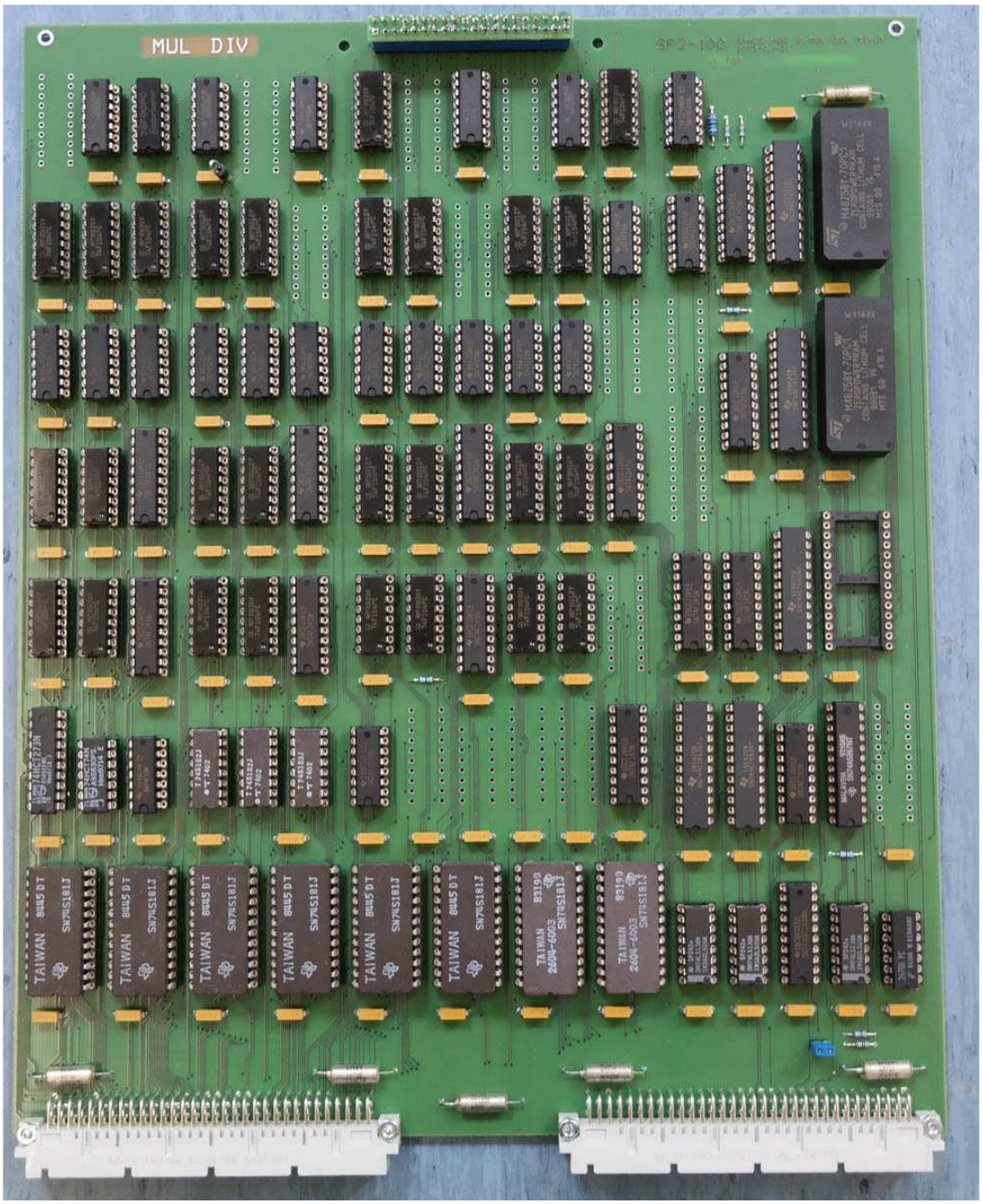
Dokumentation
MulDiv Karte
für den SpaceAge 2



1. Theorie der Rechenvorschriften

In diesem Kapitel werden die Rechenvorschriften für die binären Punktrechenarten hergeleitet und wie sich diese jeweils in Hardware abbilden lassen. Dies beginnt mit der Herleitung der vorzeichenlosen Multiplikation aus der Addition. Im Anschluss werden die Differenzen der anderen Rechenvorschriften aus Dieser herausgearbeitet.

1.1 Multiplikation ohne Vorzeichen

Eine vorzeichenlose Multiplikation besteht aus einer wiederholten Addition des gleichen Summanden B mit der Anzahl A. A wird hierbei Multiplikator genannt und B ist der Multiplikand.

Der erste Ansatz wäre nun diese wiederholte Addition durchzuführen. Dies mag bei 4 Bit Operanden noch ein verfolgbarer Ansatz sein mit maximal 15 Wiederholungen, aber bei der 32Bit Wortbreite des MIPS ergeben sich hier bis zu $2^{32}-1=4294967295$ Wiederholungen der Addition. Bei einem Prozessortakt des Spaceage 2 von 4MHz würde diese Operation 1073 Sekunden dauern.
Daher muss ein anderer Ansatz gewählt werden, das schriftliche Multiplizieren, bekannt aus der Grundschule:

```
 1 2 3 x 1 2 3
---------------
       3 6 9
     2 4 6
   1 2 3
---------------
   1 5 1 2 9
```

Abbildung 1

Bei dieser Vorschrift wird zuerst die 1er Stelle des Multiplikators mit dem Multiplikand multipliziert und unter diesen geschrieben. Im nächsten Schritt wird die 10er Stelle des Multiplikators herangezogen und das Ergebnis wird um eine Stelle nach links verschoben bei den Partialprodukten hingeschrieben. Dies wiederholt sich bis alle Stellen des Multiplikators durchiteriert wurden. Anschließend werden die Partialprodukte zum endgültigen Produkt addiert.
Diese Vorschrift lässt sich 1 zu 1 ins Binäre übertragen:

a_3	a_2	a_1	a_0	x	b_3	b_2	b_1	b_0	
					a_0b_3	a_0b_2	a_0b_1	a_0b_0	
				a_1b_3	a_1b_2	a_1b_1	a_1b_0		Partial-
			a_2b_3	a_2b_2	a_2b_1	a_2b_0			produkte
		a_3b_3	a_3b_2	a_3b_1	a_3b_0				
		…	…	…	$a_0b_1 + a_1b_0$		a_0b_0		Produkt

Abbildung 2

Dabei gilt, dass eine Multiplikation 2er Bits im Binären eine UND Verknüpfung darstellt:

axb=y	a	b	y
0x0=0	0	0	0
0x1=0	0	1	0
1x0=0	1	0	0
1x1=1	1	1	1

Abbildung 3

Dies bedeutet sobald $a_x = 0$ ist, so ist auch das dazu gehörige Partialprodukt 0.
Ein Beispiel mit 5x7=35

```
0 1 0 1 x 0 1 1 1
          0 1 1 1
        0 0 0 0
      0 1 1 1
    0 0 0 0
0 0 1 0 0 0 1 1
```

Abbildung 4

Für diese Rechenvorschrift würden jetzt 4 Speicherstellen/Register zu 4Bit benötigt. Die Partialprodukte lassen sich aber bereits nach deren Entstehung direkt mit der Summe der vorherigen Partialsummen addieren. Dabei ist zu beachten, dass ein Register zur Aufsummierung zuerst mit 0 initialisiert werden muss.

Für die Multiplikation des 32Bit Spaceage 2 wird demnach ein 64Bit Ergebnisregister benötigt, denn es gilt: $2^{32} * 2^{32} = 2^{64}$. Dieses Register ist zum Start der Multiplikation mit dem Wert 0 initialisiert. Damit der Multiplikand immer an der richtigen Stelle für das Partialprodukt steht muss dieser auch pro Rechentakt um ein Bit nach links geschoben werden. Demnach wird für den Multiplikanden auch ein 64Bit Register benötigt, in diesem Falle in der Ausführung als Schieberegister. Von dem Multiplikator wird jeweils nur das LSB benötigt. Daraus ergibt sich ein 32Bit Schieberegister, das pro Rechentakt um ein Bit nach rechts schiebt. Der Addierer muss somit auch 64Bit breit sein, da er den Inhalt des 64Bit Ergebnisregisters mit dem 64Bit Multiplikandenregister addieren muss.
Ist das LSB des Multiplikatorregisters = 1, so wird das Ergebnis der Addition in das Ergebnisregister geschrieben, sonst nicht. Anschließend werden die Schieberegister nach Vorschrift geschoben.

Daraus ergibt sich folgendes Schaltbild:

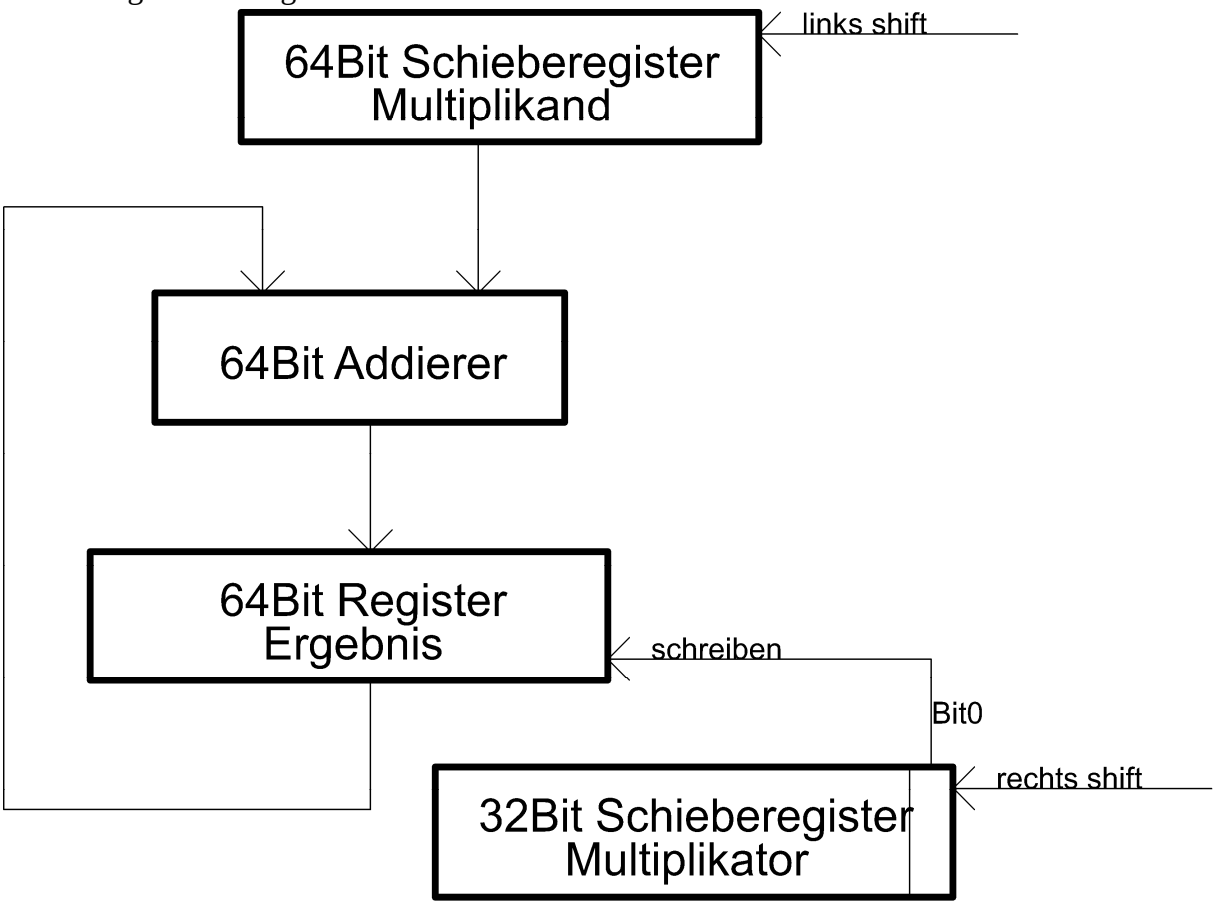


Abbildung 5

Zwei Register zu 64Bit Größe und ein 64Bit Addierer stellen einen hohen Hardwareaufwand dar. Daher muss eine Lösung gefunden werden welche weniger Hardwareressourcen verbraucht.

Wie in Abbildung 2 ersichtlich wird das LSB des ersten Partialprodukts nach der ersten Schiebung des Multiplikanden nicht mit einem weiteren Partialprodukt aufsummiert. Dies gilt nach dem zweiten Takt auch für das zweite Bit des Partialprodukts.
Anstatt den Multiplikanden nach links zu schieben bleibt dieser nun fest stehen und das Ergebnisregister des Addierers wird logisch nach rechts geschoben. Somit müssen nur noch 32Bit Additionen ausgeführt werden und das 64Bit Schieberegister des Multiplikanden entfällt. Dafür wird nun ein 64Bit Schieberegister für das Ergebnis benötigt.
Das Ergebnisregister wird aufgeteilt in 2 Schieberegister zu 32Bit, diese heißen „HI“ und „LO“. Denn HI benötigt 2 Operationen: das Ergebnis der Addition speichern und dann nach rechts schieben. Das Register LO wird nur nach rechts geschoben und nicht geschrieben.

Weiterhin ist zu beachten, dass das Ergebnis einer Addition zweier 32Bit Zahlen 33Bit beanspruchen kann. Da im neuen Schaltungsentwurf das beschreibbare Register HI dieselbe Breite hat wie die Operanden, kann es zu einem Overflow kommen der das Ergebnis verfälscht. Beispielhaft bei der Multiplikation 4294967295x7. Die Erste Addition der Partialprodukte würde bereits den 32Bit Zahlenbereich verlassen.
Daher muss das Carry Bit des Addierers mit in das Ergebnisregister geschoben werden!

Die Ressourcen sparende Schaltungsvariante des vorzeichenlosen Multiplikators sieht dementsprechend folgendermaßen aus:

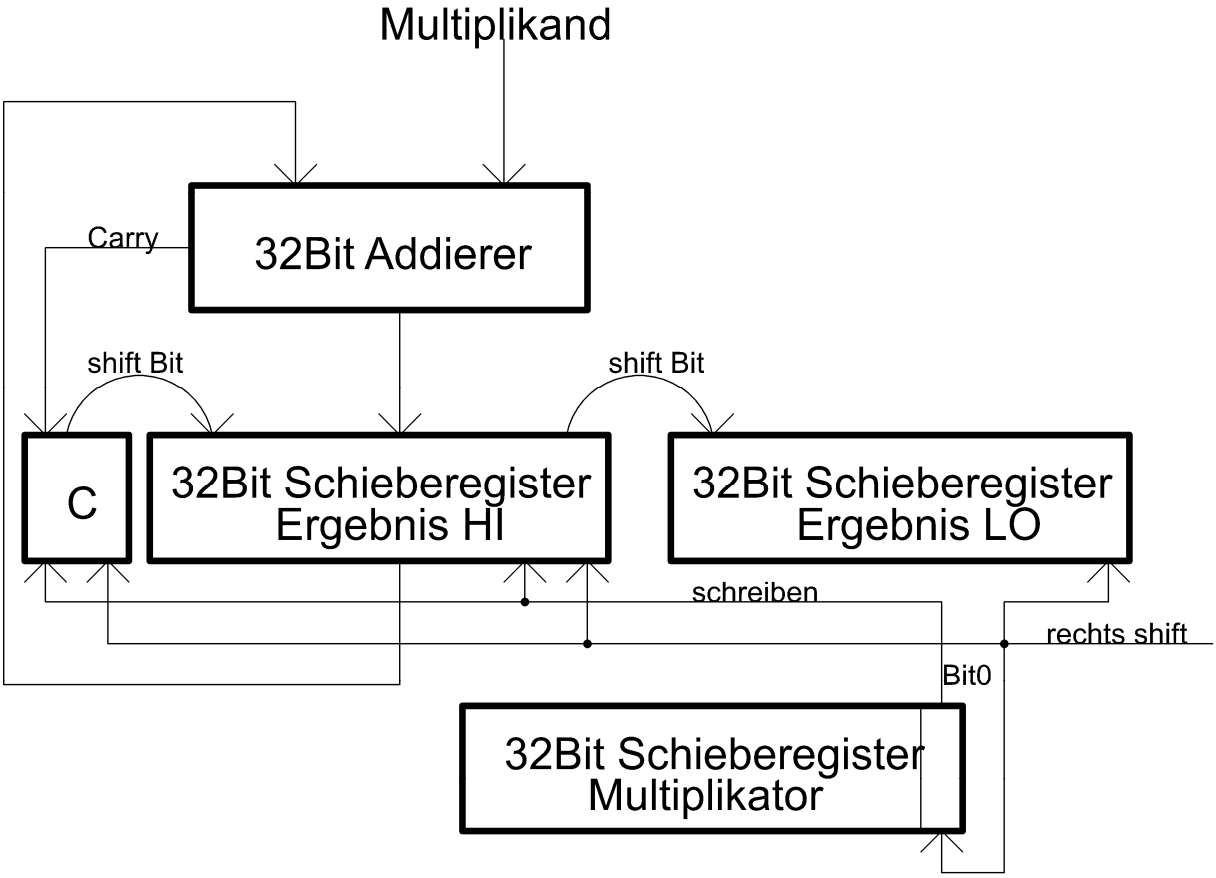


Abbildung 6

Es lässt sich aber noch ein Schieberegister sparen, denn es ist unerheblich welchen Initialwert Das Register LO besitzt, denn dieses wird nach den 32 Takten für die Berechnung komplett überschrieben. Nur das Register HI muss jetzt noch mit dem Wert 0 initialisiert werden.
Das Register LO und das Register des Multiplikators werden beide nach rechts geschoben. Daher bietet es sich an, dass der Multiplikand Initial in das Register LO geschrieben wird. Wenn ein weiteres Ergebnisbit feststeht, denn dann wird auch ein Bit des Multiplikators aus dem LO Register geschoben.

Nach 16 Takten sieht der Inhalt des LO Registers folgendermaßen aus:

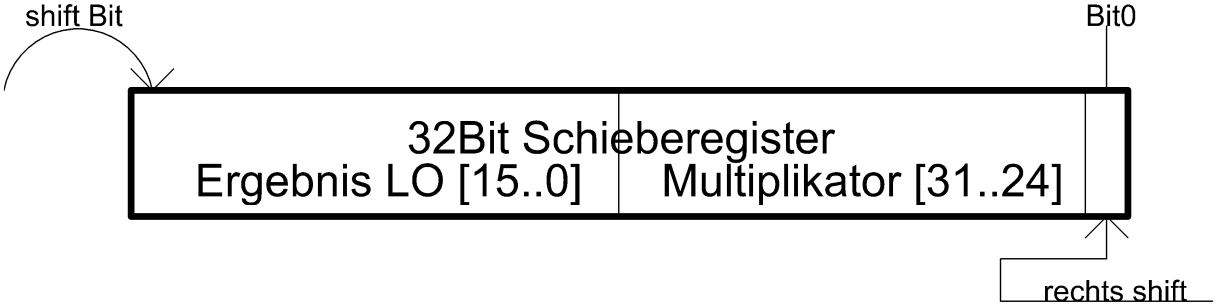


Abbildung 7

Die Grenze zwischen dem Ergebnis LO und dem Multiplikator wandert pro Takt um ein weiteres Bit nach rechts bis nur noch das Ergebnis LO im Register LO steht.

Somit ergibt sich die finale Schaltungsvariante des vorzeichenlosen Multiplizierers:

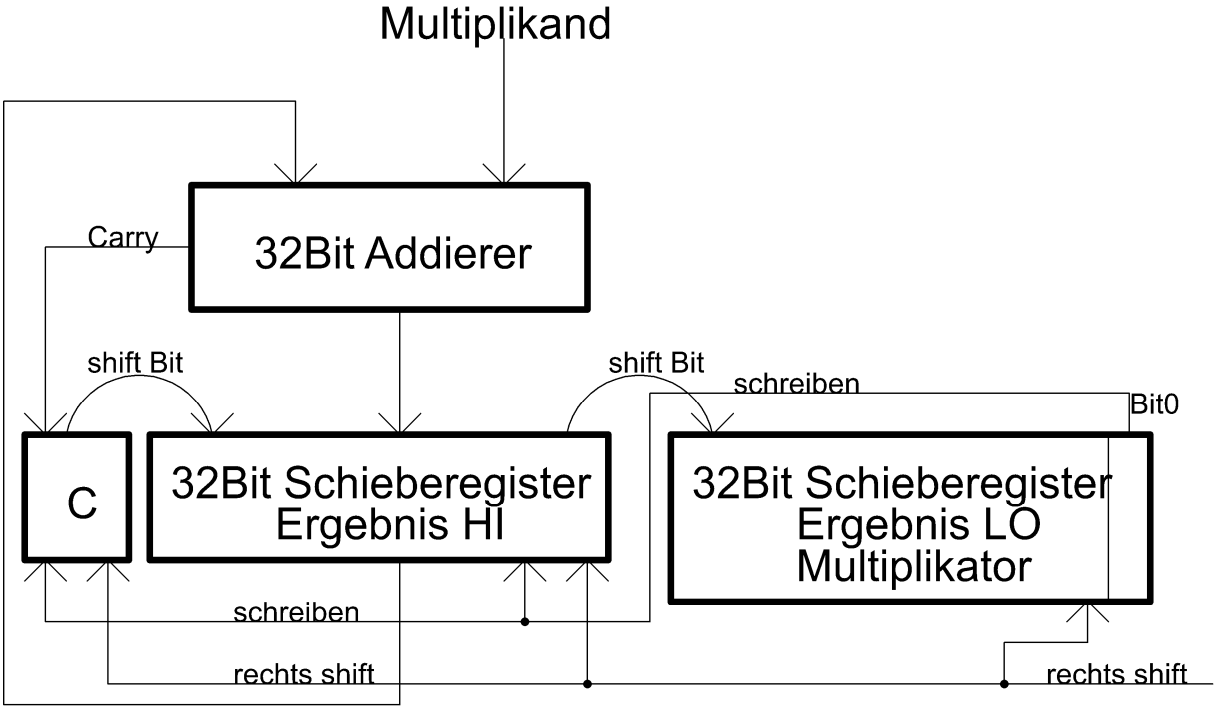


Abbildung 8

Für ein genaues Verständnis des Ablaufs, folgend das passende Flussdiagramm:

In diesem Kapitel wird nur die Rechenvorschrift betrachtet. Das Schreiben der Operanden und Auslesen der Ergebnisse benötigt zusätzliche Hardware und Datenpfade. Dies wird in Kapitel 3 betrachtet.

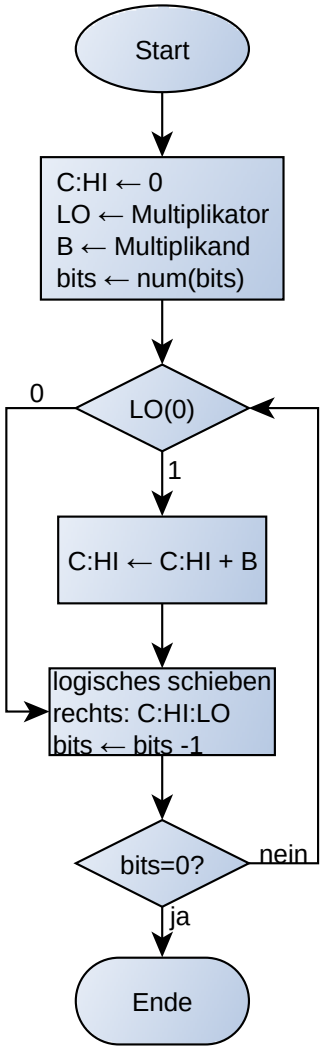


Abbildung 9

1.2 Multiplikation mit Vorzeichen

Mit der in 1.1 Multiplikation ohne Vorzeichen eingeführten Rechenvorschrift lassen sich keine vorzeichenbehafteten Zahlen Multiplizieren. Denn es wird keine Rücksicht auf die Vorzeichenbits des 2er Komplements genommen.
Zur Verdeutlichung ein Beispiel:

```
0 1 0 1 x 1 0 0 1   5*-7= -35
          1 0 0 1
        0 0 0 0
      1 0 0 1
    0 0 0 0
-------------------
  0 0 1 0 1 1 0 1       45
  1 1 0 1 1 1 0 1       -35
```

Abbildung 10

Anstatt das richtige Ergebnis von -35 zu liefern, ist das Ergebnis 45.

Nun könnte der Multiplikand einfach Vorzeichenerweitert werden, aber dann wird wieder ein 64Bit Addierer mit einem 64Bit Multiplikandenregister benötigt.
Das Beispiel aus 1.1 Multiplikation ohne Vorzeichen mit Vorzeichenerweiterung:

```
0 1 0 1 x 1 0 0 1
-----------------
  1 1 1 1 1 0 0 1
  0 0 0 0 0 0 0 0
  1 1 1 0 0 1
  0 0 0 0 0
-----------------
  1 1 0 1 1 1 0 1
```

Abbildung 11

Hier lässt sich der Booth[1] Algorithmus anwenden, daher ein kleiner Auszug aus der Wikipedia:
Dieser Algorithmus nutzt den Sachverhalt, dass jede Zahl als eine Differenz zweier anderer Zahlen darstellbar ist:
Sei $b = c - d$
Dann lässt sich jede beliebige Multiplikation von b mit einem Multiplikator a folgendermaßen umformen:
$a * b = a * (c - d) = a * c - a * d$
(Wikipedia Ende)

Das Zahlenpaar bestehend aus c und d muss von der Schaltung gefunden werden und kann mehrfach angewendet werden. Dazu wird das kürzlich raus geschobene Bit des Multiplikators aus dem Register LO, ab jetzt a_{-1} genannt, zusammen mit dem jetzigen LSB (a_0) des Multiplikators herangezogen. Diese beiden Bits geben die Bildung des Partialprodukts vor. Bei gleichbleibenden Bitfolgen b00 und b11 ist das Partialprodukt 0. Bei Übergängen, also sich ändernden Bitfolgen, wird die Negation im 2er Komplement des Multiplikanden als Partialprodukt genutzt oder der Multiplikand selbst. Der Addierer aus 1.1 Multiplikation ohne Vorzeichen wird jetzt zu einer ALU, die addieren und subtrahieren kann. Je nach Bitmuster

[1] https://de.wikipedia.org/wiki/Booth-Algorithmus

der Booth Bits. Dies wird Booth Encoding genannt, um genauer zu sein mit dem Radix 2. Höhere Radix sehen auch verdoppelte Partialprodukte vor durch eine integrierte Multiplikation mit 2, welche sich durch eine einfaches nach links schieben realisieren lassen.

Booth		Partialprodukt
a_0	a_{-1}	
0	0	PP = 0
0	1	PP * 1
1	0	PP * (-1)
1	1	PP = 0

Abbildung 12

Da sich das Booth Encoding aus den raus geschobenen Bits des Multiplikators ergibt, lassen sich diese bereits vorher tabellarisch darstellen:

a_3	a_2	a_1	a_0	a_{-1}	Booth	Partial
0	1	0	1	0	10	PP * -1
0	1	0	1		01	PP * 1
0	1	0			10	PP * -1
0	1				01	PP * 1

Abbildung 13

Ein Rechenbeispiel mit dem Booth Algorithmus:

Booth	0	1	0	1	x	1	0	0	1	Booth Bits	5*-7=-35
PP * -1		0	0	0	0	0	1	1	1	10	
PP * 1		1	1	1	1	0	0	1		01	
PP * -1		0	0	0	1	1	1			10	
PP * 1		1	1	0	0	1				01	
	1	1	0	1	1	1	0	1			-35

Abbildung 14

Wie in Abbildung 14 zu sehen ist werden die negativen Partialprodukte immer noch Vorzeichenerweitert (gelbe Markierung). Allerdings ist durch das Booth Encoding bekannt wann Vorzeichenerweitert werden muss und wann nicht. Dadurch lässt sich die Vorzeichenerweiterung während des Schiebens nach rechts realisieren indem arithmetisch oder logisch geschoben wird. Es wird immer arithmetisch geschoben außer wenn das Booth Encoding auf b10 steht für „PP * (-1)“ und der Multiplikand negativ ist, denn sonst lässt sich die Kette der Vorzeichenerweiterung nicht unterbrechen und das Ergebnis kann nie wieder positiv werden. Diese Fälle sind in Abbildung 14 beim ersten und dritten Partialprodukt zu sehen, dort sind die Vorzeichenbits 0.

Diese Erkenntnisse werden nun wieder in einem Blockschaltplan festgehalten:

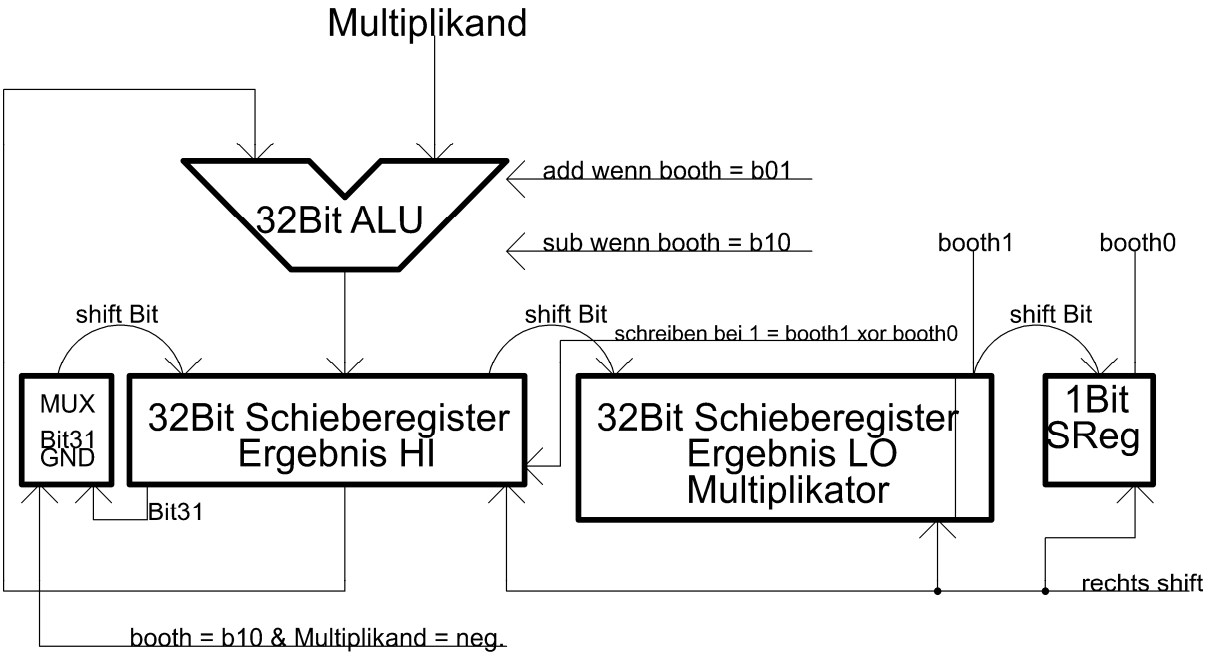


Abbildung 15

Das Carry der ALU wird nicht mehr benötigt, die Ergebnisse passen durch das 2er Komplement immer in das 32Bit Schieberegister HI. Um genauer zu sein wird auf eine negative Partialsumme immer eine positive Zahl addiert und viceversa. Das rein geschobene Bit in das HI Register wird durch einen Multiplexer bestimmt der bei dem Sonderfall das arithmetische schieben für ein logisches schieben unterbricht.
Nach dem Register LO folgt noch ein 1Bit Schieberegister um das letzte raus geschobene LSB des Multiplikators zu speichern für die Bildung des Booth Encodings.

Um die erarbeitete Rechenvorschrift mit den (Sonder-) Fällen besser zu verstehen abschließend noch ein Flussdiagramm:

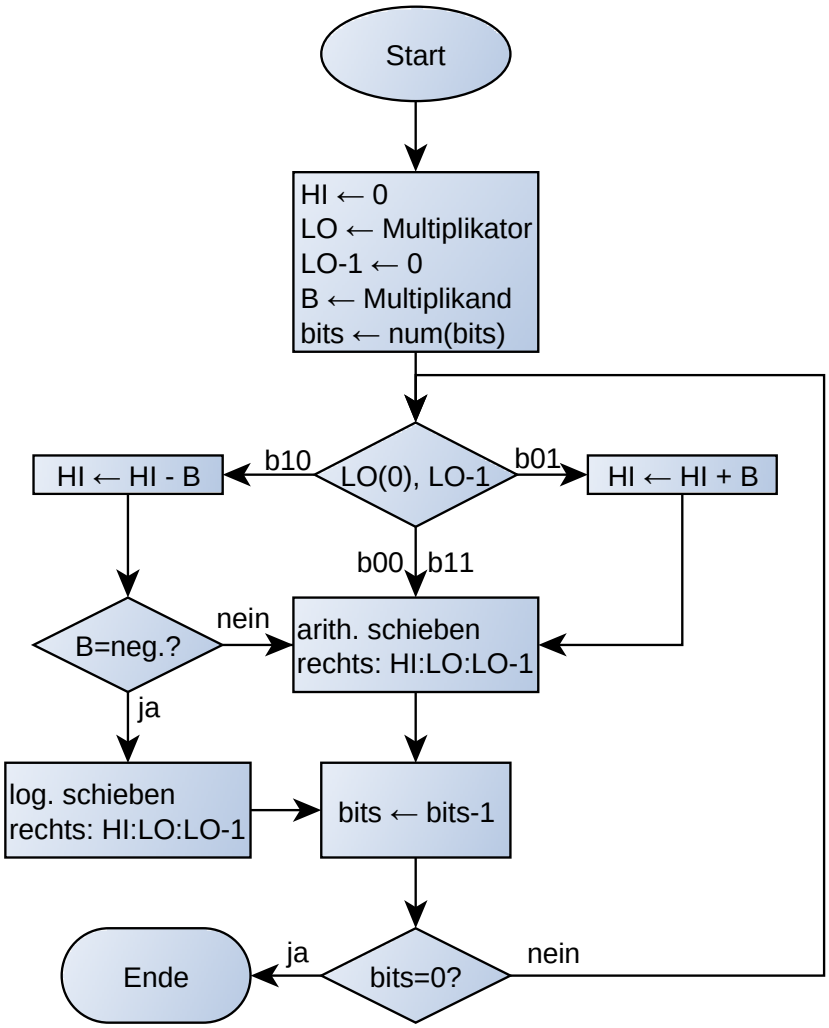


Abbildung 16

In diesem Kapitel wird nur die Rechenvorschrift betrachtet. Das Schreiben der Operanden und Auslesen der Ergebnisse benötigt zusätzliche Hardware und Datenpfade. Dies wird in Kapitel 3 betrachtet.

1.3 Division ohne Vorzeichen

Die Division ist die Umkehroperation zur Multiplikation. Dementsprechend wird bei dieser Operation nicht wiederholt addiert, sondern subtrahiert. Dabei wird erfasst wie oft der Operand B in den Operand A hineinpasst. A wird hierbei Dividend genannt und B ist der Divisor. Das Ergebnis ist kein Produkt, sondern ein Quotient. Bei A und B handelt es sich um Integer, dementsprechend kann es vorkommen, dass der Divisor nicht n mal in den Dividend passt, es bleibt ein Rest zurück.
Es gilt: $A : B = Q, R$

Wie bereits in 1.1 Multiplikation ohne Vorzeichen hergeleitet, würde eine Division von $4294967295 : 1 = 4294967295, \mathrm{R} = 0$ ungefähr 1073 Sekunden auf dem Spaceage 2 benötigen.
Dies ist auch bei einer Division nicht hinnehmbar, daher wird auch bei der Division von der Grundschulmathematik abgeschaut:

```
  2 1 0 : 8 = 0 2 6
- 0
  2 1
- 1 6
    5 0
  - 4 8
      2    Rest
```

Abbildung 17

Bei der Division wird pro Dezimalstelle des Dividenden ermittelt wie oft der Divisor in diese hineinpasst. Bei der Hunderterstelle mit dem Wert 2 passt die 8 genau 0 Mal hinein, daher ist der Wert der Hunderterstelle des Quotienten auch 0. Nach der Ermittlung der Stelle muss der Rest ermittelt werden zur Bestimmung der nächst kleineren Stelle. In diesem Fall ist 0*8 = 0, demnach wird von der Hunderterstelle auch 0 abgezogen und die 2 wird zur Berechnung der kleineren Stelle herangezogen.
Die nächst kleinere Stelle des Dividenden, hier die Zehnerstelle, wird nun nach unten gezogen um mit dem vorherigen Wert eine 21 zu bilden. In diese 21 passt der Divisor 2 Mal hinein, daher ist die Zehnerstelle des Quotienten 2. 2*8 ergibt 16 und dieser Wert wird von der Zwischensumme abgezogen um den Rest für die Einerstelle bereitzustellen.
Schlussendlich wird durch die weitere Anwendung der Rechenvorschrift als Quotient 26 berechnet und der Rest beträgt 2.

Wie lässt sich diese Vorschrift nun ins Binäre übertragen?
Erst einmal ist zu erkennen, dass sich von der höchstwertigen Stelle zur niederwertigsten Stelle durchgearbeitet wird. Bei der Multiplikation ist dies genau andersrum der Fall. Daher wird bei der Division nach links geschoben und nicht nach rechts.
Im Binären kann eine Stelle zudem nur die Werte 0 oder 1 annehmen und nicht 0 bis 9. Dadurch entfällt das Rückrechnen, wie in der Dezimalrechnung, um den Rest zur weiteren Berechnung zu finden. Die Berechnung fällt deshalb zurück auf die Ereignisse „Subtraktion ist möglich“ und „Subtraktion ist nicht möglich“, bezogen darauf, dass das Ergebnis sonst negativ wird. Sobald eine Subtraktion möglich ist wird eine 1 in der niederwertigsten Stelle eine Schieberegisters gespeichert, sonst 0. Dieses Schieberegister wird auch nach links geschoben.

Dazu ein Beispiel welches wieder 210 : 8 rechnet:

```
Q              1 1 0 1 0 0 1 0 : 1 0 0 0
0   - 1 0 0 0    Subtraktion nicht möglich
              1 1
0     - 1 0 0 0  Subtraktion nicht möglich
              1 1 0
0       - 1 0 0 0  Subtraktion nicht möglich
              1 1 0 1
1         - 1 0 0 0
              1 0 1 0
1           - 1 0 0 0
                0 1 0 0
0             - 1 0 0 0   Subtraktion nicht möglich
                1 0 0 1
1               - 1 0 0 0
                  0 0 1 0
0                 - 1 0 0 0   Subtraktion nicht möglich
                      1 0     Rest
```

Abbildung 18

Wie in der Grafik zu erkennen ist wird der Dividend Bit für Bit nach links geschoben. Anschließend wird von diesem geschobenen Dividenden der Divisor subtrahiert. Ist das Ergebnis positiv, so wird Dieses gespeichert und es wird eine 1 im Ergebnisschieberegister gespeichert. Ist das Ergebnis negativ, findet beides nicht statt.

Aus dieser Rechenvorschrift lässt sich auch wieder ein Blockschaltplan ableiten:

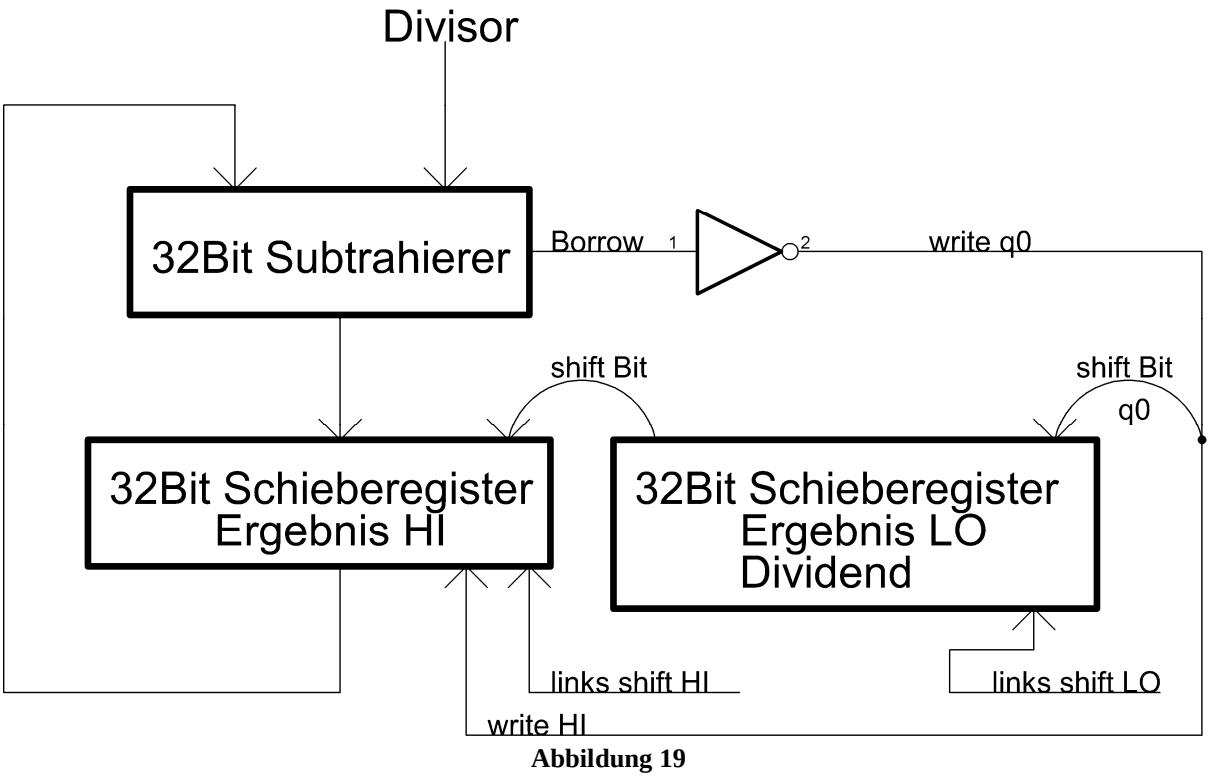


Abbildung 19

Um denselben Spartrick mit Operand A anwenden zu können wie bei der Multiplikation muss der Schiebebefehl zweigeteilt werden. Denn es muss nach Rechenvorschrift zuerst nach links geschoben werden um anschließend zu prüfen ob die Subtraktion aufgeht. Zu diesem Zeitpunkt ist q0 allerdings noch nicht bekannt.
Deshalb wird zuerst das Schieberegister HI nach links geschoben und wenn 1 Takt später q0 bekannt ist wird auch das Schieberegister LO nach links geschoben.

Ob die Subtraktion erfolgreich war oder nicht wird durch den Borrow Ausgang des Subtrahierers bestimmt, dies kann nicht durch das MSB des Ergebnisses geschehen. Bei z.B. 4Bit Wortbreite kann das Ergebnis von 1-15 nicht mehr erfasst werden. So ist 1-15 ein b0010, müsste aber ein b10010 sein. Dies geschieht, da der Subtrahierer intern 1+1 rechnet. Denn ein Subtrahierer ist intern ein Addierer, der den Subtrahenden internen in ein 2er Komplement umrechnet. Aber der Borrow Ausgang ist gesetzt bei dem Ergebnis b0010.
Daher muss das Borrow abgefragt werden zur Bestimmung des Vorzeichens des Ergebnisses. Ein 15-15 wird intern zu 15+1 -> Borrow auf LOW -> schreiben gewollt und q0 = 1.

Auch hier wieder das abschließende Flussdiagramm zur schrittweisen Nachverfolgung der Schritte:

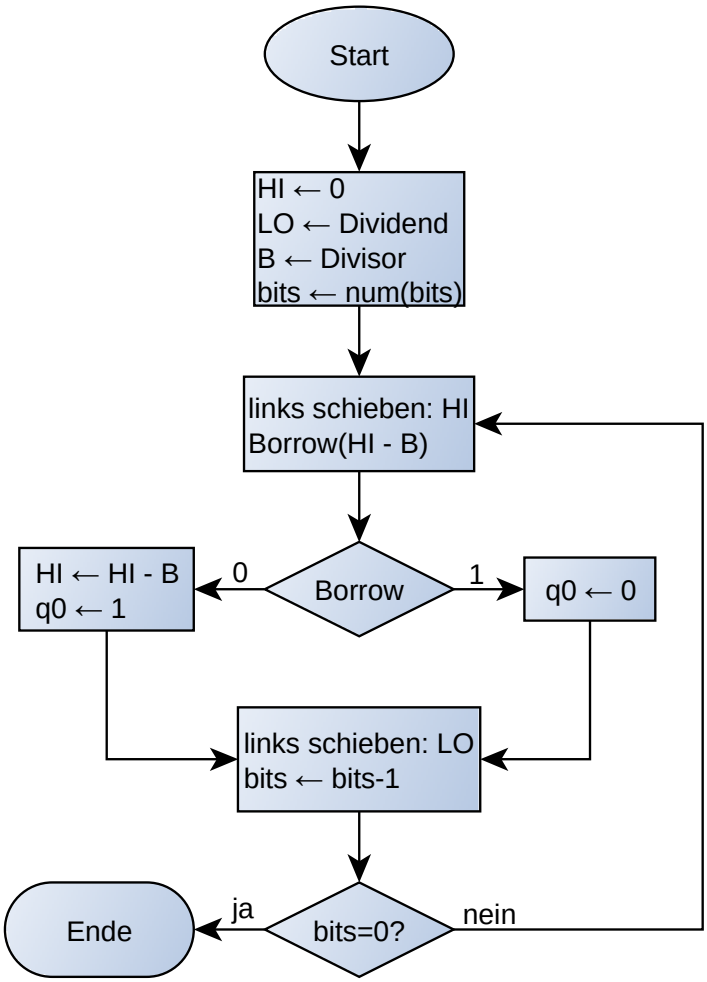


Abbildung 20

In diesem Kapitel wird nur die Rechenvorschrift betrachtet. Das Schreiben der Operanden und Auslesen der Ergebnisse benötigt zusätzliche Hardware und Datenpfade. Dies wird in Kapitel 3 betrachtet.

1.4 Division mit Vorzeichen

Bei der Division ohne Vorzeichen wird Bit für Bit getestet ob der Divisor in den bisher rein geschobenen Teil des Dividenden passt. Anders gesagt wird die Differenz zu null auf die passende Größe getestet. Ist der Dividend nun negativ, dann kann nicht subtrahiert werden sondern es muss addiert werden um diese Differenz zu überprüfen. Ansonsten wird die Zahl immer negativer anstatt ein passendes Ergebnis zu liefern.
Ist nun der Divisor und der Dividend negativ, so muss wieder subtrahiert werden, Das ergibt sich daraus, dass das negative Vorzeichen des Divisors damit in der Rechnung umgedreht wird. Weiterhin ist zu beachten, dass der Inhalt vom HI Register nun die Vorzeichenerweiterung des Dividenden im LO Register ist.

Der weitere Teil der Rechenvorschrift bleibt gleich. Ändert sich das Vorzeichen nicht -> HI schreiben und q0 setzen. Ändert sich das Vorzeichen -> HI nicht schreiben und q0 = 0.
Durch das verwendete 2er Komplement muss hier nicht mehr auf das Carry oder Borrow der ALU geprüft werden. Es reicht die MSB des Ergebnisses der ALU und das MSB des HI Schieberegisters zu vergleichen. Denn es wird sowieso eine logische Verknüpfung gebraucht wegen dem folgenden Spezialfall.
Allerdings wird der Quotient bei dieser Rechenvorschrift als Absolutwert berechnet, deshalb muss zum Ende der Berechnung eine Nachbetrachtung erfolgen. Diese prüft ob sich die Vorzeichen von Dividend und Divisor unterscheiden und negiert dann entsprechend den Quotienten.

Für den Sonderfall verfolgen wir die Rechnung $-8/2 = -4, R0$. Die Zahlen in den Klammern sind die Iterationsschritte mit den 2 Subschritten (erst rechnen, dann speichern). Das HI|LO Schieberegister ist zusammengefasst und F ist der Ausgang der ALU.

```
(S)   1111|1000
----------------------
(0-0) 1111|0000 F:0001
(0-1) 1111|0000 F:0001
----------------------
(1-0) 1110|0000 F:0000
(1-1) 1110|0000 F:0000
----------------------
(2-0) 1100|0000 F:1110
(2-1) 1110|0001 F:1110
----------------------
(3-0) 1100|0010 F:1110
(3-1) 1110|0011 F:1110
----------------------
(E)   1110|1101
```

Abbildung 21

Das Ergebnis dieser Rechnung ist -3, R -2, dies ist offensichtlich falsch. Doch was ist passiert? Im Schritt (0-1) ist das Ergebnis der ALU 0. Dies bedeutet, dass die Addition von -2+2 offensichtlich erkennt, dass die 2 des Divisors in den momentanen Tel des Dividenden passt. Aber dies hat ein Vorzeichenwechsel zur Folge, wodurch q0 nicht auf 1 gesetzt wird und das Ergebnis nicht nach HI gespeichert wird. Hier muss der Sonderfall greifen und beide Operationen doch durchführen.

Im folgender Abbildung die korrekte Berechnung mit dem Sonderfall:

```
(S)   1111|1000
----------------------
(0-0) 1111|0000 F:0001
(0-1) 1111|0000 F:0001
----------------------
(1-0) 1110|0000 F:0000
      Sonderfall!
(1-1) 0000|0001 F:0000
----------------------
(2-0) 0000|0010 F:1110
(2-1) 0000|0010 F:1110
----------------------
(3-0) 0000|0100 F:1110
(3-1) 0000|0100 F:1110
----------------------
(E)   0000|1100
```

Abbildung 22

Durch das Schreiben des Ergebnisses nach HI ist dieses nun für immer null und somit auch der Rest. Das Bit des Quotienten wird auch an die passende Stelle geschoben. Der Sonderfall darf jedoch nur dann auftreten, wenn nur noch der letzte Rest abgezogen werden muss der genau passt. Dies darf nicht also nicht passieren, wenn der restliche Dividend noch Bits besitzt. Ansonsten verfälscht dies z.B. die Berechnung $-2/1 = -2, R0$:

```
(S)   1111|1110
----------------------
(0-0) 1111|1100 F:0000
      Sonderfall!
(0-1) 0000|1101 F:0000
----------------------
(1-0) 0001|1010 F:0000
      Sonderfall!
(1-1) 0000|1011 F:0000
----------------------
(2-0) 0001|0110 F:0000
      Sonderfall!
(2-1) 0000|0111 F:0000
----------------------
(3-0) 0000|1110 F:1111
(3-1) 0000|1110 F:1111
----------------------
(E)   0000|0010
```

Abbildung 23

Mit der Konkretisierung des Sonderfalls wird auch diese Berechnung korrekt ausgeführt:

```
(S)   1111|1110
----------------------
(0-0) 1111|1100 F:0000
(0-1) 1111|1100 F:0000
----------------------
(1-0) 1111|1000 F:0000
(1-1) 1111|1000 F:0000
----------------------
(2-0) 1111|0000 F:0000
      Sonderfall!
(2-1) 0000|0001 F:0000
----------------------
(3-0) 0000|0010 F:1111
(3-1) 0000|0010 F:1111
----------------------
(E)   0000|1110
```

Abbildung 24

Durch die Einführung des „restlichen Dividenden“ im Sonderfall muss leider ein weiteres Schieberegister eingeführt werden. Obwohl bei der Herleitung der Multiplikation eines gespart werden sollte. Denn das LO Schieberegister kann nicht herangezogen werden zur Bestimmung ob der restliche Dividend bereits 0 ist, weil in das LO Schieberegister auch die Quotientenbits reingeschoben werden. Deshalb benötigt LO ein Schattenregister, welches mit Nullen aufgeschoben wird und dessen Inhalt auf gesetzte Bits überprüft wird. Gleichzeitig muss auch das ALU Ergebnis auf null überprüft werden. Da der Sonderfall bereits bekannt sein muss wenn die Schreibauswahl ansteht, wird das LO Schattenregister mit dem HI Schieberegister zusammen geschoben.

Dazu das passende Blockschaltbild:

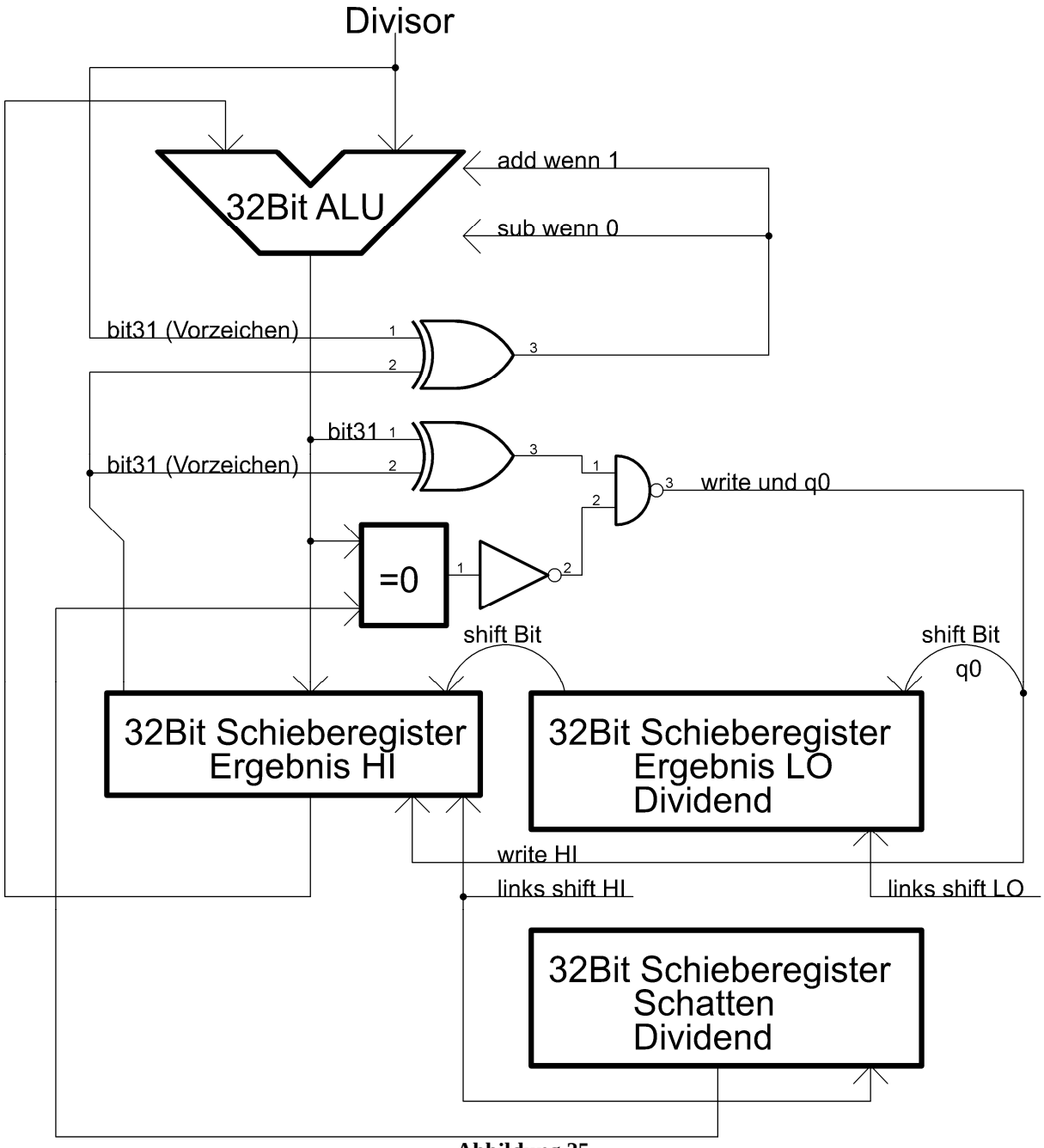


Abbildung 25

Um die erarbeitete Rechenvorschrift mit den (Sonder-) Fällen besser zu verstehen abschließend noch ein Flussdiagramm:

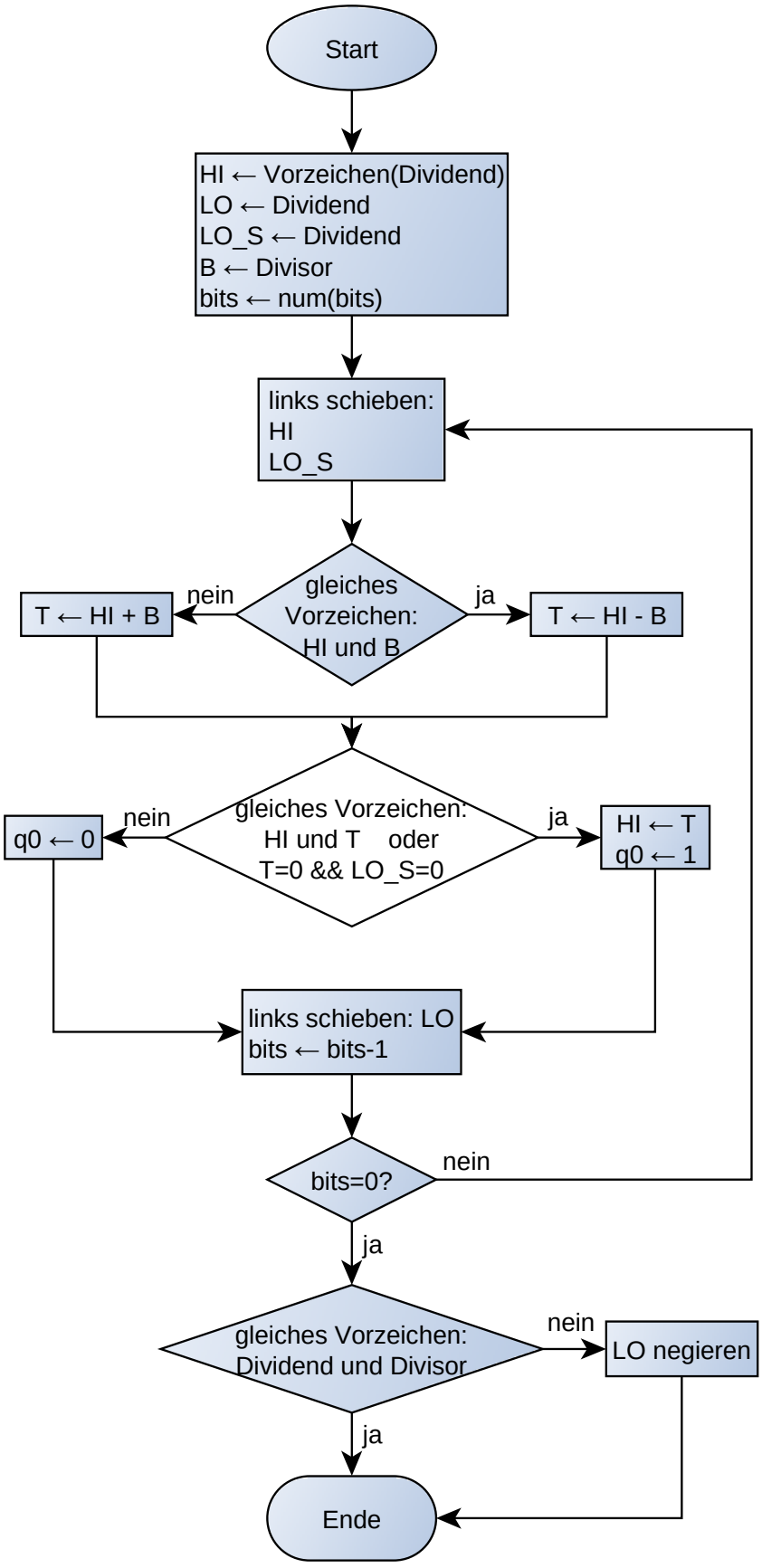


Abbildung 26

2. Theorieüberprüfung anhand eines C Programms

Die Theorie der Rechenvorschriften steht, jedoch sollten diese überprüft werden. Dazu empfiehlt sich ein kleines C Programm. Mit dem durch Bitmanipulationsoperationen die Schieberegister und die ALU simuliert werden können.

```
Not enough arguments, usage:
punkttest <func> <opA> <opB>

func:
sm (signed multiplication)
um (unsigned multiplication)
sd (signed division)
ud (unsigned division)

use t behind a func to test ALL
possible op combinations
```

Abbildung 27

Dabei kann das Programm in 4/8/32 Bit simulieren. Die 4Bit Simulation dient dazu die Rechenvorschrift zu testen und die 8Bit um noch mal sicher zu gehen. 256 oder 65536 Kombinationen sind noch per Bruteforce testbar. Die 32 Bit Simulation dient dem Nachvollziehen eventueller späterer Hardwarefehler. Damit kann überprüft werden ab welchem Rechenschritt die Hardware und die Simulation auseinander driften. Für diesen Anwendungsfall kann eine Binärausgabe der Rechenschritte aktiviert werden.

Mithilfe dieses Programms wurden die Sonderfälle bei den vorzeichenbehafteten Rechenvorschriften gefunden.

Anschließend die 4 Prozeduren zur Simulation. Anhand dieser kann der ein oder Andere die Rechenvorschriften vielleicht besser nachvollziehen.

```
uint64_t perform_um(uint32_t uopA, uint32_t uopB){

	// init right half of shift with multiplier
	uint8_t shiftregister = uopA;

	uint16_t alu_erg = 0;
	uint8_t carry = 0;
	unsigned i;

	for (i = 0; i < 4; i++){

		// if bit0 in shiftregister is set
		// -> write addition to left half of shift register
		if(shiftregister & 0x1){
			alu_erg = (shiftregister >> 4) + uopB;
			carry = (alu_erg >> 4) & 0x1;
			alu_erg &= 0xFF;
			shiftregister &= 0xF;
			shiftregister |= (alu_erg << 4);
		}

		// shift always right
		shiftregister >>= 1;
		shiftregister &= 0x7F;

		// carry needs to be shifted in!
		shiftregister |= (carry << 7);
		carry = 0;
	}

	return shiftregister;
}
```

Abbildung 28 Multiplikation ohne Vorzeichen

```
void perform_ud(uint32_t* rem, uint32_t* quot, uint32_t uopA, uint32_t uopB){

	// init right half of shift with quotient
	uint8_t shiftregister = uopA;
	shiftregister &= 0x0F;

	uint16_t alu_erg = 0;
	unsigned i;

	for (i = 0; i < 4; i++){

		// left shift
		shiftregister <<= 1;
		shiftregister &= 0xFE;

		alu_erg = (shiftregister >> 4) - (uint8_t)uopB;
		// if alu_erg is negative -> dont save and shiftregister lsb = 0
		// if alu_erg is positive -> save and shiftregister lsb = 1
		if(0 == ((alu_erg & 0x100) == 0x100)) { // borrow!
			alu_erg &= 0xFF;
			shiftregister &= 0xF;
			shiftregister |= (alu_erg << 4) | 1;
		}
	}

	*rem = (shiftregister >> 4);
	*quot = shiftregister & 0x0F;
}
```

Abbildung 29 Division ohne Vorzeichen

```
int64_t perform_sm(int32_t sopA, int32_t sopB){

	// init right half of shift with multiplier
	uint8_t shiftregister = sopA;
	shiftregister &= 0xF;

	uint16_t alu_erg = 0;
	uint8_t qmin = 0;
	uint8_t booth = 0;
	uint8_t msb = 0;
	unsigned i;

	for (i = 0; i < 4; i++){

		/*
			q0_qmin
			00/11 -> only shift
			01 -> add/shift
			10 -> sub/shift
		*/
		booth = (shiftregister & 0x1) << 1;
		booth |= qmin;

		if (booth == 0b01){
			alu_erg = (shiftregister >> 4) + (int8_t)sopB;
			alu_erg &= 0xF;
			shiftregister &= 0xF;
			shiftregister |= (alu_erg << 4);
		}
		if (booth == 0b10){
			alu_erg = (shiftregister >> 4) - (int8_t)sopB;
			alu_erg &= 0xF;
			shiftregister &= 0xF;
			shiftregister |= (alu_erg << 4);
		}

		// log right shift
		qmin = shiftregister & 0x1;
		msb = shiftregister & 0x80;
		shiftregister >>= 1;
		shiftregister &= 0x7F;

		// no arith at sub if multiplicand is negative
		if (0 == ((booth == 0b10) && ((sopB & 0x8) == 0x8))){
			shiftregister |= msb;
		}

	}

	if (shiftregister & 0x80){
		return 0xFFFFFFFFFFFFFF00 | (int64_t)shiftregister;
	}

	return (int64_t)shiftregister;
}
```

Abbildung 30 Multiplikation mit Vorzeichen

```
void perform_sd(int32_t* rem, int32_t* quot, int32_t sopA, int32_t sopB){

	// init right half of shift with quotient
	int8_t shiftregister = sopA;
	shiftregister &= 0x0F;
	int8_t remaining = shiftregister;
	remaining &= 0xF;

	// sign extention of dividend
	if(sopA & 0x8){
		shiftregister |= 0xF0;
	}

	uint16_t alu_erg = 0;
	unsigned i, sonderfall;

	for (i = 0; i < 4; i++){

		// left shift
		shiftregister <<= 1;
		shiftregister &= 0xFE;
		remaining <<= 1;
		remaining &= 0xE;

		// divisor and left shiftregister same sign -> subtract, else add
		if(((sopB & 0x8) >> 3) ^ ((shiftregister & 0x80) >> 7)){
			alu_erg = (shiftregister >> 4) + sopB;
		}else{
			alu_erg = (shiftregister >> 4) - sopB;
		}

		sonderfall = 0;
		if (((alu_erg & 0xF) == 0) && ((remaining & 0xF) == 0)){
			sonderfall = 1;
		}

		// if sign alu_erg and shiftregister is same
		// -> save and shiftregister lsb = 1
		// if sign alu_erg and shiftregister is diff -> dont save and
		// shiftregister lsb = 0
		// If the sign of A is the same before and after the operation or
		// (A=0 & remaining dividend=0)
		if(sonderfall ||
		(0 == (((alu_erg & 0x8) >> 3) ^ ((shiftregister & 0x80) >> 7)))){
			shiftregister &= 0xF;
			shiftregister |= (alu_erg << 4);
			shiftregister |= 1;
		}
	}

	// signs of divisor and dividend are different?
	// -> negate qotient
	if(((sopA & 0x8) >> 3) ^ ((sopB & 0x8) >> 3)){
		uint8_t tmp = shiftregister & 0xF;
		shiftregister &= 0xF0;
		tmp = (~tmp) + 1;
		shiftregister |= (tmp & 0xF);
	}

	// output sign extensions
	if (shiftregister & 0x80){
		*rem = 0xFFFFFFF0 | (shiftregister >> 4);
	}else{
		*rem = (shiftregister >> 4);
	}
	if (shiftregister & 0x8){
		*quot = 0xFFFFFFF0 | (shiftregister & 0xF);
	}else{
		*quot = shiftregister & 0x0F;
```

Abbildung 31 Division mit Vorzeichen

3. Herleitung der benötigten Gesamthardware

Bisher wurde die Theorie der Rechenvorschriften betrachtet, jedoch müssen die Operanden und Ergebnisse auch von und zur CPU gelangen. Bei der CPU Kern Entwicklung des Spaceage 2 wurden die Punktrechenarten noch in Software emuliert. Dazu mussten die passenden Registeroperanden in Software greifbar sein. Hierfür wurde eine Emulationskarte vor die ALU der CPU verbaut, welche die Werte der Register speicherte. Diese Karte besitzt die beiden Bussysteme „alu_in_A“ und „alu_in_B“. Währenddessen auf alu_in_A ein tristate Betrieb möglich ist, liegt auf dem Bus alu_in_B immer ein Datenwort an. Wie sich in den Rechenvorschriften herausstellte liegt Operand B sowieso immer fest an und daher stellt dies kein Problem dar. Somit bleibt alu_in_A als in/out Bus für Datenworte von und zur MulDiv Karte übrig.

Ab jetzt ist mit ALU nur noch die ALU der MulDiv Karte gemeint.
Der Bus alu_in_B wird somit an den ALU Eingang für den Operand B angeschlossen und alu_in_A an den ALU Eingang für Operand A. Gleichzeitig liegt aber auch der Ausgang des HI Schieberegisters damit an alu_in_A. Dies bedeutet, dass der Ausgang des HI Schieberegisters in den tristate geschalten werden können muss.

Weiterhin muss der Operand A, der zwangsweise über den Bus alu_in_A anliegt, in das LO Schieberegister geschrieben werden können. Um dies zu realisieren existieren zwei Möglichkeiten:
Der Eingang des LO Schieberegisters wird auch …

1) an den Bus alu_in_A angeschlossen
2) den Ergebnisbus der ALU angeschlossen

Es wurde sich für die Möglichkeit 2) entschieden, denn so ist eine eventuelle Bearbeitung des Operanden durch die ALU möglich und die Buslast von alu_in_A wird reduziert.

Die Ergebnisse der jeweiligen Berechnungen liegen zum Schluss in den Schieberegistern HI und LO. Daher müssen beide über den Bus alu_in_A auslesbar sein. Durch das Anschließen der Schleife HI Schieberegister -> ALU Operand A -> HI Schieberegister befindet sich der Ausgange dieses Registers bereits am Bus alu_in_A. Der Ausgang des LO Schieberegisters muss nun auch an diesen Bus angeschlossen werden wodurch auch dieses Schieberegister seinen Ausgang in den tristate schalten können muss,
Beide Schieberegister besitzen somit jeweils denselben Bus als Ein- und Ausgang.

Zusätzlich zu den 2 Bussystemen liegen noch folgende Steuersignale an der Schnittstelle zum Mainboard an:

mul_div_S[2..0]
Das Steuersignal zur Auswahl welches Emulationsregister geschrieben werden soll, dieses Signal kann nun zur Auswahl der Rechenart dienen. Dabei wurde folgende Belegung festgelegt:
mul_div_S2: #Speicheroperation / Berechnung
mul_div_S1: #Division / Multiplikation
mul_div_S0: #Vorzeichenlos / Vorzeichenbehaftet

#store_reg_to_mul_div
Dieses Signal löst den eigentlichen Schreibvorgang in Verbindung mit mul_div_S aus und sollte seinen „Ausführcharakter“ beibehalten. Somit dient das Signal zum Starten einer Berechnung.

#hi_to_alu_A, #lo_to_alu_A
Diese beiden Signale sind Teil des Eingangsmultiplexers der CPU ALU und dienen zum Auslesen der HI und LO Register. Ist eines der Signale LOW, so muss das entsprechende Register auf den Bus alu_in_A gelegt werden, was durch den bereits geklärten groben Schaltungsaufbau möglich ist.

Zusätzlich wird ein Signal benötigt um den CPU Kern zu stoppen, denn eine Berechnung dauert länger als die maximale Mikrocodeschrittanzahl von deren Steuerwerk. Dazu wird das Signal **halt/#run_src** vom Steuerwerk der CPU zur MulDiv Karte durchgeschleift, damit das Signal dort manipuliert werden kann.

Dementsprechend sieht der grobe Blockschaltplan folgendermaßen aus:

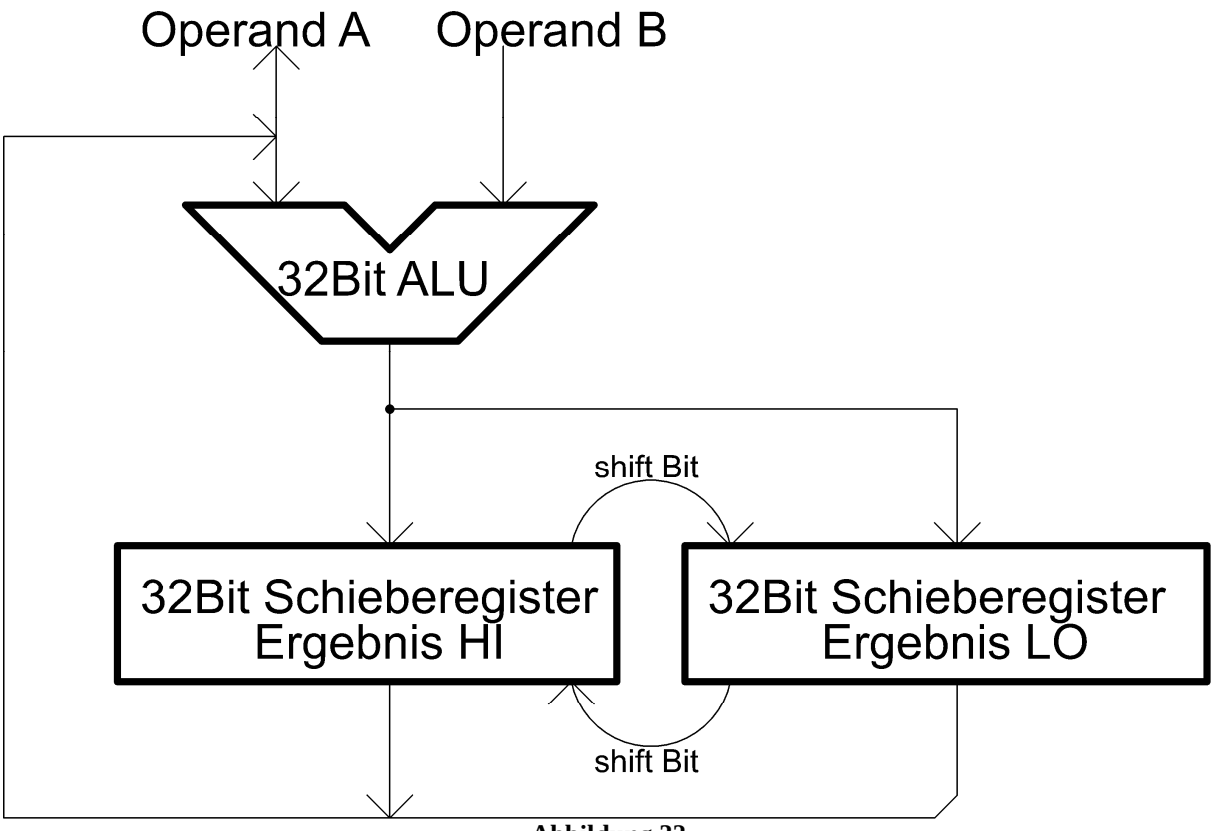


Abbildung 32

Nicht enthalten im Blockschaltplan sind die Steuersignale der Schieberegister und der ALU. Diese werden im nächsten Kapitel hergeleitet.

3.1 Hardwareblöcke

Die Hardwareblöcke und dessen Anforderungen müssen nun definiert werden. Für sämtliche Berechnungen werden die ALU sowie die beiden Schieberegister HI und LO benötigt. Sowie eine Schaltung zur Entscheidung wann das HI Register geschrieben werden muss. Ein Steuerwerk wird natürlich auch als Gesamtsteuerung der Schaltung benötigt.

Für die Multiplikationen wird noch eine Schiebesteuerung benötigt, diese kontrolliert welches Bit beim rechts schieben in das HI Register gelangt.

Folgende Hardwareblöcke und dessen Anforderungen müssen demnach definiert werden:
- Schieberegister HI und LO
- Schattenregister LO und Nullvergleich
- ALU
- Booth Controller (Bereitstellung des Booth Encodings)
- Schreibsteuerung (wann wird HI geschrieben und q0 gesetzt)
- Schiebesteuerung (welches Bit wird reingeschoben beim rechts schieben)
- Steuerwerk

3.1.1 Schieberegister HI und LO

Das Schieberegister benötigt 4 Betriebszustände die synchron zu einem Taktsignal sind:
0) halten
1) rechts schieben
2) links schieben
3) paralleles laden

Als Ein- und Ausgänge werden benötigt:
- Eingang für paralleles laden
- Eingang für Ausgang auf tristate schalten
- Eingang für Taktsignal
- Eingang für den Betriebszustand
- Eingang für das rein geschobene Bit (links und rechts)
- Ausgang für das raus geschobene Bit (links und rechts)
- Ausgang zum parallelen auslesen (tristate)

3.1.2 Schattenregister LO und Nullvergleich

Das Schattenregister wird nur bei der Division mit Vorzeichen benötigt, daher muss es nur nach links geschoben sowie parallel geladen werden können. Beim Schieben wird das Register auch immer mit 0 als LSB aufgefüllt. Das Schattenregister wird auch nicht auf einen tristate Bus geschaltet, sondern wird direkt an den Vergleicher angeschlossen. Daher entfällt auch diese Funktionalität. Am Ende wird noch ein 64Bit Vergleicher benötigt, der das Schattenregister und den ALU Ausgang auf gleich null vergleicht.

Als Ein- und Ausgänge werden benötigt:
- Eingang für paralleles laden
- Eingang für Taktsignal
- Eingang für den Betriebszustand
- Eingang für ALU Ergebnis
- Ausgang für „alle Bits = 0“

3.1.3 ALU

Die ALU sitzt immer vor dem HI und LO Schieberegister, weshalb diese auch ein einfaches Durchreichen der Bits unterstützen muss um diese zu beschreiben. Für die Berechnungen muss ein Addieren und Subtrahieren möglich sein. Weiterhin müssen noch zusätzlich Operationen für die Vor- und Nachbetrachtungen der Berechnungen möglich sein.

Die ALU muss daher folgende Operationen unterstützen:
- Addition
- Subtraktion
- Operand durchreichen (Ausgang = Operand A)
- Ausgang = 0 (HI löschen)
- Ausgang = 0xFFFFFFFF (HI als Vorzeichenerweiterung)
- Operand invertieren (Schritt 1 der Negation)
- Operand mit 1 addieren (Schritt 2 der Negation)

Als Ein- und Ausgänge werden benötigt:
- Eingang für Operand A
- Eingang für Operand B
- Eingang für ALU Steuerung
- Ausgang für Ergebnis
- Ausgang für Carry
- Ausgang für Borro

3.1.4 Booth Controller

Der Booth Controller ist nichts weiter als ein 1Bit Schieberegister. Welches das raus geschobene Bit aus dem LO Register speichert. Das nachfolgende LSB des LO Registers und das gespeicherte Booth Bit bilden dann beide das momentane Booth Encoding.

Als Ein- und Ausgänge werden benötigt:
- Eingang für „LO links schieben“
- Eingang für LO LSB
- Eingang für Taktsignal
- Ausgang für Booth Encoding

3.1.5 Schreibsteuerung

Die Schreibsteuerung übernimmt eine Doppelrolle, bei allen Rechnungen wird entschieden wann HI geschrieben wird. Bei den Divisionen wird auch entschieden ob q0 auf 1 oder 0 gesetzt wird. Für jede Rechenvorschrift existiert ein anderes Entscheidungskriterium, daher muss diesem Schaltungsteil vom Steuerwerk mitgeteilt werden welche Rechenoperation grade durchgeführt wird und ob überhaupt geschrieben werden soll.

Beim Multiplizieren ohne Vorzeichen wird geschrieben, wenn das LSB des LO Register 1 ist.
Beim Multiplizieren mit Vorzeichen wird geschrieben, wenn beide Bits des Booth Encodings unterschiedlich sind.
Beim Dividieren ohne Vorzeichen wird geschrieben, wenn der Borrow ausgang der ALU 0 ist.
Beim Dividieren mit Vorzeichen wird geschrieben, wenn sich das Vorzeichen zwischen dem Inhalt von HI und dem ALU Ergebnis nicht ändern oder der Sonderfall eintritt.
Insofern beim Dividieren geschrieben wird, dann wird auch q0 auf 1 gesetzt, sonst 0.

Als Ein- und Ausgänge werden benötigt:
- Eingang für Rechenoperation selektieren
- Eingang für Schreibfreigabe
- Eingang für Booth Encoding
- Eingang für LSB LO Register
- Eingang für Borrow
- Eingang für Vorzeichen ALU Ergebnis
- Eingang für Vorzeichen HI Register
- Eingang für Division Sonderfall Vergleicher
- Ausgang für q0
- Ausgang für HI schreiben

3.1.6 Schiebesteuerung

Die Schiebesteuerung entscheidet während einer Multiplikation ob eine 1 oder eine 0 beim rechts schieben in das HI Schieberegister geschoben wird. Zur Entscheidungsfindung wird das Carrybit der ALU benötigt, sowie ob mit oder ohne Vorzeichen gerechnet wird. Zur Erkennung des Sonderfalls beim Booth Algorithmus wird daher auch das jetzige Booth Encoding sowie das Vorzeichen von Operand B benötigt.

Beim Multiplizieren ohne Vorzeichen wird das Carrybit intern gespeichert, wenn auch das HI Register geschrieben wird und anschließend wird dieses beim nächsten schieben nach rechts ausgegeben.
Beim Multiplizieren mit Vorzeichen wird immer arithmetisch nach rechts geschoben, daher gibt dieser Schaltungsblock das letzte MSB des HI Registers aus. Es sei denn es tritt der Sonderfall ein, dann wird 0 ausgegeben für ein logisches schieben.
Beim dividieren ist der Ausgang der Schaltung nicht von relevant, denn es wird immer nach links geschoben und diese Schaltung speist nur den Eingang für das rein geschobene Bit beim rechts schieben.

Als Ein- und Ausgänge werden benötigt:
- Eingang für Carry
- Eingang für „HI schreiben“ = Carry merken
- Eingang für Taktsignal
- Eingang für MSB des HI Schieberegister (arith. schieben)
- Eingang für Steuerung Rechnung mit/ohne Vorzeichen
- Eingang für Booth Encoding
- Eingang für Vorzeichen Operand B
- Ausgang für rein zuschiebendes Bit nach HI

3.1.7 Steuerwerk

Das Steuerwerk koordiniert den sequentiellen Ablauf der Berechnung inkl. der Vor- und Nachbetrachtungen. Das Steuerwerk gibt die Signale zur Steuerung der ALU aus, sowie die Operationen der HI und LO Schieberegister und deren Steuerungen.
Das Steuerwerk muss auch intern Entscheidungen treffen um z.B. ein Addieren oder Subtrahieren auszulösen oder den Quotienten zu negieren als Nachbetrachtung.
Weiteres zum Steuerwerk wird beim Mikrocode erklärt.

Als Ein- und Ausgänge werden benötigt:
- Eingang für Taktsignal
- Eingang für mul_div_S (Auswahl der Operation)
- Eingang für #store_reg_to_mul_div (Startbedingung)
- Eingang für Booth Encoding (Addieren oder Subtrahieren)
- Eingang für LO Register Bit 31 (Vorzeichenerweiterung nach HI)
- Eingang für HI Vorzeichen und ALU Ergebnis Vorzeichen (Addieren oder Subtrahieren)
- Eingang für gespeichertes Vorzeichen Operand A (DIV Nachbetrachtung)
- Ausgang für ALU Steuerung
- Ausgang für HI, LO Steuerung
- Ausgang für Schreibsteuerung Steuerung
- Ausgang für Schiebesteuerung Steuerung
- Ausgang für anhalten der CPU

3.2 TTL Implementierung

Bei der TTL Implementierung müssen nun die in 3.1 Hardwareblöcke gefundenen Funktionen mit den verfügbaren TTL ICs abgebildet werden.

3.2.1 Schieberegister HI und LO

Für diesen Schaltungsteil bietet sich der 74F194 an. Dieser IC stellt bereits fast alle benötigten Funktionen zur Verfügung, nur der tristate Ausgang ist nicht vorhanden. Diese Funktion wird durch nachgeschaltete 74F245 realisiert.

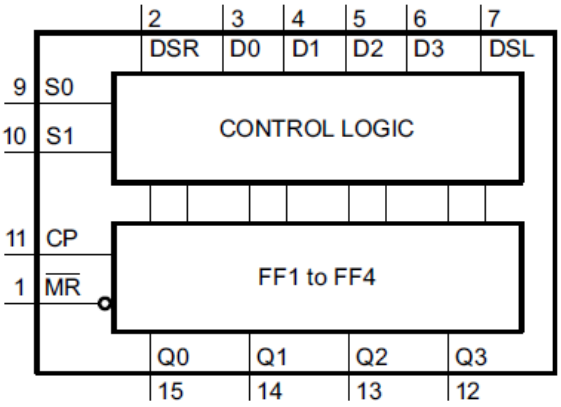


Abbildung 33 74F194 Logiksymbol

Operating mode	Inputs							Outputs			
	CP	$\overline{MR}$	S1	S0	DSR	DSL	Dn	Q0	Q1	Q2	Q3
Reset (clear)	X	L	X	X	X	X	X	L	L	L	L
Hold (do nothing)	X	H	l	l	X	X	X	q0	q1	q2	q3
Shift left	↑	H	h	l	X	l	X	q1	q2	q3	L
	↑	H	h	l	X	h	X	q1	q2	q3	H
Shift right	↑	H	l	h	l	X	X	L	q0	q1	q2
	↑	H	l	h	h	X	X	H	q0	q1	q2
Parallel load	↑	H	h	h	X	X	dn	d0	d1	d2	d3

Abbildung 34 74F194 Logiktabelle

Besonders die Codierung b11 am Steuereingang S für den Schreibbefehl ist von Vorteil, denn somit kann die Schreibsteuerung das Steuersignal des Steuerwerks simpel mit 2 ODER Gattern überschreiben.

Daraus ergibt sich folgende Schaltung als 8Bit Ausschnitt des HI Registers:

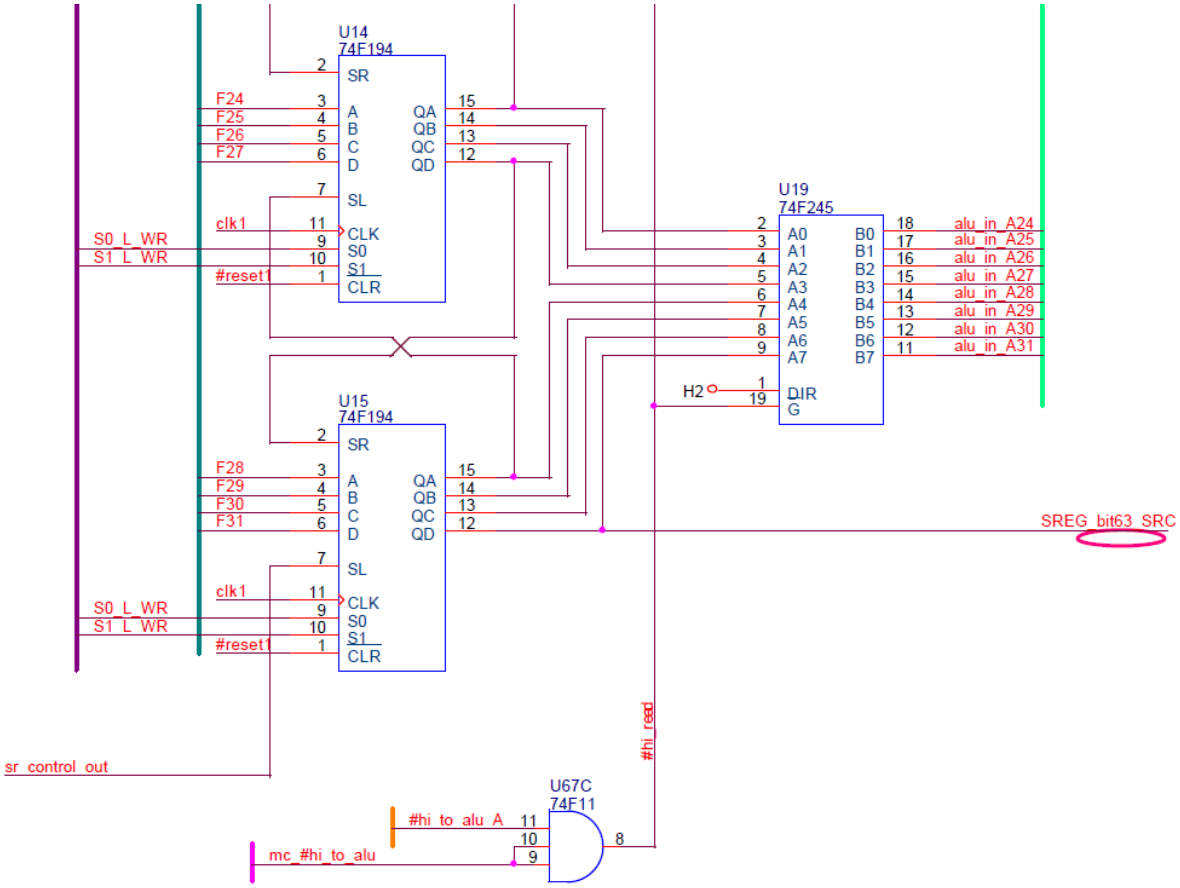


Abbildung 35 HI Register MSB Ende

Die äußeren Ausgänge QA und QD der jeweiligen Schieberegister ICs werden über Kreuz zu den Schiebebit Eingängen SR und SL der benachbarten Schieberegister ICs verbunden um 8 Stück 4Bit Schieberegister zu einem 32Bit Schieberegister zu verschalten. Die Ausgänge der ICs sind für den tristate Modus an einen nachfolgenden Bustreiber angeschlossen. Da jedoch auch dauerhaft das raus geschobene Bit anliegen soll werden QD des MSB IC und QA des LSB IC herausgeführt als „Ausgang für das raus geschobene Bit (links und rechts)“. Die äußeren Eingänge SL und SR stellen den „Eingang für das rein geschobene Bit (links und rechts)“. Die Steuersignale aus dem Steuerwerk werden zu allen ICs parallel geführt.

Das Signal **F** ist das Ergebnis der ALU und das Signal **alu_in_A** liegt am Operanden A der ALU an zudem dient dieses zum Auslesen des Registerinhalts zur CPU.
Die Signale **S0_L_WR** und **S1_L_WR** steuern das HI Register und kommen aus dem Steuerwerk. Vorher werden die beiden Signale jedoch durch die Schreibsteuerung durchgeschleift um bei den Teilberechnungen bedingt das ALU Ergebnis zu speichern. Das Signal **#hi_read** wird aus dem Steuerwerksignal **mc_#hi_to_alu** gebildet sowie dem CPU Signal **#hi_to_alu_A**. Denn jeder der beiden Signale wird benötigt zum Auslesen des HI Registers.
Das Signal **sr_control_out** kommt aus der Schiebesteuerung und stellt das in das HI Register rein zuschiebende Bit dar.
Der verbleibende Ausgang **SREG_bit63_SRC** ist das Vorzeichenbit des HI Registers.

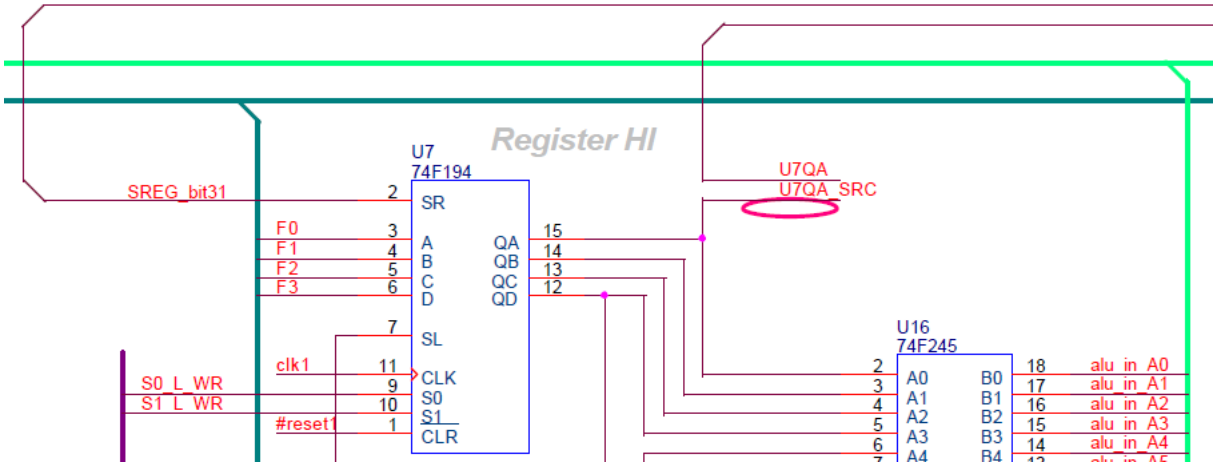


Abbildung 36 HI Register LSB Ende

In das LSB Ende des HI Registers wird das MSB des LO Registers reingeschoben beim rechts Schieben, dies ist das Signal **SREG_bit31**. Beim nach links schieben wird das LSB des HI Registers zum LO Register durchgereicht, das Signal **U7QA_SRC**.

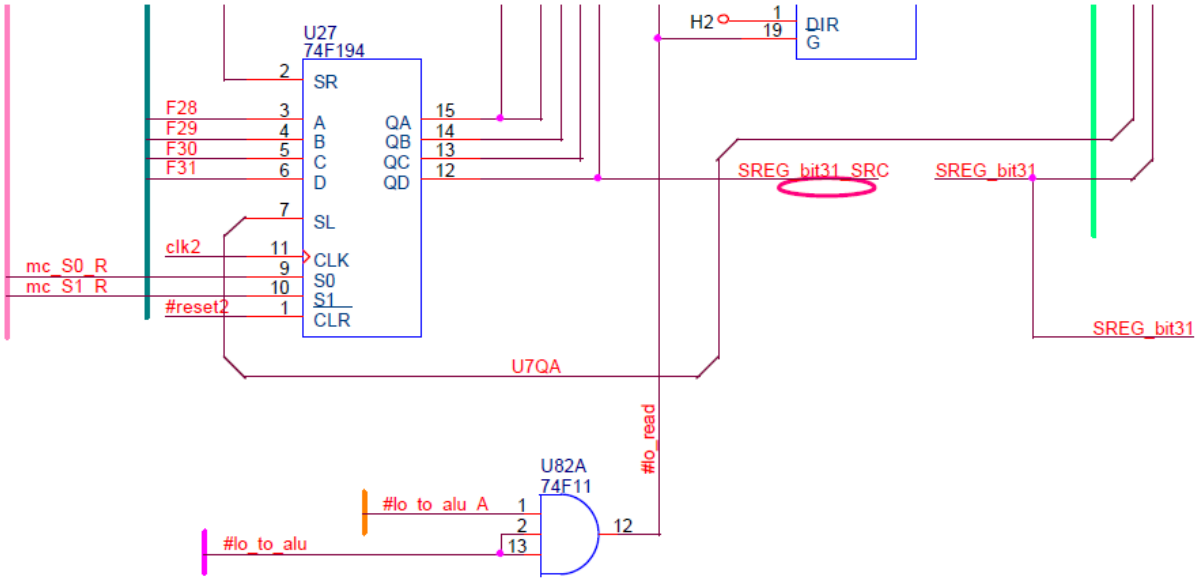


Abbildung 37 LO Register MSB Ende

Die Signale **mc_S0_L** und **mc_S1_L** steuern das LO Register und kommen direkt aus dem Steuerwerk. Das Signal **#lo_read** wird aus dem Steuerwerksignal **mc_#lo_to_alu** gebildet sowie dem CPU Signal **#lo_to_alu_A**. Denn jeder der beiden Signale wird benötigt zum Auslesen des LO Registers.
Das Signal **U7QA** ist das raus geschobene Bit aus dem HI Register wund wird beim rechts Schieben in das LO Register geschoben. Währenddessen das Signal **SREG_bit31_SRC** die andere Richtung darstellt und beim links Schieben in das HI Register geschoben wird. Gleichzeitig stellt dieses Signal das Vorzeichenbit des LO Registers zur Verfügung.

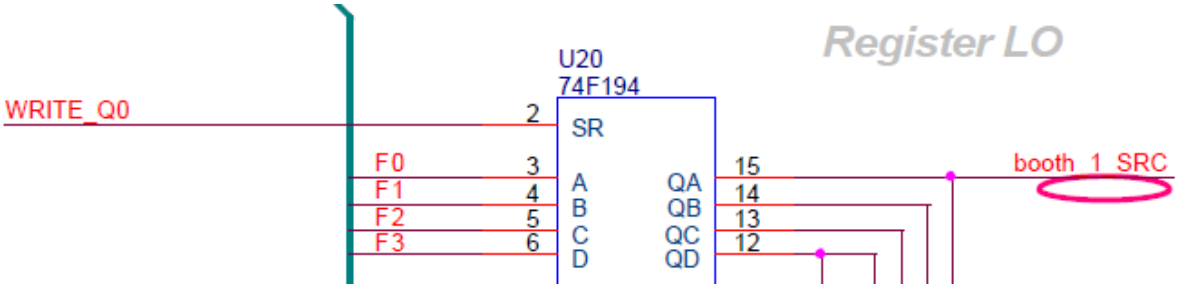


Abbildung 38 LO Register LSB Ende

Schlussendlich noch die Signale des LSB Ende vom LO Register.
Bei einem links Schieben zur Division wird das Signal **WRITE_Q0** aus der Schreibsteuerung in das LO Register geschrieben um den Quotienten zu bilden.
Das raus geschobene Bit **booth_1_SRC** beim rechts Schieben ist bei der vorzeichenlosen Multiplikation das Steuerbit zum schreiben des HI Registers oder bildet Bit1 des Booth Encodings beim vorzeichenbehafteten Multiplizieren.

3.2.2 Schattenregister LO und Nullvergleich

Das Schattenregister ist genauso aufgebaut wie das LO Register. Jedoch entfällt der Bustreiber, da dieses Register nie über einen Bus ausgelesen wird. Stattdessen schließt sich dem Ausgang der Schieberegister ICs direkt ein Vergleicher an.

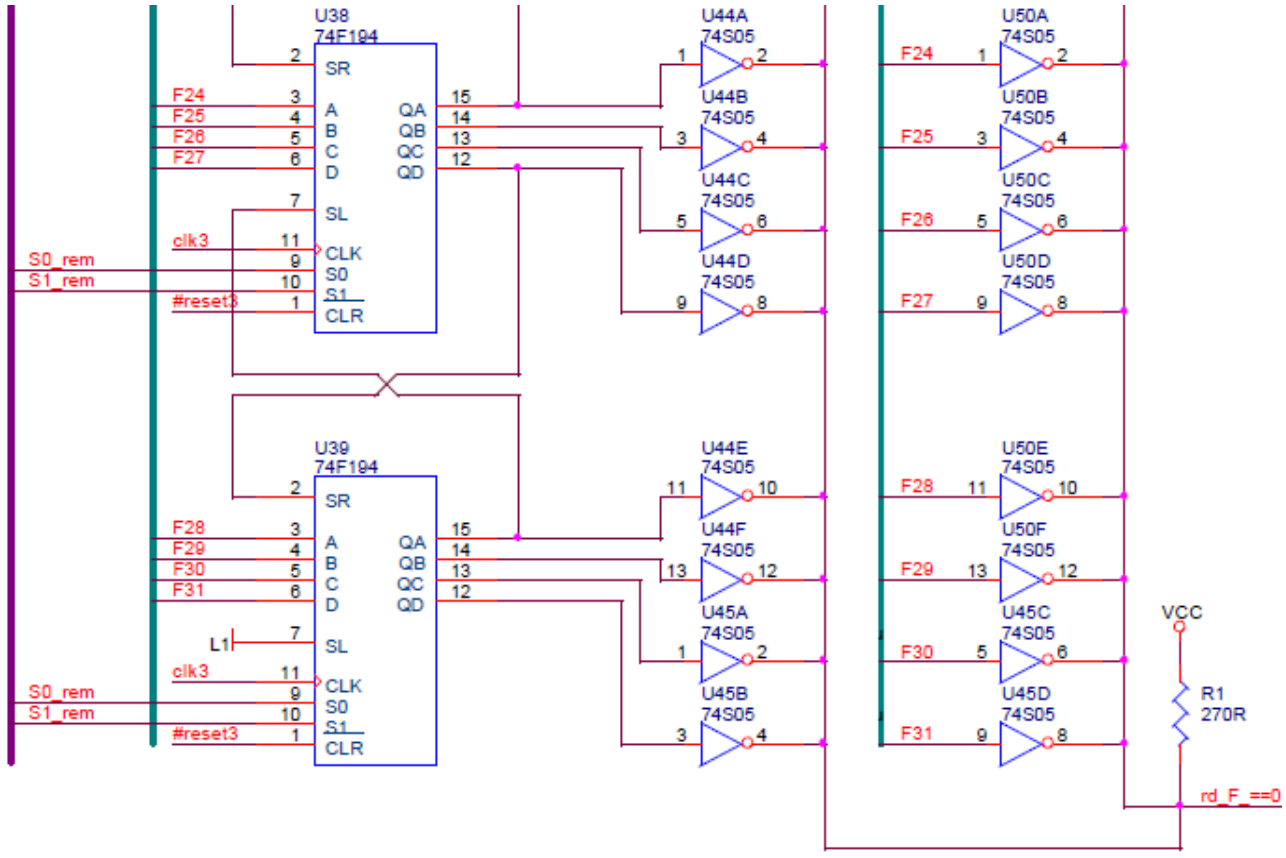


Abbildung 39

Wie in der Abbildung zu sehen ist, wird der IC 74S05 als Vergleicher benutzt. Dieser IC ist ein 6-facher Inverter mit Open Collector Ausgang. Sobald eines der 64Bits 1 ist zieht der Inverter den Ausgang **rd_F_==0** auf Lowpegel. Erst wenn alle 64Bit auf null sind besteht die Chance, dass der Ausgang über den Pullup auf Highpegel gezogen wird.
Mit dieser Methode lässt sich ein ressourcensparender und schneller Vergleicher auf null realisieren. Eine baumförmige Verschaltung aus ODER Gattern und einem finalen NOR Gatter zu je 2 Eingängen würde 32+16+8+4+2=62 ODER Gatter benötigen. Das wären bei 4 Gattern pro IC 16+1 ICs. Bei der Inverterlösung werden nur 11 ICs benötigt und auch kein Baum mit 5 Stufen welche die Laufzeit erhöhen.

Das Schattenregister muss zur gleichen Zeit geschoben werden wie das HI Register, aber muss zur gleichen Zeit geschrieben werden wie das LO Register. Daher kann nicht eines der HI, LO Steuersignale zur Ansteuerung genutzt werden sondern es fallen die internen Steuersignale **S0_rem** und **S1_rem** an. Daher besitzt das Schattenregister seine eigene kleinere Steuerung. Diese Speichert auch das Vorzeichen für die Nachbetrachtung der vorzeichenbehafteten Division. Denn dieses Vorzeichen muss zur gleichen Zeit gespeichert werden wie der Inhalt des Schattenregisters.

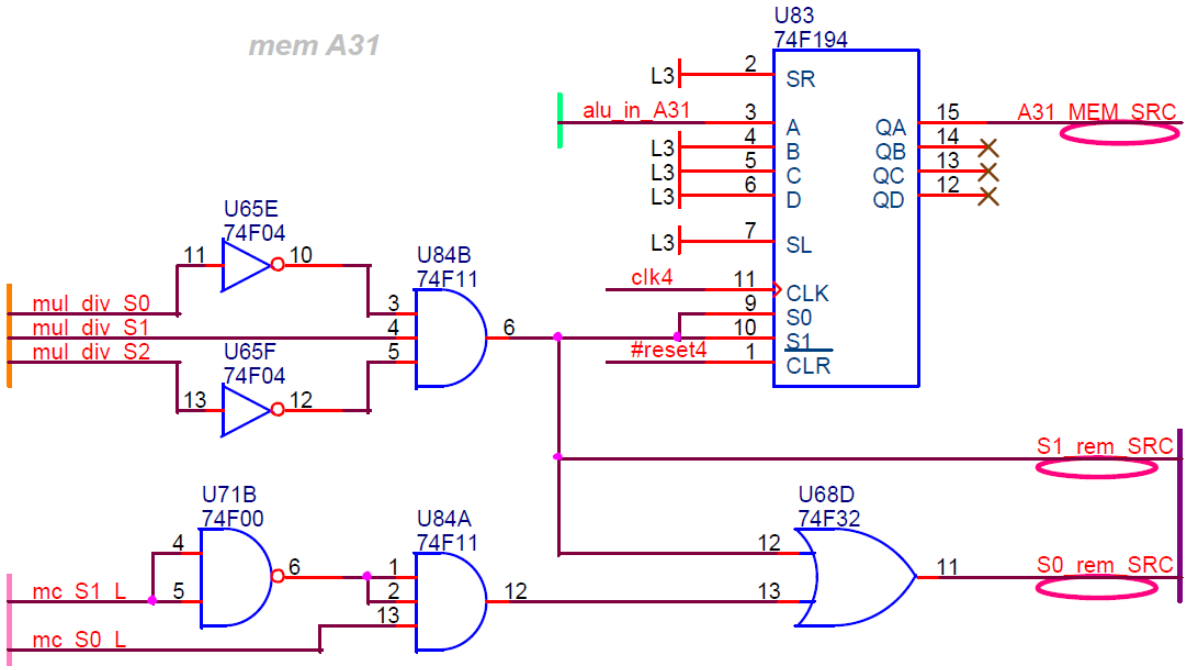


Abbildung 40

Als Speicherelement für das Vorzeichen wird auch ein 74F194 genutzt um den BOM nicht unnötig zu vergrößern. Dieses Schieberegister ist so beschalten, dass es nur speichern kann. Zum Zeitpunkt des Speicherns liegt am Signal **alu_in_A31** das Vorzeichenbit des Operanden A an. Bis zum nächsten Vorgang ist dieses Vorzeichenbit dann als **A31_MEM_SRC** in der MulDiv Schaltung verfügbar.
Das Schreibsignal für das Vorzeichenregister und das Schattenregister wird aus dem Steuersignal **mul_div_S** gebildet und über U68D ins das Steuersignal zum Schattenregister eingeschleift. Wenn dieses Signal auf b010 steht wird das Register LO beschrieben und durch die Steuerung auch das Schatten- und Vorzeichenregister. Weiterhin überprüft die Schaltung ob das Register HI nach links geschoben werden soll und schiebt dementsprechend auch das Schattenregister nach links. Dafür liegen an der internen Steuerung die Schieberegistersteuersignale **mc_S1_L** und **mc_S0_L** vom HI Register an. Der Befehl „HI speichern“ muss unbedingt herausgefiltert werden, sonst wird auch das Schattenregister geschrieben und es ist nicht mehr möglich auf den verbleibenden Dividenden zu prüfen. Diese Filterung wird durch U71B und U84A realisiert, indem nur links schieben durchgelassen wird.

3.2.3 ALU

Als ALU wird auch bei der MulDiv Karte auf die Kombination aus 74S181 und 74S182 zurückgegriffen. Durch den Carry Lookahead Aufbau ergeben sich kurze Rechenzeiten beim Addieren und Subtrahieren. Weiterhin stellt diese Kombination auch ein Carry und Borrow Signal bereit. Da die Schaltung mit positiver Logik betrieben wird ist der Carry/Borrow Ausgang negiert und bekommt dementsprechend einen Inverter. Das Signal wird im Schaltplan **CARRY_OUT_SRC** genannt. Die ALU wird direkt aus dem Steuerwerk über die Signale **mc_S[0..5]** gesteuert und besitzt keine weitere eigene Logik.
Die ALU stellt das das Ergebnis aus den Operanden Signalen **alu_in_A** und **alu_in_B** auf dem Signal **F** bereit.

3.2.4 Booth Controller

Der Booth Controller besteht aus einem weiteren Schieberegister IC um das aus LO heraus geschobene Bit aufzunehmen.

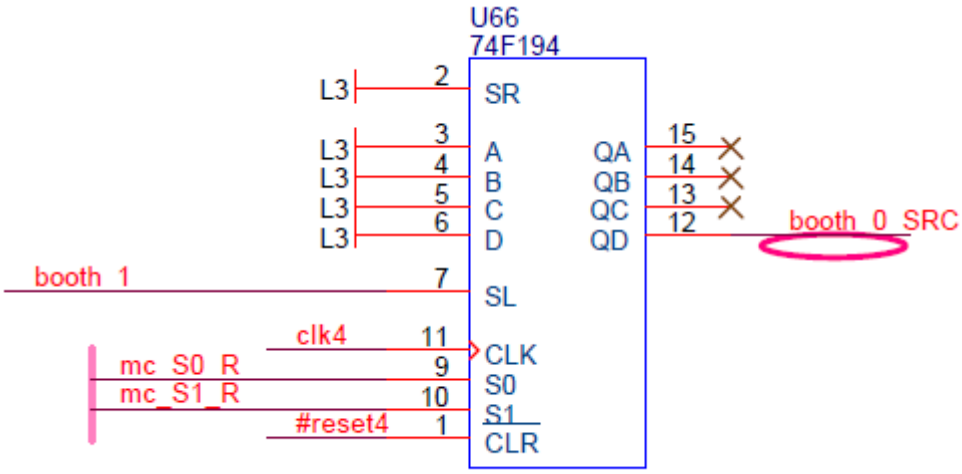


Abbildung 41

Gesteuert wird dieser IC vom selben Signal wie das LO Register, weil beide zeitgleich geschoben und initialisiert werden. Das Booth Register muss mit null initialisiert werden, weshalb die Eingänge zum parallelen laden auf GND liegen.
Das Signal **booth_1** ist das aus dem LO Register raus geschobene Signal bei einem rechts Schieben und bildet mit dem Signal **booth_0_SRC** das Booth Encoding für weitere Steueraufgaben in der Schaltung.

3.2.5 Schreibsteuerung

Die Schreibsteuerung dient der Vereinfachung des Steuerwerks indem die Entscheidungsfindung des Schreibens vom HI Register hier abgenommen wird.
Jede der 4 Rechenarten hat eine eigene Logik zur Erkennung des Schreibbefehls. Daher muss ein Multiplexer aus diesen 4 Logiken auswählen. In der Schaltung wird ein 74F151 1 aus 8 Multiplexer genutzt, weil dieser schon in der CPU genutzt wurde. Dies hält die BOM klein.

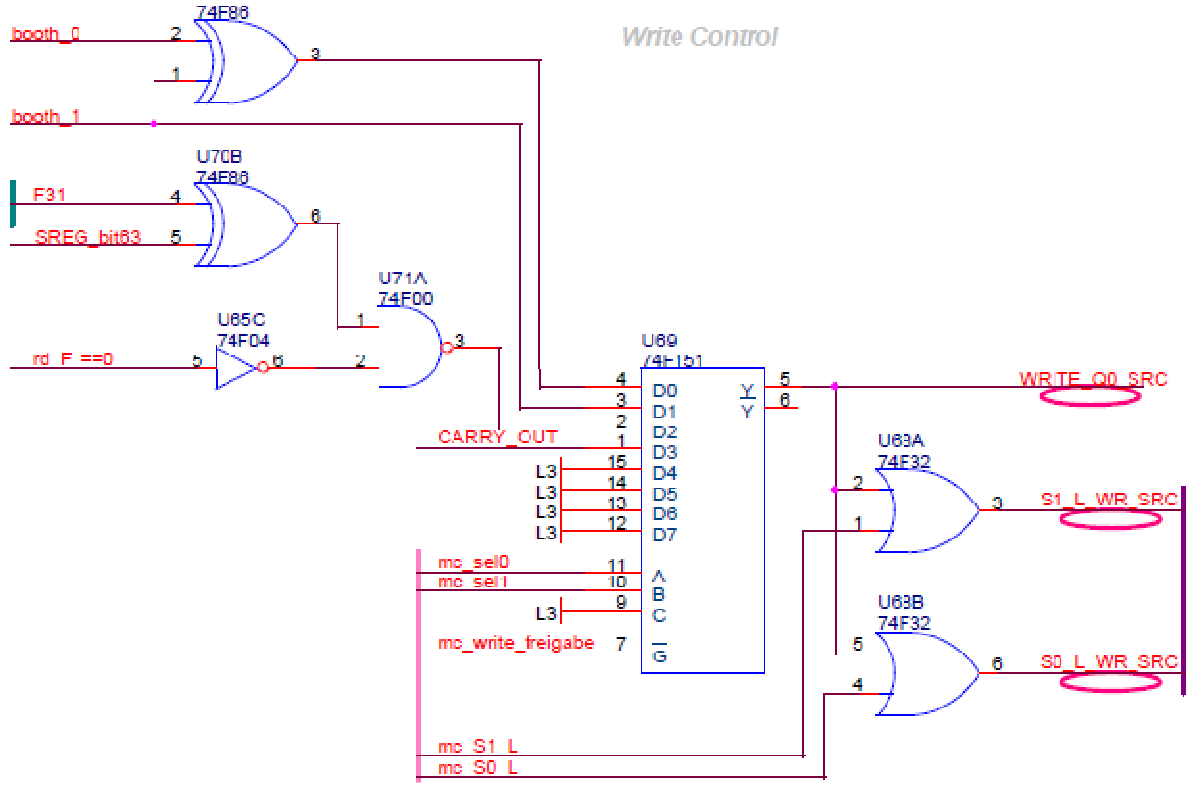


Abbildung 42

Das Steuerwerk muss mit dem Signal **mc_write_freigabe** das Schreiben in das HI Register freigeben. Welche Logik schlussendlich das Schreiben auslöst wird mit den Signalen **mc_sel0** und **mc_sel1** ausgewählt.
Es wird ausgewählt zwischen:
- dem Booth Encoding, sind beide Bits unterschiedlich so wird geschrieben
- dem LSB des LO Registers, das Signal heißt im Schaltplan **booth_1**
- dem Vorzeichenunterschied aus ALU Ergebnis und HI Register mit dem Sonderfall
- dem Borrow Bit der ALU, welches im Schaltplan **CARRY_OUT** heißt

Das Schreibsignal wird mit U68A und U68B in das Steuersignal **mc_S1_L, mc_S0_L** aus dem Steuerwerk eingespeist. Das Signal **S1_L_WR_SRC** und **S0_L_WR_SRC** steuert dann das HI Register.
Das Signal **WRITE_Q0_SRC** liegt direkt am LO Register an, denn nur beim Schieben nach links kann dieses aktiv genutzt werden und dies ist nur beim Dividieren der Fall wo dieses Signal benötigt wird.

3.2.6 Schiebesteuerung

Die Schiebesteuerung dient der Vereinfachung des Steuerwerks indem die Entscheidungsfindung, welches Bit beim rechts Schieben in das HI Register geschoben wird, dem Steuerwerk abgenommen wird.

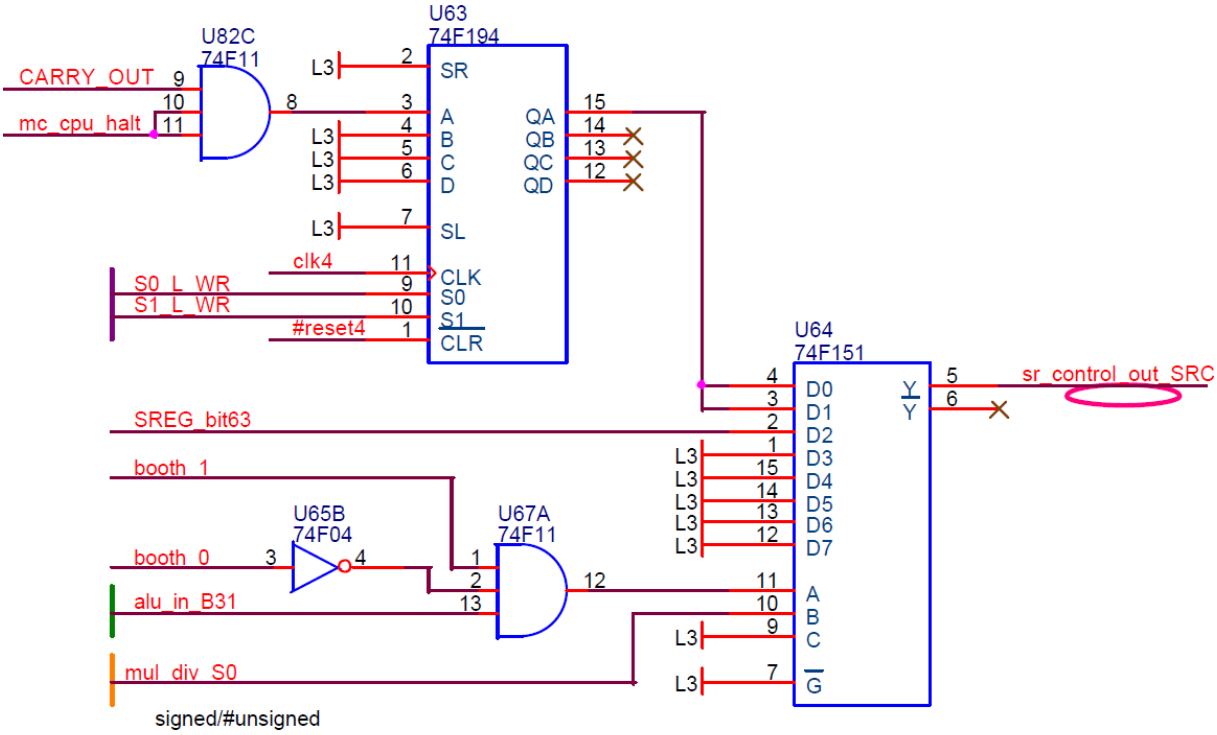


Abbildung 43

Diese Schaltung wird nur beim Schieben nach rechts aktiv, denn sonst kann kein Bit in das HI Register rein geschoben werden. Daher wird kein Steuerwerk Signal herangezogen zur Auswahl des Schiebebits, sondern direkt das Steuersignal **mul_div_S0**, dieses Signal codiert ob die Rechenoperation mit oder ohne Vorzeichen stattfindet. Das Signal ist 1 für eine Berechnung mit Vorzeichen.

Ist die Berechnung ohne Vorzeichen so ist der Multiplexersteuereingang A ohne Belang und daher besitzen der Dateneingang D0 und D1 dieselbe Quelle. Diese Quelle ist das Carrybit der letzten Berechnung, welches nun in das HI Register hineingeschoben werden muss. Das Carrybit muss dazu zuerst gespeichert werden, was auch in einem 74F194 stattfindet zur klein Haltung des BOM. Das Carryregister wird gleichzeitig mit dem HI Register beschrieben und geschoben, daher sind die Steuersignale **S0_L_WR** und **S1_L_WR** dieselben. Während des ersten Schreibens des HI Registers zum Initialisieren mit 0 führt die ALU keine valide Operation aus und selbst wenn: die Operanden sind unbekannt. Daher kann das Signal **CARRY_OUT** einen beliebigen Wert führen und muss somit mit dem Signal **mc_cpu_halt** maskiert werden.

Bei einem Multiplizieren mit Vorzeichen wird arithmetisch geschoben, dies wird erreicht indem das MSB des HI Registers, das Signal **SREG_bit63**, an den Ausgang **sr_control_out_SRC** durchgereicht wird. Nur beim Sonderfall des Aufbrechens des arithmetischen Schiebens wird eine 0 an den Ausgang durchgereicht. Die Erkennung des Sonderfalls hängt am Multiplexersteuereingang A und somit wird über D2 und D3 das Verhalten gesteuert.

3.2.7 Steuerwerk

Das Steuerwerk steuert nicht jede einzelne Funktion der Schaltung, es gibt nur die grobe Richtung vor. So wird vorgegeben in welche Richtung geschoben werden muss und welche Operation die ALU auszuführen hat. Das Steuerwerk bestimmt aber nicht direkt welche Bits in die Schieberegister geschoben werden. Es bestimmt auch nicht wann das HI Register bei einer Zwischenschrittberechnung geschrieben wird. Nur eine Freigabe für den Schreibvorgang ist möglich. Diese Logik der einzelnen Berechnungen liegt in Hardware vor die das Steuerwerk nur noch auswählen muss.
Das Steuerwerk steuert auch nicht das Auslesen der HI, LO Register durch die CPU, dafür sind eigene Signale vorgesehen. Das Schreiben nach HI, LO wird jedoch durch das Steuerwerk bestimmt, denn diese Schreibvorgänge müssen durch die ALU.

Das Steuerwerk steuert dazu den Anfang der Berechnung indem Initialwerte in das HI Register geschrieben werden und die CPU angehalten wird. Danach wird die eigentliche Rechnung ausgeführt bis alle 32Bit berechnet wurden. Zum Schluss wird z.B. bei der Division noch eine Nachbetrachtung fällig oder die CPU wird wieder laufen gelassen.
Um das Steuerwerk einfach zu halten und um nachträglich Bugs auszumerzen wurde sich für ein Mikrocodesteuerwerk entschieden. Das heißt ein Sequenzer zählt die Adressen eines ROM hoch und der Inhalt des ROM steuert die Schaltung.

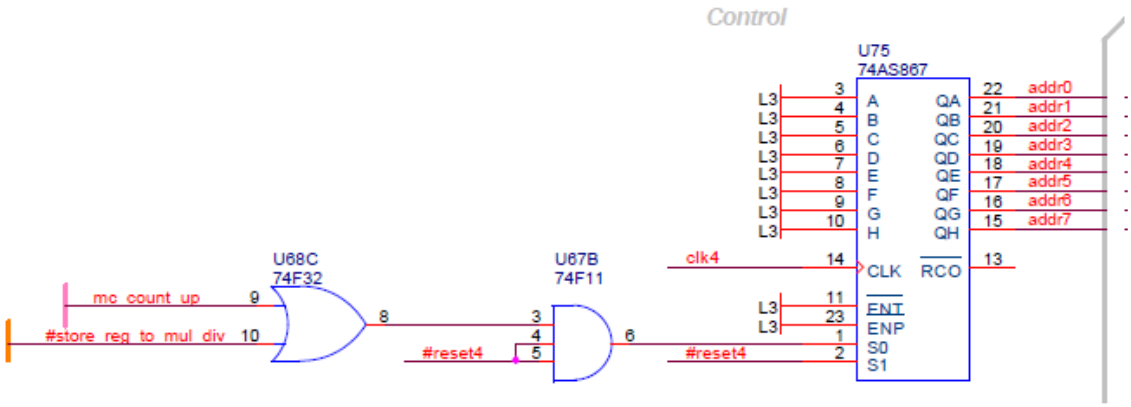


Abbildung 44 Sequenzer

Als Sequenzer wird ein 8Bit Zählerbaustein 74AS867 genutzt. Nach einem Reset steht dieser auf 0. Der Zähler beginnt erst zu zählen, wenn er durch das Signal **#store_reg_to_mul_div** freigeschalten wird. Dieses Signal wird vom CPU Kern nur kurz auf 1 gesetzt, daher muss das Steuerwerk sich selber halten durch das Signal **mc_count_up**. Sind beide Eingangssignale des Gatters U68C wieder auf 0, so wird der Zähler, wie beim Reset, wieder auf 0 zurückgesetzt für die nächste Berechnung. Nur diesmal durch den Befehl „parallel load“ anstatt einem echten Reset.

Zusammen mit dem Signal **mul_div_S**, welches nach einem Buffer nun **mds** heißt, wird as dem Zählerstand die Adresse der ROM Bausteine gebildet. Das LSB ist dabei ein Flag, als Signalname **flg**. Dieses Flag dient der Entscheidungsfindung ob addiert oder subtrahiert werden muss. Oder ob bei einer Nachbetrachtung der Quotient negiert werden muss.

Welches Flag ausgewählt wird bestimmt der Mikcrocode selbst durch die Signale **mc_sel0** und **mc_sel1**. Denn auch hier gilt, dass für jede Rechenoperation eine andere Entscheidungsfindung von Nöten ist.

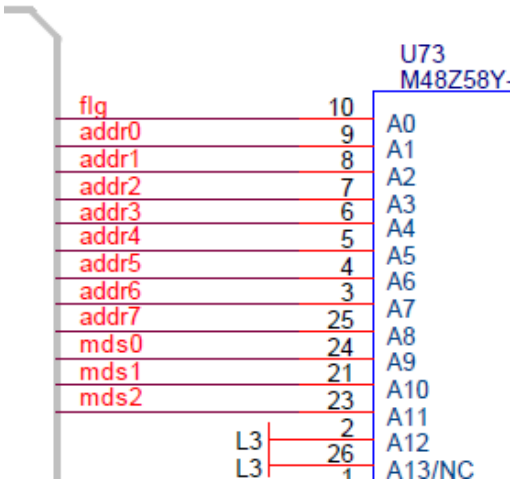


Abbildung 45 Adressbildung

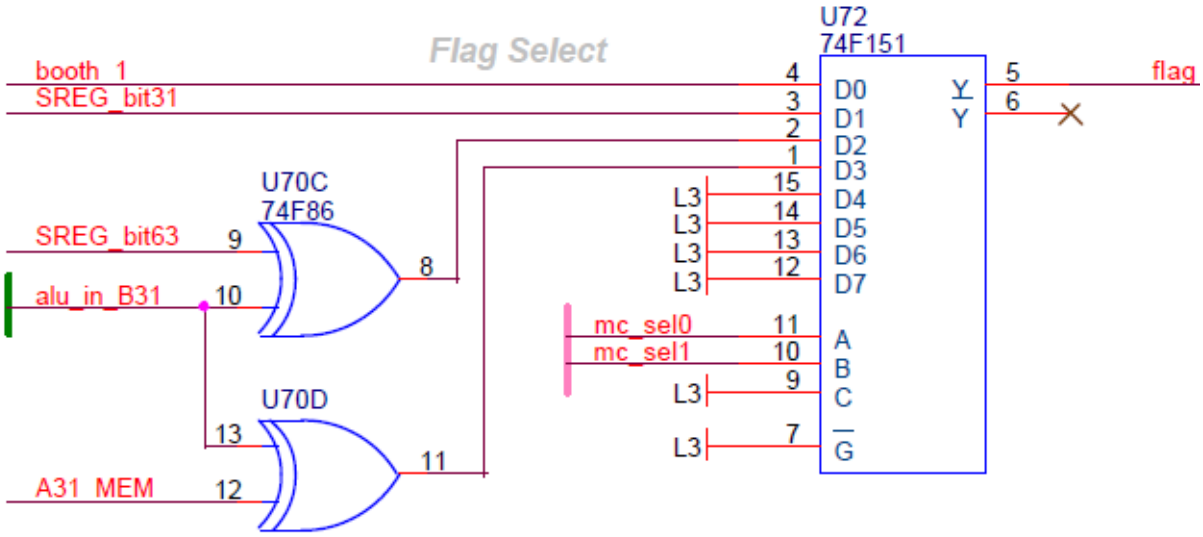
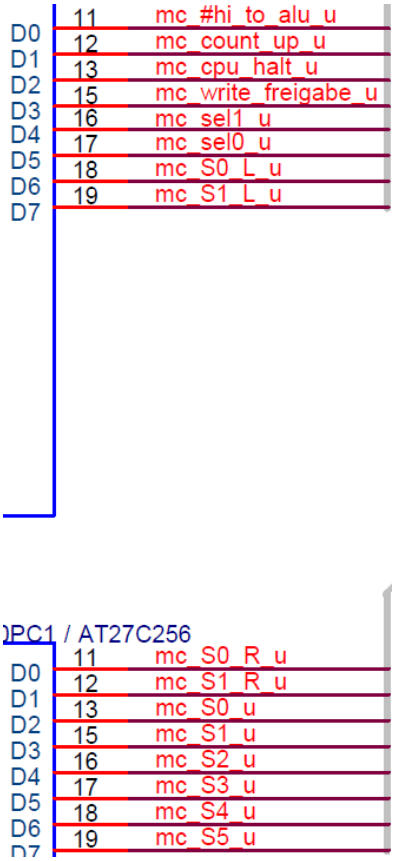


Abbildung 46 Flag Auswahl

Die Berechnungen ohne Vorzeichen benötigen kein Flag. Dafür benötigt die Division mit Vorzeichen gleich derer drei:
- D1 -> Erkennung einer negativen Zahl in LO zum Vorzeichenerweitern in HI
- D2 -> Erkennung der Vorzeichendifferenz zur ALU Operation addieren oder subtrahieren
- D3 -> Erkennung ob der Quotient zur Nachbetrachtung negiert werden muss

Die Multiplikation mit Vorzeichen nutzt nur das **booth_1** Bit zur Erkennung ob addiert oder subtrahiert werden muss.

Schlussendlich die Dokumentation der Steuersignale des Steuerwerks:



mc_#hi_to_alu
Legt das LO Register auf den Operandenbus der ALU.

mc_count_up
Startet den Sequenzer des Steuerwerks.

mc_cpu_halt
Hält die restliche CPU an durch das Aktivieren des Signals **halt/#run** des CPU Steuerwerks.

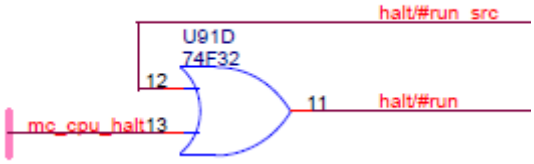


Abbildung 48 CPU HALT

Abbildung 47 Steuersignale

mc_write_freigabe
Gibt die Schreibsteuerung frei oder sperrt diese. Durch die Mehrfachnutzung des Signals **mc_sel** kann es sonst zu ungewünschten Schreibaktionen kommen während einer Vor- oder Nachbetrachtung.

mc_sel
Dieses Signal wählt das Flag des Steuerwerks aus und welche Logik den Schreibbefehl auslöst.

mc_sel[1..0]	**Schreibbefehl für Rechenoperation**
b00	Multiplikation mit Vorzeichen
b01	Multiplikation ohne Vorzeichen
b10	Division mit Vorzeichen
b11	Division ohne Vorzeichen

mc_S_L
Das Signal zum kontrollieren des HI Registers.

mc_S_R
Das Signal zum steuern des LO Registers.

mc_S[1..0]_X	**Schieberegisteroperation**
b00	halten
b01	links schieben
b10	rechts schieben
b11	schreiben

mc_S
Steuerbits der ALU. Dabei sind Bits mc_S[3..0] die Bits S[3..0] des 74S181. Das Bit mc_S4 ist das M Bit des 74S181 und das Bit mc_S5 ist das Carry Eingangsbit des 74S181.

mc_S[5..0]	**ALU Operation**
b000000	HI o. LO += 1 (F = A + 1)
b000011	HI o. LO Register löschen (F = 0)
b000110	Subtrahieren (F = A – B)
b100011	HI o. LO auf einsen setzen (F = MINUS 1)
b101001	Addieren (F = A + B)
b110000	HI o. LO invertieren (F = not A)
b111111	Operand A nach HI o. LO Register (F = A)

mc_#lo_to_alu
Dieses Signal wird kombinatorisch gebildet, weil alle 16Bit der ROM bereits genutzt wurden. Da dieses Steuersignal nur zur Nachbetrachtung der Division mit Vorzeichen benötigt wird. Während der Nachbetrachtung zum Quotient negieren steht **mc_sel a**uf 0b11 und **mc_#hi_to_alu** ist auch auf 1, denn es wird LO benötigt, nicht HI. Daraus ergibt sich ein 3-fach NAND Gatter zur Erzeugung des Signals.
Weiterhin ist es durch diese Signalgenerierung nicht möglich, dass HI und LO gleichzeitig auf den Bus **alu_in_A** geschalten werden können.

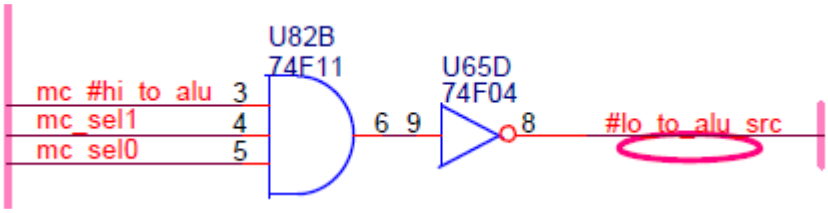


Abbildung 49 #lo_to_alu Erzeugung

Ein NAND Gatter mit 3 Eingängen wird in der Schaltung bisher nicht benötigt, aber es sind noch ein 3-fach UND Gatter sowie ein Inverter frei gewesen.

4. Microcode

Bisher ist die Theorie der Berechnung geklärt sowie die passende Hardware dazu. Das Steuerwerk muss aber noch belebt werden durch einen geeigneten Mikrocode. Der Inhalt des Mikrocodes wird in diesem Kapitel beschrieben.

Das Steuerwerk hätte auch als klassischer Automat gebaut werden können, aber dessen Gatterlaufzeit würde die eines ROM Bausteins übersteigen. Weiterhin wäre auch der Platzverbrauch größer und eventuelle Fehler wären nur schwer zu beheben.

Der Mikrocode wird in einer Exceltabelle erstellt. Das klingt zunächst abwegig, aber somit sind Kommentare und Farbmarkierungen möglich um auf die Abläufe hinzuweisen. Diese Tabelle wird durch den Spaceage 2 Programmer eingelesen und auf das Steuerwerk geschrieben. Daher muss ein Schema eingehalten werden.

			LUT2								LUT1								
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
mtlo			#carry	M	S3	S2	S1	S0	S_R1	S_R0	S_L1	S_L0	SEL0	SEL1	#write_en	cpu_halt	count_up	#hi_to_alu	
	0b010	0	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	1	A nach LO (F = A)
	1024	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	1	A nach LO (F = A)
		xx	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	1	A nach LO (F = A)

Tabelle 1

Am Beispiel des einfachen Befehls „Move To LO“ (mtlo) wird dieser Aufbau nun einmal erklärt. In der ersten Spalte links wird nach einem bekannten Mnemonik gesucht und wird dieser gefunden, dann wird von diesem aus eine Spalte nach rechts und 2 Zeilen nach unten die Basisadresse des Befehls gesucht. Die Basisadresse ist vorgegeben durch das **mul_div_S** Signal am Adresseingang des ROM.
Ab dem Mnemonik werden 2 Spalten nach rechts und 1 Zeile nach unten die Subadressen eingelesen. Diese können von 0 bis 511 reichen. Dabei ist eine Leerzeile erlaubt ohne, dass das einlesen abgebrochen wird. Erst ab 2 Leerzeilen geschieht der Abbruch. Somit kann eine Zeile mit der Basisadresse „xx“ optisch abgetrennt werden. Dies ist die Zeile mit dem alle weiteren, bisher undefinierten, Adressen dieses Befehls aufgefüllt werden.

Jeder Mikrocode für eine Berechnung besteht aus 3 Teilen. Der Vorbetrachtung, der Berechnung und zum Schluss der Nachbetrachtung. Wobei die Nachbetrachtung auch entfallen kann. Die Übergabe der Daten zu einer Berechnung wird nicht im Berechnungsmikrocode behandelt. Die Daten werden mit dem mtlo Befehl übergeben und mit den Befehlen mfhi mflo von der CPU gelesen.

Anschließend die Mikrocodebeschreibung der einzelnen Berechnungen.

4.1 Multiplikation ohne Vorzeichen

			LUT2								LUT1								
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
multu			#carry	M	S3	S2	S1	S0	S_R1	S_R0	S_L1	S_L0	SEL0	SEL1	#write_en	cpu_halt	count_up	#hi_to_alu	
	0b110	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	Idlezustand, nichts tun
	3072	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	Idlezustand, nichts tun
		2	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1	1	HI loeschen (F = 0, arithmode)
		3	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1	1	HI loeschen (F = 0, arithmode)

Tabelle 2 Vorbetrachtung

Die Vorbetrachtung besteht aus einem Idletakt, dieser ist vorhanden zur Synchronisation mit der Rest CPU. Denn diese wird bei der späteren Berechnung angehalten.
Beim zweiten Takt wird dann der Sequenzer des Mikrocodes aktiviert (count_up) und das HI Register wird mithilfe der ALU gelöscht.

	LUT2								LUT1								
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
	#carry	M	S3	S2	S1	S0	S_R1	S_R0	S_L1	S_L0	SEL0	SEL1	#write_en	cpu_halt	count_up	#hi_to_alu	
4	1	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	addieren und nach HI schreiben
5	1	0	1	0	0	1	0	0	0	0	1	0	0	1	1	0	addieren und nach HI schreiben
6	1	0	1	0	0	1	1	0	1	0	1	0	1	1	1	0	shift right
7	1	0	1	0	0	1	1	0	1	0	1	0	1	1	1	0	shift right

Tabelle 3 Berechnung

Anschließend wird die Berechnung für alle 32 Bits wiederholt ausgeführt. Diese besteht aus der Addition des HI Registers mit dem Operanden B wenn das LSB des LO Registers gesetzt ist. Danach werden HI und LO Register nach rechts geschoben.
Beim letzten schieben existiert eine Ausnahme, denn dort ist das Bit „cpu_halt“ nicht mehr gesetzt, damit die CPU nach dem Ende der Berechnung wieder übernehmen kann.

Eine Nachbetrachtung findet nicht statt.

4.2 Multiplikation mit Vorzeichen

			LUT2								LUT1								
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
mult			#carry	M	S3	S2	S1	S0	S_R1	S_R0	S_L1	S_L0	SEL0	SEL1	#write_en	cpu_halt	count_up	#hi_to_alu	
	0b111	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	Idlezustand, nichts tun
	3584	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	Idlezustand, nichts tun
		2	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1	1	HI loeschen (F = 0, arithmode)
		3	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1	1	HI loeschen (F = 0, arithmode)

Tabelle 4 Vorbetrachtung

Die Vorbetrachtung besteht aus einem Idletakt, dieser ist vorhanden zur Synchronisation mit der Rest CPU. Denn diese wird bei der späteren Berechnung angehalten.
Beim zweiten Takt wird dann der Sequenzer des Mikrocodes aktiviert (count_up) und das HI Register wird mithilfe der ALU gelöscht.

	LUT2								LUT1								
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
	#carry	M	S3	S2	S1	S0	S_R1	S_R0	S_L1	S_L0	SEL0	SEL1	#write_en	cpu_halt	count_up	#hi_to_alu	
4	1	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	addieren und nach HI schreiben
5	0	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	subtrahieren und nach HI schreiben
6	1	0	1	0	0	1	1	0	1	0	0	0	1	1	1	0	shift right
7	1	0	1	0	0	1	1	0	1	0	0	0	1	1	1	0	shift right

Tabelle 5 Berechnung

Anschließend wird die Berechnung für alle 32 Bits wiederholt ausgeführt. Diese besteht, je nach aktuellem Flag Bit, aus einer Addition oder Subtraktion des HI Register mit dem Operanden B. Das Flag Bit wird dabei aus dem Booth Encoding bestimmt. Danach werden HI und LO Register nach rechts geschoben.
Beim letzten schieben existiert eine Ausnahme, denn dort ist das Bit „cpu_halt“ nicht mehr gesetzt, damit die CPU nach dem Ende der Berechnung wieder übernehmen kann.

Eine Nachbetrachtung findet nicht statt.

4.3 Division ohne Vorzeichen

			LUT2								LUT1								
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
divu			#carry	M	S3	S2	S1	S0	S_R1	S_R0	S_L1	S_L0	SEL0	SEL1	#write_en	cpu_halt	count_up	#hi_to_alu	
	0b100	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	Idlezustand, nichts tun
	2048	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	Idlezustand, nichts tun
		2	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1	1	HI loeschen (F = 0, arithmode)
		3	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1	1	HI loeschen (F = 0, arithmode)

Tabelle 6 Vorbetrachtung

Die Vorbetrachtung besteht aus einem Idletakt, dieser ist vorhanden zur Synchronisation mit der Rest CPU. Denn diese wird bei der späteren Berechnung angehalten.
Beim zweiten Takt wird dann der Sequenzer des Mikrocodes aktiviert (count_up) und das HI Register wird mithilfe der ALU gelöscht.

	LUT2								LUT1								
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
	#carry	M	S3	S2	S1	S0	S_R1	S_R0	S_L1	S_L0	SEL0	SEL1	#write_en	cpu_halt	count_up	#hi_to_alu	
4	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1	0	HI shift left
5	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1	0	HI shift left
6	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1	0	subtrahieren
7	0	0	0	1	1	0	0	0	0	0	1	1	1	1	1	0	subtrahieren
8	0	0	0	1	1	0	0	1	0	0	1	1	0	1	1	0	HI schreiben und q0 nach LO schreiben mit LO shift left
9	0	0	0	1	1	0	0	1	0	0	1	1	0	1	1	0	HI schreiben und q0 nach LO schreiben mit LO shift left

Tabelle 7 Berechnung

Durch den getrennten Schiebevorgang von HI und LO werden 3 statt 2 Takte pro Bitberechnung benötigt.
Zuerst wird HI nach links geschoben damit die Berechnung an der richtigen Position stattfinden kann. Danach wird der Subtrahiervorgang der ALU ausgelöst um 1 Takt später geschrieben werden zu können. Zu dem Zeitpunkt ist dann auch q0 bekannt und das LO Register kann auch nach links geschoben werden.
Beim letzten schieben existiert eine Ausnahme, denn dort ist das Bit „cpu_halt“ nicht mehr gesetzt, damit die CPU nach dem Ende der Berechnung wieder übernehmen kann.

Eine Nachbetrachtung findet nicht statt.

4.4 Division mit Vorzeichen

			LUT2								LUT1								
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
div			#carry	M	S3	S2	S1	S0	S_R1	S_R0	S_L1	S_L0	SEL0	SEL1	#write_en	cpu_halt	count_up	#hi_to_alu	
	0b101	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	Idlezustand, nichts tun
	2560	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	Idlezustand, nichts tun
		2	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1	1	HI loeschen (F = 0, arithmode)
		3	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1	1	HI loeschen (F = 0, arithmode)
		4	0	0	0	0	1	1	0	0	1	1	1	0	1	1	1	1	HI loeschen (F = 0, arithmode)
		5	1	0	0	0	1	1	0	0	1	1	1	0	1	1	1	1	HI aus LO Vorzeichenerweitern (F = MINUS 1)

Tabelle 8 Vorbetrachtung

Die Vorbetrachtung besteht aus einem Idletakt, dieser ist vorhanden zur Synchronisation mit der Rest CPU. Denn diese wird bei der späteren Berechnung angehalten.
Beim zweiten Takt wird dann der Sequenzer des Mikrocodes aktiviert (count_up) und das HI Register wird mithilfe der ALU gelöscht. Das Löschen ist hierbei nur ein Platzhalter um auf die Bereitstellung des Flag Bits zu warten. Im dritten Takt wird das HI Register vorzeichenerweitert, wenn die Zahl im LO Register negativ ist.

	LUT2								LUT1								
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
	#carry	M	S3	S2	S1	S0	S_R1	S_R0	S_L1	S_L0	SEL0	SEL1	#write_en	cpu_halt	count_up	#hi_to_alu	
6	0	0	0	1	1	0	0	0	0	1	0	1	1	1	1	0	HI shift left
7	0	0	0	1	1	0	0	0	0	1	0	1	1	1	1	0	HI shift left
8	0	0	0	1	1	0	0	0	0	0	0	1	1	1	1	0	subtrahieren
9	1	0	1	0	0	1	0	0	0	0	0	1	1	1	1	0	addieren
10	0	0	0	1	1	0	0	1	0	0	0	1	0	1	1	0	HI schreiben und q0 nach LO schreiben mit LO shift left (sub)
11	1	0	1	0	0	1	0	1	0	0	0	1	0	1	1	0	HI schreiben und q0 nach LO schreiben mit LO shift left (add)

Tabelle 9 Berechnung

Durch den getrennten Schiebevorgang von HI und LO werden 3 statt 2 Takte pro Bitberechnung benötigt.
Zuerst wird HI nach links geschoben damit die Berechnung an der richtigen Position stattfinden kann. Danach wird, je nach Flagstatus, der Subtrahiervorgang oder Sddiervorgnag der ALU ausgelöst um 1 Takt später geschrieben werden zu können. Zu dem Zeitpunkt ist dann auch q0 bekannt und das LO Register kann auch nach links geschoben werden.

	LUT2								LUT1								
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
	#carry	M	S3	S2	S1	S0	S_R1	S_R0	S_L1	S_L0	SEL0	SEL1	#write_en	cpu_halt	count_up	#hi_to_alu	
198	1	0	0	0	0	0	1	1	0	0	1	1	1	1	1	1	divisor und dividend gleich, Ergebnis fertig (F = A)
199	1	1	0	0	0	0	1	1	0	0	1	1	1	1	1	1	nicht gleich -> 2K (Schritt1 A invertieren)
200	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	1	divisor und dividend gleich, Ergebnis fertig (F = A)
201	0	0	0	0	0	0	1	1	0	0	1	1	1	0	0	1	nicht gleich -> 2K (Schritt2 A PLUS 1)
																	SEL0 and SEL1 and #hi_to_alu in Hardware für lo_to_alu

Tabelle 10 Nachbetrachtung

Einzig bei der Division mit Vorzeichen wird eine Nachbetrachtung des Ergebnisses benötigt. Denn der Quotient wird von der Rechenvorschrift immer als positive Zahl berechnet. Aber wenn Divisor und Dividend unterschiedliche Vorzeichen besitzen, dann muss der Quotient negiert werden. Dies üben die letzten zwei Schritte aus. Diese bilden das 2er Komplement des Quotienten wenn nötig.
Schritt 1 invertiert den Quotienten und Schritt 2 addiert noch eine 1 drauf.

5. Inbetriebnahme

Zur Inbetriebnahme wurde die Karte zunächst auf Lötfehler und Fertigungsfehler geprüft. Dazu existiert der Spaceage 2 Testadapter „das Mäuseklavier“ mit dem sämtliche Signale der beiden 96 poligen Verbinder gesetzt und gelesen werden können. Dazu existiert ein extra Testprotokoll, weil es den Rahmen der Dokumentation sprengen würde.

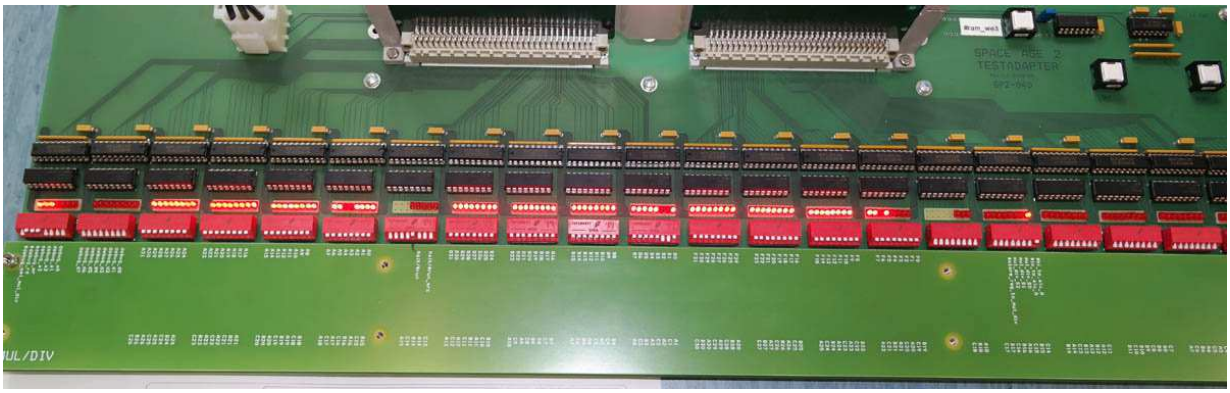


Abbildung 50

Anschließend wurden einfache Berechnungen auf der Karte ausgeführt um grob die Funktionsfähigkeit zu überprüfen vor dem Einbau.

Für die CPU wurde ein kleines Testprogramm geschrieben welches auf der MulDiv Karte rechnet und die Ergebnisse mit vorgefertigten Ergebnissen zum Vergleicht. Dieses Programm fand auch prompt Fehler.

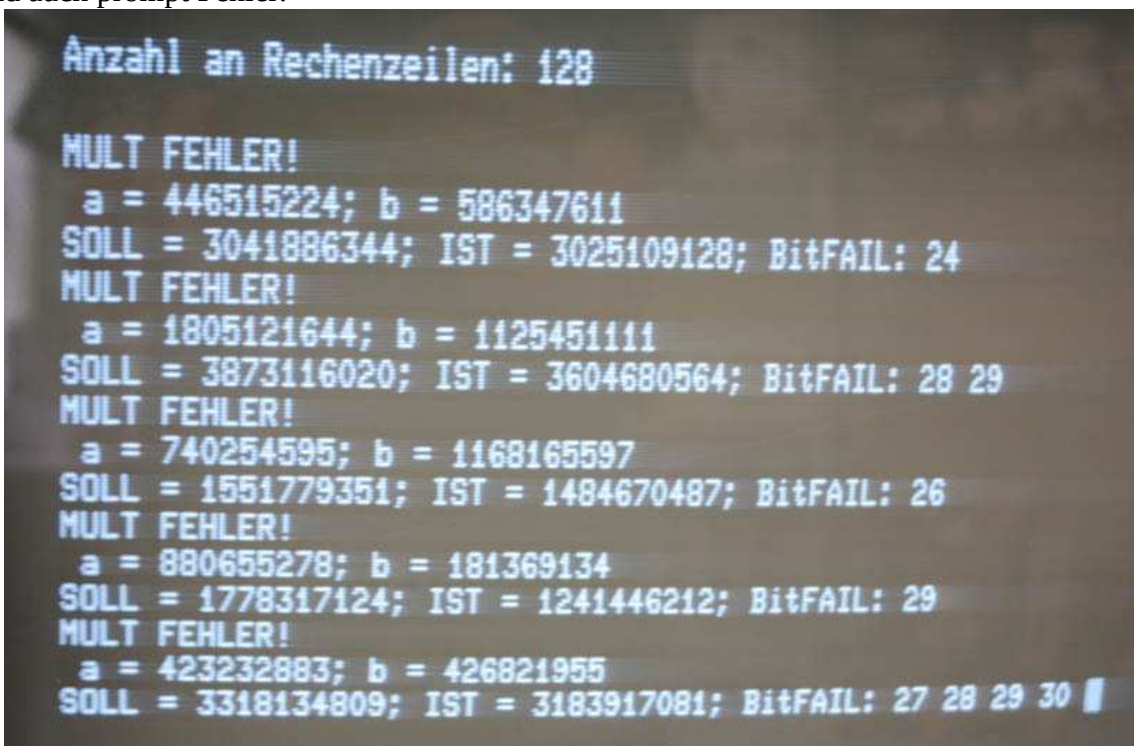


Abbildung 51

Auffällig ist, dass die Fehler erst ab Bit 24 auftreten. Mit dem Testadapter ließen sich die Fehler auch reproduzieren und beim Durchgehen der Takt kam es zu Oszillationen. Diese

waren an einer gedimmten Anzeige LED zu erkennen. Die Oszillation konnte zu einem IC mit verbogenem Masse Pin zurückverfolgt werden.

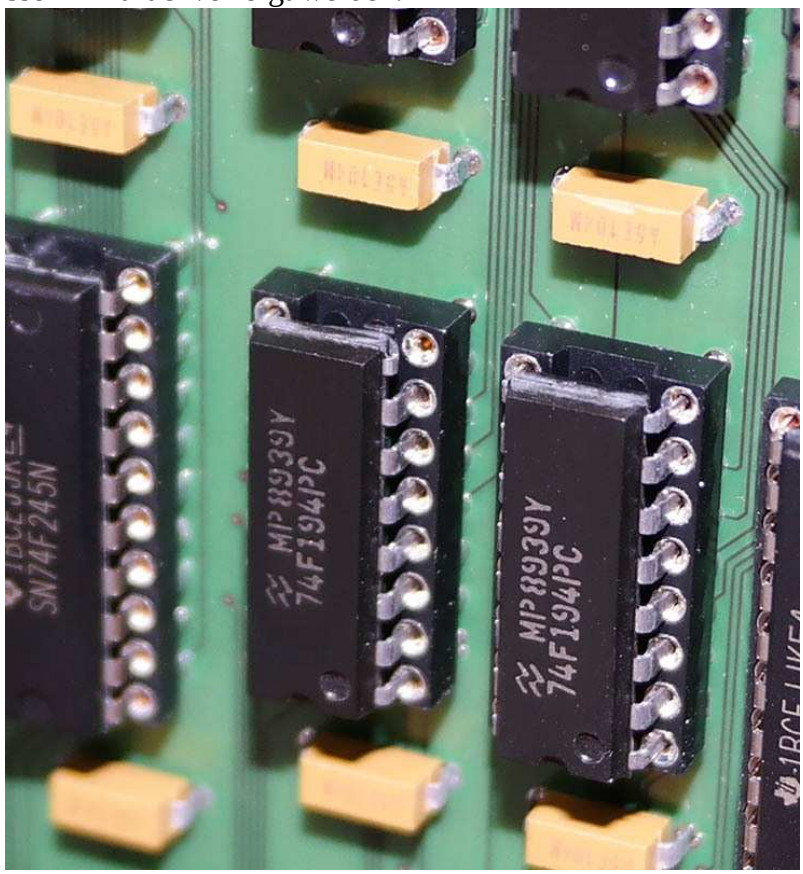


Abbildung 52

Nachdem richtigen Einbau des ICs lief das Testprogramm nun durch.

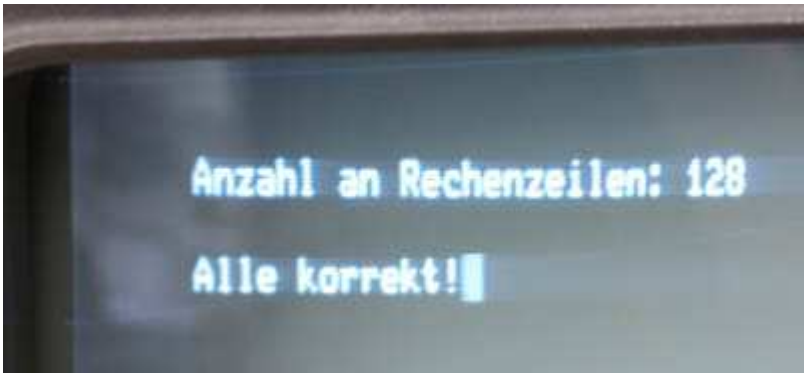


Abbildung 53

Alle Programme auf dem SpaceAge 2 liefen nun durch bis auf das Taschenrechnerprogramm. Dieses rechnet intern mit doubles und 5*5 ergab nicht 25, sondern 25,00001. Der Nachkommafehler von double kommt eigentlich viel später und wird durch ein printf gefiltert. Da sich dieser Fehler auch nicht mit weiteren Durchläufen des Testprogramms mit anderen zufällig erzeugten Zahlen finden ließ, musste der Debugger her. Der Debugger wurde so programmiert, dass er live die Operanden und Ergebnisse der MulDiv Karte auslesen konnte.

Dazu hier ein Video: https://www.youtube.com/watch?v=68j4Y1E-bhk

Der Debugger erfasste beispielhaft folgenden Fehler:

```
------------------------MulDiv-------------------------------
MULTU:
2279119104 *  390561520
10000111|11011000|10011001|00000000 * 00010111|01000111|01111110|11110000

HI_ist:   207250990; LO_ist:  1700622336
HI_soll:  207250989; LO_soll: 1700622336

HI_ist:  00001100|01011010|01100110|00101110; LO_ist:  01100101|01011101|01110000|00000000
HI_diff: 00000000|00000000|00000000|00000011; LO_diff: 00000000|00000000|00000000|00000000
HI_soll: 00001100|01011010|01100110|00101101; LO_soll: 01100101|01011101|01110000|00000000
```

Abbildung 54

Alle Fehlerbits waren immer zusammenhängend und auch immer am LSB Ende des HI Registers. Dies bedeutet, dass zum Anfang der Berechnung etwas aus dem Ruder lief.
Schlussendlich war es ein Synchronisationsproblem beim HI Register löschen, das Carry Bit war noch nicht gelöscht. Ein Fehler, der durch eine Änderung im Mikrocode behebbar ist.