(*tag*) beinhaltet die Bits $n = 2 \ldots 7$ bis 31 der Speicheradresse (implementierungsabhängig). Die dicker gekennzeichneten Zustandsübergänge kennzeichnen die normale Vorgehensweise. Zuerst wird durch den Befehl **LDREX** eine Speicherstelle ausgelesen und „markiert". Anschließend berechnet man den neuen Wert und schreibt diesen durch den Befehl **STREX** zurück. Die Speicheroperation kann zu Erfolg führen (schwarz) oder auch nicht (rot). Für den Fall, das die Speicheroperation nicht erfolgreich war, muss der gesamte Vorgang inklusive der Ladeoperation wiederholt werden.

Der globale[45] Exclusive–Monitors ist im Prinzip gleich aufgebaut, jedoch wird neben der Adresse auch die Nummer der CPU geprüft. Da jeder Prozessor einen eigenen globalen Exclusive–Monitor besitzt, muss dieser auch die Zugriffe der anderen Prozessoren auf den gemeinsamen Speicher erkennen können. Wenn ein anderer Prozessor einen erfolgreichen Schreibvorgang auf eine „markierte" Speicherstelle durchgeführt hat, muss der globale Monitor des beobachtenden Prozessors in den Zustand *open access* wechseln.

Bei einem Wechsel des Kontextes durch das Betriebssystem oder bei Ende einer ISR muss sichergestellt werden, dass sich anschließend beide Exclusive–Monitore im Zustand *open access* befinden. Aus diesem Grund ist es auch sinnvoll, dass nach einem Load–linked–Befehl der zugehörige Store–Conditional–Befehl möglichst innerhalb weniger Instruktionszyklen ausgeführt wird. Diese Befehle sollen auch nur auf normalem Speicher ausgeführt werden, nicht für Memory–mapped–I/O.

4.8.4.4 Anwendung

Um eine Variable auch auf Mehrkernsystemen thread–sicher zu inkrementieren dient folgendes Programm.

Assembler #049

```
1 increment:                    ; R0 zeigt auf die Variable
2 LDREX   R1, [R0]              ; Lade die Variable
3 ADD     R1, R1, #1            ; R1 ← R1 + 1
4 STREX   R2, R1, [R0]          ; Ergebnis versuchsweise zurückschreiben
5 CMP     R2, #0x0              ; Prüfen, ob die Operation erfolgreich war
6 BNE     increment             ; Nochmal probieren wenn kein Erfolg
7 MOV     PC, LR
```

Dieser thread–sichere Algorithmus ist **nicht–blockierend** und kann deshalb auch ohne Einschränkung in einer ISR, in Signal–Handlern oder Funktionen des Betriebssystemkerns verwendet werden.

Zur Implementierung weiterer nicht–blockierender Algorithmen wird oft eine thread–sichere, nicht–blockierende Version einer Austauschfunktion benötigt. Hierbei wird der Inhalt einer Variable im Speicher durch einen neuen Wert ersetzt und der alte Wert der Speicherstelle wird zurückgegeben.

Assembler #050

```
1 exchange:                     ; R0 zeigt auf die Variable
2                               ; R1 ist der neue Wert
3 STM     SP!, {R3, R4, LR}
4 exchange_load:
5 LDREX   R3, [R0]              ; lade die Variable
6 STREX   R4, R1, [R0]          ; neuen Wert versuchsweise zurückschreiben
7 CMP     R4, #0                ; prüfen, ob die Operation erfolgreich war
8 BNE     exchange_load         ; nochmal probieren wenn kein Erfolg
9 MOV     R0, R3                ; alten Wert in R0 zurückgeben.
10 LDM    SP!, {R3, R4, PC}
```

Diese Funktion führt die Operationen **R0** ← *Variable* und *Variable* ← **R1** auch auf Mehrkernprozessoren thread–sicher aus. Die Anweisungen in Zeile 5 bis 9 ersetzen den veralteten Befehl **SWP**. Durch diese Anweisungen wird jedoch der Bus nicht blockiert.

[45] Der globale Exclusive–Monitors überwacht die Speicherbereiche, die von mehreren Prozessoren verwendet werden (*shared*).

Durch die Load–linked/Store–Conditional–Befehle kann man einfache, nicht–blockierende Algorithmen implementieren. Es kann jedoch leider immer nur ein Wort modifiziert werden. Man kann z. B. nicht–blockierende Operationen zur Verwaltung einfach–verketteter Listen implementieren, wenn entweder nur am Anfang oder nur am Ende der Liste Objekte eingefügt oder entfernt werden. Komplexere, nicht–blockierende aber dennoch thread–sichere Algorithmen, wie z. B. die Verwaltung doppelt–verketteter Listen, die Implementierung von Speicherverwaltungen (`free()` und `malloc()`) oder die Realisierung von Smart–Pointer lassen sich durch diese Architektur leider nicht oder nur für sehr spezielle Sonderfälle realisieren.

Eine nützliche Operation zur Implementierung von Locking– oder Synchronisationsfunktionen ist die von anderen Prozessoren bekannte Anweisung *compare and swap* (CAS). CAS vergleicht eine Variable und tauscht ggf. den Wert aus, $x = x_\mathrm{a} \Rightarrow x \leftarrow x_\mathrm{n}$. Für $x \neq x_\mathrm{a}$ wird die Variable x nicht geändert. Diese Operation muss atomar durchgeführt werden.

Assembler #051

```
1  compare_and_swap:                ; R0 zeigt auf die Variable x
2                                   ; R1: x_n
3                                   ; R2: x_a
4  STM      SP!, {R3, R4, LR}
5
6  compare_and_swap_load:
7  LDREX    R3, [R0]                ; lade die Variable, R3 ← x
8  CMP      R3, R2                  ; vergleiche x mit x_a
9  BNE      compare_and_swap_exit   ; abbrechen, wenn x ≠ x_a
10 STREX    R4, R1, [R0]            ; neuen Wert versuchsweise zurückschreiben (x ← x_n)
11 CMP      R4, #0                  ; prüfen, ob die Operation erfolgreich war
12 BNE      compare_and_swap_load   ; nochmal probieren wenn Speichern nicht erfolgreich war
13
14 compare_and_swap_exit:
15 LDM      SP!, {R3, R4, PC}       ; Z-Flag: 1 bei Erfolg (x = x_a ⇒ x ← x_n), 0 wenn x ≠ x_a
```

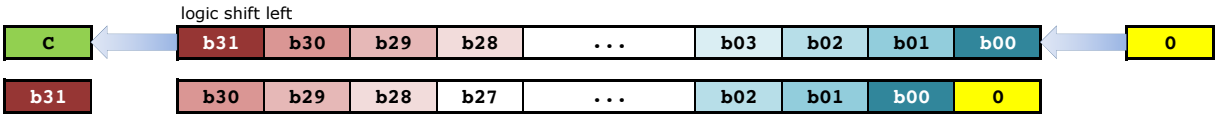
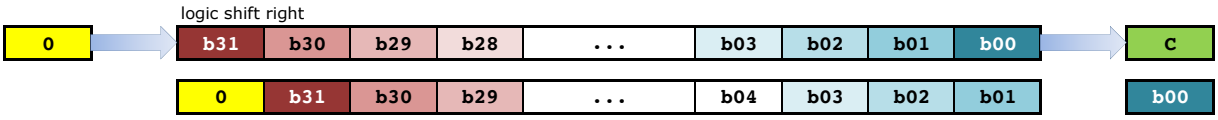
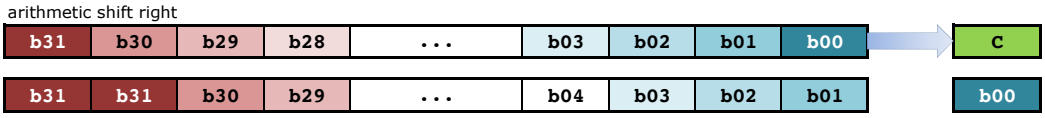
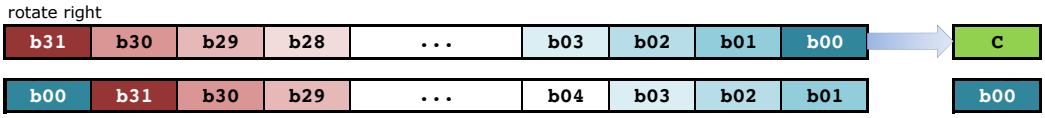
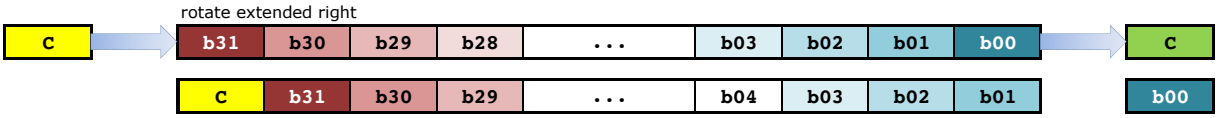
Im Gegensatz zu der CAS–Implementierung vieler bekannter Prozessoren, leidet diese Implementierung jedoch nicht unter dem sog. ABA–Problem. Natürlich dürfen keine normale Speicheroperationen zur Modifikation der Variable x verwendet werden.

Das nachfolgende Beispiel zeigt die Implementierung eines einfachen (blockierenden) Mutex für Mehrkernprozessoren inklusive der notwendigen Speicherbarrieren (mit ARMv7–Befehlen).

Assembler #052

```
1  LOCKED   EQU 1
2  UNLOCKED EQU 0
3
4  lock_mutex:
5  LDREX    R1, [R0]                ; Check if locked
6  CMP      R1, #LOCKED             ; Compare with 'locked'
7  WFEEQ                            ; Mutex is locked, go into standby
8  BEQ      lock_mutex              ; On waking re-check the mutex (ARMv7-M)
9  MOV      R1, #LOCKED
10 STREX    R2, R1, [R0]            ; Attempt to lock mutex
11 CMP      R2, #0x0                ; Check whether store completed
12 BNE      lock_mutex              ; If store failed, try again
13 DMB                              ; Required before accessing resource
14 MOV      PC, LR
15
16 unlock_mutex:
17 DMB                              ; Ensure accesses to resource have completed
18 MOV      R1, #UNLOCKED           ; Write 'unlocked' into lock field
19 STR      R1, [R0]
20 DSB                              ; Ensure update of the mutex must be
21                                  ; visible to other CPUs
22 SEV                              ; Send event to other CPUs (ARMv7-M),
23                                  ; wakes any other CPU waiting on using WFE
24 MOV      PC, LR
```

Die Funktion `lock_mutex` belegt das Mutex und wartet hierbei ggf. unendlich lange. Die Funktion `unlock_mutex` gibt das Mutex wieder frei. Die Speicherbarrieren **DMB** und **DSB** stellen sicher, dass die Aktualisierung der Mutex–Variable auch von allen anderen Prozessoren gesehen wird.

Mnemonik	Beschreibung
	
LSL	Es wird um n Bits nach links geschoben und mit 0 aufgefüllt. Dies entspricht einer Multiplikation mit 2^n. Am Ende der Operation steht Bit $32-n$ optional im Carry-Flag.
	
LSR	Es wird um n Bits nach rechts geschoben und von links mit 0 aufgefüllt. Dies entspricht einer vorzeichenlosen Division durch 2^n. Am Ende der Operation steht Bit $n-1$ optional im C–Flag.
	
ASR	Es wird um n Bits nach rechts geschoben. Entsprechend des Vorzeichen-Bits 31 wird 1 (negativ) oder 0 nachgeschoben. Dies entspricht einer vorzeichenrichtigen Division durch 2^n, **ggf. mit Abrunden** ($+3$ `ASR` $1 = +1$, -3 `ASR` $1 = -2$). Am Ende der Operation steht Bit $n-1$ optional im C–Flag.
	
ROR	Es wird um n Bits nach rechts rotiert (auch von Bit 0 nach Bit 31). Am Ende der Operation steht Bit $n-1$ in Bit 31 sowie optional im Carry-Flag.
	
RRX	Es wird immer um ein Bit nach rechts rotiert (die Anzahl der Bits ist nicht kommandierbar). Das C–Flag wird dabei wie ein 33–tes Bit mit einbezogen.

Tab. 3: *Die verschiedenen Verschiebeoperationen* `LSL`, `LSR`, `ASR`, `ROR` *und* `RRX`, *des Barrel–Shifters und deren Interpretation*

1 Zyklen

1.1 Data Processing

(AND, EOR, ORR, ADD, SUB, MOV, CMP, TST, etc)

Base	**Barrel Shift**	$R_d = PC$
1	+1	+2

1.2 Multiply

Base	**Accumulate**	**Long**	$R_d = PC$
2-5	+1	+1	+2

1.3 Sonstige

Instruction	**Base**	$R_d = PC$
Branch	3	
Load	3	+2
Store	2	
Load Multiple	$n + 2$	+2
Store Multiple	$n + 1$	

1.4 Conditional

Nichtausführung	**Ausführung**
=1	+0