beispiel 2.2:
Es sollen folgende Zahlen addiert werden:

```
+58760.75d : 476588C0h
+ 7214.50d : 45E17400h
+65975.25d : 4780DBA0h
```

Bild 2.3 zeigt das Resultat der Simulation. Der Exponent von bfp wird angepasst, d.h. manb wird gemäß countb = 03 um drei Stellen nach rechts geshiftet. Das Signal manbshft beinhaltet das Resultat dieser Operation. Über das L-aktive Signal start_add wird die Addition gestartet. Sowie der Automat in den Zustand z1 übergeht, weist das Signal ready_add L-Pegel auf. Da beide Operanden von Null verschieden sind und gleiche Vorzeichen haben, geht der Automat in den Zustand z2 über, in welchem die Korrektur für den aufgetretenen Mantissenübertrag stattfindet. Den Mantissenübertrag erkennt man an der höchstwertigen hexadezimalen Stelle im Signal qmans. Dieser Wert ist in Bild 2.3 gleich 1, d.h. die Bitstellen 23 bis 1 (nachfolgend rot dargestellt) sowie der um 1 erhöhte Exponent (nachfolgend blau dargestellt) werden in qsfp gespeichert:

```
qmans = X“101B740“ = 1|0000|0001|1011|0111|0100|0000
qexps + 1 = X“8F“ = 1000|1111
qsfp = 0100|0111|1000|0000|1101|1011|1010|0000 = X“4780DBA0“
```

Rechnung und Simulation stimmen somit überein. Geht der Automat wieder in den Zustand z0, dann wird das Signal ready_add auf H-Pegel gesetzt, d.h. die Gleitkomma-Addition ist abgeschlossen.

Beispiel 2.3:
Es sollen folgende Zahlen addiert werden:

```
+3.789d : 40727EFAh
-2.283d : C0121CACh
+1.506d : 3FC0C49Ch
```

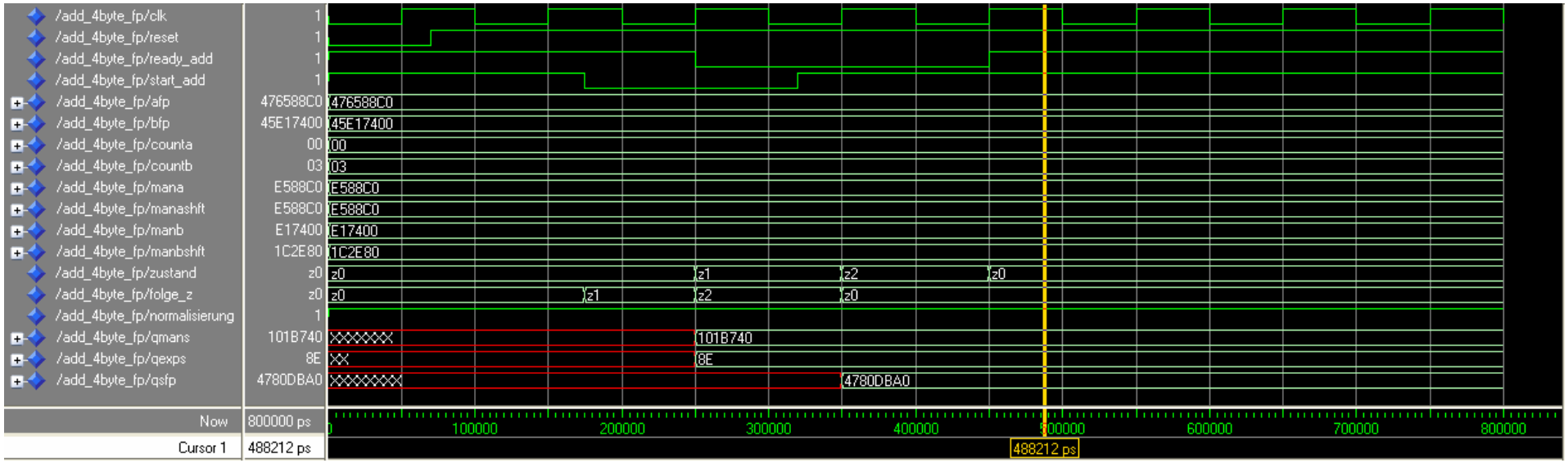


Bild 2.3: Addition von +58760.75 und +7214.50

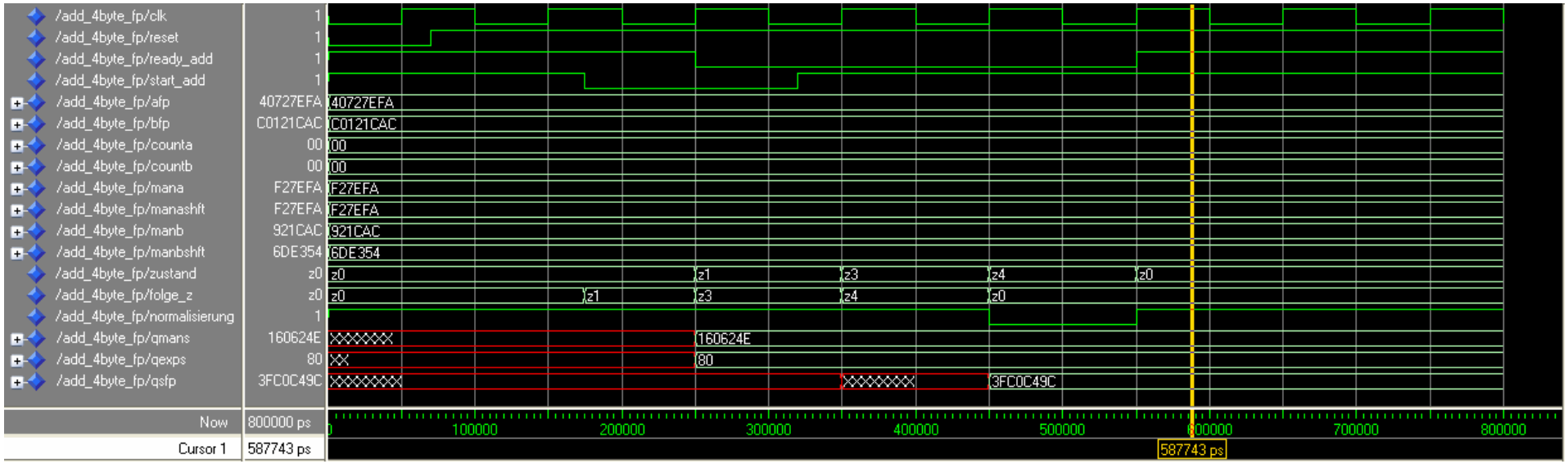


Bild 2.4: Addition von +3.789 und -2.283

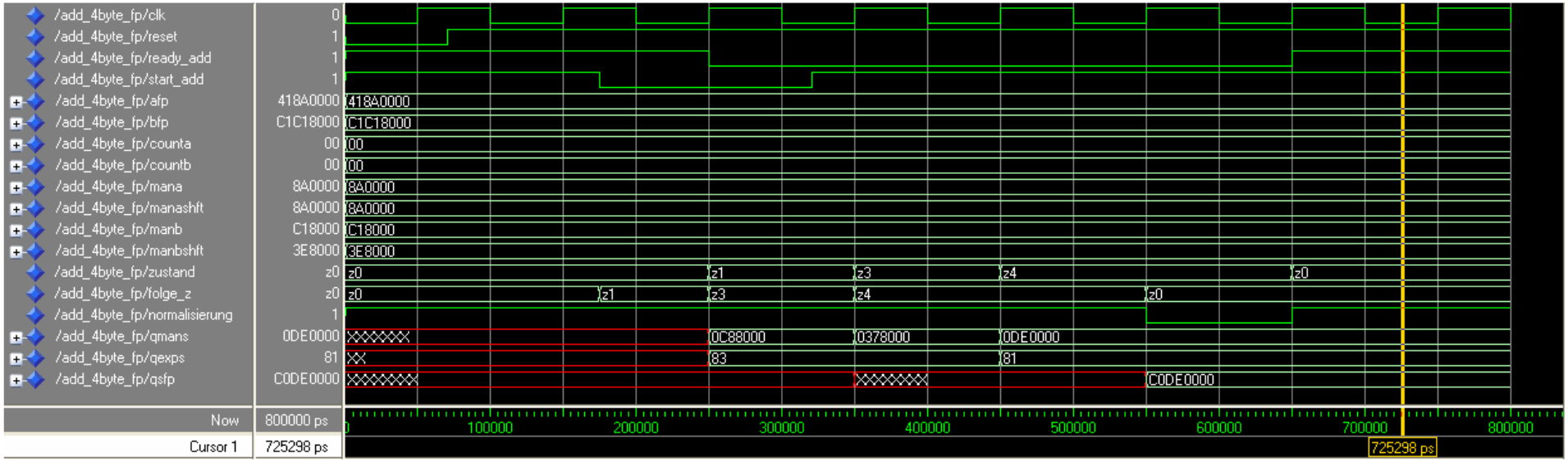


Bild 2.5: Addition von +17.25 und -24.1875

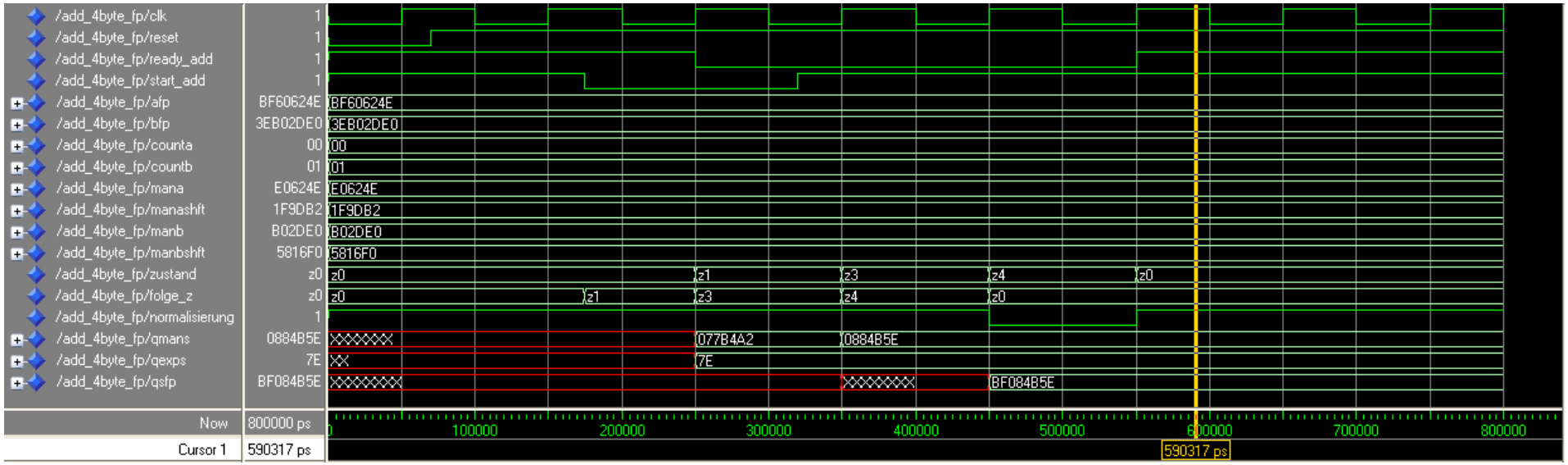


Bild 2.6: Addition von -0.8765 und +0.3441

Die Simulation nach Bild 2.4 zeigt einen Wechsel in den Zustand z3, da die Vorzeichen der Operanden nicht identisch sind. In diesem Zustand erfolgt keine Komplementbildung von qmans, d.h. das Ergebnis der Addition ist positiv. Die Normalisierung erfolgt im Zustand z4 und wird mit einem L-Pegel des Signals normalisierung abgeschlossen, so dass der Automat wieder in den Ausgangszustand z0 übergehen kann.

Beispiel 2.4:
Es sollen folgende Zahlen addiert werden:

```
+17.2500d : 418A0000h
-24.1875d : C1C18000h
- 6.9375d : C0DE0000h
```

Auch hier sind die Vorzeichen der Operanden unterschiedlich und es wird in den Zustand z3 gewechselt, siehe Bild 2.5. Hier erkennt man, dass sich der Wert des Signals qmans von 0C88000 auf 0378000 ändert, d.h. im Zustand z3 erfolgt eine Komplementbildung aufgrund des negativen Ergebnisses der Mantissenaddition (kein Mantissenüberlauf). Die Position qmans(24) wird hier nicht berücksichtigt, da sie jetzt nicht mehr benötigt wird:

```
qmans = X“0C88000“ = 0|1100|1000|1000|0000|0000|0000

Komplementbildung:   0|0011|0111|0111|1111|1111|1111
                   +                               1
                     0|0011|0111|1000|0000|0000|0000 = X“0378000“
```

Dieses Ergebnis der Komplementbildung verdeutlicht die Vorgehensweise bei der nun folgenden Normalisierung der Mantisse. Im Zustand z4 werden die Bitpositionen qmans(23) und qmans(22) auf 1 abgefragt. Da beide Bits 0 sind, wird qmans um zwei Stellen nach links geshiftet und der Exponent qexps um 2 dekrementiert:

```
    qmans = X“0378000“ = 0|0011|0111|1000|0000|0000|0000
Linksshift um 2 Stellen: 0|1101|1110|0000|0000|0000|0000 = X“0DE0000“
                qexps = qexps – 2 = X“83“ – 2 = X“81“
```

Im nun noch einmal zu durchlaufenden Zustand z5 wird qmans(23) auf 1 abgefragt und die Bitstellen qmans(22) bis qmans(0) sowie der Exponent in qsfp gespeichert.

Beispiel 2.5:
Es sollen folgende Zahlen addiert werden:

```
-0.8765d : BF60624Eh
+0.3441d : 3EB02DE0h
-0.5324d : BF084B5Eh
```

Auch diese Simulation nach Bild 2.6 läuft analog zu den bisherigen Beispielen ab. Aufgrund unterschiedlicher Vorzeichen der Operanden wechselt der Automat in den Zustand z3. Da keine Komplementbildung erfolgt, ist das Ergebnis negativ. Es folgen die Normalisierung der Mantisse im Zustand z4 und die Rückkehr in den Ausgangszustand z0.

2.2 Multiplikation von Gleitkommazahlen

Bevor wir auf die Multiplikation von Zahlen im Gleitkommaformat eingehen, soll zunächst die Multiplikation von positiven Dualzahlen betrachtet werden. Um die theoretischen Ausführungen sowie den schaltungstechnischen Aufwand gering zu halten, beschränken wir uns hierbei lediglich auf den seriellen Multiplizierer. Die Berechnungszeit bei dieser Realisierungsform dauert zwar üblicherweise zu lange, jedoch kann so die Erweiterung der Multiplikation positiver Zahlen auf das Gleitkommaformat relativ einfach verstanden und in VHDL umgesetzt werden.

2.2.1 Multiplikation positiver Dualzahlen

Die Multiplikation zweier nicht vorzeichenbehafteter Dualzahlen A[n] und B[m] ergibt ein (n+m)-stelliges Produkt:

$$P[n+m] = A[n] \cdot B[m] \tag{2.4}$$

Meist wird die gleiche Stellenzahl n=m gewählt, wovon wir hier ausgehen. Weiterhin soll analog zu [Hoff93] aus Gründen der Übersichtlichkeit im Folgenden A=A[n] und B=B[n] vereinbart werden. Die Multiplikation kann über die Summation der Partialprodukte (Teilprodukte) ausgedrückt werden, indem der Multiplikator B mittels Gl. (1.1) in eine Summe zerlegt wird:

$$\begin{aligned} P = A \cdot B &= A \cdot \sum_{j=0}^{n-1} b_j \cdot 2^j = \sum_{j=0}^{n-1} A \cdot b_j \cdot 2^j \\ &= A \cdot b_{n-1} \cdot 2^{n-1} + A \cdot b_{n-2} \cdot 2^{n-2} + \ldots + A \cdot b_1 \cdot 2^1 + A \cdot b_0 \cdot 2^0 \end{aligned} \tag{2.5}$$

Gl. (2.5) lässt sich nach [Hoff93] durch Ausklammern umformen, wobei es zwei Möglichkeiten gibt. Man kann zuerst das höchstwertige Bit b_{n-1} auswerten und das Teilprodukt $A \cdot b_{n-1}$ bilden, oder man wertet zuerst das niederwertigste Bit b_0 aus.

Fall 1: Beginn mit der höchstwertigen Stelle
Ausklammern von Gl. (2.5) liefert:

$$P = \underbrace{(\ldots(\underbrace{\underbrace{(A \cdot b_{n-1})}_{P_1} \cdot 2 + A \cdot b_{n-2}}_{P_2}) \cdot 2 + \ldots + A \cdot b_1) \cdot 2 + A \cdot b_0}_{P_n} \tag{2.6a}$$

Daraus ergibt sich folgendes rekursives Gleichungssystem:

$$\begin{aligned} P_0 &= 0 \\ P_1 &= 2 \cdot P_0 + A \cdot b_{n-1} \\ P_2 &= 2 \cdot P_1 + A \cdot b_{n-2} \\ &\vdots \\ P_n &= 2 \cdot P_{n-1} + A \cdot b_0 \end{aligned} \tag{2.6b}$$

Beispiel 2.6:
Es sollen die Rekursionsgleichungen nach Gl. (2.6b) für n=4 angegeben werden.

$$\begin{aligned} P_0 &= 0 \\ P_1 &= A \cdot b_3 \\ P_2 &= 2 \cdot P_1 + A \cdot b_2 = A \cdot b_3 \cdot 2 + A \cdot b_2 \\ P_3 &= 2 \cdot P_2 + A \cdot b_1 = A \cdot b_3 \cdot 2^2 + A \cdot b_2 \cdot 2 + A \cdot b_1 \end{aligned}$$

$$P_4 = 2 \cdot P_3 + A \cdot b_0 = A \cdot b_3 \cdot 2^3 + A \cdot b_2 \cdot 2^2 + A \cdot b_1 \cdot 2 + A \cdot b_0$$

Eine Hardware-Realisierung in Form eines Zustandsgraphen zeigt Bild 2.7. Ein Linksshift um eine Stelle entspricht einer Multiplikation mit dem Faktor 2, wobei von rechts eine '0' nachgezogen wird. Der zirkulare Linksshift des Multiplikators B in jedem Takt hat zur Folge, dass am Ende der Multiplikation der anfängliche Wert in diesem Register erhalten bleibt und nicht 000…00 beträgt.

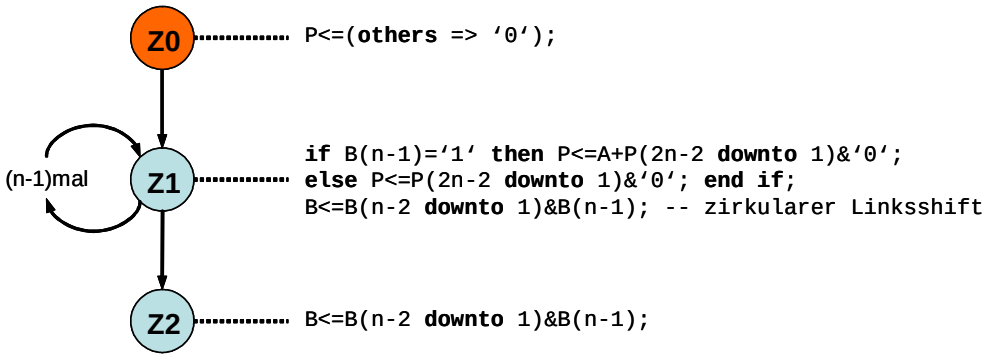


Bild 2.7: Serielle Multiplikation, Fall 1

Das Rechenwerk, das sich aus den in Bild 2.7 aufgeführten Ausgangsberechnungen des Automaten herleitet, ist jedoch zu aufwändig, da es einen 2n-stelligen Addierer erfordert. Aus diesem Grunde wird man die Multiplikation nicht mit dem höchstwertigen Bit b_{n-1} beginnen [Hoff93].

Fall 2: Beginn mit der niederwertigsten Stelle

Aus Gl. (2.5) ergibt sich:

$$P = (\ldots(\underbrace{(A \cdot 2^n \cdot b_0)}_{P_1} \cdot 2^{-1} + A \cdot 2^n \cdot b_1) \cdot 2^{-1} + \ldots + A \cdot 2^n \cdot b_{n-1}) \cdot 2^{-1} \qquad \text{(2.7a)}$$

P_2 und P_n (Klammerungen)

Daraus folgt das Rekursionsschema:

$$\begin{aligned} P_0 &= 0 \\ P_1 &= (P_0 + A \cdot 2^n \cdot b_0) \cdot 2^{-1} \\ P_2 &= (P_1 + A \cdot 2^n \cdot b_1) \cdot 2^{-1} \\ &\vdots \\ P_n &= (P_{n-1} + A \cdot 2^n \cdot b_{n-1}) \cdot 2^{-1} \end{aligned} \qquad \text{(2.7b)}$$

Beispiel 2.7:

Es sollen die Rekursionsgleichungen nach Gl. (2.7b) für n=4 angegeben werden.

$$P_0 = 0$$

$$P_1 = A \cdot 2^4 \cdot b_0 \cdot 2^{-1} = A \cdot 2^3 \cdot b_0$$

$$P_2 = (A \cdot 2^3 \cdot b_0 + A \cdot 2^4 \cdot b_1) \cdot 2^{-1} = A \cdot 2^2 \cdot b_0 + A \cdot 2^3 \cdot b_1$$

$$P_3 = (A \cdot 2^2 \cdot b_0 + A \cdot 2^3 \cdot b_1 + A \cdot 2^4 \cdot b_2) \cdot 2^{-1} = A \cdot 2^1 \cdot b_0 + A \cdot 2^2 \cdot b_1 + A \cdot 2^3 \cdot b_2$$

$$P_4 = (A \cdot 2^1 \cdot b_0 + A \cdot 2^2 \cdot b_1 + A \cdot 2^3 \cdot b_2 + A \cdot 2^4 \cdot b_3) \cdot 2^{-1} = A \cdot b_0 + A \cdot 2 \cdot b_1 + A \cdot 2^2 \cdot b_2 + A \cdot 2^3 \cdot b_3$$

Man sieht, dass das Ergebnis mit dem aus Beispiel 2.6 identisch ist.

Aus dem Rekursionsschema lässt sich wieder ein Zustandsgraph entwickeln, siehe Bild 2.8. Hierbei wird das Produktregister P[2n] aufgeteilt in zwei gleich große Register PL[n] und PH[n]. Für die kombinatorisch realisierte (n+1)-Bit-Addition werden die Register PH[n] und A[n] mit einer führenden '0' auf (n+1)-Bits erweitert. Die Summe steht im (n+1)-Bit-Vektor H[n+1]. Der Term 2^{-1} in Gl. (2.7b) stellt eine Division durch 2 dar, entspricht somit einem Shift nach rechts um eine Stelle. Ist die niederwertigste Stelle PL(0)='0', dann wird beim Rechtsshift von links eine '0' nachgezogen. Andernfalls steht an der Stelle PH(n-1) das höchstwertige Bit der (n+1)-Bit-Addition, d.h. H(n).

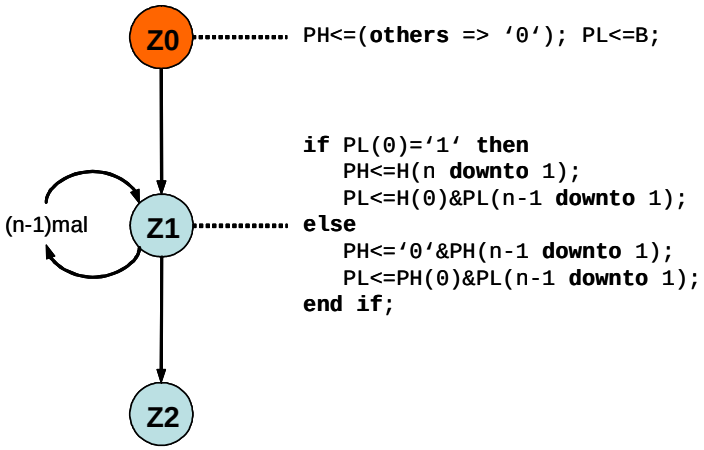


Bild 2.8: Serielle Multiplikation, Fall 2

Bild 2.9 zeigt das sich aus den Ausgangsberechnungen des Automaten ergebende Rechenwerk. Für PL(0)='1' wird zu PH der Wert von A addiert, andernfalls der Wert "000…00". Die Unterscheidung, welcher Wert addiert wird, erfolgt über eine UND-Verknüpfung für jede Stelle von A. Bei einem nach diesem Prinzip konstruierten Multiplikationsrechenwerk entspricht PH dem Akkumulator und PL dem erweiterten Akkumulator. Nur Register PL stellt ein Schieberegister dar. Das Register für B kann eingespart werden, wenn B am Anfang in PL steht. Die Arbeitsweise dieses Rechenwerks lässt sich an einem Zahlenbeispiel sehr einfach demonstrieren.

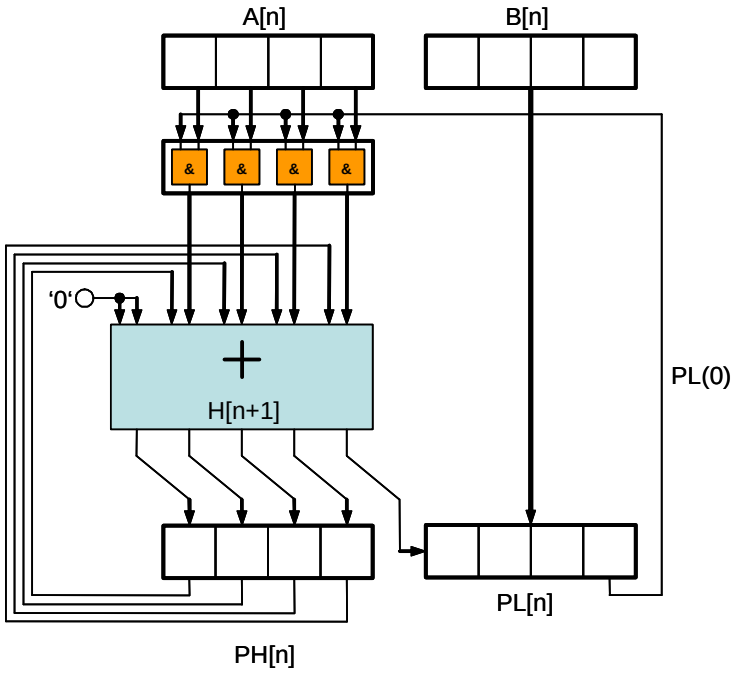


Bild 2.9: Rechenwerk zur seriellen Multiplikation, Fall 2

Beispiel 2.8:
Die Dezimalzahlen 15 und 9 sollen mit dem Rechenwerk nach Bild 2.9 multipliziert werden.

```
A = 15d = 1111b
B =  9d = 1001b   ;   15d * 9d = 135d

Zustand
   0                  PH_PL ← 0000|1001
   1     H = 0|1111   PH_PL ← 0111|1100
   1     H = 0|0111   PH_PL ← 0011|1110
   1     H = 0|0011   PH_PL ← 0001|1111
   1     H = 1|0000   PH_PL ← 1000|0111 = 135d
   2
```

2.2.2 VHDL-Beschreibung der seriellen Multiplikation

Das in Bild 2.9 gezeigte Rechenwerk für die serielle Multiplikation soll nun konkret für den Fall zweier 8-Bit-Operanden betrachtet werden. Bild 2.10 zeigt das Rechenwerk für n=8.

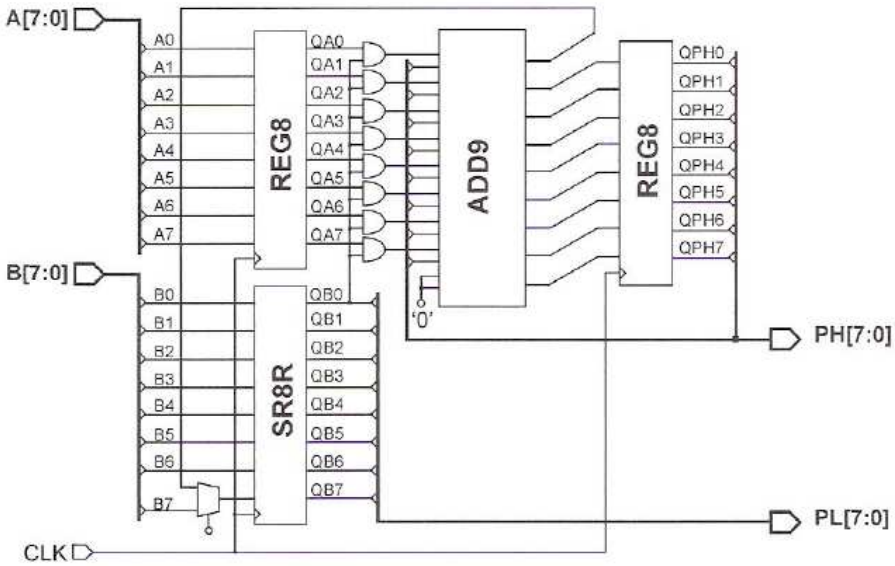


Bild 2.10: Rechenwerk zur seriellen Multiplikation mit $n = 8$ [Jork04]

In der Schaltung nach Bild 2.10 werden der Operand A[7:0] im Register QA[7:0] und der Operand B[7:0] im Schieberegister QB[7:0] abgespeichert, das gleichzeitig auch als Register PL[7:0] fungiert. Über einen Multiplexer kann entschieden werden, ob das Bit B(7) oder die niederwertigste Stelle des 9-stufigen Addierers nach QB(7) geladen wird. Ansonsten entspricht der Aufbau in Bild 2.10 dem Rechenwerk nach Bild 2.9.

Mit jeder Addition werden nur die oberen 8 Bit des Zwischenergebnisses neu berechnet. Diese oberen 8 Bit des Zwischenergebnisses werden in dem Ergebnisregister QPH[7:0] gespeichert. Das unterste Bit jeder Addition wird nicht weiter verändert und kann in das Ergebnis für die unteren 8 Bit des 16-Bit-Ergebniswertes eingehen. Nach 8 Takten steht das Produkt mit den oberen 8 Bit im Register QPH[7:0] und mit den unteren 8 Bit im Schieberegister QB[7:0]. Der gespeicherte Operand B[7:0] wurde überschrieben.

Das noch erforderliche Steuerwerk orientiert sich ebenfalls an den Ausführungen in [Jork04]. Gemäß Bild 2.11 werden nach einem Ladesignal für die Operanden (oder für den letzten der beiden Operanden) 8 Verarbeitungstakte durch ein Zustands-Flipflop QBUSY markiert. In diesen 8 Takten finden die Additionen und Schiebeoperationen statt. Die 8 Verarbeitungstakte werden mit einem 3-stufigen Zähler QBIT[2:0] codiert.

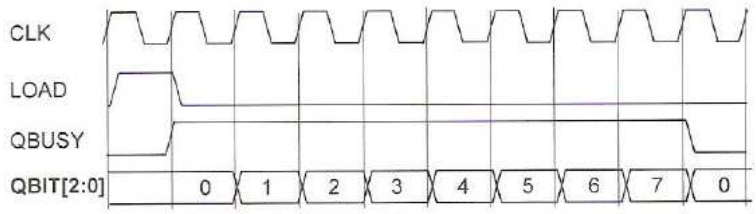


Bild 2.11: Steuerwerk-Signale für die 8-Bit-Multiplikation [Jork04]

Das in Bild 2.12 gezeigte Steuerwerk enthält den 3-stufigen Zustandszähler mit synchronem Rücksetzeingang. Der Ladeimpuls schaltet das Zustands-RS-Flipflop über den S-Eingang ein. Das Flipflop gibt den Zähler frei und dieser setzt das Flipflop nach 8 Takten zurück.

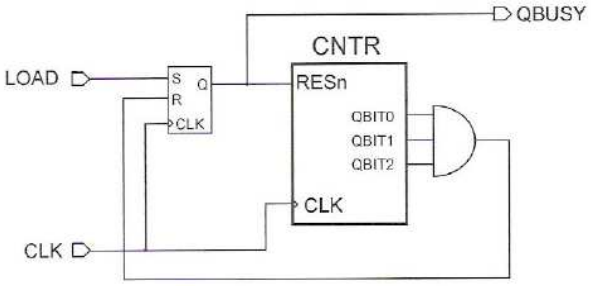


Bild 2.12: Steuerwerk für die serielle Multiplikation [Jork04]

In dem im Anhang A2 aufgeführten VHDL-File Mult_n_Bit.vhd sind alle Operationen zusammengefasst [Jork04]. Das VHDL-Modell kann auf größere Operanden übertragen werden. Zur variablen Angabe der Wortbreite wird die Parameterübergabe für variable Vektoren mittels generic-Anweisung genutzt. Die 8-Bit-Multiplikation lässt sich so leicht auf die Operandengrößen 2^n umstellen. Hierbei können die Parameter n und w entweder bei der VHDL-Beschreibung initialisiert werden oder bei der Komponenten-Instanziierung. Innerhalb von process-Anweisungen können Wertzuweisungen und Operationen für eine variable Anzahl von Bitstellen mit loop-Schleifen formuliert werden [Jork04, ReSc03]. Diese Anweisungen werden sequentiell wirksam.

Die Zustandscodierung des Steuerwerks nach Bild 2.12 wird in der process-Anweisung states zusammengefasst. Wenn das LOAD-Signal gesetzt ist, dann wird QBUSY gesetzt und alle Bits des Zählers werden in der ersten loop-Schleife gelöscht. Der darauf folgende if-Zweig wird noch nicht durchlaufen, da die Aktualisierung aller Signale erst am Prozessende erfolgt (nur Prozessvariablen werden unmittelbar aktualisiert)! So wird der Zähler bei der nächsten steigenden Taktflanke auch dann um 1 inkrementiert, wenn das LOAD-Signal immer noch aktiv ist, man vergleiche mit Bild 2.13. Sind alle Bits des Zählers auf ’1’ gesetzt, dann ist die im Prozess definierte Variable vsw am Ende der zweiten loop-Schleife immer noch ’1’ und das Signal QBUSY wird gelöscht. Nachfolgend ist der VHDL-Code des Prozesses states dargestellt:

```
states: process(CLK)
variable vsw: std_logic;
begin
  if rising_edge(CLK) then
     if LOAD = '1' then
        QBUSY <= '1';
        for I in 0 to n-1 loop
          QBIT(I) <= '0';
         end loop;
     end if;
     if QBUSY = '1' then
        QBIT <= QBIT + 1;
        vsw:='1';
        for I in 0 to n-1 loop
          vsw:=vsw and QBIT(I);
        end loop;
        if (vsw = '1') then QBUSY <= '0'; end if;
     end if;
  end if;
end process states;
```

In der zweiten process-Anweisung mult werden das Laden der Operanden, das Initialisieren des Ergebnisregisters, die Addition und die Schiebeoperationen beschrieben. Das Partialprodukt aus dem Vektor QA[7:0] und der Bitstelle QB(0) wird zuerst in der Prozess-Variablen SUM[8:0] zusammengefasst. Es ist “00000000“, falls die aktuelle Bitstelle QB(0) = ’0’ ist, andernfalls QA[7:0].

Für die 9-Bit-Addition wird das Partialprodukt mit einer führenden ’0’ auf 9 Bit erweitert. Das Partialprodukt wird auf die Zwischensumme aufaddiert. Das Register QPH[7:0] übernimmt die oberen 8 Bit der Summe. Das unterste Bit jeder 9-Bit-Addition wird in das Schieberegister QB[7:0] eingeschoben. Der VHDL-Code für den Prozess mult lautet wie folgt:

```
mult: process(CLK)
variable SUM: std_logic_vector(w downto 0);
begin
  if rising_edge(CLK) then
     if LOAD = '1' then
         QA <= A; QB <= B;
         for I in 0 to w-1 loop
           QPH(I) <= '0';
         end loop;
     end if;
     if QBUSY = '1' then
         for I in 0 to w-1 loop
           SUM(I):=QA(I) and QB(0);
         end loop;
        SUM(w):= '0';
        SUM := SUM + ('0' & QPH);
        QB <= SUM(0) & QB(w-1 downto 1);
        QPH <= SUM(w downto 1);
     end if;
  end if;
end process mult;
```

Beispiel 2.9:
Es sollen folgende 8-Bit-Dualzahlen multipliziert werden:

```
A = 213d = D5h = 1101|0101
B = 179d = B3h = 1011|0011

P = A*B = 38127 = 94EF = 1001|0100|1110|1111
```

Die Stimulidaten zur Simulation stehen wieder in einer do-Datei:

```
# Kommandodatei für 8-Bit-Multiplikation
restart -f
radix hex
force clk 0 0, 1 50ns -repeat 100ns
force a x"D5"
force b x"B3"
force load 0 0, 1 180ns, 0 465ns
run 1.2us
```

Die Simulation mit ModelSim liefert das in Bild 2.13 gezeigte 16-Bit-Ergebnis 94EFh, wobei das obere Byte in qph und das untere Byte in qb steht. Die Multiplikation dauert erwartungsgemäß 8 Takte.

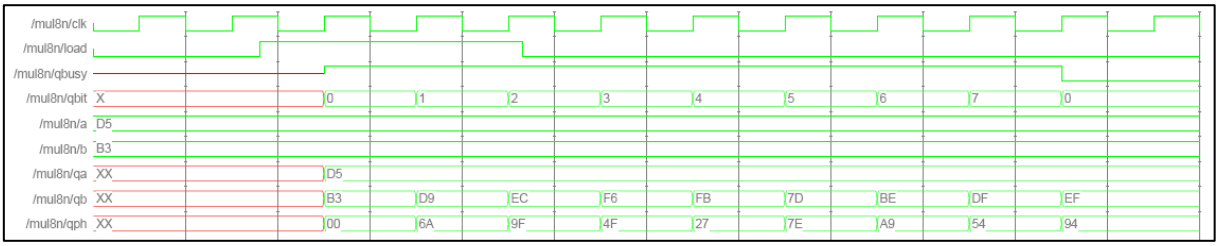


Bild 2.13: 8-Bit-Multiplikation der Zahlen 213d und 179d

Im nächsten Abschnitt wird die Multiplikation von 32-Bit-Gleitkommazahlen behandelt. Hier werden wir für die Multiplikation zweier 24-Bit-Mantissen einen 24-Bit-Multiplizierer benötigen. Das im Anhang A2 angegebene VHDL-File ist dafür nicht direkt nutzbar, da die Operandenlänge keine Zweierpotenz ist.

Für die fest vorgegebene Wortlänge von 24 Bit ist eine entsprechend angepasste VHDL-Beschreibung mit dem Namen Mult_24_Bit.vhd im Anhang A3 zu finden. Das in A2 in der Architektur definierte Signal QBUSY wird hier als bidirektionales Signal (inout) innerhalb der Entity beschrieben. Der Grund hierfür wird im nächsten Abschnitt an entsprechender Stelle erläutert.

2.2.3 Erweiterung auf Gleitkommazahlen

Die Multiplikation von (normalisierten) Gleitkommazahlen ist im Vergleich zur Addition von Gleitkommazahlen einfach. Die Mantissen der beiden Operanden werden miteinander multipliziert und die Exponenten zur Basis 2 werden unabhängig davon addiert, d.h. ein Angleichen der Exponenten entfällt [Hoff93]. Aus den Vorzeichen der beiden Mantissen wird das Vorzeichen des Resultats ermittelt.

$$p = a \cdot b = mp \cdot 2^{ep} = \left(ma \cdot 2^{ea}\right) \cdot \left(mb \cdot 2^{eb}\right) = (ma \cdot mb) \cdot 2^{ea+eb} \tag{2.8}$$

Geht man von der Kommastellung des IEEE-Gleitkommaformats aus (siehe Abschnitt 1.3), dann erfüllen die Mantissen ma und mb die Normalisierungsbedingung nach Gl. (1.10):

$$1 \le \overline{m}_{norm} \le 2 - 2^{-r} < 2$$

Die Produkt-Mantisse liegt dann zwischen folgenden Grenzen:

$$1 \le \overline{mp} \le \left(2 - 2^{-r}\right)^2 < 4 \tag{2.9}$$

Das bedeutet, dass das Produkt nicht notwendigerweise normalisiert ist. Betrachten wir hierzu ein Beispiel mit reduzierter Stellenanzahl.

Beispiel 2.10:
Die folgenden normalisierten Mantissen mit r=5 sollen (ohne Rundung) multipliziert werden:

```
a) ma = 1.00110b = 1.1875d
   mb = 1.00001b = 1.03125d

b) ma = 1.11110b = 1.9375d
   mb = 1.11100b = 1.875d

Zu a)  1.00110*1.00001 = 01.00111100110b = 1.224609375d
```

Der Wertebereich der Normalisierungsbedingung nach Gl. (1.10) wird hier nicht überschritten, d.h. die Produkt-Mantisse ist normalisiert. Bei einer praktischen Realisierung werden durch einen Linksshift um eine Stelle die rot dargestellten Dualziffern als 6-Bit-Ergebnis interpretiert, man vergleiche mit Bild 2.14. Diese Shiftoperation hat auf den hier nicht aufgeführten Exponenten keinen Einfluss, da Normalisierung bereits vorliegt und durch das Shiften lediglich das Hidden-Bit an die höchstwertige Stelle im Register gebracht wird.

```
Zu b)  1.11110*1.11100 = 11.1010001000b = 3.6328125d
```

Jetzt wird der Wertebereich der Normalisierungsbedingung überschritten, d.h. die Mantisse muss normalisiert werden. Für die praktische Realisierung bedeutet dies, dass das Komma um eine Stelle nach links wandert, d.h. der Exponent muss um Eins erhöht werden. Auch hier bilden die rot dargestellten Dualziffern das 6-Bit-Ergebnis.

Bild 2.14 zeigt das Rechenwerk für die 32-Bit-Gleitkomma-Multiplikation. Hierbei werden die Operanden gemäß Gl. (2.8) in die Bestandteile Mantisse und Exponent aufgeteilt. Die jeweils um das Hidden-Bit auf 24 Stellen erweiterten Mantissen können mit dem 24-Bit-Multiplizierer aus Anhang A3 berechnet werden, d.h. es wird eine Instanz des VHDL-Moduls MUL24 eingesetzt.

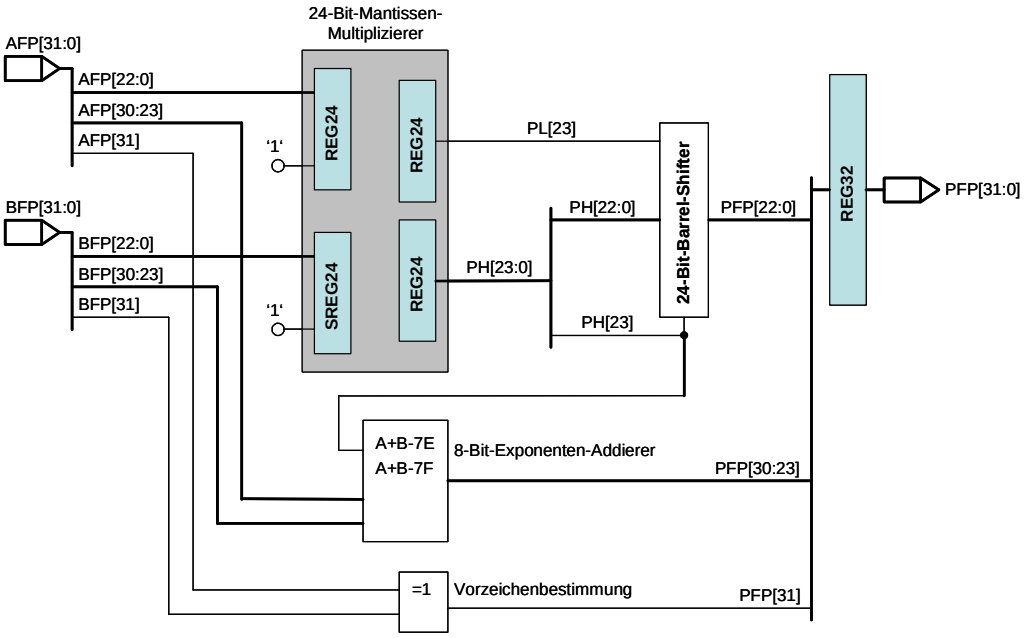


Bild 2.14: Rechenwerk der 32-Bit-Gleitkomma-Multiplikation (in Anlehnung an [Jork04])

Beispiel 2.10 hat deutlich gemacht, dass bei der Multiplikation nur dann ein normalisierter Mantissenwert auftritt, wenn das höchstwertige Bit Null ist, also PH(23)='0'. Der 24-Bit-Barrelshifter führt dann einen Linksshift um eine Stelle aus, wobei das Bit PL(23) von rechts nachgezogen wird. Durch die Addition der Exponenten muss in jedem Falle ein Basiswert (bias = 127d = 7Fh) abgezogen werden. Bei bereits normalisierter Mantisse ist keine weitere Korrektur des Exponenten erforderlich. Ist hingegen PH(23)='1', dann wird gemäß Beispiel 2.10 der Wertebereich der Normalisierungsbedingung überschritten. In diesem Falle ist keine Shiftoperation erforderlich, jedoch muss zur Normalisierung der Mantisse der Exponent neben der Basiswert-Korrektur (Abzug von 7Fh) um Eins erhöht werden (Verschiebung des Kommas um eine Stelle nach links), d.h. letztlich wird der Exponent um den Wert 7Eh verringert.

Das Ergebnis der Gleitkomma-Multiplikation setzt sich nach Bild 2.14 aus dem Exponenten- und dem verkürzten Mantissen-Ergebnis zusammen. Die Operanden können vorzeichenbehaftet sein. Das Ergebnisvorzeichen ergibt sich durch die EXOR-Verknüpfung der Vorzeichen der Operanden [Jork04]. Das Ergebnis wird in einem Ausgangsregister QPFP[31:0] abgespeichert. Der vollständige VHDL-Quellcode besteht aus dem in A4 dargestellten File Top_Level_Modul.vhd sowie dem File Mult_24_Bit.vhd nach A3. Die Instanziierung der Komponente MUL24 sowie die process-Anweisung validation sehen wie folgt aus:

```
ManMul: MUL24 port map (ManA,ManB,LOAD,CLK,PH,PL,BUSY);
validation: process(CLK)
begin
  if rising_edge(CLK) then
     QD <= BUSY;
     if LOAD='1' then QVALID<= '0'; end if;
     if (QD='1' and BUSY='0') then QVALID <= '1'; end if;
  end if;
end process validation;
```

Durch das Ladesignal LOAD für die Operanden wird die Multiplikation gestartet. Die zeitliche Steuerung der seriellen 24-Bit-Multiplikation ist bereits im Modul MUL24 enthalten. Im Prozess validation wird dem Signal QD das Signal BUSY zugewiesen, welches dem bidirektionalen Signal QBUSY des Moduls MUL24 entspricht. Durch diese Verwendung außerhalb der zugehörigen Architektur erklärt sich auch, warum QBUSY in A3 als bidirektionales Signal definiert ist. Durch das Ladesignal LOAD wird das Markierungssignal QVALID zurückgesetzt. Wenn die Mantissen-Multiplikation abgeschlossen und somit BUSY='0' ist, wird die process-Anweisung validation bei der nächsten steigenden Flanke mit den Werten QD='1' und BUSY='0' ausgeführt. QD wird dann auf '0' (BUSY-Wert) und das Flipflop QVALID auf '1' gesetzt. Letztere zeigt so die Gültigkeit der berechneten Ergebnisse an. Die Simulation nach Bild 2.15 verdeutlicht dies noch einmal.

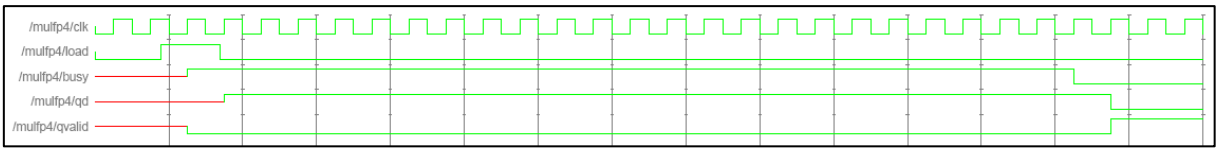


Bild 2.15: Generierung der Signale QD und QVALID im Prozess validation

Man erkennt, dass nach 24 Verarbeitungstakten für die serielle Multiplikation die Signale qd und qvalid erwartungsgemäß einen Takt später ihre Pegel wechseln. In diesem 25. Takt finden die Normalisierung des Ergebnisses und die Abspeicherung statt, die in der Process-Anweisung correct zusammengefasst sind:

```
correct: process(CLK)
begin
  if rising_edge(CLK) then
     if (QD='1' and BUSY='0') then
         if PH(23)='1' then
            QPFP(22 downto 0) <= PH(22 downto 0);
            QPFP(30 downto 23) <= AFP(30 downto 23)
                                  + BFP(30 downto 23) - X"7E";
        else
            QPFP(22 downto 0) <= PH(21 downto 0) & PL(23);
            QPFP(30 downto 23) <= AFP(30 downto 23)
                                  + BFP(30 downto 23) - X"7F";
        end if;
        QPFP(31) <= AFP(31) xor BFP(31);
     end if;
  end if;
end process correct;
```

Die Normalisierung und Abspeicherung findet durch die UND-Verknüpfung der Signale QD und BUSY nur im 25. Takt statt.

2.2.4 Simulation

Zur Umwandlung der Dezimalzahlen in das 32-Bit-Gleitkommaformat wird wieder ein Java-Applet verwendet, dessen URL in Abschnitt 2.1.1 angegeben ist. Die do-Datei für die Stimulidaten sieht wie folgt aus:

```
# Kommandodatei für Gleitkomma-Multiplikation
restart -f
radix hex
force clk 0 0, 1 50ns -repeat 100ns
force afp x"XXXXXXXX"  # Operand einsetzen
force bfp x"XXXXXXXX"  # Operand einsetzen
force load 0 0, 1 180ns, 0 340ns
run 3us
```

Folgende Dezimalzahlen sollen miteinander multipliziert werden:

```
a)  afp = +1.500d = 3FC00000h
    bfp = +1.125d = 3F900000h

           => pfp = +1.6875d = 3FD80000h

b)  afp = +1.75d = 3FE00000h
    bfp = -1.50d = BFC00000h

          => pfp = -2.625d = C0280000h
```

Die Simulationsergebnisse sind in den Bildern 2.16 und 2.17 zu sehen. Im Fall a) steht nach der Mantissen-Multiplikation im Register ph der Wert 6C0000h, d.h. das höchstwertige Bit ist eine ’0’. Somit muss hier eine Shiftoperation durchgeführt werden. Im Fall b) entfällt diese, da nach der Multiplikation im Register ph der Wert A80000h steht, ph(23) ist somit ’1’.

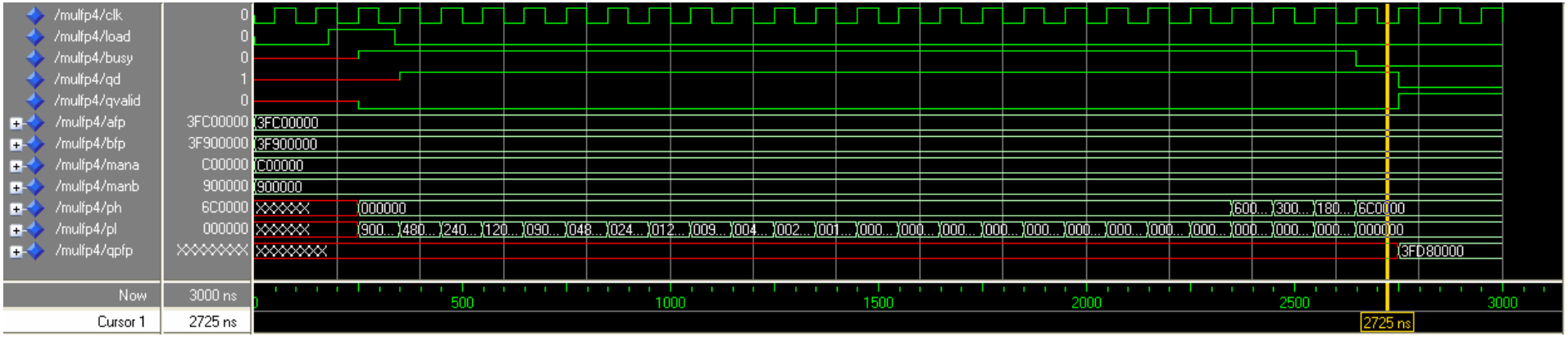


Bild 2.16: Multiplikation von +1.500 und +1.125

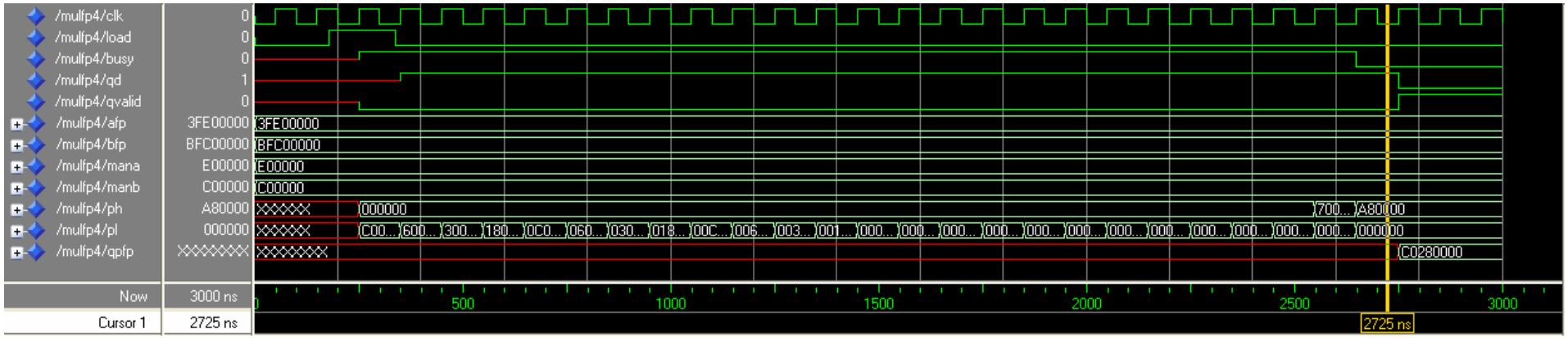


Bild 2.17: Multiplikation von +1.75 und -1.50

2.3 Division von Gleitkommazahlen

Die Division ist die schwierigste Grundrechenart und stellt die inverse Operation zur Multiplikation dar. Auch hier soll zur Reduktion des Hardwareaufwandes nur eine serielle Umsetzung besprochen werden, wobei man in der Literatur grundsätzlich zwei Verfahren unterscheidet [Flik93]:

- Division mit Rückstellen des Zwischenrestes
- Division ohne Rückstellen des Zwischenrestes

Die Methode “Division mit Rückstellen des Zwischenrestes“ wird, zur Steigerung der Rechengeschwindigkeit leicht abgewandelt, in VHDL implementiert und stellt hier die Basis für die eigentliche Gleitkomma-Division dar.

2.3.1 Division vorzeichenloser Dualzahlen

Allgemein ist die Division definiert als die Bestimmung eines Quotienten, welcher durch Multiplikation mit dem Divisor den Dividenden ergibt [Pirs96]. Selten ergeben sich für ganzzahlige Divisoren und ganzzahlige Dividenden auch ganzzahlige Quotienten, so dass ein Rest eingeführt wird. Es sei A der Dividend, B der Divisor, Q der Quotient und R der Rest. Dann gilt:

$$Q = \frac{A - R}{B} \quad \Rightarrow \quad R = A - Q \cdot B \tag{2.10}$$

Wir gehen nachfolgend davon aus, dass der Dividend 2n-stellig und der Divisor sowie der Quotient jeweils n-stellig sind. Für den n-stelligen Quotienten Q können wir nach Gl. (1.1) schreiben:

$$Q = \sum_{i=0}^{n-1} q_i \cdot 2^i \tag{2.11}$$

Durch Einsetzen von Gl. (2.11) in Gl. (2.10) folgt:

$$\begin{aligned} R &= A - q_{n-1} \cdot B \cdot 2^{n-1} - q_{n-2} \cdot B \cdot 2^{n-2} - \ldots - q_0 \cdot B \\ &= \underbrace{\underbrace{\underbrace{(A \cdot 2^{-(n-1)} - q_{n-1} \cdot B)}_{R_1} \cdot 2 - q_{n-2} \cdot B}_{R_2} \cdot 2 - \ldots - q_0 \cdot B}_{R_n} \end{aligned} \tag{2.12}$$

Ausgangspunkt sei folgender Startwert:

$$R_0 = A \cdot 2^{-n} \tag{2.13}$$

Dies bedeutet, dass das Komma des Dividenden A um n Stellen nach links verschoben wird. Der jeweils neue Rest (Zwischenrest) ergibt sich aus dem vorherigen Rest entsprechend der Rekursionsbeziehung:

$$\begin{aligned} R_1 &= R_0 \cdot 2 - q_{n-1} \cdot B \\ R_2 &= R_1 \cdot 2 - q_{n-2} \cdot B \\ &\vdots \\ R_n &= R_{n-1} \cdot 2 - q_0 \cdot B \end{aligned} \tag{2.14}$$

Bei der Division kann ein Quotientenüberlauf entstehen, wenn der Quotient nicht mit den verfügbaren Stellen dargestellt werden kann. Wenn n Stellen für den Betrag des Quotienten vorgesehen sind, dann darf er nicht größer als 2^n-1 werden. Wenn kein Überlauf auftreten soll, dann muss folgende Bedingung erfüllt sein:

$$A < 2^n \cdot B \quad \Rightarrow \quad A \cdot 2^{-n} = R_0 < B \tag{2.15}$$

Gl. (2.15) ist gleichbedeutend mit der Aussage, dass der um n Stellen (nach links) verschobene Divisor B größer sein muss als der Dividend A. Die Überprüfung des Quotientenüberlaufes kann durch einen vorgeschalteten Zyklus vor Beginn der eigentlichen Division erfolgen.

Die Division vorzeichenbehafteter Zahlen kann – analog zur Multiplikation – durch die Division der Beträge und separater Behandlung der Vorzeichen durchgeführt werden. Es seien a_n, b_n, q_n, r_n die Vorzeichen der Zahlen A, B, Q, R. Für die Vorzeichen gilt dann [Pirs96]:

$$\begin{aligned} q_n &= a_n \oplus b_n \\ r_n &= a_n \end{aligned} \tag{2.16}$$

Die Beziehung $r_n=a_n$ kann an einem einfachen Zahlenbeispiel verdeutlicht werden:

$$(-15):(+6) = -2 \quad \text{Rest: } -3$$

mit Gl. (2.10): $A = Q \cdot B + R = (-2)\cdot(+6) + (-3) = -15$

$$(+15):(-6) = -2 \quad \text{Rest: } +3$$

$$A = Q \cdot B + R = (-2)\cdot(-6) + (+3) = +15$$

2.3.2 Division mit Rückstellen des Zwischenrestes

Bei diesem Verfahren wird in jedem Schritt der Divisor B stellenverschoben gemäß der Rekursionsbeziehung nach Gl. (2.14) vom Dividenden "auf Verdacht" subtrahiert und anschließend kontrolliert, ob die Null unterschritten wurde (negativer Zwischenrest). Entsteht bei dieser Subtraktion ein negativer Zwischenrest, so muss diese Subtraktion durch eine nachfolgende Addition wieder rückgängig gemacht werden, deshalb spricht man vom Divisionsverfahren mit Rückstellen des Zwischenrestes [Flik93]. Diese Wiederherstellung der ursprünglichen Zahl wird als Restoring bezeichnet. Q entsteht bei diesem Verfahren ziffernweise, beginnend mit der höchsten Stelle, gemäß den erfolglosen Subtraktionen (negativer Zwischenrest, q_i=0) und den erfolgreichen Subtraktionen (positiver Zwischenrest, q_i=1).

Beispiel 2.11:

```
n = 4
A = 147d = 10010011b
B =  14d = 1110b

R0    01001.0011   Überlaufkontrolle
-D   -01110.
      11011.0011   R0<0 => kein Überlauf

R0    01001.0011   Restore

2*R0  10010.0110   Beginn Divisionszyklus
 -D   -01110.
 R1    00100.0110   R1>0 => q3=1

2*R1  01000.1100
 -D   -01110.
 R2    11010.1100   R2<0 => q2=0

R2    01000.1100   Restore
```

```
2*R2    10001.1000
  -D    -01110.
  R3     00011.1000   R3>0 => q1=1

2*R3    00111.0000
  -D    -01110.
  R4     11001.0000   R4<0 => q0=0

  R4    00111.0000   Restore

   Q = 1010b = 10d
   R = 0111b =  7d
```

Bei einer praktischen Realisierung kann das umständliche Rückstellen des Zwischenrestes mittels Addition des Divisors umgangen werden, indem über einen Multiplexer die höchstwertige Stelle der (n+1)-Bit-Subtraktion ausgewertet wird. Bild 2.18 zeigt ein solches Rechenwerk für n=4 Stellen mit Berücksichtigung eines möglichen Quotientenüberlaufs.

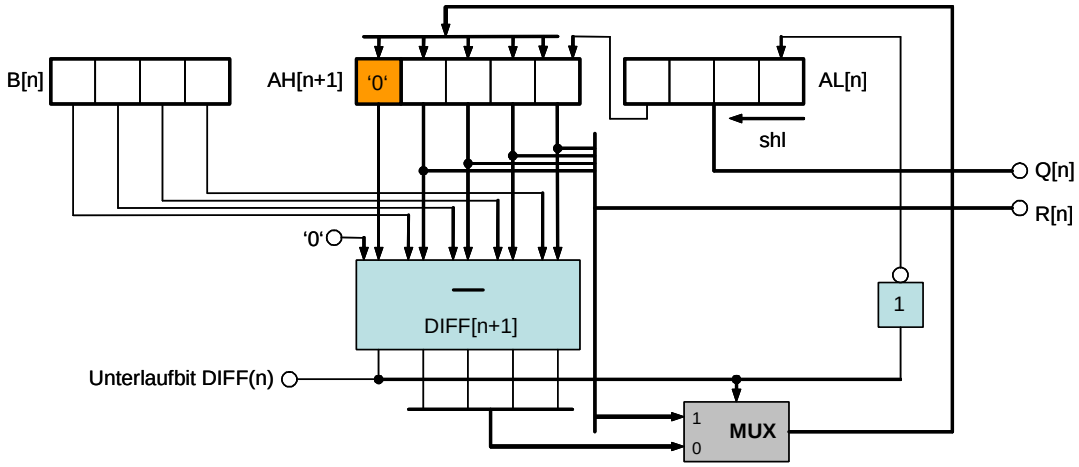


Bild 2.18: Rechenwerk zur Division mit Rückstellen des Zwischenrestes, leicht abgewandelt

Der 2n-stellige Dividend wird in den unteren Anteil AL[n] und den oberen Anteil AH[n+1] aufgeteilt, wobei das werthöchste Bit des AH-Registers mit '0' initialisiert wird. Der n-stellige Divisor ist im Register B[n] abgelegt. Im 1. Takt wird durch eine (n+1)-Bit-Subtraktion geprüft, ob ein möglicher Quotientenüberlauf aufgetreten ist. Falls ja, dann ist das Unterlaufbit DIFF(n)=0 und die Division wird abgebrochen. Wenn kein Quotientenüberlauf auftritt, dann werden das AH- und das AL-Register um eine Stelle nach links geshiftet, wobei das höchstwertige Bit von AL[n] in die niederwertigste Stelle von AH[n+1] wandert. In der frei werdenden niederwertigsten Stelle des AL-Registers wird das invertierte Unterlaufbit abgelegt.

Nun wird (n-1)-Mal eine (n+1)-Bit-Subtraktion mit anschließender Auswertung des Unterlaufbits DIFF(n) ausgeführt. Ist DIFF(n)=0, dann legt der Multiplexer die Differenz DIFF[n-1:0] auf AH[n:1], andernfalls erfolgt die Shiftoperation AH[n-1:0]→AH[n:1]. Zusätzlich wird nach jeder Subtraktion das AL-Register um eine Stelle nach links geshiftet, wobei auch hier wieder das höchstwertige Bit von AL[n] in die niederwertigste Stelle von AH[n+1] wandert.

Im letzten Zyklus (Takt n+1) wird ebenfalls eine (n+1)-Bit-Subtraktion ausgeführt und das Unterlaufbit ausgewertet. Ist jetzt DIFF(n)=0, dann legt der Multiplexer die Differenz DIFF[n-1:0] auf AH[n-1:0], da im letzten Schritt keine Shiftoperation im AH-Register erforderlich ist. Für DIFF(n)=1 bleiben die aktuellen Werte im AH-Register erhalten. Das AL-Register hingegen wird im letzten Zyklus wieder um eine Stelle nach links geshiftet, wobei nun das höchstwertige Bit AL(n-1) verloren geht. Dieses Bit wird nicht benötigt, da es mit einer 0 in der (n+1)-ten Ergebnisstelle nur aussagt, dass kein Quotientenüberlauf aufgetreten ist. Bei der Shiftoperation wird das invertierte Unterlaufbit als niederwertigste Stelle im AL-Register nachgezogen. Das Ergebnis der Division steht nun im Register AL[n-1:0], der Divisionsrest im Register AH[n-1:0].

Bild 2.19 zeigt den Zustandsgraphen für das Rechenwerk nach Bild 2.18 zur Division mit Rückstellen des Zwischenrestes.

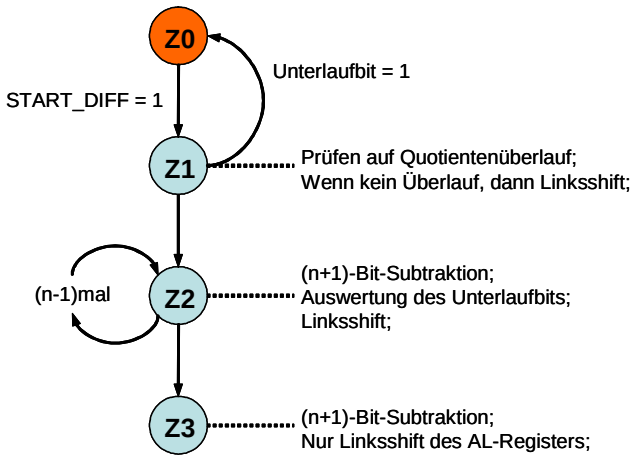


Bild 2.19: Zustandsgraph für das Rechenwerk nach Bild 2.18

Mit den Werten aus Beispiel 2.11 ergeben sich folgende taktabhängige Registerzustände:

Takt	AH[4:0]	AL[3:0]	Bemerkung
0	01001	0011|	Initialisierung;
1 (2R0)	10010	011|0	Subtraktion; kein Quotientenüberlauf; Linksshift;
2 (2R1)	01000	11|01	Subtraktion; DIFF(4)=0 => q3=1;
3 (2R2)	10001	1|010	Subtraktion; DIFF(4)=1 => q2=0;
4 (2R3)	00111	|0101	Subtraktion; DIFF(4)=0 => q1=1;
5 (R4)	**00111**	**1010**	Subtraktion; DIFF(4)=1 => q0=0;

Tabelle 2.1: Taktabhängige Registerzustände für Beispiel 2.11

2.3.3 Division ohne Rückstellen des Zwischenrestes

Bei diesem Verfahren wird ein negativer Zwischenrest nicht zurückgestellt, sondern im nächsten Rekursionsschritt weiterverarbeitet [Hoff93]. Immer wenn ein negativer Zwischenrest entsteht, wird im darauf folgenden Rekursionsschritt der Divisor B addiert und umgekehrt. Entsteht zum Schluss des Verfahrens ein negativer Rest, so muss dieser noch durch Addition des Divisors korrigiert werden.

Beispiel 2.12:

```
 n = 4
 A = 147d = 10010011b
 B =  14d = 1110b

 R0    01001.0011   Überlaufkontrolle
 -D   -01110.
       11011.0011   negativ => kein Überlauf

shl    10110.0110   Beginn Divisionszyklus
 +D   +01110.
       00100.0110   positiv => q3=1

shl    01000.1100
 -D   -01110.
       11010.1100   negativ => q2=0
```

```
  shl    10101.1000
   +D   +01110.
         00011.1000   positiv => q1=1

  shl    00111.0000
   -D   -01110.
   R4    11001.0000   negativ => q0=0

Korr.     1001.
         +1110.
          0111.       => Rest

    Q = 1010b = 10d
    R = 0111b =  7d
```

Man sieht, dass das Ergebnis identisch ist zu Beispiel 2.11.

Bild 2.20 zeigt ein Rechenwerk zur Division ohne Rückstellen des Zwischenrestes, das im Prinzip dem Aufbau aus Bild 2.18 entspricht. Der Multiplexer entfällt hier, stattdessen wird neben dem (n+1)-Bit-Subtrahierer noch ein (n+1)-Bit-Addierer benötigt. Das Ergebnis der jeweiligen (n+1)-Bit-Operation soll hier mit H[n+1] bezeichnet werden.

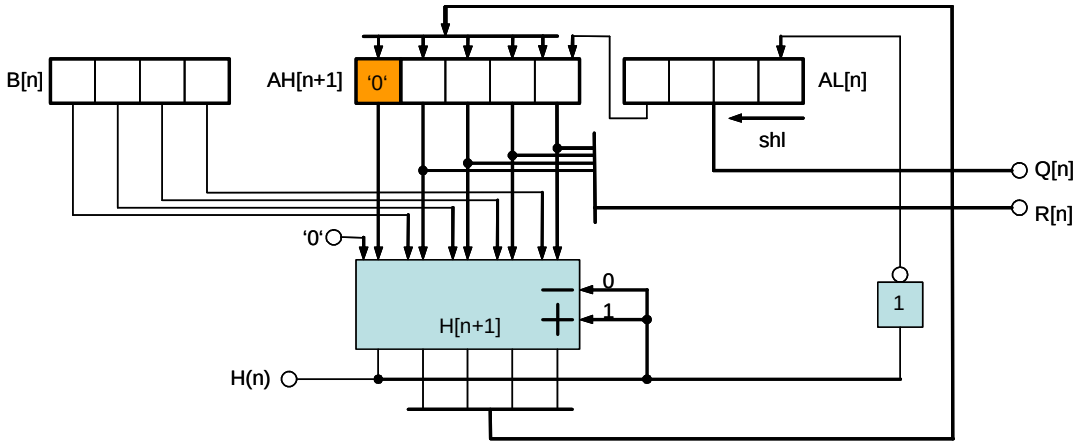


Bild 2.20: Rechenwerk zur Division ohne Rückstellen des Zwischenrestes

Im 1. Takt wird stets eine Subtraktion durchgeführt. Ist das Ergebnis mit H(n)=0 positiv, dann ist ein Quotientenüberlauf aufgetreten und die Division wird abgebrochen. Andernfalls wird das Ergebnis stellenverschoben im AH-Register gespeichert und das AL-Register – analog zu Bild 2.18 – um eine Stelle nach links geshiftet. Das mit H(n)=1 negative Ergebnis signalisiert nun, dass im nächsten Takt eine (n+1)-Bit-Addition ausgeführt wird usw.

Auch bei diesem Rechenwerk ist – analog zu Bild 2.18 – im (n+1)-ten Takt keine Shiftoperation mehr im AH-Register erforderlich. Das AL-Register hingegen wird in diesem Zyklus wieder um eine Stelle nach links geshiftet, wobei auch hier das nicht benötigte höchstwertige Bit AL(n-1) verloren geht.

Charakteristisch für dieses Verfahren ist, dass alle geraden Quotienten Q einen negativen Rest haben, der durch eine zu Beginn dieses Abschnitts erwähnte zusätzliche Addition des Divisors B zum Rest korrigiert werden muss [Flik93]. In diesen Fällen benötigt das hier beschriebene Rechenwerk eine um einen Takt längere Verarbeitungszeit als das Rechenwerk nach Bild 2.18. Aus diesem Grunde wird auf die Angabe eines Zustandsgraphen sowie eine VHDL-Quellcodeentwicklung für dieses Verfahren verzichtet. Abschließend werden noch die taktabhängigen Registerzustände für die Zahlenwerte aus Beispiel 2.12 angegeben:

Takt	AH[4:0]	AL[3:0]	Bemerkung
0	01001	0011|	Initialisierung;
1	10110	011|0	Subtraktion; kein Quotientenüberlauf; Linksshift;
2	01000	11|01	Addition; H(4)=0 => q3=1;
3	10101	1|010	Subtraktion; H(4)=1 => q2=0;
4	00111	|0101	Addition; H(4)=0 => q1=1;
5	11001	1010	Subtraktion; H(4)=1 => q0=0;
6	**10111**	**1010**	Korrektur des Restes durch Divisor-Addition;

Tabelle 2.2: Taktabhängige Registerzustände für Beispiel 2.12

2.3.4 VHDL-Beschreibung einer 8-Bit-Division

In diesem Abschnitt soll der VHDL-Quellcode für das unter 2.3.2 beschriebene Divisionsverfahren mit Rückstellen des Zwischenrestes entwickelt werden. Für den Dividenden wird eine feste Breite von 16 Bit angenommen, wobei dieser sich aufteilt in ein unteres Byte AL und ein oberes Byte AH. Diese Werte sowie der Divisor B werden in den Registern QAH[8:0], QAL[7:0] und QB[7:0] gespeichert. Der Automat ist wieder in drei nebenläufige Prozesse zur Zustandsaktualisierung (Z_SPEICHER), zur Folgezustandsberechnung (F_BEST) und zur Ausgangsberechnung (A_BEST) gegliedert. Die Interprozesskommunikation erfolgt wieder über die internen Signale ZUSTAND und FOLGE_Z, insgesamt werden vier Automatenzustände (Z0,Z1,Z2,Z3) deklariert.

Mit einem Reset-Signal geht der Automat in den Zustand Z0 über. Zusätzlich wird das Signal QS[2:0] im Prozess Z_SPEICHER mit “000“ initialisiert. Im Zustand Z0 werden die Register QAH, QAL und QB geladen, wobei das höchstwertige Bit von QAH gelöscht wird. Mit dem Signal START_DIFF, das in der Empfindlichkeitsliste des nachfolgend aufgeführten Prozesses F_BEST steht, wird die Division gestartet:

```
F_BEST: process(START_DIFF,ZUSTAND,QS)
variable Q_ÜBERLAUF: std_logic_vector(8 downto 0);
begin
  case ZUSTAND is
    when Z0 =>   if START_DIFF = '1' then FOLGE_Z <= Z1; end if;

    when Z1 =>   if START_DIFF = '1' then
                    Q_ÜBERLAUF := QAH  - ('0' & QB);
                    if Q_ÜBERLAUF(8) = '0' then
                       FOLGE_Z <= Z0;
                    else
                       FOLGE_Z <= Z2;
                    end if;
                 end if;

    when Z2 =>   if QS = "110" then FOLGE_Z <= Z3; end if;

    when Z3 =>   FOLGE_Z <= Z0;

  end case;
end process;
```

Im Zustand Z1 wird eine (n+1)-Bit-Subtraktion ausgeführt. Da das Ergebnis in der Variablen Q_ÜBERLAUF steht, kann das höchstwertige Bit Q_ÜBERLAUF(8) unmittelbar zur Prüfung auf einen Quotientenüberlauf ausgewertet werden. Falls kein Überlauf auftritt, ist der Folgezustand Z2.

Im Zustand Z1 wird im Prozess A_BEST ebenfalls eine (n+1)-Bit-Subtraktion ausgeführt, wobei das Ergebnis hier in der Variablen DIFF steht. Ist DIFF(8)=’1’ (kein Quotientenüberlauf), dann werden die Register QAH und QAL um eine Stelle nach links geshiftet, wobei das invertierte Unterlaufbit der Subtraktion nachgezogen wird. Der Prozess A_BEST ist nachfolgend dargestellt:

```
A_BEST: process(ZUSTAND,QS)
variable DIFF: std_logic_vector(8 downto 0);
begin
  case ZUSTAND is
    when Z0 =>   QAL <= AL; QAH <= '0'&AH; QB <= B;

    when Z1 =>   DIFF := QAH  - ('0' & QB);
                 if DIFF(8) = '1' then
                    -- nur Linksshift
                    QAH(8 downto 0) <= QAH(7 downto 0) & QAL(7);
                    QAL(7 downto 0) <= QAL(6 downto 0) & not(DIFF(8));
                 end if;

    when Z2 =>   DIFF := QAH  - ('0' & QB);
                 if DIFF(8) = '0' then
                    QAH(8 downto 0) <= DIFF(7 downto 0) & QAL(7);
                 else
                    QAH(8 downto 0) <= QAH(7 downto 0) & QAL(7);
                 end if;
                 QAL(7 downto 0) <= QAL(6 downto 0) & not(DIFF(8));

    when Z3 =>   DIFF := QAH  - ('0' & QB);
                 if DIFF(8) = '0' then
                    QAH(7 downto 0) <= DIFF(7 downto 0);
                    REST <= DIFF(7 downto 0);
                 else
                    REST <= QAH(7 downto 0);
                 end if;
                 QAL(7 downto 0) <= QAL(6 downto 0) & not(DIFF(8));
                 Q <= QAL(6 downto 0) & not(DIFF(8));
  end case;
end process;
```

Im Zustand Z2 wird im Prozess Z_SPEICHER mit jeder steigenden Flanke des Clocksignals der Vektor QS[2:0] um 1 inkrementiert. Da QS in den Empfindlichkeitslisten der Prozesse F_BEST und A_BEST steht, werden diese bei jeder Änderung von QS ausgeführt, auch wenn der Automat im Zustand Z2 verweilt und somit keine Zustandsänderung eintritt. Im Prozess F_BEST wird QS auf den Wert “110“ abgefragt und in diesem Fall der Folgezustand auf Z3 gesetzt. Auf diese Weise kann der Automat n-1=8-1=7 Takte im Zustand Z2 verweilen und jeweils im Prozess A_BEST eine (n+1)-Bit-Subtraktion mit anschließender Auswertung der höchsten Bitstelle der Variablen DIFF[8:0] durchführen. Ist DIFF(8)=’0’, dann wird die Differenz der Subtraktion stellenverschoben ins Register QAH übernommen, andernfalls erfolgt ein Linksshift des QAH-Registers. QAL wird in beiden Fällen um eine Stelle nach links geshiftet, wobei das invertierte Unterlaufbit der Subtraktion an der freien Position QAL(0) abgelegt wird.

Im Zustand Z3 wird der letzte Zyklus ausgeführt, d.h. eine (n+1)-Bit-Subtraktion ohne anschließende Stellenverschiebung/Linksshift des QAH-Registers (siehe Abschnitt 2.3.2). Der ganzzahlige Quotient steht nun in QAL und wird über das Port-Signal Q[7:0], der Rest steht in QAH und wird über das Port-Signal REST[7:0] nach außen geführt. Der vollständige VHDL-Quellcode für die hier beschriebene 8-Bit-Division besteht aus dem in A5 dargestellten File Div_8_Bit.vhd.

Beispiel 2.13:
Es sollen folgende Dualzahlen dividiert werden:

```
A = 17521d = 4471h (16 Bit)
B =   100d =   64h ( 8 Bit)

R = A – Q*B => Q = 175d = AFh
               R =  21d = 15h
```

```
# Kommandodatei für 8-Bit-Division
restart -f
radix hex
force clk 0 0, 1 50ns -repeat 100ns
force reset 1 0, 0 20ns, 1 180ns
force start_diff 0 0, 1 220ns, 0 320ns
#force start_diff 0 0, 1 220ns, 0 520ns
force ah x"44"
force al x"71"
force b  x"64"
run 1.3us
```

Bild 2.21 zeigt das Ergebnis der Simulation mit ModelSim. Einschließlich der Prüfung auf einen möglichen Quotientenüberlauf beträgt die Verarbeitungszeit 9 Takte.

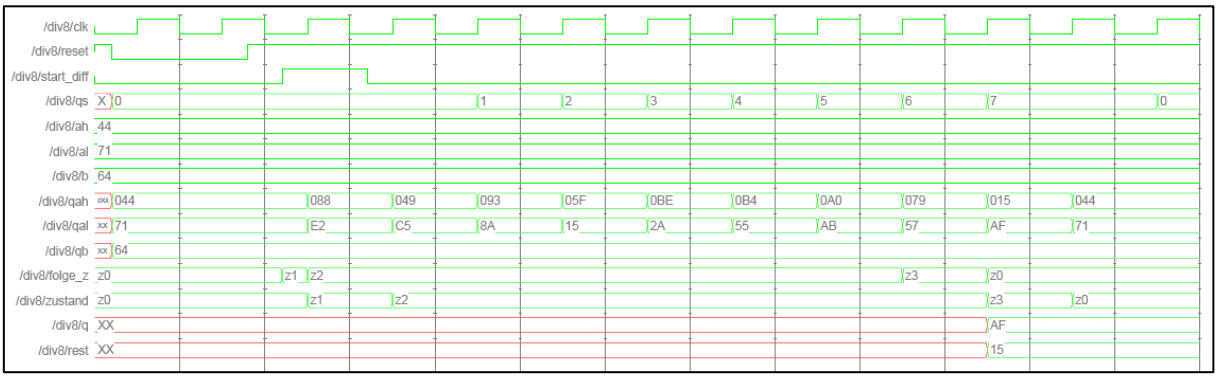


Bild 2.21: 8-Bit Division der Zahlen 17521d und 100d

2.3.5 Division gebrochener Dualzahlen

Prinzipiell können die Algorithmen für die Division ganzer Zahlen auf gebrochene Dualzahlen übertragen werden. Im Hinblick auf die Gleitkomma-Division sollen hier nur unecht gebrochene Zahlen betrachtet werden, also Zahlen mit einem ganzzahligen Anteil. Zur Demonstration gehen wir von zwei unecht gebrochenen Dualzahlen A und B mit jeweils einer Vor- und vier Nachkommastellen aus, wobei die Vorkommastelle stets den Wert '1' aufweist (entspricht später dem hidden Bit der erweiterten 24-Bit-Mantisse).

Beispiel 2.14:
Es sind folgende Quotienten zu bilden:

```
a)  1.8125d / 1.5625d = 1.16d
b)  1.5625d / 1.8125d = 0.86206896…d
```

Für den Dividend bzw. Divisor ergibt sich (n=5; eine Vor- und vier Nachkommastellen):

```
    1.8125d = 1.1101b   ;   1.5625d = 1.1001b
```

Zu a)

```
          01.1101
         -01.1001
          00.0100    pos. =>  q0 = 1

   shl    00.1000
         -01.1001
          10.1111    neg. =>  q-1 = 0

Restore   00.1000

   shl    01.0000
         -01.1001
          10.0111    neg. =>  q-2 = 0

Restore   01.0000

   shl    10.0000
         -01.1001
          00.0111    pos. =>  q-3 = 1
```

```
   shl   00.1110
        -01.1001
         11.0101   neg. =>  q-4 = 0

Restore  00.1110

   shl   01.1100
        -01.1001
         00.0011   pos. =>  q-5 = 1
```

Wir erhalten somit folgendes (n+1)-stelliges Ergebnis für den Quotienten Q:

```
   Q[n+1] = 1.00101b = 1.15625
```

Da die Bitstelle q_0=1 einem ganzzahligen Anteil entspricht, ist der Quotient eine unecht gebrochene Dualzahl. Für die im nächsten Abschnitt betrachtete Gleitkomma-Division bedeutet dies, dass der Quotient bereits in normalisierter Form vorliegt. Um ein n-stelliges Ergebnis zu erhalten, wird die niederwertigste Stelle q_{-5} einfach abgeschnitten:

```
   Q[n] = 1.0010b = 1.125d
```

Es zeigt sich, dass aufgrund der begrenzten Stellenzahl in diesem Beispiel das exakte Ergebnis von 1.16 nur schlecht angenähert werden kann.

Zu b)

```
         01.1001
        -01.1101
         11.1100   neg. =>  q0 = 0

Restore  01.1001

   shl   11.0010
        -01.1101
         01.0101   pos. =>  q-1 = 1

   shl   10.1010
        -01.1101
         00.1101   pos. =>  q-2 = 1

   shl   01.1010
        -01.1101
         11.1101   neg. =>  q-3 = 0

Restore  01.1010

   shl   11.0100
        -01.1101
         01.0111   pos. =>  q-4 = 1

   shl   10.1110
        -01.1101
         01.0001   pos. =>  q-5 = 1
```

Das (n+1)-stellige Ergebnis für den Quotienten lautet:

```
   Q[n+1] = 0.11011b = 0.84375d
```

Bezogen auf die Gleitkomma-Division muss der Quotient aufgrund des fehlenden ganzzahligen Anteils normalisiert werden, d.h. das Komma wird – bei gleichzeitiger Verminderung des Exponenten – um eine Stelle nach rechts geschoben. Das n-stellige Mantissen-Ergebnis lautet folglich:

```
   Q[n] = 1.1011b
```

Beispiel 2.14 macht deutlich, dass bei der Division gebrochener Dualzahlen kein Divisionsrest angegeben wird. Weiterhin besitzen die Vorgabeoperanden, anders als bei ganzen Zahlen, die gleiche Wortlänge [Jork04]. Bild 2.22 zeigt ein an Bild 2.18 angelehntes Rechenwerk mit n=5 (eine Vor- und vier Nachkommastellen) zur Division gebrochener Dualzahlen.

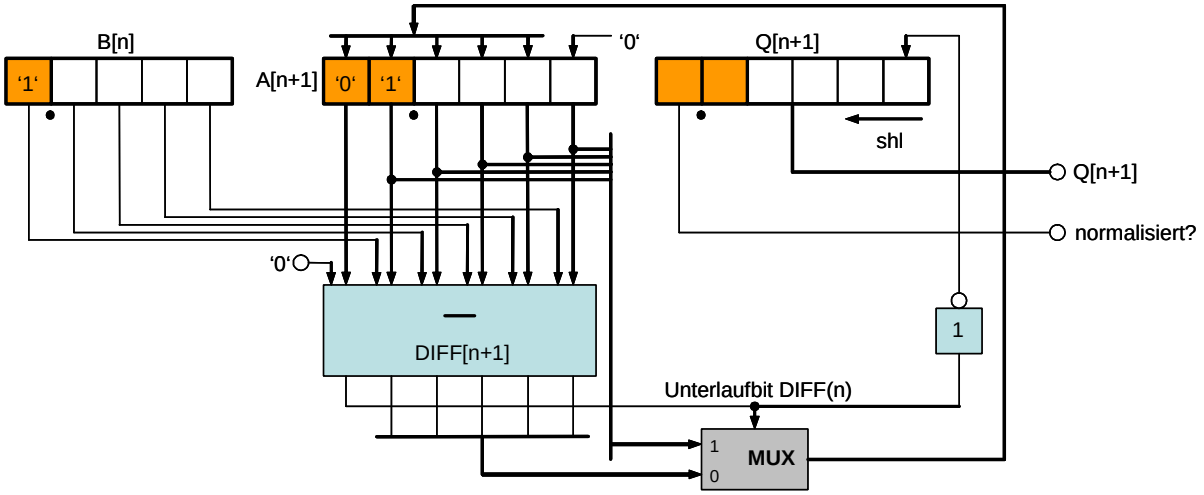


Bild 2.22: Rechenwerk zur Division gebrochener Dualzahlen mit Rückstellen des Zwischenrestes

Die Arbeitsweise ist ähnlich zu Bild 2.18. Die Prüfung auf einen Quotientenüberlauf entfällt, da dieser hier nicht auftreten kann. Insgesamt wird (n+1)-mal eine (n+1)-Bit-Subtraktion ausgeführt. Ausgehend von dem Unterlaufbit DIFF(n) legt der Multiplexer entweder die Differenz DIFF[n-1:0] oder den Restore-Wert A[n-1:0] auf A(n:1), in beiden Fällen wird von rechts eine Null nachgezogen. Die Abspeicherung des invertierten Unterlaufbits erfolgt in jedem Takt im Quotienten-Register Q[n+1].

Im Hinblick auf die Gleitkomma-Division zeigt das Rechenwerk nach Bild 2.22 bereits die Abfrage auf ein normalisiertes Ergebnis. Der nachfolgend entwickelte VHDL-Quellcode berücksichtigt dies noch nicht. Er ist jedoch auf n=24 erweitert, so dass er unmittelbar zur 32-Bit-Gleitpunkt-Division im nächsten Abschnitt als Komponente instanziiert werden kann.

Vergleichbar zur Multiplikation in Abschnitt 2.2.2 werden nach einem Ladesignal für die Operanden (oder für den letzten der beiden Operanden) n+1=25 Verarbeitungstakte durch das Zustands-Flipflop QBUSY markiert. In diesen 25 Takten finden die Subtraktionen und Schiebeoperationen statt. Diese Verarbeitungstakte werden in Anlehnung an [Jork04] mit einem 5-stufigen Zähler QS[4:0] im Prozess states codiert:

```
states: process(CLK)
begin
  if rising_edge(CLK) then
     if LOAD = '1' then
        QBUSY <= '1'; QS <= "00000";
     end if;
     if QBUSY = '1' then
        QS <= QS + 1;
        if QS = "11000" then
           QBUSY <= '0';
        end if;
     end if;
  end if;
end process states;
```

Die Subtraktionen und Schiebeoperationen werden im Prozess divide ausgeführt. In diesem Prozess werden nach dem Ladesignal LOAD die 24-Bit-Operanden um ’0’ auf 25 Bit erweitert. Mit gesetztem Markierungssignal QBUSY wird in jedem Schritt die Differenz über die Prozess-Variable DIFF[24:0] gebildet und abhängig von der Auswertung des Unterlaufbits entweder die Differenz oder der Restore-Wert, jeweils um eine Stelle nach links mit nachgezogener Null geshiftet, im Register QA abgelegt. Gleichzeitig werden die invertierten Unterlaufbits in das Register QQ übernommen.

```
divide: process(CLK)
variable DIFF: std_logic_vector(24 downto 0);
begin
  if rising_edge(CLK) then
     if LOAD = '1' then
        QA <= "0" & A; QB <= "0" & B;
        QQ <= "00000000000000000000000000";
     end if;
     if QBUSY = '1' then
        DIFF:= QA - QB;
        if DIFF(24) = '1' then
           QA <= QA(23 downto 0) & '0';
        else
           QA <= DIFF(23 downto 0) & '0';
        end if;
        QQ <= QQ(23 downto 0) & not(DIFF(24));
     end if;
  end if;
end process divide;
```

Die vollständige VHDL-Beschreibung mit dem Namen DIV_24_Bit.vhd für die Division gebrochener Dualzahlen ist im Anhang A6 zu finden.

Beispiel 2.15:
Es sollen folgende unecht gebrochene Dualzahlen mit $n = 24$ Stellen dividiert werden:

```
A = 1.59423828125d = 1.10011000001000000000000b
B = 1.97412109375d = 1.11111001011000000000000b

Q = A / B = 0.80756863715063d
```

Die Stimulidaten zur Simulation stehen in der nachfolgenden do-Datei:

```
# Kommandodatei für 24-Bit-Division
restart -f
radix hex
force clk 0 0, 1 50ns -repeat 100ns
force load 0 0, 1 70ns, 0 360ns
force a x"CC1000"
force b x"FCB000"
run 3us
```

Bild 2.23 zeigt das Ergebnis der Simulation mit ModelSim. Die Verarbeitungszeit beträgt erwartungsgemäß 25 Takte.

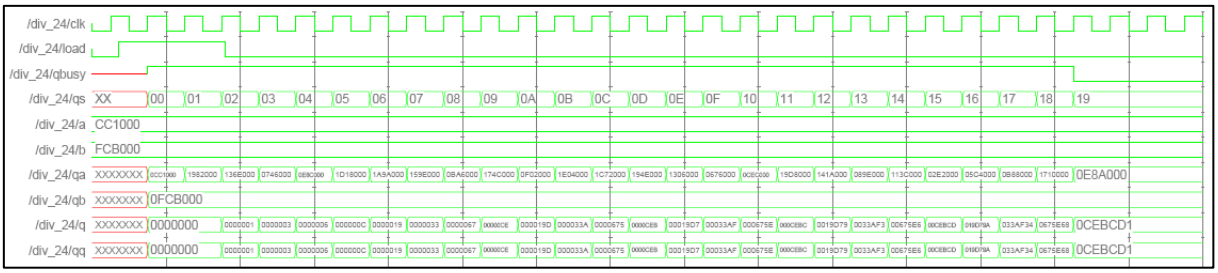


Bild 2.23: 24-Bit-Division der Zahlen 1.59423828125d und 1.97412109375d

Das im Register QQ stehende 25-Bit-Divisionsergebnis wird wie folgt interpretiert:

```
QQ = 0CEBCD1h = 0.1100|1110|1011|1100|1101|0001
```

Wird das denormalisierte Ergebnis nicht normalisiert, dann bleibt das niederwertigste Bit unberücksichtigt und man erhält folgendes 24-Bit-Ergebnis:

```
0.110011101011110011101000b = 0.80756855010987d
```

In Bezug auf Gleitkommazahlen muss eine Normalisierung durchgeführt werden, d.h. das Komma wird um eine Stelle nach rechts verschoben und der Exponent um 1 verringert. Durch diese Kommaverschiebung wird nun das zuvor nicht berücksichtigte niederwertigste Bit im Register QQ die niederwertigste Stelle der 23-Bit-Mantisse:

```
1.10011101011110011010001b * 2^(-1) = 0.8075686097145d
```

Man erkennt, dass in beiden Fällen das Ergebnis auf 6 Stellen nach dem Komma genau berechnet wird.

2.3.6 Erweiterung auf Gleitkommazahlen

Die Division von (normalisierten) Gleitkommazahlen kann – analog zur Multiplikation – in eine Mantissen- und eine Exponenten-Operation aufgeteilt werden. Die Mantissen der beiden Operanden werden dividiert und die Exponenten zur Basis 2 unabhängig davon subtrahiert, d.h. auch hier entfällt ein Angleichen der Exponenten. Aus den Vorzeichen der beiden Mantissen wird wieder das Vorzeichen des Resultats ermittelt.

$$q = \frac{a}{b} = mq \cdot 2^{eq} = \frac{ma \cdot 2^{ea}}{mb \cdot 2^{eb}} = \left(\frac{ma}{mb}\right) \cdot 2^{ea-eb} \tag{2.17}$$

Sind die Operanden gemäß Gl. (1.10) normalisiert, dann kann die Quotienten-Mantisse zwischen folgenden Grenzen liegen:

$$0.5 < \overline{mq} \leq 2 - 2^{-r} < 2 \tag{2.18}$$

Wie bereits im letzten Abschnitt erläutert (siehe Beispiel 2.14), ist der Quotient nicht notwendigerweise normalisiert. Ist er <1.0, dann wird das Komma um eine Stelle nach rechts verschoben und der Exponent entsprechend um 1 verringert. Bild 2.24 zeigt das zu Bild 2.14 sehr ähnlich aufgebaute Rechenwerk für die 32-Bit-Gleitkomma-Division. Mit dem 24-Bit-Dividierer aus Anhang A6 können die jeweils um das Hidden-Bit auf 24 Stellen erweiterten Mantissen dividiert werden. Es wird somit eine Instanz des VHDL-Moduls DIV24 eingesetzt, die in Bild 2.24 das 25-stellige Ergebnis Q[24:0] liefert. Für die Normalisierung muss die höchstwertige Bitstelle Q[24] ausgewertet werden.

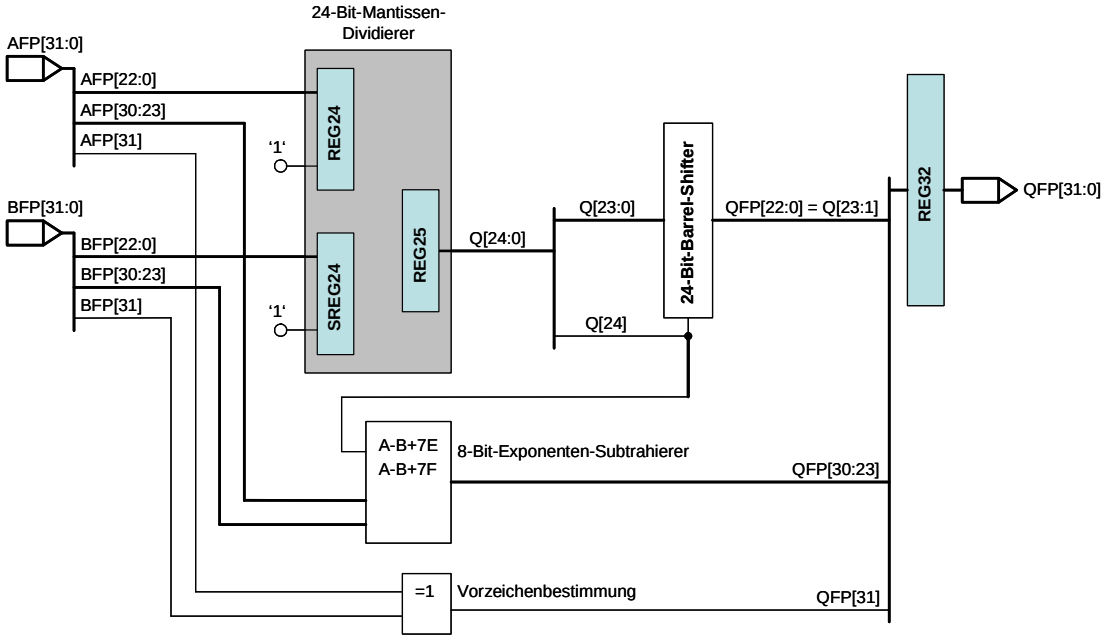


Bild 2.24: Rechenwerk für die 32-Bit-Gleitkomma-Division (in Anlehnung an [Jork04])

Ist diese gleich Null, dann erfolgt ein Linksshift um eine Stelle (entspricht Kommaverschiebung nach rechts), im anderen Fall wird keine Shiftoperation ausgeführt. Für die Ergebnis-Mantisse werden die 23 Bitstellen Q[23:1] übernommen. Durch die Subtraktion der Exponenten muss in jedem Falle ein Basiswert (bias=127d=7Fh) hinzugefügt werden. Bei bereits normalisierter Mantisse ist keine weitere Korrektur des Exponenten erforderlich. Ist hingegen Q[24] = '0', dann muss aufgrund der Kommaverschiebung nach rechts neben der Basiswert-Korrektur (Addition von 7Fh) eine Dekrementierung des Exponenten um Eins erfolgen, d.h. letztlich wird der Exponent um den Wert 7Eh erhöht.

Da die Operanden vorzeichenbehaftet sein können, wird das Ergebnisvorzeichen auch hier wieder durch eine EXOR-Verknüpfung gebildet. Das 32-Bit-Divisionsergebnis wird in einem Ausgangsregister QQFP[31:0] abgespeichert. Der vollständige VHDL-Quellcode besteht aus dem in A7 dargestellten File Top_Level_Modul.vhd sowie dem File DIV_24_Bit.vhd nach A5. Die Instanziierung der Komponente DIV24 und der Prozess validation sehen wie folgt aus:

```
ManDiv: DIV_24 port map (ManA,ManB,LOAD,CLK,ManQ,BUSY);
validation: process(CLK)
begin
  if rising_edge(CLK) then
     QD <= BUSY;
     if LOAD = '1' then QVALID <= '0'; end if;
     if (QD = '1' and BUSY = '0') then QVALID <= '1'; end if;
  end if;
end process validation;
```

Dieser Prozess ist identisch zu dem der Gleitkomma-Multiplikation (siehe Abschnitt 2.2.3), d.h. das Ladesignal LOAD für die Operanden startet die Division, deren zeitliche Steuerung im Modul DIV24 enthalten ist. Ist diese abgeschlossen (BUSY='0'), dann wird auch hier der Prozess validation bei der nächsten steigenden Flanke mit den Werten QD='1' und BUSY='0' ausgeführt. QD wird somit gelöscht (BUSY-Wert) und das Flipflop QVALID zur Signalisierung der Gültigkeit der Ergebnisse auf '1' gesetzt.

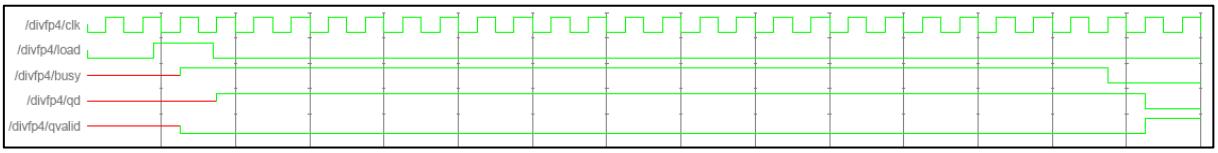


Bild 2.25: Generierung der Signale QD und QVALID im Prozess validation

Die Simulation nach Bild 2.25 zeigt, dass nach 25 Verarbeitungstakten für die Division auch hier die Signale qd und qvalid ihre Pegel einen Takt später wechseln. In diesem 26. Takt wird das Mantissen-Ergebnis – falls notwendig – normalisiert, der Exponent bestimmt und das komplette 32-Bit-Gleitkomma-Ergebnis einschließlich Vorzeichen abgespeichert. Diese Anweisungen erfolgen im Prozess correct:

```
correct: process(CLK)
begin
  if rising_edge(CLK) then
     if (QD = '1' and BUSY = '0') then
        if ManQ(24) = '0' then
           QQFP(22 downto 0) <= ManQ(22 downto 0);
           QQFP(30 downto 23) <= AFP(30 downto 23)
                              - BFP(30 downto 23) + X"7F" - 1;  -- Verschiebung des Kommas
                                                                -- berücksichtigen
        else
           QQFP(22 downto 0) <= ManQ(23 downto 1);
           QQFP(30 downto 23) <= AFP(30 downto 23)
                              - BFP(30 downto 23) + X"7F";
        end if;
        QQFP(31) <= AFP(31) xor BFP(31);
     end if;
  end if;
end process correct;
```

Die Normalisierung und Abspeicherung findet durch die UND-Verknüpfung der Signale QD und BUSY nur im 26. Takt statt.

Bei der Division können nach [Hoff93] folgende Fehler auftreten:

- Undefiniertes Ergebnis (0/0)
- Division durch 0 (x/0=∞)
- Quotient zu groß (Exponentenüberlauf)
- Quotient zu klein (Exponentenunterlauf)

Die beiden ersten Fälle lassen sich zu dem Fehler “Division durch Null“ zusammenfassen, bei einem Exponentenunterlauf (unechte Null) kann der Quotient gleich Null gesetzt werden. Der in diesem Abschnitt erläuterte VHDL-Quellcode fängt diese möglichen Fehler nicht ab, d.h. für eine praxistaugliche Floating-Point Unit sind hier noch Ergänzungen notwendig!

2.3.7 Simulation

Für die Simulation der Gleitkomma-Division sollen die zuvor bei der Multiplikation benutzten Zahlenwerte verwendet werden:

```
a)  afp = +1.6875d = 3FD80000h
    bfp = +1.5000d = 3FC00000h

           => qfp = +1.125d = 3F900000h

b)  afp = -2.625d = C0280000h
    bfp = +1.750d = 3FE00000h

          => qfp = -1.500d = BFC00000h
```

Die do-Datei für die Stimulidaten sieht wie folgt aus:

```
# Kommandodatei für Gleitkomma-Division
restart -f
radix hex
force clk 0 0, 1 50ns -repeat 100ns
force afp x"XXXXXXXX"  # Operand einsetzen
force bfp x"XXXXXXXX"  # Operand einsetzen
force load 0 0, 1 180ns, 0 340ns
run 3us
```

Die Bilder 2.26 und 2.27 zeigen die Ergebnisse der Simulation. Man erkennt, dass der Quotient der Gleitkomma-Division jeweils nach 26 Taktzyklen im Register QQFP steht.

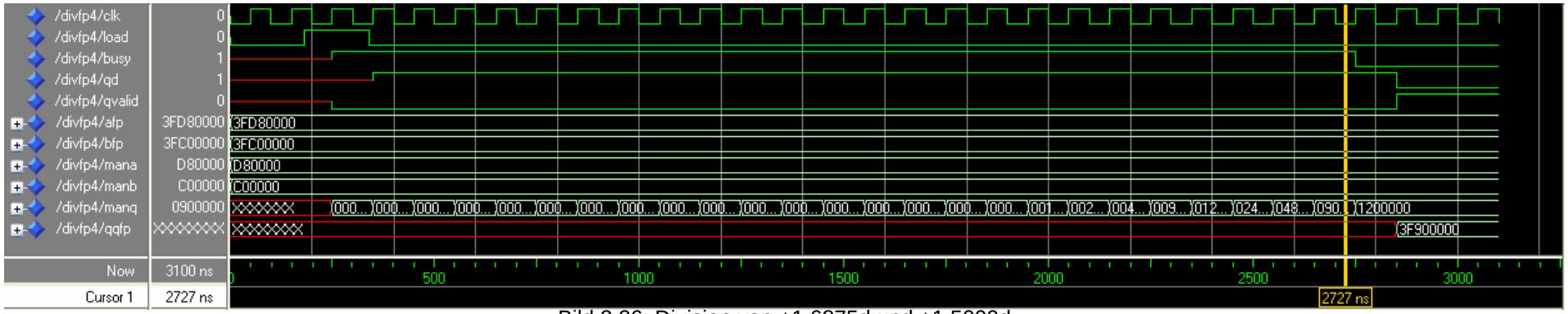


Bild 2.26: Division von +1.6875d und +1.5000d

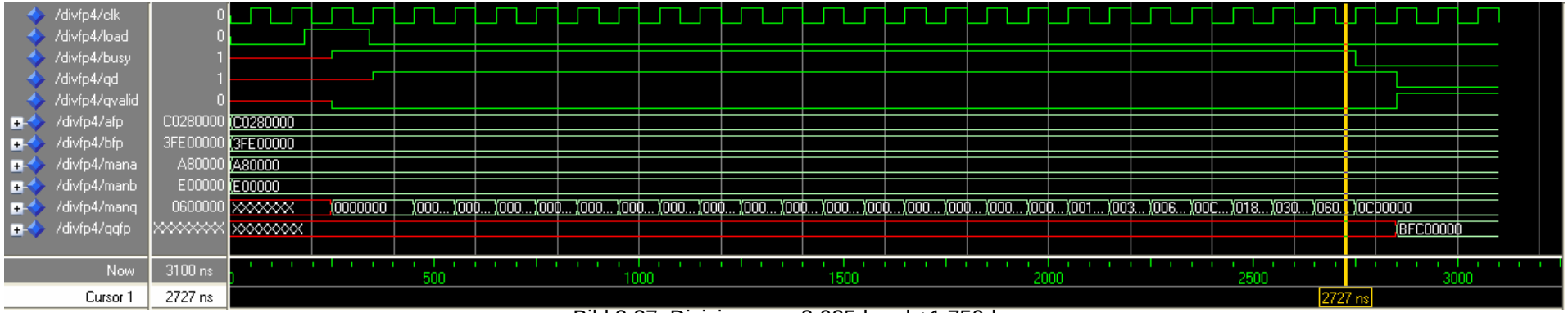


Bild 2.27: Division von -2.625d und +1.750d

A Anhang

A1 VHDL-Module für die 32-Bit-Gleitkomma-Addition

```
-- Top_Level_Modul.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ADD_4BYTE_FP is
port(CLK,RESET,START_ADD: in std_logic;
     AFP,BFP: in std_logic_vector(31 downto 0);
     READY_ADD: out std_logic;
     SFP: out std_logic_vector(31 downto 0));
end ADD_4BYTE_FP;

architecture ADD_4BYTE_FP_A of ADD_4BYTE_FP is

component SHFT24R
port(X: in std_logic_vector(23 downto 0);
     COUNT: in std_logic_vector(7 downto 0);
     SB_1,SB_2: in std_logic;
     Y: out std_logic_vector(23 downto 0));
end component SHFT24R;

type ZUSTAENDE is (Z0,Z1,Z2,Z3,Z4,Z5);    --Aufzählungstyp
signal ZUSTAND,FOLGE_Z: ZUSTAENDE;        --Interprozesskommunikation

signal AgtB,BgtA,Azero,Bzero: std_logic;
signal ManA,ManB,ManAshft,ManBshft: std_logic_vector(23 downto 0);
signal CountA,CountB: std_logic_vector(7 downto 0);
signal QManS: std_logic_vector(24 downto 0);
signal QExpS: std_logic_vector(7 downto 0);
signal QSFP: std_logic_vector(31 downto 0);
signal QS: std_logic_vector(3 downto 0);
signal NORMALISIERUNG: std_logic;

begin
  ManA <= '1' & AFP(22 downto 0);
  ManB <= '1' & BFP(22 downto 0);
  AgtB <= '1' when AFP(30 downto 23) > BFP(30 downto 23) else '0';
  BgtA <= '1' when BFP(30 downto 23) > AFP(30 downto 23) else '0';
  CountA <= BFP(30 downto 23) - AFP(30 downto 23) when BgtA = '1' else "00000000";
  CountB <= AFP(30 downto 23) - BFP(30 downto 23) when AgtB = '1' else "00000000";
  shftA: SHFT24R port map (ManA,CountA,AFP(31),BFP(31),ManAshft);
  shftB: SHFT24R port map (ManB,CountB,BFP(31),AFP(31),ManBshft);
  Azero <= '1' when (AFP(30 downto 0) = "00000000000000000000000000000000") else '0';
  Bzero <= '1' when (BFP(30 downto 0) = "00000000000000000000000000000000") else '0';

  Z_SPEICHER: process(CLK,RESET)
  begin
    if RESET = '0' then
       ZUSTAND <= Z0;
       QS <= "0000";
    elsif CLK = '1' and CLK'event then
       ZUSTAND <= FOLGE_Z;
         case ZUSTAND is
           when Z0 => QS <= "0000";
           when Z4 => QS <= QS + 1;
           when others =>
         end case;
    end if;
  end process Z_SPEICHER;

  F_BEST: process(START_ADD,ZUSTAND,NORMALISIERUNG)
  begin
    case ZUSTAND is
      when Z0 =>   if START_ADD = '0' then FOLGE_Z <= Z1;
                   else FOLGE_Z <= Z0;
                   end if;

      when Z1 =>   if (Azero = '1' or Bzero = '1') then FOLGE_Z <= Z5;
```

```
                    -- mindestens ein Operand ist 0
                    elsif (AFP(31) = BFP(31)) then FOLGE_Z <= Z2;
                    -- beide Vorzeichen identisch
                    else FOLGE_Z <= Z3;      --beide Vorzeichen ungleich
                    end if;

       when Z2 =>   FOLGE_Z <= Z0;

       when Z3 =>   FOLGE_Z <= Z4;

       when Z4 =>   if NORMALISIERUNG = '1' then FOLGE_Z <= Z4;
                    else FOLGE_Z <= Z0;
                    end if;

       when Z5 =>   FOLGE_Z <= Z0;
     end case;
   end process F_BEST;

   A_BEST: process(ZUSTAND,QS)
   begin
     case ZUSTAND is
       when Z0 =>   NORMALISIERUNG <= '1';
                    READY_ADD <= '1';

       when Z1 =>   READY_ADD <= '0';
                    QManS <= ('0' & ManAshft) + ('0' & ManBshft);
                    if AgtB = '1' then
                       QExpS <= AFP(30 downto 23);
                    else
                       QExpS <= BFP(30 downto 23);
                    end if;

       when Z2 =>   if QManS(24) = '1' then      -- Mantissenübertrag
                       QSFP(22 downto 0) <= QManS(23 downto 1);
                       QSFP(30 downto 23) <= QExpS + 1;
                    else
                       QSFP(22 downto 0) <= QManS(22 downto 0);
                       QSFP(30 downto 23) <= QExpS;
                    end if;
                    QSFP(31) <= BFP(31);

       when Z3 =>   if QManS(24) = '0' then
                       -- kein Mantissenübertrag => Ergebnis negativ
                       -- Komplementbildung
                       QManS(23 downto 0) <= (not QManS(23 downto 0) + 1);
                       QSFP(31) <= '1';  -- Ergebnis negativ
                    else
                       QSFP(31) <= '0';  -- Ergebnis positiv
                    end if;

       when Z4 =>   if QManS(23 downto 0) = X"000000" then
                       -- Ausnahmebehandlung
                       -- QSFP wird hier auf X"00000000" gesetzt
                       QSFP <= X"00000000";
                       NORMALISIERUNG <= '0';
                    else
                       if QManS(23) = '1' then
                          QSFP(22 downto 0) <= QManS(22 downto 0);
                          QSFP(30 downto 23) <= QExpS;
                          NORMALISIERUNG <= '0';
                       elsif QManS(22) = '1' then
                          QSFP(22 downto 0) <= QManS(21 downto 0) & '0';
                          QSFP(30 downto 23) <= QExpS - 1;
                          NORMALISIERUNG <= '0';
                       else
                          QManS(23 downto 0) <= QManS(21 downto 0) & "00";
                          QExpS <= QExpS - 2;
                       end if;
                    end if;

       when Z5 =>   if (Azero = '1' and Bzero = '0') then QSFP <= BFP;     -- Operand AFP ist 0
                    else QSFP <= AFP;  -- Operand BFP ist 0 oder beide Operanden sind 0
                    end if;
     end case;
   end process A_BEST;
   SFP <= QSFP;

end ADD_4BYTE_FP_A;
```

```
-- 24_Bit_Barrelshifter.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity SHFT24R is
port(X: in std_logic_vector(23 downto 0);
     COUNT: in std_logic_vector(7 downto 0);
     SB_1,SB_2: in std_logic;   -- signed bits
     Y: out std_logic_vector (23 downto 0));
end SHFT24R;

architecture SHFT24R_A of SHFT24R is
signal M: std_logic_vector(23 downto 0);
signal SELECTOR: std_logic_vector(1 downto 0);
begin
  SELECTOR <= SB_1 & SB_2;
  with COUNT select
    M <=      X              when X"00",
         '0' & X(23 downto  1) when X"01",
         "00" & X(23 downto  2) when X"02",
         "000" & X(23 downto  3) when X"03",
         "0000" & X(23 downto  4) when X"04",
         "00000" & X(23 downto  5) when X"05",
         "000000" & X(23 downto  6) when X"06",
         "0000000" & X(23 downto  7) when X"07",
         "00000000" & X(23 downto  8) when X"08",
         "000000000" & X(23 downto  9) when X"09",
         "0000000000" & X(23 downto 10) when X"0A",
         "00000000000" & X(23 downto 11) when X"0B",
         "000000000000" & X(23 downto 12) when X"0C",
         "0000000000000" & X(23 downto 13) when X"0D",
         "00000000000000" & X(23 downto 14) when X"0E",
         "000000000000000" & X(23 downto 15) when X"0F",
         "0000000000000000" & X(23 downto 16) when X"10",
         "00000000000000000" & X(23 downto 17) when X"11",
         "000000000000000000" & X(23 downto 18) when X"12",
         "0000000000000000000" & X(23 downto 19) when X"13",
         "00000000000000000000" & X(23 downto 20) when X"14",
         "000000000000000000000" & X(23 downto 21) when X"15",
         "0000000000000000000000" & X(23 downto 22) when X"16",
         "00000000000000000000000" & X(23)           when X"17",
         "000000000000000000000000"                   when others;

  with SELECTOR select
    Y <= ((not M) + 1) when "10",
         M when others;

end SHFT24R_A;
```

A2 VHDL-Modul für die 8-Bit-Multiplikation

```
-- Mult_n_Bit.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity MUL8N is
generic (n: integer:=3; w: integer:=2**3);
port (A,B: in std_logic_vector(w-1 downto 0);
      LOAD,CLK: in std_logic;
      PH,PL: out std_logic_vector(w-1 downto 0));
end MUL8N;

architecture MUL8N_A of MUL8N is
signal QA,QB,QPH: std_logic_vector(w-1 downto 0);
signal QBIT: std_logic_vector(n-1 downto 0);
signal QBUSY: std_logic;
begin
  states: process(CLK)
  variable vsw: std_logic;
  begin
    if rising_edge(CLK) then
       if LOAD = '1' then
          QBUSY <= '1';
          for I in 0 to n-1 loop
            QBIT(I) <= '0';
          end loop;
       end if;
       if QBUSY = '1' then
          QBIT <= QBIT + 1;
          vsw:='1';
          for I in 0 to n-1 loop
            vsw:=vsw and QBIT(I);
          end loop;
          if (vsw = '1') then QBUSY <= '0'; end if;
       end if;
    end if;
  end process states;

  mult: process(CLK)
  variable SUM: std_logic_vector(w downto 0);
  begin
    if rising_edge(CLK) then
       if LOAD = '1' then
          QA <= A; QB <= B;
          for I in 0 to w-1 loop
            QPH(I) <= '0';
          end loop;
       end if;
       if QBUSY = '1' then
          for I in 0 to w-1 loop
            SUM(I):=QA(I) and QB(0);
          end loop;
          SUM(w):= '0';
          SUM := SUM + ('0' & QPH);
          QB <= SUM(0) & QB(w-1 downto 1);
          QPH <= SUM(w downto 1);
       end if;
    end if;
  end process mult;
  PH <= QPH; PL <= QB;
end MUL8N_A;
```

A3 VHDL-Modul für die 24-Bit-Multiplikation

```
-- Mult_24_Bit.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity MUL24 is
port (A,B: in std_logic_vector(23 downto 0);
      LOAD,CLK: in std_logic;
      PH,PL: out std_logic_vector(23 downto 0);
      QBUSY: inout std_logic);
end MUL24;

architecture MUL24_A of MUL24 is
signal QA,QB,QPH: std_logic_vector(23 downto 0);
signal QBIT: std_logic_vector(4 downto 0);
begin
  states: process(CLK)
  variable vsw: std_logic;
  begin
    if rising_edge(CLK) then
       if LOAD = '1' then
          QBUSY <= '1';
          QBIT <= "00000";
       elsif QBUSY = '1' then
          QBIT <= QBIT + 1;
          if QBIT = "10111" then
             QBUSY <= '0';
          end if;
       else
          QBIT <= "00000";
       end if;
    end if;
  end process states;

  mult: process(CLK)
  variable PrtPrd, SUM: std_logic_vector(24 downto 0);
  begin
    if rising_edge(CLK) then
       if LOAD = '1' then
          QA <= A; QB <= B; QPH <= X"000000";
       elsif QBUSY = '1' then
          PrtPrd:='0'&(QA and
            (QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&
             QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&
             QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)&QB(0)));
          SUM:=PrtPrd + ('0' & QPH);
          QB <= SUM(0) & QB(23 downto 1);
          QPH <= SUM(24 downto 1);
       end if;
    end if;
  end process mult;
  PH <= QPH; PL <= QB;
end MUL24_A;
```

A4 VHDL-Modul für die 32-Bit-Gleitkomma-Multiplikation

```
-- Top_Level_Modul.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity MULFP4 is
port (AFP,BFP: in std_logic_vector(31 downto 0);
      LOAD,CLK: in std_logic;
      PFP: out std_logic_vector(31 downto 0);
      VALID: out std_logic);
end MULFP4;

architecture MULFP4_A of MULFP4 is
component MUL24 is
port (A,B: in std_logic_vector(23 downto 0);
      LOAD,CLK: in std_logic;
      PH, PL: out std_logic_vector(23 downto 0);
      QBUSY: inout std_logic);
end component MUL24;

signal ManA,ManB,PH,PL: std_logic_vector(23 downto 0);
signal BUSY: std_logic;
signal QPFP: std_logic_vector(31 downto 0);
signal QVALID, QD: std_logic;

begin
  ManA <= '1' & AFP(22 downto 0);
  ManB <= '1' & BFP(22 downto 0);
  ManMul: MUL24 port map (ManA,ManB,LOAD,CLK,PH,PL,BUSY);
  validation: process(CLK)
  begin
    if rising_edge(CLK) then
       QD <= BUSY;
       if LOAD='1' then QVALID<= '0'; end if;
       if (QD='1' and BUSY='0') then QVALID <= '1'; end if;
    end if;
  end process validation;

  correct: process(CLK)
  begin
    if rising_edge(CLK) then
       if (QD='1' and BUSY='0') then
          if PH(23)='1' then
             QPFP(22 downto 0) <= PH(22 downto 0);
             QPFP(30 downto 23) <= AFP(30 downto 23)
                                   + BFP(30 downto 23) - X"7E";
          else
             QPFP(22 downto 0) <= PH(21 downto 0) & PL(23);
             QPFP(30 downto 23) <= AFP(30 downto 23)
                                   + BFP(30 downto 23) - X"7F";
          end if;
          QPFP(31) <= AFP(31) xor BFP(31);
       end if;
    end if;
  end process correct;
  PFP <= QPFP; VALID <= QVALID;
end MULFP4_A;
```

A5 VHDL-Modul für die 8-Bit-Division

```
-- Div_8_Bit.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity DIV8 is
port (AL,AH,B: in std_logic_vector(7 downto 0);
      START_DIFF,CLK,RESET: in std_logic;
      Q,REST: out std_logic_vector(7 downto 0));
end DIV8;

architecture DIV8_A of DIV8 is
signal QAL,QB: std_logic_vector(7 downto 0);
signal QAH: std_logic_vector(8 downto 0);
signal QS: std_logic_vector(2 downto 0);
type ZUSTAENDE is (Z0,Z1,Z2,Z3);            --Aufzählungstyp
signal ZUSTAND,FOLGE_Z: ZUSTAENDE;          --Interprozesskommunikation

begin
Z_SPEICHER: process(CLK,RESET)
begin
  if RESET = '0' then
     ZUSTAND <= Z0;
     QS <= "000";
  elsif CLK = '1' and CLK'event then
        ZUSTAND <= FOLGE_Z;
          case ZUSTAND is
            when Z0 => QS <= "000";

            when Z2 => QS <= QS + 1;

            when others =>
          end case;
  end if;
end process Z_SPEICHER;

F_BEST: process(START_DIFF,ZUSTAND,QS)
variable Q_ÜBERLAUF: std_logic_vector(8 downto 0);
begin
  case ZUSTAND is
    when Z0 =>   if START_DIFF = '1' then FOLGE_Z <= Z1; end if;

    when Z1 =>   if START_DIFF = '1' then
                    Q_ÜBERLAUF := QAH  - ('0' & QB);
                    if Q_ÜBERLAUF(8) = '0' then
                       FOLGE_Z <= Z0;
                    else
                       FOLGE_Z <= Z2;
                    end if;
                 end if;

    when Z2 =>   if QS = "110" then FOLGE_Z <= Z3; end if;

    when Z3 =>   FOLGE_Z <= Z0;

  end case;
end process;

A_BEST: process(ZUSTAND,QS)
variable DIFF: std_logic_vector(8 downto 0);
begin
  case ZUSTAND is
    when Z0 =>   QAL <= AL; QAH <= '0'&AH; QB <= B;

    when Z1 =>   DIFF := QAH  - ('0' & QB);
                 if DIFF(8) = '1' then
                    -- nur Linksshift
                    QAH(8 downto 0) <= QAH(7 downto 0) & QAL(7);
                    QAL(7 downto 0) <= QAL(6 downto 0) & not(DIFF(8));
                 end if;

    when Z2 =>   DIFF := QAH  - ('0' & QB);
                 if DIFF(8) = '0' then
                    QAH(8 downto 0) <= DIFF(7 downto 0) & QAL(7);
```

```
                else
                   QAH(8 downto 0) <= QAH(7 downto 0) & QAL(7);
                end if;
                QAL(7 downto 0) <= QAL(6 downto 0) & not(DIFF(8));

    when Z3 =>  DIFF := QAH  - ('0' & QB);
                if DIFF(8) = '0' then
                   QAH(7 downto 0) <= DIFF(7 downto 0);
                   REST <= DIFF(7 downto 0);
                else
                   REST <= QAH(7 downto 0);
                end if;
                QAL(7 downto 0) <= QAL(6 downto 0) & not(DIFF(8));
                Q <= QAL(6 downto 0) & not(DIFF(8));
    end case;
  end process;
end DIV8_A;
```

A6 VHDL-Modul für die 24-Bit-Division gebrochener Dualzahlen

```
-- DIV_24_Bit.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity DIV24 is
port (A,B: in std_logic_vector(23 downto 0);
      LOAD,CLK: in std_logic;
      Q: out std_logic_vector(24 downto 0);
      QBUSY: inout std_logic);
end DIV24;

architecture DIV_24_A of DIV_24 is
signal QA,QB: std_logic_vector(24 downto 0);
signal QQ: std_logic_vector(24 downto 0);
signal QS: std_logic_vector(4 downto 0);
begin
  states: process(CLK)
  begin
    if rising_edge(CLK) then
       if LOAD = '1' then
          QBUSY <= '1'; QS <= "00000";
       end if;
       if QBUSY = '1' then
          QS <= QS + 1;
          if QS = "11000" then
             QBUSY <= '0';
          end if;
       end if;
    end if;
  end process states;

  divide: process(CLK)
  variable DIFF: std_logic_vector(24 downto 0);
  begin
    if rising_edge(CLK) then
       if LOAD = '1' then
          QA <= "0" & A; QB <= "0" & B;
          QQ <= "00000000000000000000000000";
       end if;
       if QBUSY = '1' then
          DIFF:= QA - QB;
          if DIFF(24) = '1' then
             QA <= QA(23 downto 0) & '0';
          else
             QA <= DIFF(23 downto 0) & '0';
          end if;
          QQ <= QQ(23 downto 0) & not(DIFF(24));
       end if;
    end if;
  end process divide;
  Q <= QQ;
end DIV_24_A;
```

A7 VHDL-Modul für die 32-Bit-Gleitkomma-Division

```
-- Top_Level_Modul.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity DIVFP4 is
port (AFP,BFP: in std_logic_vector(31 downto 0);
      LOAD,CLK: in std_logic;
      QFP: out std_logic_vector(31 downto 0);
      VALID: out std_logic);
end DIVFP4;

architecture DIVFP4_A of DIVFP4 is
component DIV24
port (A,B: in std_logic_vector(23 downto 0);
      LOAD,CLK: in std_logic;
      Q: out std_logic_vector(24 downto 0);
      QBUSY: inout std_logic);
end component DIV24;

signal ManA,ManB: std_logic_vector(23 downto 0);   -- Mantissen
signal ManQ: std_logic_vector(24 downto 0);        -- Mantissen-Quotient
signal QQFP: std_logic_vector(31 downto 0);        -- Ausgangs-Register
signal BUSY: std_logic;                            -- Zustands-Flipflop
signal QVALID,QD: std_logic;                       -- Flipflops

begin
  ManA <= '1' & AFP(22 downto 0);
  ManB <= '1' & BFP(22 downto 0);
  ManDiv: DIV24 port map (ManA,ManB,LOAD,CLK,ManQ,BUSY);

  validation: process(CLK)
  begin
    if rising_edge(CLK) then
       QD <= BUSY;
       if LOAD = '1' then QVALID <= '0'; end if;
       if (QD = '1' and BUSY = '0') then QVALID <= '1'; end if;
    end if;
  end process validation;

  correct: process(CLK)
  begin
    if rising_edge(CLK) then
       if (QD = '1' and BUSY = '0') then
          if ManQ(24) = '0' then
             QQFP(22 downto 0) <= ManQ(22 downto 0);
             QQFP(30 downto 23) <= AFP(30 downto 23)
                                - BFP(30 downto 23) + X"7F" - 1;  -- Verschiebung des Kommas
                                                                  -- berücksichtigen
          else
             QQFP(22 downto 0) <= ManQ(23 downto 1);
             QQFP(30 downto 23) <= AFP(30 downto 23)
                                - BFP(30 downto 23) + X"7F";
          end if;
          QQFP(31) <= AFP(31) xor BFP(31);
       end if;
    end if;
  end process correct;
  QFP <= QQFP; VALID <= QVALID;
end DIVFP4_A;
```

Literaturverzeichnis

[Flik93] Flik, T.; Liebig, H.: **Rechnerorganisation – Prinzipien, Strukturen, Algorithmen**; 2. Auflage, Springer Verlag, 1993.

[Flik98] Flik, T.; Liebig, H.: **Mikroprozessortechnik**; 5. Auflage, Springer Verlag, 1998.

[Hoff93] Hoffmann, R.: **Rechnerentwurf: Rechenwerke, Mikroprogrammierung, RISC**; 3. Auflage, Oldenbourg Verlag, 1993.

[Jork04] Jorke, G.: **Rechnergestützter Entwurf digitaler Schaltungen – Schaltungssynthese mit VHDL**; 1. Auflage, Carl Hanser Verlag, 2004.

[Pirs96] Pirsch, P.: **Architekturen der digitalen Signalverarbeitung**; 1. Auflage, Teubner Verlag, 1996.

[ReSc03] Reichardt, J.; Schwarz, B.: **VHDL-Synthese – Entwurf digitaler Schaltungen und Systeme**; 3. Auflage, Oldenbourg Verlag, 2003.

[ScSc99] Schiffmann, W.; Schmitz, R.: **Technische Informatik 2 – Grundlagen der Computertechnik**; 3. Auflage, Springer Verlag, 1999.

[SeBe98] Seifart, M.; Beikirch, H.: **Digitale Schaltungen**; 5. Auflage, Verlag Technik, 1998.