Befehlsübersicht ADSP-BF 561

Inhalt

Die folgende Befehlsübersicht gibt einen Überblick über eine kleine Auswahl wichtiger Assemblerbefehle des DSPs ADSP-BF561. Auf eine ausführliche Diskussion der Einschränkungen und Besonderheiten der einzelnen Befehle wird zugunsten einer besseren Übersichtlichkeit und leichteren Lesbarkeit verzichtet. Mehr Details zu den Befehlen erklärt das Handbuch „ADSP-BF53x-BF56x Blackfin® Processor Programming Reference“.

1 Ladebefehle (Load Instructions)

Unmittelbare Ladebefehle (load immediate) laden Konstanten direkt aus dem Programmcode in ein Register des DSP-Kerns. Im Gegensatz dazu laden indirekte Ladebefehle Daten aus dem Datenspeicher des DSPs in ein Register des DSP-Kerns. Die Adresse der Speicherstelle wird dabei meist in einem Zeigerregister gespeichert und in der Form [Preg] als indirekte Adresse verwendet. Die meisten indirekten Ladebefehle erlauben die folgenden Optionen:

- post-increment, z.B. r5 = [p0 ++]; p0 wird nachträglich erhöht.
- post-decrement, z.B. r2 = [p2 --]; p2 wird nachträglich um 4 (wg. des 4-Byte-Transfers, s.u.) reduziert.
- Offsetadressierung, z.B. r6 = [p2 + 12]; Hier wird der Inhalt der Speicherstelle, die 12 Bytes nach der in p2 gespeicherten Adresse liegt, nach r6 geladen. Der Inhalt von p2 bleibt dabei unverändert.

Die Größe der Speicherstelle, die durch die indirekte Adresse [Preg] angesprochen wird, wird in den Assemblerbefehlen folgendermaßen beschrieben:

- [Preg] steht für eine 32-Bit Speicherstelle, also 4 Bytes.
- W[Preg] steht für eine 16-Bit Speicherstelle, also 2 Bytes.
- B[Preg] steht für eine 8-Bit Speicherstelle, also 1 Byte.

Die Erhöhung / Reduktion eines post-/pre-increments/decrements richtet sich nach der Größe der Speicherstelle, die angesprochen wird:

- z.B. r5 = [p0 ++]; der Pointer p0 wird um 4 erhöht (4-Byte-Wort Transfer).
- z.B. r2 = W[p2 --]; der Pointer p2 wird um 2 reduziert (2-Byte-Wort Transfer).
- z.B. r0 = B[p1 ++]; der Pointer p1 wird um 1 erhöht (1-Byte-Wort Transfer).

Wenn die Daten, die durch einen Ladebefehl in ein Register kopiert werden, nicht die gesamte Breite des Registers ausfüllen (z.B. Laden eines 16-Bit Datenwortes in ein 32-Bit Register), dann werden die Daten in der Regel (wenn nicht durch den Befehl anders festgelegt) in die unteren Bits des Registers gespeichert. Wie dann mit den oberen Bits zu verfahren ist, muss durch zusätzliche Optionen festgelegt werden:

- (Z): Zero-extended: Hier werden die oberen Bits mit Nullen aufgefüllt.
- (X): Sign-extended: Hier werden die oberen Bits mit Vorzeichenbits aufgefüllt. Das bedeutet, dass die oberen Bits mit Nullen aufgefüllt werden, falls das MSB der kopierten Daten eine Null war (positive Zahl). Falls das MSB der kopierten Daten eine Eins war (negative Zahl), werden die oberen Bits mit Einsen aufgefüllt. Dadurch bleibt das Vorzeichen der kopierten Zahl erhalten, auch dann wenn das ganze Register ausgelesen wird.

Die folgenden Tabellen fassen die wichtigsten Ladebefehle kurz zusammen:

Load Immediate	**register = constant**	
Lädt eine Konstante in ein Register		
Beispiele:	r7 = 63 (z) ;	Lädt die Konstante 63 (Dezimalzahl) in das 32-bit Datenregister R7, die dafür nicht benötigten höherwertigen Bits werden mit Nullen aufgefüllt (zero-extended).
	r0 = -344 (x) ;	Lädt -344 in das Datenregister r0, (x) steht für sign-extended, d.h. die nicht benötigten Bits werden mit dem Vorzeichen der Kon-

		stante, hier also mit Einsen aufgefüllt.
	p1 = 0x1234 (z) ;	Lädt die Hexadezimalzahl 0x1234 in das Zeigerregister p1, nicht benötige Bits werden mit Nullen aufgefüllt.
	l3.h = 0xbcde ;	Lädt die Hexadezimalzahl 0xbcde in die höherwertigen 16 Bits des Adressregister l3. Da dabei alle 16 Bits von l3.h benötigt werden, muss kein Extension-Mode spezifiziert werden.

Load Pointer Register	**Preg_dst = [Preg_src]**	
Lädt den Inhalt der Speicherstelle, auf die das Quell-Zeigerregister Preg_src zeigt in das Ziel-Zeigerregister Preg_dst. Achtung: Preg_src muss als Adresse ein Vielfaches von 4 enthalten.		
Beispiele:	p3 = [p2] ;	Lädt den Inhalt der Speicherstelle, auf die p2 zeigt, nach p3.
	p5 = [p0 ++] ;	Lädt den Inhalt der Speicherstelle, auf die p0 zeigt, nach p5. p0 wird danach um 4 inkrementiert.
	p2 = [sp --] ;	Lädt den Inhalt der Speicherstelle, auf die der Stackpointer sp zeigt nach p2, sp wird danach um 4 dekrementiert.
	p3 = [p2 + 8] ;	Lädt den 32-bit Inhalt der Speicherstelle, die 8 Byte nach der in p2 gespeicherten Adresse liegt, nach p3.

Load Data Register	**Dreg = [Preg]**	
Lädt den Inhalt der Speicherstelle, auf die das Zeigerregister Preg zeigt in das Datenregister Dreg. Achtung: Preg muss als Adresse ein Vielfaches von 4 enthalten.		
Beispiele:	r3 = [p0] ;	Lädt den Inhalt der Speicherstelle, auf die p0 zeigt, nach r3.
	r7 = [p1 ++] ;	Lädt den Inhalt der Speicherstelle, auf die p1 zeigt, nach r7. p1 wird danach um 4 inkrementiert.
	r2 = [sp --] ;	Lädt den Inhalt der Speicherstelle, auf die der Stackpointer sp zeigt nach r2, sp wird danach um 4 dekrementiert.
	r6 = [p2 + 12] ;	Lädt den 32-bit Inhalt der Speicherstelle, die 12 Byte nach der in p2 gespeicherten Adresse liegt, nach r6.

Load Half Word	**Dreg = W[Preg](Z)**	**zero-extended**
	Dreg = W[Preg](X)	**sign-extended**
Lädt 16 Bits von der Speicherstelle, auf die das Zeigerregister Preg zeigt, in die untere (niederwertige) Hälfte des Datenregisters Dreg. Die obere Hälfte von Dreg wird entweder mit Nullen (Z) oder mit Vorzeichenbits (X) aufgefüllt. Achtung: Preg muss als Adresse ein Vielfaches von 2 enthalten.		
Beispiele:	r3 = w [p0] (z) ;	Lädt 16 Bits von der Adresse, auf die p0 zeigt, in die untere Hälfte von r3 (also in r3.l). Die obere Hälfte (r3.h) wird mit Nullen aufgefüllt
	r3 = w [p0] (x) ;	Hier wird r3.h mit Vorzeichenbits aufgefüllt.

Load Byte	**Dreg = B[Preg](Z)**	**zero-extended**
	Dreg = B[Preg](X)	**sign-extended**
Lädt ein Byte (8 Bits) von der Speicherstelle, auf die das Zeigerregister Preg zeigt, in die unteren 8 Bits des Datenregisters Dreg. Die oberen 24 Bits von Dreg werden entweder mit Nullen (Z) oder mit Vorzeichenbits (X) aufgefüllt.		
Beispiele:	r3 = b [p0] (z) ;	Lädt 8 Bits von der Adresse, auf die p0 zeigt, in die unteren 8 Bits von r3. Die oberen 24 Bits werden mit Nullen aufgefüllt.

r3 = b [p0] (x) ;	Hier werden die oberen 24 Bits von r3 mit Vorzeichenbits aufgefüllt.

Load High Data Register Half	**Dreg.h = W[Preg]**
Lädt 16 Bits von der Speicherstelle, auf die das Zeigerregister Preg zeigt, in die obere Hälfte des Datenregisters Dreg, also nach Dreg.h. Dreg.l wird dabei nicht verändert. Achtung: Preg muss als Adresse ein Vielfaches von 2 enthalten.	
Beispiel: r2.h = w [p4] ;	Lädt 16 Bits von der Adresse, auf die p4 zeigt, nach r2.h.

Load Low Data Register Half	**Dreg.l = W[Preg]**
Lädt 16 Bits von der Speicherstelle, auf die das Zeigerregister Preg zeigt, in die untere Hälfte des Datenregisters Dreg, also nach Dreg.l. Dreg.h wird dabei nicht verändert. Achtung: Preg muss als Adresse ein Vielfaches von 2 enthalten.	
Beispiel: r2.l = w [p4] ;	Lädt 16 Bits von der Adresse, auf die p4 zeigt, nach r2.l.

2 Speicherbefehle (Store Instructions)

Speicherbefehle schreiben den Inhalt eines Registers in den Speicher des DSPs. Die Zieladresse wird dabei meist in einem Zeigerregister gespeichert und in der Form [Preg] als indirekte Adresse vom Assemblerbefehl verwendet. Auch bei den Speicherbefehlen lassen sich reduzierte Speichergrößen, d.h. 16-Bit-Worte und Bytes, durch ein prefix W[ ] bzw. B[ ] ansprechen. Ebenso gibt es natürlich die post- und pre-increment/decrement-Syntax, z.B. [p1++] bzw. [--p2], und auch hier richtet sich die Erhöhung/Reduktion des Pointers nach der Größe des adressierten Speicherbereichs.

Store Pointer Register	**[Preg_dst] = Preg_src**	
Speichert den Inhalt des Quell-Zeigerregisters Preg_src in die 32-Bit Speicherstelle, auf die das Ziel-Zeigerregister Preg_dst zeigt. Achtung: Preg_dst muss als Adresse ein Vielfaches von 4 enthalten.		
Beispiele:	[p2] = p3 ;	Speichert den Inhalt von p3 in die Speicherstelle, auf die p2 zeigt.

Store Data Register	**[Preg] = Dreg**	
Speichert den Inhalt des Datenregisters Dreg in die 32-Bit Speicherstelle, auf die das Zeigerregister Preg zeigt. Achtung: Preg muss als Adresse ein Vielfaches von 4 enthalten.		
Beispiel:	[p2] = r3 ;	Speichert den Inhalt von r3 in die Speicherstelle auf die p2 zeigt.

Store High Data Register Half	**W[Preg] = Dreg.h**	
Speichert den Inhalt der oberen 16 Bit des Datenregisters Dreg (also Dreg.h) in die 16-Bit Speicherstelle, auf die das Zeigerregister Preg zeigt. Achtung: Preg muss als Adresse ein Vielfaches von 2 enthalten.		
Beispiele:	w [p4] = r2.h ;	Speichert den Inhalt von r2.h in die Speicherstelle auf die p4 zeigt.

Store Low Data Register Half	**W[Preg] = Dreg.l oder W[Preg] = Dreg**	
Speichert den Inhalt der unteren 16 Bit des Datenregisters Dreg (also Dreg.l) in die 16-Bit Speicherstelle, auf die das Zeigerregister Preg zeigt. Achtung: Preg muss als Adresse ein Vielfaches von 2 enthalten.		
Beispiel:	w [p0] = r3.l ;	Speichert den Inhalt von r3.l in die Speicherstelle, auf die p0 zeigt.
	w [p0] = r3 ;	Speichert den Inhalt von r3.l in die Speicherstelle, auf die p0 zeigt. Der Zusatz .l kann hier also auch weggelassen werden.

Store Byte	**B[Preg] = Dreg**	
Speichert die unteren 8 Bit des Datenregisters Dreg in die 8-Bit Speicherstelle, auf die das Zeigerregister Preg zeigt.		
Beispiel:	b [p0] = r3 ;	Speichert die unteren 8 Bit von r3 in die Speicherstelle, auf die p0 zeigt

3 Kopierbefehle (Move Instructions)

Kopierbefehle kopieren den Inhalt eines Quellregisters in ein Zielregister. Das Quellregister bleibt dabei unverändert. Der Begriff „Copy Instruction“ wäre deshalb wohl treffender. Im Handbuch werden diese Befehle aber als „Move Instructions“ bezeichnet.

Move Register	**dst_reg = src_reg**	
Kopiert den Inhalt des Quellregisters src_reg in das Zielregister dst_reg. Kopierbefehle von kleineren Registern in größere Register arbeiten mit sign-extension. Kopierbefehle von den Akkumulatoren in die Datenregister unterstützen Saturation. Der Inhalt des Quellregisters bleibt unverändert.		
Beispiele:	r3 = r0 ;	Kopiert den Inhalt von r0 nach r3.
	r2 = a0 ;	Kopiert den Inhalt von a0 nach r2 (mit Sättigung), a0 arbeitet nur mit den geraden Datenregistern zusammen.
	r7 = a1 ;	Kopiert den Inhalt von a1 nach r7 (mit Sättigung), a1 arbeitet nur mit den ungeraden Datenregistern zusammen.

Move Half to Full Word	**dst_reg = src_reg.l (Z)** **dst_reg = src_reg.l (X)**	**zero-extended** **sign-extended**
Kopiert den Inhalt der unteren 16 Bits des Quellregisters src_reg (also den Inhalt von src_reg.l) in die unteren 16 Bits des Zielregisters dst_reg. Die oberen 16 Bits werden entweder mit Nullen aufgefüllt (zero-extended) oder mit Vorzeichenbits (sign-extended).		
Beispiele:	r4 = r0.l (z)	Der Inhalt von r0.l wird in die unteren 16 Bits von r4 kopiert, die oberen 16 Bits von r4 werden mit Nullen aufgefüllt.
	r4 = r0.l (x)	Der Inhalt von r0.l wird in die unteren 16 Bits von r4 kopiert, die oberen 16 Bits von r4 werden mit Vorzeichenbits aufgefüllt.

4 Schiebebefehle (Shift Operations)

Schiebebefehle schieben den Inhalt eines Registers um eine im Befehlsopcode festgelegte Anzahl von Bits nach links oder rechts. Wenn bei dieser Schiebeoperation das ursprüngliche Vorzeichen des Registerinhalts erhalten wird, spricht man von einem arithmetischen Shift. Im Gegensatz dazu wird bei einem logischen Shift keine Rücksicht auf das Vorzeichen des Registerinhalts genommen. Solange es nicht zu Überläufen kommt, entspricht ein Linksshift um *n* Bits einer Multiplikation mit 2^n, ein Rechtsshift um *n* Bits einer Division mit 2^n, wobei der Divisionsrest ignoriert wird.

Bei arithmetischen Rechtsshifts werden die auf der linken Seite frei werdenden Bits durch Vorzeichenbits aufgefüllt, also durch Nullen, falls der ursprüngliche Registerinhalt eine positive Zahl war, oder durch Einsen, falls der ursprüngliche Registerinhalt eine negative Zahl war. Bei logischen Rechtsshifts werden die links frei werdenden Bits immer mit Nullen aufgefüllt. Aufgrund dieses Unterschieds werden arithmetische Rechtsshifts durch „>>>“ gekennzeichnet und logische Rechtsshifts durch „>>“.

Bei Linksshifts besteht kein Unterschied zwischen arithmetischen und logischen Shifts, solange es durch das Schieben nicht zu einem Überlauf kommt. Deshalb gibt es keinen expliziten arithmetischen Linksshiftoperator („<<<“ gibt es also nicht). Wenn es allerdings durch den Linksshift zu einem Überlauf kommt, dann gibt es sehr wohl einen Unterschied zwischen arithmetischem und logischem Shift. Beim logischen Shift wird dann das Vorzeichenbit aus dem Register herausgeschoben, so dass sich das Vorzeichen des Registerinhalts ändern kann. Beim arithmetischen Shift wird bei einem Überlauf das Vorzeichenbit erhalten. Dazu wird bei ursprünglich positivem Registerinhalt der Registerinhalt nach der Schiebeoperation auf die betragsmäßig größte positive Zahl gesetzt. Bei ursprünglich negativem Registerinhalt wird der Registerinhalt nach der Schiebeoperation auf die betragsmäßig größte negative Zahl gesetzt.

Arithmetic Shift
dst_reg >>>= shift_magnitude dst_reg = src_reg >>> shift_magnitude (opt_sat) dst_reg = src_reg << shift_magnitude (S) accumulator = accumulator >>> shift_magnitude
Arithmetische Schiebeoperationen (arithmetic shifts) schieben den Inhalt eines Registers um eine gewisse Anzahl von Bits nach rechts oder links. Dabei wird das Vorzeichen des Registerinhalts beibehalten. Bei Rechts-Shifts bedeutet das, dass die rechts freigewordenen Bits durch Vorzeichenbits aufgefüllt werden. Bei Links-Shifts tritt eine Sättigung ein, wenn zu weit geschoben wird.
Beispiele: r0 >>>= 19 ; /* 16-bit instruction length arithmetic right shift */ r3.l = r0.h >>> 7 ; /* arithmetic right shift, half-word */ r3.l = r0.h >>> 7(s) ; /* arithmetic right shift, half-word, saturated */ r4 = r2 >>> 20 ; /* arithmetic right shift, word */ A0 = A0 >>> 1 ; /* arithmetic right shift, Accumulator */ r0 >>>= r2 ; /* 16-bit instruction length arithmetic right shift */ r3.l = r0.h << 12 (S) ; /* arithmetic left shift */ r5 = r2 << 24(S) ; /* arithmetic left shift */

Logical Shift
dest_reg >>= shift_magnitude dest_reg <<= shift_magnitude dest_reg = src_reg >> shift_magnitude dest_reg = src_reg << shift_magnitude
Logische Schiebeoperationen (logical shifts) schieben den Inhalt eines Registers um eine gewisse Anzahl von Bits nach rechts oder links. Dabei werden Bits, die aus dem Register herausgeschoben werden verworfen, frei werdende Bits werden durch Nullen aufgefüllt. Durch logische Schiebeoperationen kann sich also das Vorzeichen des Registerinhalts ändern.
Beispiele: r3 >>= 17 ; /* data right shift */ r3 <<= 17 ; /* data left shift */ r3.l = r0.l >> 4 ; /* data right shift, half-word register */ r3.h = r0.l << 12 ; /* data left shift, half-word register */ r3 = r6 >> 4 ; /* right shift, 32-bit word */ r3 = r6 << 4 ; /* left shift, 32-bit word */ a0 = a0 >> 7 ; /* Accumulator right shift */ a0 = a0 << 7 ; /* Accumulator left shift */ r3 >>= r0 ; /* data right shift */ r3 <<= r1 ; /* data left shift */

5 Multiplizier- und MAC-Befehle

Die Multiplizier- und MAC-Befehle (Multiply and Accumulate) sind die wichtigsten arithmetischen Befehle des DSPs. Die MAC Befehle führen zunächst eine Multiplikation durch und addieren dann das Produkt zum Inhalt eines Akkumulators oder subtrahieren das Produkt vom Inhalt eines Akkumulators. Die folgenden Tabellen zeigen die wichtigsten vier Versionen dieser Befehle. In der ersten Version wird das Ergebnis in einem Akkumulator gespeichert, in der zweiten Version wird das Ergebnis in einem Akkumulator gespeichert und zusätzlich in eine Datenregisterhälfte extrahiert. In der dritten Version wird das Ergebnis sowohl in einem Akkumulator gespeichert als auch in ein (ganzes) 32-Bit Datenregister kopiert. Die vierte Version speichert das Ergebnis einer Multiplikation direkt in einem Datenregister, ohne dass dabei einer der Akkumulatoren verändert wird.

Jeder DSP-Kern verfügt über zwei Akkumulatoren und zwei MAC-Einheiten. Welche der beiden MAC-Einheiten benutzt wird, hängt von den Details des jeweiligen Befehls ab. Dabei gelten die folgenden Regeln:

- Wird der Akkumulator a0 als Ziel gewählt, werden die Operationen von der MAC0-Einheit ausgeführt. Als Zielhalbregister wird hier stets die untere Hälfte (lo) gewählt. Als Zieldatenregister (32-Bit) werden stets geradzahlige Register gewählt.
- Wird der Akkumulator a1 als Ziel gewählt, werden die Operationen von der MAC1-Einheit ausgeführt. Als Zielhalbregister wird hier stets die obere Hälfte (hi) gewählt. Als Zieldatenregister (32-Bit) werden stets ungeradzahlige Register gewählt.

Welches Zahlenformat bei der Multiplikation verwendet werden soll, kann durch einen nachgestellten optionalen Modus festgelegt werden. Dieser opt_mode wird in Klammern eingeschlossen, die wichtigsten (opt_modes) sind:

- Default: Signed fraction – Multiplikation mit „left-shift correction“: 1.15 * 1.15 = 1.31.
  Wenn das Ergebnis in ein 16-Bit Zielhalbregister (rn.h oder rn.l) gespeichert werden soll, wird das Ergebnis bei Bit 16 gerundet und dann auf das 1.15 Format gesättigt. D.h. es werden die höherwertigen 16 Bit ins Zielhalbregister extrahiert, bei einer Bereichsüberschreitung werden die jeweiligen Extremwerte ins Zielhalbregister geschrieben.
- (FU): Unsigned fraction – Multiplikation ohne „left-shift correction“: 0.16 * 0.16 = 0.32
  Wenn das Ergebnis in ein 16-Bit Zielhalbregister (rn.h oder rn.l) gespeichert werden soll, wird das Ergebnis bei Bit 16 gerundet und dann auf das 0.16 Format gesättigt. D.h. Es werden die höherwertigen 16 Bit ins Zielhalbregister extrahiert, bei einer Bereichsüberschreitung wird der Maximalwert ins Zielhalbregister geschrieben.
- (IS): Signed integer – Multiplikation ohne „left-shift correction“: 16.0 * 16.0 = 32.0
  Wenn das Ergebnis in ein 16-Bit Zielhalbregister (rn.h oder rn.l) gespeichert werden soll, werden die niederwertigen 16 Bit ins Zielhalbregister extrahiert, bei einer Bereichsüberschreitung werden die jeweiligen Extremwerte ins Zielhalbregister geschrieben.
- (IU): Unsigned integer – Multiplikation ohne „left-shift correction“: 16.0 * 16.0 = 32.0
  Wenn das Ergebnis in ein 16-Bit Zielhalbregister (rn.h oder rn.l) gespeichert werden soll, werden die niederwertigen 16 Bit ins Zielhalbregister extrahiert, bei einer Bereichsüberschreitung wird der Maximalwert ins Zielhalbregister geschrieben.

1) Multiply and Multiply-Accumulate to Accumulator	**accumulator = src_reg_0 * src_reg_1 (opt_mode)** **accumulator += src_reg_0 * src_reg_1 (opt_mode)** **accumulator –= src_reg_0 * src_reg_1 (opt_mode)**
Multipliziert den Inhalt des Quellregisters src_reg_0 mit dem Inhalt des Quellregisters src_reg_1. Das Produkt wird im Akkumulator gespeichert, zum Akkumulator addiert oder vom Akkumulator abgezogen (mit Sättigung).	
Beispiele: a0=r3.h*r2.h ;	Multipliziert in der MAC0-Einheit den signed-1.15-Operanden in r3.h mit dem signed-1.15-Operanden in r2.h (Default Modus). Das 1.31 Ergebnis wird durch Voranstellen von Vorzeichenbits auf 9.31 erweitert und im Akkumulator a0 gespeichert.
a1+=r6.h*r4.l (fu) ;	Multipliziert in der MAC1-Einheit den unsigned-0.16-Operanden in r6.h mit dem unsigned-0.16-Operanden in r4.l. Das 0.32-Ergebnis wird durch Voranstellen von Nullen auf 8.32 erweitert und zum Inhalt des Akkumulators a1 addiert. Tritt dabei ein Überlauf auf, kommt es zur Sättigung von a1.

2) Multiply and Multiply-Accumulate to Half-Register	**dst_reg_half = (accumulator = src_reg_0 * src_reg_1) (opt_mode)** **dst_reg_half = (accumulator += src_reg_0 * src_reg_1) (opt_mode)** **dst_reg_half = (accumulator –= src_reg_0 * src_reg_1) (opt_mode)**
Multipliziert den Inhalt des Quellregisters src_reg_0 mit dem Inhalt des Quellregisters src_reg_1. Das Produkt wird im Akkumulator gespeichert, zum Akkumulator addiert oder vom Akkumulator abgezogen (mit Sättigung). Anschließend werden 16 Bit des Ergebnisses in das Zielregister (Registerhälfte, hi oder lo) gespeichert.	
Beispiele: r3.l=(a0=r3.h*r2.h) ;	Multipliziert in der MAC0-Einheit den signed-1.15-Operanden in r3.h mit dem signed-1.15-Operanden in r2.h. Das 1.31 Ergebnis wird durch Voranstellen von Vorzeichenbits auf 9.31 erweitert und im Akkumulator a0 gespeichert. Die 16-Bit von a0.h werden gerundet und nach r3.l geschrieben.
r3.h=(a1+=r6.h*r4.l) (fu) ;	Multipliziert in der MAC1-Einheit den unsigned-0.16-Operanden in r6.h mit dem unsigned-0.16-Operanden in r4.l. Das 0.32-Ergebnis wird durch Voranstellen von Nullen auf 8.32 erweitert und zum Inhalt des Akkumulators a1 addiert. Tritt dabei ein Überlauf auf, kommt es zur Sättigung von a1. Das Ergebnis wird dann anschließend nach r3.h extrahiert. Dazu wird es bei Bit 16 gerundet und auf das Format 0.16 gesättigt.

3) Multiply and Multiply-Accumulate to Data Register	**dst_reg = (accumulator = src_reg_0 * src_reg_1) (opt_mode)** **dst_reg = (accumulator += src_reg_0 * src_reg_1) (opt_mode)** **dst_reg = (accumulator –= src_reg_0 * src_reg_1) (opt_mode)**
Multipliziert den Inhalt des Quellregisters src_reg_0 mit dem Inhalt des Quellregisters src_reg_1. Das Produkt wird im Akkumulator gespeichert, zum Akkumulator addiert oder vom Akkumulator abgezogen (mit Sättigung). Anschließend werden 32 Bit des Ergebnisses in das Zielregister (ganzes 32-Bit Datenregister) gespeichert.	
Beispiele: r4=(a0=r3.h*r2.h) ;	Multipliziert in der MAC0-Einheit den signed-1.15-Operanden in r3.h mit dem signed-1.15-Operanden in r2.h. Das 1.31 Ergebnis wird durch Voranstellen von Vorzeichenbits auf 9.31 erweitert und im Akkumulator a0 gespeichert. Die 32-Bit von a0.w werden dann nach r4 geschrieben.
r3 =(a1+=r6.h*r4.l) (fu) ;	Multipliziert in der MAC1-Einheit den unsigned-0.16-Operanden in r6.h mit dem unsigned-0.16-Operanden in r4.l. Das 0.32-Ergebnis wird durch Voranstellen von Nullen auf 8.32 erweitert und zum Inhalt des Akkumulators a1 addiert. Tritt dabei ein Überlauf auf, kommt es zur Sättigung von a1. Das Ergebnis wird dann anschließend nach r3 extrahiert. Dazu wird auf das Format 0.32 gesättigt.

4) Multiply 16-Bit Operands	**dst_reg = src_reg_0 * src_reg_1**
Multipliziert den Inhalt des Quellregisters src_reg_0 mit dem Inhalt des Quellregisters src_reg_1 und speichert das Ergebnis direkt im Zielregister dst_reg mit Sättigung. Die Akkumulatoren werden dabei nicht verändert.	
Beispiele: r3.l=r3.h*r2.h ;	Multipliziert in der MAC0-Einheit den signed-1.15-Operanden in r3.h mit dem signed-1.15-Operanden in r2.h. Die niederwertigen 16 Bit des 1.31-Ergebnisses werden durch Runden entfernt (RND_MOD im ASTAT Register entscheidet, welche Art der Rundung durchgeführt wird). Die höherwertigen 16 Bit werden als 1.15 Ergebnis in r3.l gespeichert.
r3.h=r6.h*r4.l (fu) ;	Multipliziert in der MAC1-Einheit den unsigned-0.16-Operanden in r6.h mit dem unsigned-0.16-Operanden in r4.l. Die niederwertigen 16 Bit des 0.32-Ergebnisses werden durch Runden entfernt. Die höherwertigen 16 Bit werden als 0.16 Ergebnis in r3.h gespeichert.
r6=r3.h*r4.h ;	Multipliziert in der MAC0-Einheit den signed-1.15-Operanden in r3.h mit dem signed-1.15-Operanden in r4.h. Das 1.31-Ergebnis wird in r6 gespeichert.

6 Zyklische Adressierung

Der DSP verfügt über eine Hardwareunterstützung für die Implementierung von Ringspeichern (cyclic buffers). Ringspeicher werden beispielsweise für die Programmierung von FIR-Filtern verwendet. Ringspeicher besitzen eine vorher festgelegte Größe und werden über Zeiger beschrieben oder ausgelesen. Wenn der Zeiger das Ende des Speichers erreicht und weiter erhöht werden soll, wird die Adresse, auf die er zeigt, um die Länge des Ringspeichers erniedrigt. Damit zeigt der Speicher wieder auf eine Adresse innerhalb des Ringspeichers.

Für die zyklische Adressierung werden die „Data Address Registers“ eingesetzt. Als Beispiel werden im Folgenden die 4 Register I0, L0, B0 und M0 verwendet. Das Basisregister B0 (base register) enthält die Anfangsadresse des Ringspeichers. Das Längenregister (length register) L0 enthält die Länge des Ringspeichers in Bytes. Das Modifikationsregister (modify register) M0 enthält den Wert, um den das I0-Register bei jedem Speicherzugriff geändert wird (in Bytes, positiver oder negativer Wert, Beispiel: R0=[I0++M0]). Das Index Register (index register) I0 enthält die aktuelle Adresse des Ringspeichers, also die Adresse auf die geschrieben wird bzw. aus der gelesen wird. Hinweis: Die Verwendung des Modifikationsregisters M0 ist für die Realisierung eines Ringspeichers nicht unbedingt erforderlich.

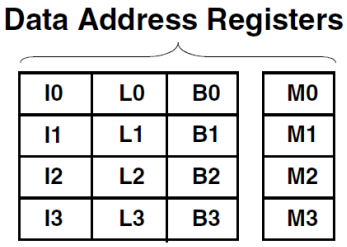

Abbildung 1: Data Address Registers.

Zur Adressierung des Ringspeichers wird das Indexregister I0 nach jedem Speicherzugriff verändert (meist wird es erhöht). Wenn die dadurch entstehende Adresse außerhalb des Ringspeichers liegt, wird I0 um den Inhalt von L0 reduziert. Damit zeigt I0 wieder auf eine Adresse innerhalb des Ringspeichers. Die folgende Abbildung zeigt ein Beispiel für die Adressierung eines Ringspeichers.

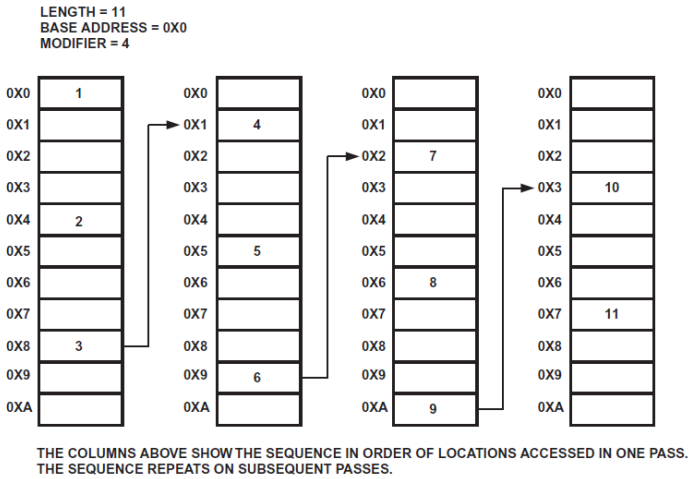

Abbildung 2: Beispiel für die Adressierung eines Ringspeichers.

7 Schleifenprogrammierung

Der DSP verfügt über eine Hardwareunterstützung zur Schleifenprogrammierung. Dadurch können Schleifen ohne aufwändige Software-Abfragen implementiert werden. Insgesamt stehen zwei Registersätze, mit denen jeweils eine Schleife programmiert werden kann, zur Verfügung: Satz 0 besteht aus den Registern LC0, LT0 und LB0; Satz 1 besteht aus den Registern LC1, LT1 und LB1. LC steht für Loop Counter, LT für Loop Top und LB für Loop Bottom. LC muss vor der Ausführung der Schleife auf die gewünschte Zahl der Durchläufe gesetzt werden. LT enthält die Adresse des ersten Befehls der Schleife, und LB enthält die Adresse des letzten Schleifenbefehls. Wenn die Adresse, auf die LB zeigt, ausgeführt wird, erfolgt ein Rücksprung zur Adresse, auf die LT zeigt, sofern LC größer gleich zwei ist. Zur Veranschaulichung soll folgendes Beispiel dienen: Anfangs ist LC auf 2 gesetzt. Wenn zum ersten Mal das Schleifenende LB erreicht wird, erfolgt ein Rücksprung zum Schleifenanfang LT. Außerdem wird LC um 1 reduziert, so dass LC jetzt den Wert 1 hat. Beim zweiten Erreichen des Schleifenendes erfolgt kein Rücksprung mehr zum Schleifenanfang, weil LC bereits kleiner als 2 ist. LC wird um 1 reduziert und hat dann den Wert 0. Zum Setzen aller drei Schleifenregister eignet sich der Befehl LSETUP. Das folgende Beispiel zeigt eine Schleife, die aus zwei Befehlen besteht und 32 Mal durchlaufen wird.

Beispiel:

```
P5 = 0x20 ;
LSETUP ( lp_start, lp_end ) LCO = P5 ;
lp_start: R5 = R0 + R1(ns) || R2 = [P2++] || R3 = [I1++] ;
lp_end: R5 = R5 + R2 ;
```

Bei einer Schleife mit nur einem Befehl ist lp_start und lp_end identisch.

8 Vektorbefehle (SIMD Instructions)

Bei Vektorbefehlen wird der Inhalt eines 32-Bit Datenregisters als mehrere individuelle Datenworte (z.B. zwei 16-Bit Worte) behandelt. Dadurch lassen sich mit einem Befehl beispielsweise zwei Multiplikationen und oder zwei Shift-Befehle gleichzeitig ausführen. Im Folgenden werden einige wenige wichtige Beispiele für Vektorbefehle exemplarisch besprochen.

Vector Arithmetic Shift

Dreg_dst = ASHIFT *Dreg_src* BY *Dreg_lo_shift* (V) ;

Der Inhalt des 32-Bit Datenregisters *Dreg_src* wird als zwei getrennte 16-Bit Operanden betrachtet, die jeweils um den Inhalt von *Dreg_lo_shift* Bits verschoben werden. Wenn der Inhalt von *Dreg_lo_shift* eine negative Zahl ist, handelt es sich um einen arithmetischen Rechtsshift. Ist der Inhalt positiv, handelt es sich um einen logischen Linksshift.

Beispiele:

1) R0 = ASHIFT R2 BY R3.L (V); (Inhalt von R3.L ist 0x0002):
   Betrachtet die Inhalte von R2.L und R2.H als 2 unabhängige 16-Bit Operanden. Diese werden jeweils um 2 Bit nach links (logischer Shift) geschoben. Die Ergebnisse werden in R0.L und R0.H gespeichert.
2) R0 = ASHIFT R2 BY R3.L (V); (Inhalt von R3.L ist 0xFFFE):
   Betrachtet die Inhalte von R2.L und R2.H als 2 unabhängige 16-Bit Operanden. Diese werden jeweils um 2 Bit nach rechts (arithmetischer Shift) geschoben. Die Ergebnisse werden in R0.L und R0.H gespeichert.

Dreg_dst = ASHIFT Dreg_src BY Dreg_lo_shift (V, S) ;

Gleiche Funktion wie oben, nur dass jetzt bei positivem Inhalt von *Dreg_lo_shift* ein arithmetischer Linksshift (mit Sättigung) durchgeführt wird.

Beispiel:

3) R0 = ASHIFT R2 BY R3.L (V, S); (Inhalt von R3.L ist 0x0002):
   Betrachtet die Inhalte von R2.L und R2.H als 2 unabhängige 16-Bit Operanden. Diese werden jeweils um 2 Bit nach links (arithmetischer Shift/Sättigung) geschoben. Die Ergebnisse werden in R0.L und R0.H gespeichert.

Vector Multiply and Multiply-Accumulate

Zwei gewöhnliche (skalare) Multiplizierbefehle können zu einem Vektormultiplizierbefehl kombiniert werden. Ähnlich können zwei gewöhnliche (skalare) MAC-Befehle zu einem Vektor-MAC-Befehl kombiniert werden. Die Vektoroperation führt dabei beide skalare Operationen gleichzeitig aus. Die beiden skalaren Operationen werden durch ein Komma getrennt und ein Semikolon abgeschlossen. Damit eine Kombination zweier skalarer Befehle möglich ist müssen einige Bedingungen erfüllt sein:

- Eine skalare Operation muss von MAC0 ausgeführt werden, die andere von MAC1
- Beide skalare Operationen müssen den gleichen opt_mode haben, z.B. (default) oder (IS)
- Wenn die Ergebnisse in 16-Bit Datenregistern gespeichert werden, müssen diese die obere und untere Hälfte des gleichen 32-Bit Datenregisters sein.

Beispiele:

1) a1=r2.l*r3.h, a0=r2.h*r3.h ;
   /* both multiply signed fractions into separate Accumulators */
2) a0=r1.l*r0.l, a1+=r1.h*r0.h ;
   /* multiplication of signed fractions, store in A0 and sum into A1. MAC order is arbitrary */
3) a1+=r3.h*r3.l, a0-=r3.h*r3.h ;
   /* multiplication of signed fractions, sum product into A1, subtract product from A0 */

9 Parallelbefehle

Der DSP ist in der Lage bis zu drei kompatible Befehle parallel zu verarbeiten. Dabei sind einige Einschränkungen zu beachten: Der erste Befehl muss dabei eine Befehlswortlänge von 32 Bit haben, der zweite und dritte Befehl müssen jeweils eine Befehlswortlänge von 16 Bit haben. Die Ausführungszeit aller drei Befehle ist identisch mit der Ausführungszeit des langsamsten Befehls. Syntaktisch werden die Einzelbefehle durch das ||-Zeichen getrennt.

```
32-bit ALU/MAC instruction || 16-bit instruction || 16-bit instruction ;
```

Beispiele:

1) A1=R2.L*R1.L, A0=R2.H*R1.H || R2.H=W[I2++] || [I3++]=R3 ; /* Multiply and accumulate to Accumulator while loading a data register and storing a data register using an Ireg pointer. */
2) A1+=R0.L*R2.H,A0+=R0.L*R2.L || R2.L=W[I2++] || R0=[I1--] ;
   R3.H=(A1+=R0.L*R1.H), R3.L=(A0+=R0.L*R1.L) || R0=[P0++] || R1=[I0] ;
   /* Multiply and accumulate while loading two data registers. One load uses an Ireg pointer. */

10 Aufruf von Assemblerfunktionen aus C

Während die Programmierung in Assembler deutlich effizienter ist, ermöglicht eine Hochsprache wie C eine übersichtlichere und weniger fehleranfällige Programmierung. Wenn weniger rechenintensive Programmteile in C programmiert werden, kann in der Regel die Software-Entwicklungszeit reduziert werden. Es empfiehlt sich aber, besonders rechenintensive Abschnitte in Assembler zu programmieren. Die Assembler-Routinen können dann direkt aus dem C-Programm aufgerufen werden. Im Prinzip wird der Aufruf einer C-Funktion *function(…)* vom Compiler lediglich in einen Sprungbefehl zu einer Programmadresse *_function* umgesetzt, unter der wiederum eine Assemblerroutine definiert sein kann. Für die Parameterübergabe gelten dabei die folgenden Regeln:

- Die ersten drei 32-Bit Wörter der Parameterliste werden über die Register R0 – R2 übergeben.
- Jeder Parameter, der mehr als 32 Bit benötigt, belegt mehr als eines dieser Register.
- Weitere Parameter werden über den Stack übergeben.
- Rückgabewerte, die höchstens 32 Bit benötigen, werden in R0 zurückgegeben.
- Rückgabewerte, die zwischen 32 und 64 Bit benötigen, werden in R0 und R1 zurückgegeben.
- Noch größere Rückgabewerte werden über den Stack zurückgegeben.

Beispiele:

Function Prototype	Parameters Passed as	Return Location
`int test(int a, int b, int c)`	a in R0, b in R1, c in R2	in R0
`char test(int a, char b, char c)`	a in R0, b in R1, c in R2	in R0
`int test(int a)`	a in R0	in R0
`int test(char a, char b, char c, char d, char e)`	a in R0, b in R1, c in R2, d in [FP+20], e in [FP+24]	in R0
`int test(struct *a, int b, int c)`	a (addr) in R0, b in R1, c in R2	in R0

`struct s2a {`
`char ta;`
`char ub;`
`int vc;}`
`int test(struct s2a x, int b, int c)`
	x.ta and x.ub in R0,
	x.vc in R1,
	b in R2,
	c in [FP+20]
		in R0

`struct foo *test(int a, int b, int c)`	a in R0, b in R1, c in R2	(address) in R0