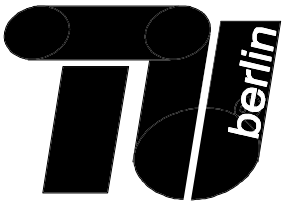
Technische Universität Berlin



Hardwarepraktikum Technische Informatik - Übersicht über die ARM-Befehlssatzarchitektur

Version 1.0
14. Oktober 2010

Technische Universität Berlin
Fakultät IV • Informatik und Elektrotechnik
Institut für Technische Informatik und Mikroelektronik
Fachgebiet Rechnertechnologie
Franklinstraße 28/29 • D-10587 Berlin

1 Die ARM-Befehlssatzarchitektur

Grundlage für jeden Prozessorentwurf ist eine formale Verhaltensspezifikation bzgl. der Ausführung eines Programms, die sogenannte *Befehlssatzarchitektur*, auch als *Programmiermodell* und *Instruction Set Architecture (ISA)* bezeichnet [6]. Die Befehlssatzarchitektur abstrahiert von den Implementierungsdetails der Prozessorhardware. Sie entspricht der Sicht des Programmierers auf einen Prozessor und umfasst alle Informationen, „die zum Schreiben eines ordnungsgemäß laufenden Programms in Maschinensprache erforderlich sind“ [7, S. 555]. Dem Leser sind sicher einige populäre ISAs oder ihnen entsprechende Prozessoren bekannt, beispielsweise IA-32 (Intel Pentium, AMD Phenom usw.), SPARC (Sun UltraSPARC T1, SPARC64V) und POWER (IBM POWER6). Alle Befehlssatzarchitekturen umfassen Informationen zu ähnlichen Teilbereichen.

- Unterstützte Datentypen: Breite, Codierung und verwendete Zahlensysteme,
- Auflistung aller Betriebsmodi, Auswirkungen jedes Betriebsmodus und Mechanismen für den Wechsel zwischen den Modi,
- alle für ein Programm sichtbaren – also les- und/oder beschreibbaren – Register *(Registersatz)*,
- die Speicherorganisation in Bezug auf Adressräume und die verwendete Byteordnung,
- Unterstützung unerwarteter Ereignisse *(Ausnahmen/Unterbrechungen)*, Bedingungen für das Eintreten dieser Ereignisse und dadurch verursachtes Verhalten des Prozessors,
- Auflistung aller Instruktionen *(Instruktionssatz)*, Codierung und Wirkung der Instruktionen auf den Maschinenzustand,
- für einige ISAs auch: Schnittstellen für Coprozessoren und evtl. eine detaillierte Spezifikation zwingend vorhandener Coprozessoren.

Die Verhaltensspezifikation der Prozessor**implementierung**, zum Beispiel auf der *Registertransferebene (register transfer level, RTL)*, wird im Gegensatz zur Befehlssatzarchitektur als *Mikroarchitektur* bezeichnet. Die Mikroarchitektur ist die „Organisation des Prozessors, die die wichtigsten Funktionseinheiten, deren Verbindungen untereinander und deren Steuerung beinhaltet“ [7, S. 369].
Das Programmiermodell aller ARM-Prozessoren, die *ARM instruction set architecture (ARM-ISA)*, ist im *ARM Architecture Reference Manual* festgehalten. Die ARM-ISA hat

zahlreiche Versionen durchlaufen und es ist üblich, diese verkürzt als ARMv4 (ARM-ISA Version 4), ARMv5 usw. zu bezeichnen. Dieses Dokument bezieht sich primär auf ARMv4. Wird die Bezeichnung ARM verwendet, ist abhängig vom Kontext entweder ARMv4, das Unternehmen *Advanced Risc Machines Ltd.* oder eine konkrete Implementierung gemeint, also ein zur ISA kompatibler Prozessor.
Die nun folgenden Ausführungen stützen sich auf das Buch *ARM System Architecture* [4] von Steve Furber, einem der Entwickler der ersten ARM-Prozessoren, sowie das ARM Architecture Reference Manual für ARMv6 [1]. Darin sind alle Details der Architekturversionen 4, 5 und 6 beschrieben.

1.1 Geschichte und Grundzüge der ARM-ISA

Die ersten ARM-Prozessoren wurden in der Mitte der 80er Jahre entwickelt. Für das verantwortliche Unternehmen, Acorn Computers Limited, handelte es sich um den ersten großen IC-Entwurf, sodass die Komplexität des Prozessors schon aufgrund mangelnder Erfahrung gering gehalten werden musste [4, S. 37f.]. Wenige Jahre zuvor waren an den Universitäten Berkeley und Stanford der RISC 1- und der MIPS-Prozessor entstanden. Beide entsprachen einer neuen Entwurfsphilosophie, die seitdem *RISC—Reduced Instruction Set Computer* genannt wird. Die Bezeichnung spielt darauf an, dass die Menge und Mächtigkeit der Befehle, die derartige Prozessoren ausführen konnten, geringer war, als bei anderen zu diesem Zeitpunkt verfügbaren Architekturen. Zur Abgrenzung werden diese als *CISC—Complex Instruction Set Computer* bezeichnet.
Durch die Vereinfachung des Instruktionssatzes reduziert sich auch der Entwicklungsaufwand für den Prozessor. ARM-Prozessoren folgten daher von Beginn an dem RISC-Paradigma, dessen Merkmale in unterschiedlich starker Ausprägung die ISA und die ARM-Mikroarchitekturen beeinflussen:

- Kompakte Befehlsworte einheitlicher Länge (üblich: 32 Bit),
- wenige unterschiedliche Befehlswortformate zwecks vereinfachter Befehlsdecodierung,
- 3-Adress-Befehlsformat, in den Befehlsworten sind zwei Quelloperanden und ein Zieloperand codiert,
- Datenverarbeitung von Operanden ausschließlich aus Prozessorregistern und daher:
  - Einführung vergleichsweise großer Registerspeicher und
  - Einführung dedizierter Instruktionen für den Transfer von Daten zwischen Registerspeicher und Hauptspeicher *(Load/Store-Architektur)*.
- „gleichzeitig überlappende Ausführung von Teilen mehrerer Befehle“ [3, S. 57] in einem *Fließband (Pipeline)*,
- Ausführung von Instruktionen in einem Prozessortakt.

Der letzte Punkt bedarf einer Erläuterung. Es ist nicht gemeint, dass eine Instruktion den Prozessor in einem Takt vollständig durchläuft. Tatsächlich benötigt der Befehlszyklus (*instruction cycle*) einer Instruktion in RISC-Prozessoren mehrere Takte. Dabei durchläuft die Instruktion in jedem Takt einen anderen Teil (Stufe) des Prozessors. Aufgrund der *Fließbandverarbeitung* kann im Idealfall aber eine Instruktion pro Takt beendet werden. In der englischsprachigen Literatur wird dieses Verhalten etwas ungenau als *single-cycle execution* bezeichnet. Präziser und weit verbreitet ist die Angabe in CPI (*clock cycles per instruction*). Ein Prozessor, der im Schnitt pro Takt eine Instruktion beendet, leistet 1 CPI.

$$CPI = \frac{\text{Taktzyklen zur Ausführung eines Programms}}{\text{Anzahl der Maschinenbefehle im Programm}}$$

1 CPI stellt für skalare RISC-Prozessoren, das (praktisch nicht erreichbare) Optimum dar. Skalare Prozessoren können in jedem Takt mit der Bearbeitung maximal einer Instruktion beginnen. Die hier nicht weiter berücksichtigten superskalaren (*multiple-issue*) Prozessoren sind in der Lage, pro Takt mit der Bearbeitung mehrerer Instruktionen zu beginnen und auch mehrere Instruktionen pro Takt zu beenden [5, S. 114].

Da RISC-Instruktionen einfacher gehalten und die Auswirkungen auf den Maschinenzustand daher geringer sind, vergrößert sich die Zahl der in einem Programm notwendigen Befehle um die gleiche Wirkung zu erzielen, die *Codedichte* sinkt. Ein Beispiel dazu:

> RISC-Prozessoren sind nicht unmittelbar in der Lage, ein Datum A aus einem Register zu einem Datum B aus dem Speicher zu addieren, wie dies in anderen Architekturen mit einem Befehl möglich ist. Stattdessen ist eine Instruktion notwendig um A aus dem Speicher in ein weiteres Register zu kopieren und eine zweite Instruktion, um die Registerinhalte zu addieren.

Das RISC-Prinzip hat sich im Prozessorbau weitgehend durchgesetzt, wenn auch nicht unbedingt in seiner reinsten Form. Der Forderung nach single-cycle execution wird für einige Instruktionsarten häufig vernachlässigt.
IA-32 als populärste aller Befehlssatzarchitekturen für leistungsfähige Computer folgt zwar nicht dem RISC-Prinzip, die IA-32-Mikroarchitekturen seit der Mitte der 90er Jahre (Pentium Pro 1995, AMD K5 1996) haben aber RISC-artige Kerne. Die IA-32-Instruktionen werden im Prozessor auf einfachere Befehle abgebildet und diese dann in RISC-Pipelines ausgeführt.

ARM folgt dem RISC-Grundsatz der einfachen Instruktionen zur Annäherung an 1 CPI nicht vollständig. Herr Furber begründet dies v.a. damit, dass frühe Mikroarchitekturen aus Kostengründen nur eine Speicherschnittstelle für Instruktionen und Daten haben durften, im Fließband bei der Ausführung von Load/Store-Befehlen also ohnehin eine Lücke entstand (während eines Datenspeicherzugriffs kann nicht gleichzeitig eine neue Instruktion geholt werden). Dieser zeitliche Leerlauf bis zur nächsten Instruktion wurde für zusätzliche Wirkungen der Load/Store-Instruktionen genutzt, die RISC-untypisch sind

(siehe dazu Abschnitt 1.8.8). Die Instruktionssatzarchitektur spiegelt also auch Überlegungen zur Mikroarchitektur wider.

Die älteste noch von ARM Ltd. durch Dokumentationen und Werkzeuge unterstützte ISA-Version ist ARMv4, alle vorherigen sahen nur einen 26-Bit-Adressraum vor und gelten als veraltet. ARMv5 sollte v.a. Anwendungen der Signalverarbeitung beschleunigen und die Mikroarchitekturen sind etwas komplexer, diese Version fügte dem Konzept der ARM-Prozessoren aber keine entscheidenden Merkmale hinzu. ARMv4T und ARMv5T sind Versionen der ISA, die zusätzlich einen alternativen Instruktionssatz definieren, den *Thumb*-Instruktionssatz. Thumb-Instruktionen sind nur 16 Bit breit und bilden bzgl. ihrer Wirkung eine Teilmenge der ARM-Instruktionen. Programme, die von den mächtigeren ARM-Instruktionen nicht profitieren, belegen in Thumb-Code weniger Speicher.
ARMv6 schreibt das Vorhandensein zweier spezieller Coprozessoren (s.u.) sowie die Unterstützung des Thumb-Modus vor, darüber hinaus gibt es relevante Änderungen im Bereich der Ausnahmeverarbeitung. ARMv6 beschreibt auch detailliert, wie die Speicherhierarchie eines kompatiblen Prozessors aussehen muss und wie z.B. die Übersetzung virtueller Speicheradressen zu erfolgen hat. Eine Nachfrage bei ARM Ltd. hat ergeben, dass die Implementierung eines Prozessors, der Thumb-Befehle unterstützt, ohne Lizenz nicht gestattet ist. Im Rahmen dieser Arbeit kommt die Implementierung eines ARMv6-Prozessors daher nicht infrage.
Geräte mit ARMv4-Prozessoren werden weiterhin produziert, z.B. die Nintendo DSi-Spielekonsole. Die ISA in dieser Version enthält bereits alle wesentlichen Merkmale des ARM-Konzepts und ist mit begrenztem Zeit- und Ressourcenaufwand implementierbar. Sie bildet daher die Grundlage für den Prozessor des Hardwarepraktikums.
Die nun folgende Zusammenfassung der ARMv4-ISA bildet der Kern des Prozessorentwurfs, die hier definierten Vorgaben müssen unabhängig von allen übrigen Entwurfszielen erfüllt werden.

1.2 Datentypen

ARMv4-Prozessoren unterstützen folgende Datentypen:

Datentyp	Breite (Bit)
Byte	8
Halbwort (Halfword)	16
Wort (Word)	32

Tab. 1.1: Die Datentypen der ARM-Architektur (Version 4).

Des Weiteren gilt:

- Alle Prozessorregister sind 32 Bit breit, alle internen Berechnungen, von optionalen

64-Bit-Multiplikationsbefehlen abgesehen, erfolgen mit 32 Bit breiten Operanden und Ergebnissen.

- ARM-Instruktionen sind grundsätzlich 32 Bit breit und an Wortgrenzen im Speicher ausgerichtet.
- Datenspeicherzugriffe erfolgen in Byte-, Halbwort- oder Wortgröße. Aus dem Speicher gelesene Bytes und Halbworte werden auf 32 Bit erweitert, wahlweise mit Nullen oder dem Vorzeichen.
- Die Zahl-Interpretation von Bitvektoren erfolgt im Dualcode oder als Zweierkomplement. Ein n Bit breiter Vektor (n= 8, 16 oder 32) repräsentiert daher Zahlen im Intervall $[0, 2^n - 1]$ oder $[-2^{n-1}, 2^{n-1} - 1]$.

1.3 Adressraum

ARM-Prozessoren verwenden einen gemeinsamen, flachen Adressraum für Instruktionen und Daten. Instruktions- und Datenadressen sind 32-Bit-Dualzahlen. Die kleinste adressierbare Einheit ist ein Byte, insgesamt adressierbar sind damit 2^{32} Byte. Zum gegenwärtigen Zeitpunkt deuten keine Veröffentlichungen darauf hin, dass ARM eine Vergrößerung des Adressraums plant, wie sie bei vielen anderen populären Instruktionssatzarchitekturen vollzogen worden ist, um mehr Arbeitsspeicher adressieren zu können.
Wortadressen sind ohne Rest durch vier teilbar, Halbwortadressen ohne Rest durch zwei. Das heißt, Datenspeicherzugriffe müssen ausgerichtet (*aligned*) sein. Bis inkl. ARMv5 ist implementierungsabhängig, welche Byteordnung der Datenspeicher aufweist. ARM-Prozessoren dürfen sowohl nur *little endian byte ordering* als auch nur *big endian byte ordering* unterstüzen oder frei konfigurierbar sein. Der Prozessor des Praktikums verwendet little endian byte ordering, allerdings nicht aus technischen Gründen. Vielmehr geht es um die einheitliche Darstellung von Prozessorregistern, Datenbus und dem Arbeitsspeicher, wenn dieser in Wortbreite betrachtet wird. Das folgende Beispiel und Abbildung 1.1 verdeutlichen diesen Sachverhalt.

> Die Frage der Byteordnung stellt sich, sobald auf Daten im Speicher zugegriffen wird, die breiter sind als die Organisationseinheit des Speichers selbst, also z.B. auf Worte in byteadressiertem Speicher. Die vier Bytes des Datenwortes können unterschiedlich auf die vier Bytes des Speichers verteilt werden. Dabei sind zwei Varianten gebräuchlich:
> *little endian*: Das niederwertigste Byte innerhalb des Wortes wird an die kleinste Byteadresse geschrieben.
> *big endian*: Das höchstwertige Byte innerhalb des Wortes wird an die kleinste Byteadresse geschrieben.
> Im Beispiel soll der Wert `0x87654321` aus einem Prozessorregister an die (zur besseren Darstellbarkeit auf 16 Bit verkürzte) Wortadresse `0xFF00` geschrieben werden. Dabei wird der Speicher an den Byteadressen `0xFF00`, `0xFF01`, `0xFF02` und `0xFF03` belegt.

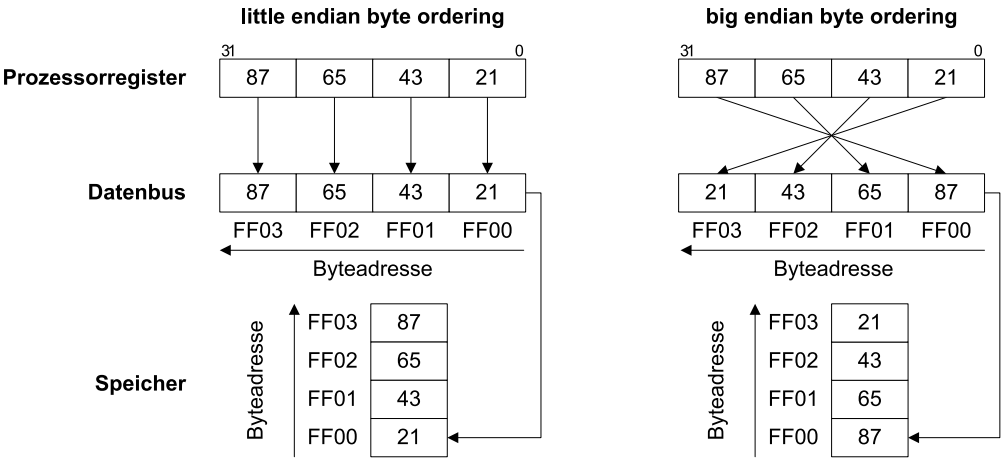


Abb. 1.1: Auswirkung der Byteordnung auf das Ablegen von Worten im Speicher

ARM-Prozessoren kommunizieren mit Peripheriekomponenten, z.B. Timern oder Kommunikationsschnittstellen, generell durch *speicherabgebildete Ein-/Ausgabe (speicherbezogene Adressierung, memory-mapped I/O*). Die Komponenten sind an den Datenbus angeschlossen und Teile des 32-Bit-Adressraums werden einzelnen Komponenten zugeordnet. „Lese- und Schreiboperationen für diese Adressen werden als Befehle für das [...] Gerät interpretiert“ [7, S.486].

1.4 Statusregister und Betriebsmodi

ARM-Prozessoren verfügen über sechs 32-Bit-Statusregister (Abbildung 1.2). Jederzeit sichtbar ist das *Current Program Status Register (CPSR)*.

Abb. 1.2: ARMv4-Statusregister.

- Die Bits 31:28 des CPSRs sind Statusbits (*condition code flags* oder kurz der *Condition-Code, CC*) des Prozessors. Sie werden von einzelnen Instruktionen auf Basis des Ergebnisses einer arithmetischen oder logischen Verknüpfung zweier Operanden gesetzt. Im Allgemeinen gilt dabei:
  - N = 1 (Negative): Das Ergebnis einer Berechnung war, als Zweierkomplementzahl interpretiert, negativ.
  - Z = 1 (Zero): Das Ergebnis einer Berechnung war 0x00000000.
  - C = 1 (Carry): Eine Berechnung hat einen Übertrag über die höchste Ergebnisstelle erzeugt. Hier gelten allerdings zahlreiche Ausnahmen: Bei Subtraktionen

wird das C-Bit gerade dann gesetzt, wenn kein Übertrag aufgetreten ist. Einige Instruktionen verändern ausschließlich das C-Bit und lassen N, Z und V unangetastet oder verändern nur N und Z.

  - V = 1 (Overflow): Eine Berechnung hat einen Überlauf verursacht.

- Die Bits 7:6 dienen dem Maskieren (Unterdrücken) von Unterbrechungen. ARMv4-Prozessoren haben zwei Unterbrechungsarten, normale *Interrupts (IRQ)* und *Fast Interrupts (FIQ)*. Ist Bit 7 (I) gesetzt, reagiert der Prozessor nicht mehr auf IRQs, bei gesetztem Bit 6 nicht mehr auf FIQs.

- Bit 5 (T) zeigt an, ob aktuell Thumb-Instruktionen ausgeführt werden. In Prozessoren, die den Thumb-Modus nicht unterstützen, ist das Bit immer 0.

- Die in der Abbildung grau hinterlegten Abschnitte des CPSRs werden in ARMv4 nicht verwendet, erhielten aber z.T. in neueren Architekturversionen bis inkl. ARMv7 eine Bedeutung. Schreibzugriffe auf nicht verwendete Abschnitte des CPSRs sind wirkungslos, beim Lesen des Statusregisters enthalten sie Nullen.

- In den Bits 4:0 schließlich ist der aktuelle Betriebsmodus des Prozessors codiert.

Es gibt sieben Betriebsmodi, sechs davon sind *privilegiert*. Sie unterscheiden sich vom einzig nicht privilegierten Modus in der Möglichkeit, das CPSR frei beschreiben zu können. Abhängig vom Design der Speicheranbindung kann auch der Zugriff auf bestimmte Speicherbereiche oder Peripheriekomponenten auf privilegierte Modi beschränkt bleiben, dies ist aber nicht durch die ISA festgeschrieben. Tabelle 1.2 fasst alle erlaubten Modusvektoren und ihre Bedeutung zusammen.

Modus	**CPSR[4:0]**	**Verwendung**	**privilegiert/Ausnahmemodus**
User	`10000`	Betriebsmodus für Benutzerprogramme	– /–
System	`11111`	privilegierter Betriebsmodus	✓/–
Supervisor	`10011`	Behandlung von Software Interrupts	✓/✓
FIQ	`10001`	Behandlung von FIQs	✓/✓
IRQ	`10010`	Behandlung von IRQs	✓/✓
Abort	`10111`	Behandlung von Speicherzugriffsfehlern	✓/✓
Undefined	`11011`	Behandlung undefinierter Instruktionen	✓/✓

Tab. 1.2: Betriebsmodi von ARM-Prozessoren.

Fünf der Betriebsmodi dienen der Behandlung verschiedener Ausnahmesituationen. Jeder Ausnahmemodus verfügt über eine eigene Kopie des CPSRs, das sogenannte *Saved Program Status Register (SPSR)*. Bei Bedarf kann innerhalb eines Ausnahmemodus lesend und schreibend auf das zum Modus gehörende SPSR zugegriffen werden. Weitere Informationen zur Behandlung von Ausnahmen folgen in Abschnitt 1.6. Wird in diesem

Dokument die Bezeichnung *PSR* verwendet, so bezieht sich dies auf den Statusregistersatz im Allgemeinen. Ob speziell das CPSR oder ein SPSR gemeint ist, hängt von weiteren Randbedingungen ab.

1.5 Registerspeicher

Die in ARM-Instruktionen verwendeten Registeradressen sind 4 Bit breit, mithin also 16 Universalregister (*general-purpose registers*, im Folgenden nur als *Register* bezeichnet) sichtbar. Die sichtbaren Register werden zukünftig mit R0 für Adresse `0000` bis R15 für Adresse `1111` bezeichnet, alle Register sind 32 Bit breit.
Der vollständige Registerspeicher umfasst 31 Register, einigen Registeradressen sind mehrere physische Register zugeordnet, es existieren *Registerbänke*. Welches Register durch eine Registeradresse angesprochen wird, hängt ggf. vom aktuellen Betriebsmodus ab. Dabei gilt der in Abbildung 1.3 verdeutlichte Zusammenhang. Die Registeradressen R0–R7

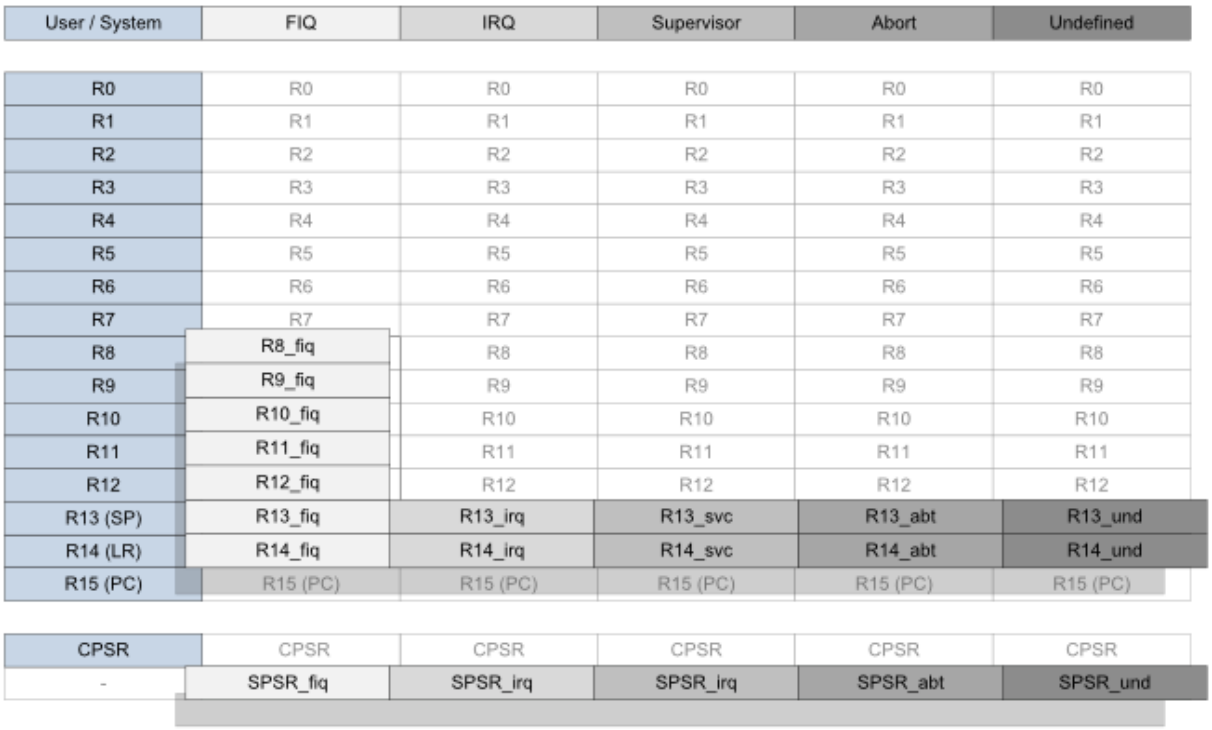


Abb. 1.3: Betriebsmodus (oben), jeweils sichtbare Register (mitte) und Statusregister (unten).

und R15 beziehen sich in jedem Modus auf dieselben Register (*unbanked registers*). Die Modi User und System verwenden dieselben Register, sie unterscheiden sich nur beim Zugriff auf das CPSR und ggf. auf den Speicher.
R8 bis R14 heißen *banked registers*. Alle Ausnahmemodi haben jeweils einen eigenen Satz der Register R13 und R14. Im FIQ-Modus stehen zusätzlich exklusive Register für die Adressen R8–R12 bereit. Dass im FIQ-Modus ein eigener Satz von R8–R12 verwendet

wird, beschleunigt die Behandlung von Fast Interrupts gegenüber normalen Interrupts.
Die Register R14 und R15 haben eine durch die ISA vorgegebene ausgezeichnete Bedeutung. R15 ist der *Befehlszähler (program counter, PC)* von ARM-Prozessoren. Weil der PC als gewöhnliches Register sichtbar ist, kann die aktuelle Programmadresse jederzeit von Instruktionen ausgelesen und verwendet werden, zum Beispiel für die PC-relative Adressierung von Daten im Speicher. Beim Lesen des PCs gilt generell: Eine Instruktion erhält dabei ihre eigene Instruktionsadresse plus acht (Byte), also die Adresse des übernächsten Befehlswortes. Dieses Verhalten erklärt sich aus der Struktur der ARM-Mikroarchitekturen, die bei der ISA-Gestaltung berücksichtigt wurde. Wird R15 als Zielregister einer Instruktion verwendet, verursacht das Schreiben in den PC einen Sprung. Hierfür gelten allerdings zahlreiche, in Kapitel 1.8 aufgeführte Einschränkungen.
R14 ist das Link-Register (*Link Register*). Bei der Ausführung bestimmter Sprungbefehle zwecks Unterprogrammaufruf und bei der Behandlung von Ausnahmen wird der PC oder ein davon abgeleiteter Wert nach R14 kopiert. Um nach einem Unterprogrammaufruf zum aufrufenden Programm zurückzukehren, genügt es, R14 nach R15 zu kopieren. Zur Rückkehr aus einer Ausnahmebehandlung muss R14 ggf. noch geringfügig angepasst werden.

R13 ist kein ausgezeichnetes Register im Sinne der ISA, seine besondere Bedeutung ergibt sich durch den *Procedure Call Standard for the ARM Architecture* (*AAPCS*) [2]. Der AAPCS definiert einen einheitlichen Standard für den Ablauf von Unterprogrammaufrufen. Dort ist festgelegt, dass die Übergabe bestimmter Parameter und das Sichern von Registern in einem Unterprogramm auf einem Stapelspeicher (*Stack*) zu erfolgen hat und dass R13 als Zeiger auf den Stack (*Stack Pointer, SP*) dient. Weil R13 modusspezifisch ist, existieren also bis zu 6 hardwareunterstützte Stacks gleichzeitig.
Wie oben gezeigt, ist neben der in einer Instruktion verwendeten Registeradresse auch immer der Betriebsmodus zu berücksichtigen, wenn ein ARM-Register adressiert wird. Darüber hinaus gibt es noch einen weiteren Einflussfaktor: Die in bestimmten Instruktionen verwendeten Registeradressen können sich auf Register des User-Modus beziehen, während sich der Prozessor in einem Ausnahmemodus befindet. So ist es möglich, innerhalb einer Ausnahmebehandlung die Register des User-Modus zu sichern oder wiederherzustellen. Das ist z.B. nützlich beim Taskwechsel zwischen mehreren User-Modus-Anwendungen innerhalb eines Betriebssystems, dessen Routinen im Supervisor-Modus ausgeführt werden.
Um das Verständnis für die Besonderheiten des Registerspeichers von ARM-Prozessoren zu verbessern, bietet sich folgende Vorstellung an: Es existiert eine Funktion, die eine 4-Bit-Registeradresse einer Instruktion, u.a. unter Berücksichtigung des Betriebsmodus, auf eines von 31 physischen Registern abbildet. Zur eindeutigen Adressierung der physischen Register genügt ein 5-Bit-Adressraum, wobei eine der 32 möglichen Adressen unbenutzt bleibt. In Anlehnung an Begriffe aus der (Haupt)speicherverwaltung [3, S.407] wird die 4-Bit-Registeradresse in dieser Arbeit als *virtuelle Registeradresse* bezeichnet und die davon abgeleitete 5-Bit-Registeradresse als *reale Registeradresse*. Weil diese Sicht auf den Registerspeicher dessen Implementierung und allgemein den Umgang mit Operandenadressen im Prozessor vereinfacht, enthält der Hardwarepraktikums-Prozessor Schaltungen zur Abbildung von virtuellen auf reale Registeradressen.

1.6 Ausnahmebehandlung

Prozessoren sind gewöhnlich „in der Lage vom Programm unabhängige, *synchrone* oder *asynchrone Ereignisse* zu bearbeiten. Synchrone Ereignisse [Traps] sind z.B. Fehler, die infolge der Ausführung eines Befehls [...] auftreten“ [6, Kapitel 1.4.1]. Ein mögliches synchrones Ereignis in einem ARM-Prozessor ist der Versuch, auf Daten- oder Programmspeicheradressen zuzugreifen, an denen kein physischer Speicher existiert.
Asynchrone Ereignisse (*Unterbrechungen/Interrupts*) treten dagegen unabhängig vom aktuell bearbeiteten Programm auf. Die ARM-ISA spezifiziert bis inkl. Version 5 nur zwei Quellen für Interrupts, zusätzlich fällt auch der Neustart (*Reset*) des Prozessors in diese Kategorie. Die ISA fasst alle synchronen und asynchronen Ereignisse unter dem Oberbegriff der *Ausnahme (Exception)* zusammen. Da mehrere Ausnahmen zeitgleich auftreten können, sind sie untereinander priorisiert und sie werden entsprechend ihrer Prioritäten nacheinander bearbeitet. Die Reaktion auf alle Ausnahmen ist sehr ähnlich, im Allgemeinen gilt:

- Instruktionen, deren Bearbeitung vor Eintritt der Ausnahme begonnen wurde, werden soweit wie möglich korrekt beendet.
- Der Prozessor wechselt in den Betriebsmodus, der für die Behandlung der jeweiligen Ausnahme vorgesehen ist. Dies geschieht durch das Schreiben des neuen Modus in das CPSR.
- Das I-Bit des CPSRs wird gesetzt und IRQs damit maskiert. Das F-Bit wird in Abhängigkeit vom Ausnahmetyp gesetzt oder nicht verändert.
- Der bisherige Inhalt des CPSRs, vor Änderung von Modus- und Maskenbits, wird in das SPSR des Ausnahmemodus kopiert.
- In das Link-Register des Ausnahmemodus wird eine Rücksprungadresse geschrieben. Der konkrete Wert ist vom Typ der Ausnahme und der Adresse der Instruktion abhängig, die die Ausnahme verursacht hat.
- Der Prozessor verzweigt an eine ausnahmespezifische Speicheradresse (*exception vector*), dort muss eine Instruktion hinterlegt sein, die den Sprung in ein Programm zur Ausnahmebehandlung (*Exception-Handler*) verursacht.
- Der Exception-Handler sichert alle Register, die im jeweiligen Ausnahmemodus nicht exklusiv zur Verfügung stehen, in der Ausnahmebehandlung aber dennoch verwendet werden. Dazu kopiert er alle fraglichen Register auf den Stack des Ausnahmemodus. Gemeinsam mit der CPSR-Kopie im SPSR ist damit der Kontext des unterbrochenen Programms gesichert.
- Am Ende der Ausnahmebehandlung werden die Registerinhalte vom Stack des Ausnahmemodus rekonstruiert. Anschließend kehrt der Prozessor durch eine spezielle Form einer gewöhnlichen Instruktion zum normalen Programmablauf zurück. Dabei wird der Inhalt des SPSRs des Ausnahmemodus in das CPSR kopiert. Somit ist

der Kontext (Registerspeicher und Statusregister) des unterbrochenen Programms wiederhergestellt.

Der Sprung in die Ausnahmebehandlung ist für Anwendungen transparent und wird durch die Prozessorhardware autonom durchgeführt. Das Sichern der Register auf den Stack ist anschließend mit einer einzigen Instruktion (STM, siehe 1.8.8) möglich. Das Hinterlegen von Sprungbefehlen an den Exception-Vektoren geschieht z.B. durch das Betriebssystem. Die Rückkehr aus der Ausnahmebehandlung erfolgt durch bestimmte Varianten von arithmetischen, logischen oder Ladeinstruktion. Der Zieloperand dieser Instruktionen ist R15, der erste Quelloperand das Link-Register des Ausnahmemodus oder ein Speicherwort, in das der Inhalt des Link-Registers kopiert wurde.
Folgende Ausnahmen werden durch ARM-Prozessoren verarbeitet:

- Reset Exception
  Ein prozessorexternes Signal verursacht den sofortigen Abbruch der Bearbeitung aller Instruktionen im Prozessor, nach Rücknahme des Signals beginnt die Befehlsausführung an Adresse 0 im Supervisor-Modus neu. Nach einem Reset kann nicht wie nach allen übrigen Ausnahmen zu einem Programm „zurückgekehrt“ werden. Der einzige Weg, nach einem Neustart in einen Nicht-Ausnahmemodus zu wechseln besteht darin, den Modusvektor der Modi User oder Sytem explizit in die Modusbits des CPSRs zu schreiben.

- Undefined Exception
  Der Prozessor reagiert mit dieser Ausnahme auf den Versuch, einen Instruktionscode auszuführen, dem keine Bedeutung zugeordnet ist. Siehe dazu auch Kapitel 1.8.2. Gleiches gilt für Coprozessorinstruktionen, auf die kein Coprozessor reagiert. Die verursachende Instruktion wird durch den Prozessor gegen einen Sprung zum Undefined-Vektor ersetzt.

- Software Interrupt Exception
  Der Software Interrupt wird durch die Ausführung einer besonderen Instruktion (SWI) ausgelöst. Er dient in erster Linie dazu, aus einem Programm im User-Modus heraus Dienste anzufordern, die höhere Zugriffsrechte auf Systemkomponenten erfordern. Ein Beispiel wäre der Zugriff auf eine Ein-/Ausgabeschnittstelle, der ggf. im User-Modus nicht gestattet ist.

- Prefetch Abort Exception
  Prefetch Aborts sind Speicherzugriffsfehler beim Versuch, eine Instruktion aus dem Instruktionsspeicher zu lesen. Die (nicht) gelesene Instruktion wird vorerst nur als ungültig gekennzeichnet. Der Prozessor verzweigt in die Ausnahmebehandlung sobald versucht wird, die ungültige Instruktion tatsächlich auszuführen. Liegt sie beispielsweise im Schatten eines verzweigenden (bedingten) Sprungbefehls, tritt keine Ausnahme auf.

- Data Abort Exception
  Data-Aborts sind Seiteneffekte von Datenspeicherzugriffen. Sie müssen vom Speichersystem angezeigt werden, während der Zugriff erfolgt (*precise data abort*). Im

Gegensatz zu allen anderen Ausnahmen befindet sich die verursachende Instruktion hier bereits in Ausführung. Sie wird nicht beendet, d.h. sie hat keine Auswirkungen auf sichtbare Prozessorregister, möglicherweise aber bereits den Speicherinhalt irreversibel verändert. Auch alle Instruktionen, deren Bearbeitung nach dem Speicherzugriffsbefehl begonnen wurde, werden abgebrochen.

- IRQ Exception
  Der Prozessor verzweigt in einen IRQ-Handler, wenn ein Interrupt am IRQ-Eingang des Prozessors angezeigt wird und das I-Bit im CPSR nicht gesetzt ist. Die Reaktion auf den Interrupt erfolgt asynchron zu den zu diesem Zeitpunkt bearbeiteten Instruktionen. Der Sprung in die Ausnahmebehandlung verdrängt die Instruktion, die ohne Interrupt als nächste ausgeführt würde.

- FIQ Exception
  Es gilt im Wesentlichen das für die IRQ Exception gesagte, wobei der Interrupt hier am FIQ-Eingang des Prozessors signalisiert wird und das F-Bit des CPSRs nicht gesetzt sein darf.

Tabelle 1.3 fasst alle sieben Ausnahmetypen, ihre Prioritäten, die jeweils zur Behandlung bestimmten Modi und die Exception-Vektoren zusammen.

Exception-Typ	**Modus**	**FIQ maskiert**	**Ausnahmevektor**	**Priorität**
Reset	Supervisor	ja	0x00000000	1
Undefined Instruction	Undefined	nein	0x00000004	6
Software Interrupt	Supervisor	nein	0x00000008	6
Prefetch Abort	Abort	nein	0x0000000C	5
Data Abort	Abort	nein	0x00000010	2
IRQ	IRQ	nein	0x00000018	4
FIQ	FIQ	ja	0x0000001C	3

Tab. 1.3: Ausnahmemodi von ARM-Prozessoren.

Neben dem bisher Gesagten sind folgende Sachverhalte zu beachten:

- Der Neustart des Prozessors (Reset) hat die höchste Priorität. Abweichend von den zuvor genannten Regeln sind die Werte im Link-Register und dem SPSR des Supervisor-Modus nach einem Reset nicht spezifiziert.

- Nach dem Reset sind beide Interruptquellen maskiert.

- Undefined Instruktion Exception und Software Interrupt Exception haben dieselbe (niedrigste) Priorität, weil sie nicht gleichzeitig auftreten können. Eine Instruktion kann nicht ein SWI-Befehl, und damit definiert, gleichzeitig aber undefiniert sein.

- Adresse 0x00000014 gehört zu einer ab ARMv4 obsoleten Ausnahme und wird bis inkl. ARMv6 nicht für Ausnahmevektoren verwendet.
- Der Ausnahmevektor der FIQs liegt an der höchsten aller Vektoradressen. Der FIQ-Handler kann daher direkt an dieser Adresse beginnen.
- Data Aborts haben zwar die zweithöchste Priorität, maskieren aber FIQs nicht. Treten beide Ausnahmen gleichzeitig auf, reagiert der Prozessor zuerst auf den Data Abort und verzweigt an den zugehörigen Ausnahmevektor. Die dortige Instruktion wird jedoch nicht ausgeführt, sondern sofort die Behandlung des FIQs begonnen. Nach der Rückkehr aus dem FIQ-Handler springt der Prozessor in den Abort-Handler.
- Prefetch Aborts und Data Aborts werden im selben Modus (Abort) behandelt, haben aber eigene Ausnahmevektoren.

Abschließend gibt Tabelle 1.4 einen Überblick darüber, welcher Wert beim Sprung in die Ausnahme jeweils nach R14 geschrieben wird mit welchen Instruktionen die Rückkehr zum normalen Programm erfolgt.

1.7 Coprozessorschnittstelle

Die Funktionalität von ARM-Prozessoren ist durch Coprozessoren erweiterbar, wobei eine Mikroarchitektur diese Möglichkeit nicht zwingend bieten muss. Die ISA lässt Struktur und Funktionsweise möglicher Coprozessoren völlig offen. Auch die Anbindung an den ARM-Kern ist bzgl. der verwendeten Signale und ausgetauschter Informationen nicht vorgeschrieben. In der Konsequenz ändert sich die Schnittstelle in praktisch jeder Generation gewerblicher ARM-Prozessoren und wird zusehends komplizierter, sofern sie überhaupt vorhanden ist. Durch die ISA vorgegeben sind lediglich einige Instruktionen, die zur Steuerung von Coprozessoren verwendet werden müssen:

- Instruktionen, die eine Datenverarbeitung im Coprozessor auslösen,
- Instruktionen, die den Austausch von Registerinhalten zwischen ARM-Kern und Coprozessor ermöglichen,
- Instruktionen, die einen Hauptspeicherzugriff des Coprozessors ermöglichen. Die Zugriffsadressen werden dabei aber vom ARM-Kern erzeugt.

Die Art der vorhandenen Instruktionen zeigt, dass Coprozessoren ebenso wie der ARM-Kern als Load/Store-Architektur umgesetzt werden sollen. Sie erhalten ihre Operanden vom ARM-Hauptprozessor oder aus dem Hauptspeicher, legen sie in Registern ab, verknüpfen die Registerinhalte und schreiben Ergebnisse dann wieder in den Hauptspeicher oder den ARM-Kern. Ein Coprozessor muss nicht zwingend an den Hauptspeicher angebunden sein, kann dann aber Operanden nur aus den ARM-Registern erhalten.
Coprozessorinstruktionen werden auch vom ARM-Kern ausgewertet. Ein 4-Bit-Feld innerhalb der Instruktionen gibt an, welcher Coprozessor sie ausführen soll. Es kann also bis

Ausnahme	In R14 gesicherter Wert	Rückkehr aus der Behandlung
Reset	undefiniert	nicht möglich
Undefined	Adresse der Instruktion, die auf die undefinierte Instruktion folgt	**MOVS PC, R14**; Sprung hinter die undefinierte Instruktion
SWI	Adresse der Instruktion nach der SWI-Instruktion	**MOVS PC, R14**; Sprung hinter die SWI-Instruktion
Prefetch Abort	Adresse der verursachenden Instruktion + 4	**SUBS PC, R14, #4**; Sprung zur verursachenden Instruktion
Data Abort	Adresse der ursächlichen Instruktion + 8	**SUBS PC, R14, #8**; um die Instruktion nach Beheben der Fehlerursache erneut auszuführen, **SUBS PC, R14, #4**; Sprung hinter die verursachende Instruktion
IRQ/FIQ	Adresse der nächsten auszuführenden Instruktion + 4	**SUBS PC, R14, #4**; Sprung zur durch den Interrupt verdrängten Instruktion

Tab. 1.4: In R14 gesicherter Wert und Möglichkeiten zur Rückkehr aus einer Ausnahme. MOVS und SUBS sind spezielle Varianten der Move- und Subtraktionsinstruktionen MOV und SUB.

zu 16 Coprozessoren pro Hauptprozessor geben, sowohl als Teil des ICs als auch in Form externer Bausteine. Ein Coprozessor darf dabei mehr als eine Nummer haben, sodass Instruktionen formal unterschiedlicher Coprozessoren durch denselben Baustein ausgeführt werden.
Ein Coprozessor kann über maximal 16 beliebig breite Register verfügen. Besetzt ein Coprozessor n Coprozessornummern, kann er folglich $n \cdot 16$ Register verwalten. Die übliche ARM-Fließkommaeinheit, der *Vector Floating Point Processor*, ist beispielsweise in Form von zwei Coprozessoren realisiert und hat daher 32 Register.
ARM reserviert die Coprozessornummern 0–3 und 8–15 für eigene Erweiterungen, die Nummern 4 bis 7 dürfen frei verwendet werden. Coprozessor 15 ist traditionell der *System Control coprocessor*, der vor allem die Caches und die Speicherverwaltung steuert. Coprozessor 14 wird als *debug architecture interface* bezeichnet und dient der Fehlersuche in Anwendungen. Ab ARMv6 muss jeder ARM-Prozessor über diese beiden detailliert spezifizierten Coprozessoren verfügen.
Wie Coprozessoren ihre Instruktionen erhalten, ist nicht vorgeschrieben. In einigen Mikroarchitekturen lesen Sie parallel zum Hauptprozessor alle Befehle vom Instruktionsbus,

in anderen leitet der Hauptprozessor die Instruktionen aktiv über eine eigene Schnittstelle weiter. Ist kein Coprozessor vorhanden, der eine Instruktion ausführen kann, reagiert der ARM-Kern mit einer Undefined Exception. Innerhalb der Ausnahmebehandlung kann dann die Wirkung des Coprozessors in Software – langsamer – nachgebildet werden. Aufgrund dieses Verhaltens eignen sich Coprozessoren gut, um den Befehlssatz abwärtskompatibel zu älteren oder einfacheren ARM-Implementierungen zu erweitern.

1.8 Instruktionssatz

Ausgangspunkt für den Prozessorentwurf ist eine detaillierte Analyse der in der ISA festgeschriebenen ARM-Instruktionen. Ein ARMv4-kompatibler Prozessor führt mindestens 44 verschiedene Instruktionen aus, bestimmte Typen bis zu 49. Ein ARM-Befehlswort umfasst 32 Bit. Um eine ARMv4-Instruktion eindeutig zu identifizieren, müssen maximal die 12 Instruktionsbits 27:20 und 7:4 betrachtet werden, alle übrigen Stellen codieren z.B. Operanden oder haben keinen Einfluss auf die Instruktionsausführung. Viele ARM-

Abb. 1.4: 12 Bit innerhalb des Befehlswortes codieren die Instruktion.

Instruktionen entsprechen dem RISC-typischen 3-Adress-Befehlsformat. Dabei sind Ziel- und erster Quelloperand immer Registeradressen. Der zweite Quelloperand ist üblicherweise in den Instruktionsbits 11:0 codiert und kann beispielsweise ein Direktoperand oder eine weitere Registeradresse sein.

1.8.1 Begriffsvereinbarung für die Instruktionsdarstellung

- Bitposition, die innerhalb eines Befehls einen bestimmten Wert haben müssen, sind mit 1 oder 0 explizit angegeben.
- Als *SBZ* (Should Be Zeros) oder *SBO* (Should Be Ones) gekennzeichnete Bereiche sollten ausschließlich Nullen bzw. Einsen enthalten, werden von Prozessoren aber nicht ausgewertet.
- *Rd*, *Rn*, *Rm* und *Rs* geben jeweils die 4 Bit breite Adresse eines beliebigen ARM-Registers an.
- *CRd*, *CRn* und *CRm* sind Felder ohne vorgeschriebene Wirkung. In der ISA wird die Verwendung zur Codierung von Coprozessorregisteradressen empfohlen.
- *Cop1*, *Cop2* sind Felder ohne vorgeschriebene Wirkung. In der ISA wird empfohlen, diese Felder zur Codierung von Coprozessor-Steuerinformationen zu verwenden.
- Mit *P, U, S, W, N, L, H, B, I, Opcode, Mask* und *Mul* bezeichnete Felder unterscheiden Instruktionen voneinander oder steuern die Wirkung einzelner Instruktionen.
- *CP#* ist die Nummer eines Coprozessors im Dualcode.
- *Register/Adresse:=Wert* ist eine Zuweisung eines Wertes an eine Speicheradresse oder ein Register.
- *Rn** bezeichnet ein Datum, das (optional) in das Register mit Adresse Rn geschrieben wird und dabei u.a. vom bisherigen Inhalt von Rn abhängt.

1.8.2 Lücken im Instruktionssatz

Der ARMv4-Instruktionssatz enthält einige Lücken. Bestimmte Bitmuster lassen sich keiner Instruktion zuordnen. Soll ein ARM-Prozessor derartige Befehlsvektoren verarbeiten, wird eine Undefined Exception erzeugt. Viele Lücken wurden in neueren Versionen der ISA durch weitere Instruktionen gefüllt, ARMv6 umfasst beispielsweise 112 Instruktionen.
In anderen Fällen ist ein Bitmuster zwar einer Instruktion eindeutig zugeordnet, die Wirkung auf einen ARM-Prozessor aber nicht für jede Kombination der im Befehl vorhandenen Kontrollbits oder jeden möglichen Operanden definiert. Häufig ergeben sich dadurch Probleme, dass auf den PC als gewöhnliches Register zugegriffen werden kann. Fast alle Instruktionen sehen einen, ggf. optionalen, Schreibzugriff auf ein Zielregister vor. Wird dabei in den PC geschrieben, entspricht das einem Sprung. Für viele Instruktionen ist dieses Verhalten nicht sinnvoll und verkompliziert den Prozessorentwurf. In solchen und ähnlich gelagerten Fällen beschreibt die Instruktionsreferenz die Wirkung der Instruktion als unvorhersehbar (*UNPREDICTABLE*). Für Softwareentwickler, die entweder einen Compiler für ARM entwerfen oder Programme in ARM-Assembler schreiben, werden damit Einschränkungen vorgegeben, die sie zu berücksichtigen haben. Der Entwurf eines ARM-Prozessors wird dadurch aber vereinfacht, weil das Verhalten in diesen Randfällen frei gewählt und bestimmten Entwurfszielen wie Komplexität, Geschwindigkeit, Fläche und Leistungsaufnahme untergeordnet werden kann. Die Instruktionsübersicht in diesem Kapitel umfasst alle Instruktionen und beschreibt jeweils grob deren Wirkung. Da in der Mikroarchitektur das Verhalten von „unvorhersehbaren“ Instruktionen sehr wohl determiniert ist, werden die Sonderfälle in dieser Arbeit als „unspezifiziert“ statt „unvorhersehbar“ bezeichnet. In einem Vorgriff auf die Implementierung sei hier beschrieben, wie der Prozessor auf einige der Randfälle reagiert.

- R15 als Quelloperand entspricht immer der Adresse der R15 lesenden Instruktion + 8.

- Schreibzugriffe auf R15 führen nur zu einem Sprung, wo die ISA dies explizit vorsieht und werden sonst ignoriert.

- Einige Instruktionen können das gleichzeitige Schreiben zweier Werte in dasselbe Register verlangen. Der erste Wert stammt dabei aus einem Register und wurde ggf. während der Befehlsausführung modifiziert, der zweite ist ein aus dem Speicher gelesenes Datum. Letzteres wird in diesem Fall nachrangig behandelt, also nicht in das Prozessorregister geschrieben.

- Schreibzugriffe auf das SPSR in den Betriebsmodi User oder System werden ignoriert, Lesezugriffe liefern einen definierten, hier aber als zufällig zu betrachtenden Wert.

1.8.3 Besonderheiten des Instruktionssatzes

Bedingte Befehlsausführung

In vielen ISAs (MIPS, IA-32 etc.) werden Programmverzweigungen durch bedingte Sprungbefehle realisiert. Die bedingte Ausführung von Instruktionen entspricht dabei einem bedingten Sprung über diese Instruktionen. Sollen nur kurze Anweisungsfolgen bedingt ausgeführt werden, kann der notwendige Sprung die Ausführungszeit erheblich vergrößern. Ein Beispiel:

```
0    if(Bedingung){
1        a = a + b;
2    }else{
3        a = a - b;
4    }
```

Die Ausführungszeit der im Beispiel verwendeten arithmetischen Instruktionen sollte in den meisten Mikroarchitekturen bei einem Takt liegen. Abhängig von Mikroarchitektur und Speicherhierarchie wird die Ausführung des notwendigen Sprungs jedoch erheblich mehr Zeit in Anspruch nehmen. Um diesen Mehraufwand zu vermeiden, ist allen ARMv4-Instruktionen die bedingte Ausführbarkeit (*conditional execution*) gemein.
Die Bits 31:28 **aller** Instruktionen (Abbildung 1.5), das sogenannte *Bedingungsfeld* (*condition field*), codieren eine Bedingung, die durch den Wert des Condition-Codes erfüllt sein muss. Beispielsweise kann gefordert sein, dass das Z-Bit gesetzt ist. Ist die Bedingung nicht erfüllt, verändert die Instruktion den Prozessorzustand nicht und wird zu einem *NOP* (*no operation*). In Assemblerbefehlen wird das Bedingungsfeld durch ein Suffix am Befehlskür-

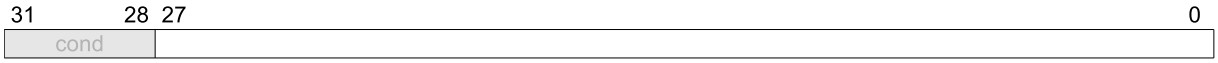


Abb. 1.5: Das Bedingungsfeld aller ARMv4-Instruktionen.

zel (*Mnemonic*) dargestellt, aus `B` (*branch*) für einen unbedingten Sprung wird z.B. `BEQ` (*branch equal*) oder `BNE` (*branch not equal*). Soll eine Instruktion unbedingt ausgeführt werden, ist im Bedingungsfeld die Bedingung *Always* codiert, die Assemblermnemonics erhalten dann üblicherweise keine Erweiterung, obwohl *AL* angehängt werden darf. Tabelle 1.5 fasst die möglichen Einträge im Bedingungsfeld, das jeweilige Assemblersuffix und die damit an die Statusbits gestellten Bedingungen zusammen.
Durch das Bedingungsfeld verringert sich die effektiv für die Codierung von Instruktionen zur Verfügung stehende Befehlswortbreite auf 28 Bit. Bereits in ARMv4 darf daher die Bedingung NV (*Never*) formal nicht mehr verwendet werden, die Wirkung ist nicht spezifiziert. Ab ARMv5 steht die Bedingung NV im Bedingungsfeld für neue, **unbedingt** auszuführende Instruktionen. Das Konzept der bedingten Befehlsausführung wurde also aufgeweicht, um Raum für weitere Instruktionscodes zu schaffen.
Eine Übersicht über die ARM-Assemblersprache und zahlreiche Programmierbeispiele enthält [8].

Bedingungsfeld	Mnemonic-erweiterung	Bedeutung	Bedingung an den Condition-Code
0000	EQ	EQual	Z = 1
0001	NE	Not Equal	Z = 0
0010	CS/HS	Carry Set / unsigned Higher or Same	C = 1
0011	CC/LO	Carry Clear / unsigned LOwer	C = 0
0100	MI	MInus / negative	N = 1
0101	PL	PLus / postive or zero	N = 0
0110	VS	V flag Set / overflow	V = 1
0111	VC	V flag Clear/ no overflow	V = 0
1000	HI	unsigned HIgher	C = 1 AND Z = 0
1001	LS	unsigned Lower or Same	C = 0 OR Z = 1
1010	GE	signed Greater than or Equal	N = V
1011	LT	signed Less Than	N != V
1100	GT	signed Greater Than or equal	Z = 0 AND (N = V)
1101	LE	signed Less than or Equal	Z = 1 OR (N != V)
1110	AL	ALways	beliebig
1111	NV	NeVer	unerfüllbar

Tab. 1.5: Bedeutung der Bedingungsfeld-Codes.

Verzicht auf Schiebeoperationen

Eine weitere Besonderheit von ARM ist der Verzicht auf dedizierte Schiebeoperationen (*Shifts*). Stattdessen ist es in einigen Instruktionen möglich, einen Shift oder eine Rotation auf einen der beteiligten Operanden anzuwenden, bevor dieser in die eigentliche Operation, z.B. eine Addition mit einem weiteren Operanden, eingeht. Je nach Rolle, die der Operand in der Instruktion einnimmt, wird dieser Teil des Instruktionsvektors als *Offset*, *Operand* oder *Operand 2* bezeichnet. Er umfasst aber immer die Instruktionsbits 11:0. Die folgenden Formate dieses Feldes treten neben weiteren Sonderfällen auf.

Immediate-Format (Abbildung 1.6) Operand 2 ist ein 8-Bit-Direktoperand. Er wird auf 32 Bit nullerweitert und um **das Doppelte** des im #rot-Feld dual codierten Betrages rechtsrotiert. Die Rotation erfolgt also um 0,2,...,30 Stellen.

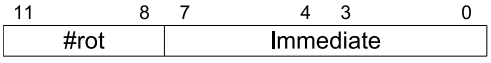


Abb. 1.6: Immediate-Format von Operand 2.

Register-Format (Abbildung 1.7) Operand 2 codiert eine Registeradresse (Rm). Auf den Inhalt des Registers wird in Abhängigkeit vom Sh-Feld eine der Operationen

- Sh = 00: LSL (Linksshift),
- Sh = 01: LSR (logischer Rechtsshift),
- Sh = 10: ASR (arithmetischer Rechtsshift) oder
- Sh = 11: ROR (Rechtsrotation)

angewendet.

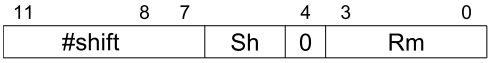


Abb. 1.7: Register-Format von Operand 2.

Die Stellenzahl des Shifts bzw. der Rotation ist im Feld #shift codiert. Dabei gelten einige Ausnahmen für #shift = `00000`. So wird die vorhandene Menge der Instruktionsvektoren besser ausgenutzt, denn #shift = `00000` bedeutete für jeden Wert des Sh-Feldes, dass der Operand nicht verändert wird. Es gilt für #shift = `00000` und die verschiedenen Operationen:

LSL
: Es findet ein Linksshift um 0 Stellen statt, also kein Shift.

LSR/ASR
: Es findet ein logischer/arithemtischer Rechtsshift um **32** Stellen statt.

ROR
: Es findet keine Rechtsrotation, sondern eine RXX-Operation (Rotate right with extend) um **eine** Stelle statt.

Alle Shifts und Rotationen setzen, sofern sie um mindestens eine Stelle durchgeführt werden, dass Carry-Bit neu. Dabei geht der vorherige Wert des Carry-Bits nicht in das Ergebnis ein. RXX bildet die einzige Ausnahme. Das Verhalten der RXX-Operation im Vergleich zur ROR-Operation verdeutlicht Abbildung 1.8: Die RXX-Operation verwendet

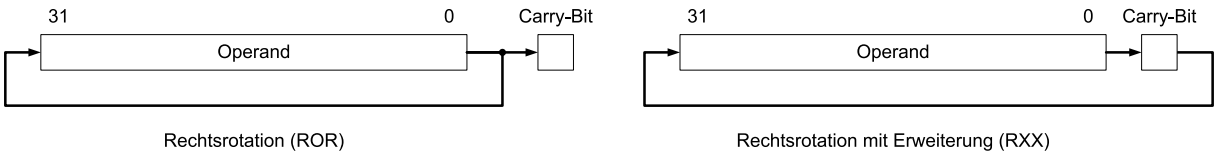


Abb. 1.8: Wirkung von ROR und RXX auf einen Registerinhalt und das Carry-Bit.

das Carry-Bit als zusätzliches, 33. Operandenbit und führt auf dem erweiterten Operanden eine Rechtsrotation um genau eine Stelle durch.

Für jeden Shift und jede andere Rotation entspricht das neue C-Bit der zuletzt aus dem Operanden geschobenen oder rotierten Stelle. Der übrige Condition-Code wird durch Operand 2 nicht beeinflusst.

Register-Register-Format (Abbildung 1.9) Operand 2 ist eine Registeradresse (Rm). Auf den Registerinhalt wird eine der Operationen LSL, LSR, ASR oder ROR in Abhängigkeit von Sh angewendet, die Stellenzahl ist im niederwertigsten Byte von Register Rs hinterlegt. Die Ausnahmen des Register-Formats gelten hier nicht, es gibt keine RXX-Operation. Es sind Shifts und Rotationen um 0 bis 255 Stellen möglich.

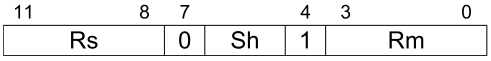


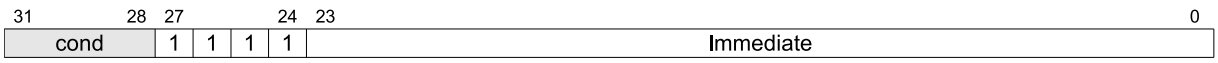
Abb. 1.9: Register-Register-Format von Operand 2.

Die Instruktionen werden im Folgenden nach ihrer Funktion in Klassen eingeteilt und klassenweise erläutert. Jede Klasse beinhaltet mindestens eine Instruktion. Zahlreiche Instruktionen unterscheiden sich in der Codierung und Wirkung nur marginal und werden daher, wo das sinnvoll ist, gemeinsam betrachtet.

1.8.4 Ausnahmeverursachende Instruktionen

Instruktionen dieser Klasse verursachen den Wechsel in einen Ausnahmemodus, ARMv4 kennt nur eine solche Instruktion.

SWI - *Software Interrupt*



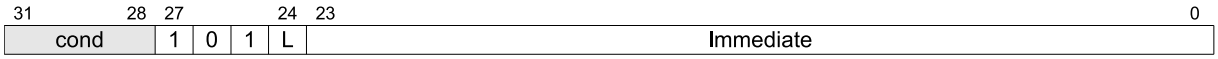
Die Ausführung der Instruktion verursacht einen Software Interrupt mit der in 1.6 beschriebenen Wirkung. Das Immediate-Feld ist nicht für den Prozessor von Bedeutung, sondern für den Exception-Handler. Er kann aus dem Link-Register (R14) die Adresse der verursachenden SWI-Instruktion ermitteln, die Instruktion als Datum in ein Register laden und das Immediate-Feld auswerten. Dieser Mechanismus ist für den Aufruf von Betriebssystemfunktionen aus einem Programm im User-Modus heraus vorgesehen.

1.8.5 Sprungbefehle

Diese Klasse fasst alle Instruktionen zusammen, die dediziert der Programmverzweigung dienen und Sprünge nicht nur als Seiteneffekt verursachen. ARMv4 fordert die Unterstützung von zwei Sprungbefehlen, ein weiterer kommt in ARMv4T hinzu und wird hier nicht

berücksichtigt.

B - *Branch*
BL - *Branch and Link*



Der 24 Bit Direktoperand wird als Zweierkomplementzahl interpretiert, um zwei Stellen nach links geschoben, vorzeichenerweitert und auf den PC addiert. Durch das Schieben entspricht der Direktoperand einer Wortadresse und wegen der Vorzeichenerweiterung sind Sprünge relativ zum PC zu höheren oder niedrigeren Adressen im Bereich $[PC - 2^{25}, PC + (2^{25} - 1)]$ möglich. Der für die Addition verwendete Inhalt des PC entspricht der Adresse des Sprungbefehls + 8. Das Verhalten bei Überschreitung der Adressraumgrenzen durch die Addition auf den PC ist nicht spezifiziert.

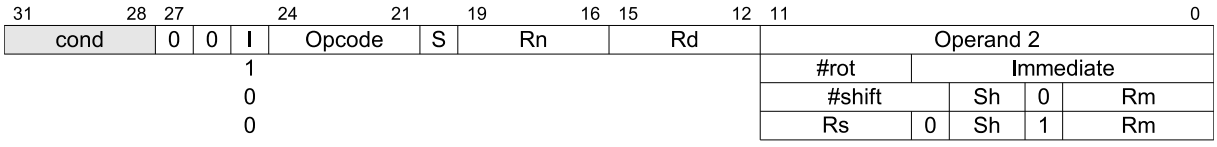
L

Die Adresse der auf den Sprung folgenden Instruktion (effektiv die Adresse des Sprungbefehls + 4) wird in R14 abgelegt, wenn L gesetzt ist. Dies entspricht dem Sichern der Rücksprungadresse im Rahmen eines Unterprogrammaufrufs.

1.8.6 Arithmetisch-logische Instruktionen

Die Instruktionen dieser Klasse dienen der arithmetischen oder logischen Verknüpfung von Registerinhalten. Ergebnisse werden wieder in einem Register abgelegt und können zusätzlich die Statusbits beeinflussen. Diese Klasse umfasst 16 Instruktionen.

AND - *AND*
EOR - *Exclusive OR*
SUB - *Subtract*
RSB - *Reverse Subtract*
ADD - *Add*
ADC - *Add with Carry*
SBC - *Subtract with Carry*
RSC - *Reverse Subtract with Carry*
TST - *Test*
TEQ - *Test Equivalence*
CMP - *Compare*
CMN - *Compare Negative*
ORR - *Logical OR*
MOV - *Move*
BIC - *Bit Clear*
MVN - *Move not*



Die 16 Instruktionen unterscheiden sich nur in der Codierung des Opcode-Feldes.

Rn
: Rn ist der erste Quelloperand der Verknüpfung.

Rd
: Rd ist die Adresse des Zielregisters der Verknüpfung.

Opcode
: Das Opcode-Feld codiert, welche der 16 arithmetischen oder logischen Verknüpfungen auf den Inhalt von Rn und Operand 2 angewendet wird.

S
: Ist S gesetzt, so werden die Statusbits auf Basis des Operationsergebnisses aktualisiert. Arithmetische Verknüpfungen setzen den gesamten Condition-Code auf Basis des Ergebnisses neu, logische Verknüpfungen aktualisieren nur N und Z.

In welchem Format Operand 2 zu interpretieren ist, hängt mindestens vom I-Bit, ggf. aber auch von den Instruktionsbits 4 und/oder 7 ab.

I = 1
: Immediate-Format

I = 0, Bit 4 = 0
: Register-Format

I = 0, Bit 4 = 1, Bit 7 = 0
: Register-Register-Format

I = 0, Bit 4 = 1, Bit 7 = 1
: Dieses Befehlswort entspricht keiner arithmetisch-logischen Instruktion, sondern gehört in den Bereich der Datentransferinstruktionen.

Abweichende Bedeutung des S-Bits bei Vergleichsoperationen Die vier Opcodes `1000, 1001, 1010, 1011` (zusammengefasst: `10--`) codieren reine Vergleichsoperationen. Hierbei werden Rn und Operand 2 miteinander verknüpft, die Statusbits aktualisiert und das Ergebnis **nicht** nach Rd zurückgeschrieben. Das S-Bit ist per Konvention immer gesetzt.
Für S = 0 und Opcode = `10--` handelt es sich nicht um eine arithmetisch-logische Instruktion, für die tatsächliche Bedeutung ist zusätzlich das I-Bit zu berücksichtigen (Tabelle 1.6).

Abweichende Bedeutung des S-Bits wenn Rd = PC Die vier Vergleichsoperationen werten die Zielregisteradresse (Rd) nicht aus (dort sollte in diesem Fall `0000` eingetragen sein). Für die übrigen Opcodes ergibt sich ein Sonderfall, wenn Rd = `1111`, d.h. R15 (der PC) ist. Das Schreiben in den PC verursacht einen Sprung, das Sprungziel entspricht dem Ergebnis der Verknüpfung von Rn und Operand 2. Mit S = 1 wird zusätzlich der Inhalt des SPSR des aktuellen Betriebsmodus in das CPSR kopiert. Dies entspricht dem Rücksprung

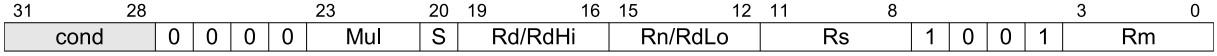
S	I	Opcode	Bedeutung
0	0	10--	Undefiniert, verursacht Undefined Exception.
0	1	10-0	Undefiniert, verursacht Undefined Exception.
0	1	10-1	Codiert die MSR-Instruktion, siehe 1.8.9.

Tab. 1.6: Abweichende Bedeutung von Instruktionscodes für S = 0.

aus einer Ausnahmebehandlung. Das Verhalten in den Modi User und System, die nicht über ein SPSR verfügen, ist nicht spezifiziert.

1.8.7 Multiplikationsinstruktionen

MUL - *Multiply*
MLA - *Multiply Accumulate*



ARMv4 umfasst zwei obligatorische 32-Bit-Multiplikationsinstruktionen sowie vier 64-Bit-Multiplikationsinstruktionen. Das **Mul**-Feld codiert die Art der Multiplikationsinstruktionen. Zwei Codes ist keine Instruktion zugeordnet. Tabelle 1.7 beschreibt die Wirkung der unbedingt unterstützten MUL- und MLA-Instruktionen. Beide legen die niederwertigen 32 Bit des Multiplikationsergebnisses im Zielregister **Rd** ab. Die Instruktionen aktualisieren den Condition Code, wenn das **S**-Bit gesetzt ist. Die Felder **Rd** und **Rn** bilden gemeinsam das Ziel von 64-Bit-Multiplikationen und werden deshalb auch als **RdHi** und **RdLo** bezeichnet.

Mul	Mnemonic	Wirkung
000	MUL	Rd := (Rm · Rs)[31:0]
001	MLA	Rd := (Rm · Rs + Rn)[31:0]

Tab. 1.7: Wirkung und Codierung der 32-Bit-Multiplikationsinstruktionen.

Aktualisierung der Statusbits bei gesetztem S-Bit (32-Bit-Multiplikation)

N

Das neue N-Bit entspricht dem höchsten Ergebnisbit (Rd[31]).

Z

Das Z-Bit wird gesetzt, falls das 32-Bit-Ergebnis gleich 0 ist.

C

ARMv4 spezifiziert die Wirkung der Multiplikationsinstruktionen auf das C-Bit nicht. Ab ARMv5 dürfen sie das Bit nicht verändern.

V

Das V-Bit wird durch die Instruktionen nicht verändert.

Operandeneinschränkung Das Verhalten bei Verwendung von R15 als Operandenregister oder Zielregister ist unspezifiziert. Das Ergebnis der Operation ist ebenfalls unspezifiziert falls Rd und Rm identisch sind.

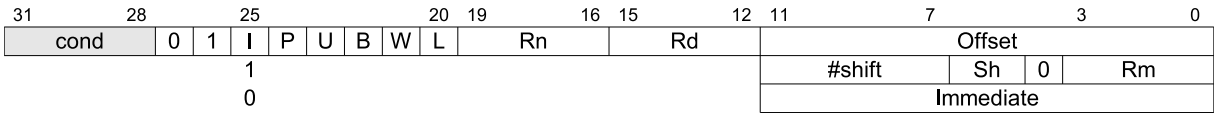
Fakultative Multiplikationsinstruktionen Die übrigen vier Multiplikationsinstruktionen müssen ARMv4-Prozessoren nicht unterstützen, weshalb sie an dieser Stelle nicht vorgestellt werden. Der Prozessor wird auf diese Befehle ebenso mit einer Undefined Excepion reagieren wie auf die beiden völlig undefinierten Codes des Mul-Feldes.

1.8.8 Datentransferbefehle

Instruktionen dieser Klasse transferieren Daten in Byte-, Halbwort- oder Wortgröße zwischen Registern und Speicher, der hier neben dem Arbeitsspeicher auch speicherbezogene Ein-/Ausgabegeräte umfasst. 14 Instruktionen dieser Klasse sind zu implementieren.

Wort- und Bytetransferinstruktionen

LDR	-	*Load Register*
LDRT	-	*Load Register with Translation*
LDRB	-	*Load Register Byte*
LDRBT	-	*Load Register Byte with Translation*
STR	-	*Store Register*
STRT	-	*Store Register with Translation*
STRB	-	*Store Register Byte*
STRBT	-	*Store Register Byte with Translation*



Diese Instruktionen kopieren ein Wort oder ein auf 32 Bit nullerweiteres Byte aus dem Speicher in ein Register (load) oder den Inhalt eines Registers in ein Wort des Arbeitsspeichers oder das niederwertigste Byte eines Registerinhalts in ein Byte des Arbeitsspeichers (store). Durch Addition eines Offsets auf den Inhalt eines Basisregisters oder Subtraktion des Offsets vom Inhalt des Basisregisters wird die Adresse des Speicherzugriffs gebildet. Das Basisregister kann parallel zum Speicherzugriff mit einem neuen Wert aktualisiert werden.

B

B legt fest, ob der Speicherzugriff in Wortgröße (B = 0) oder in Bytegröße (B = 1) stattfindet.

L

L legt fest, ob der Speicherzugriff schreibend (L = 0) oder lesend (L = 1) erfolgt.

Rd

Rd ist das Quellregister eines Schreibzugriffs bzw. das Zielregister eines Lesezugriffs.

Rn

Rn ist das Basisregister der Speicherzugriffsadresse.

Offset

Der Offset der Adressberechnung, der gemeinsam mit dem Inhalt von Rn die Speicheradresse bildet.

I

I bestimmt die Interpretation des Offset-Feldes. Für I = 0 entspricht der Offset einem 12-Bit-Direktoperanden, welcher nullerweitert und mit dem Basisregisterinhalt verknüpft wird. Für I = 1 entspricht das Offset-Feld dem Register-Format.

PU

P (pre-index/post-index) und U (up/down) legen die Adressierungsart des Speicherzugriffs und damit die Kombination von Rn und Offset zur Bildung der Speicheradresse fest. Auch der Wert, der ggf. in das Basisregister zurückgeschrieben wird (Aktualisierung des Basisregisters), hängt von der Adressierungsart ab. Tabelle 1.8 fasst die Wirkung aller Adressierungsarten zusammen.

W

Für P = 1 steuert das W-Bit, ob die aus Rn und Offset gebildete aktualisierte Basisadresse nach Rn zurückgeschrieben (*base register write-back*, W = 1) oder verworfen (W = 0) wird. Die Verwendung der Adressierungsarten Postdekrement und Postinkrement mit einem Offset, aber ohne das Rückschreiben der modifizierten Basisadresse wäre offensichtlich sinnlos, da Rn unmodifiziert als Speicherzugriffsadresse verwendet wird und sich der Offset erst in Rn* auswirkt. Daher gilt für P = 0: Die modifizierte Basisadresse wird unabhängig von W nach Rn zurückgeschrieben (*auto-indexing*). Das W-Bit ist damit frei für eine alternative Funktion.
Für P = 0 und W = 0 findet der Speicherzugriff mit der Speichersicht des aktuellen Betriebsmodus statt. Bei P = 0 und W = 1 findet der Speicherzugriff mit der Speichersicht des User-Modus statt, es handelt sich hier um die vier Instruktionen mit T-Suffix (T = Translation). Die Speichersicht hängt davon ab, welcher Modus den Komponenten am Prozessor-Datenbus präsentiert wird. Im User-Modus kann beispielsweise der Zugriff auf Ein-/Ausgabegeräte gesperrt oder eine andere Adressübersetzung aktiv sein. Bei einer sehr einfachen Speicherhierarchie ohne

jede Form von Speicherzugriffsschutz (Memory Protection Unit, MPU) oder Adressübersetzung (Memory Management Unit, MMU) ergibt sich durch die veränderte Speichersicht kein Unterschied.

PU	Adressierungsart	Zugriffsadresse	aktualisierte Basisadresse
10	decrement before	addr := Rn − Offset	Rn* := Rn − Offset
11	increment before	addr := Rn + Offset	Rn* := Rn + Offset
00	decrement after	addr := Rn	Rn* := Rn − Offset
01	increment after	addr := Rn	Rn* := Rn + Offset

Tab. 1.8: Berechnung von Zugriffsadresse und aktualisierter Basisadresse.

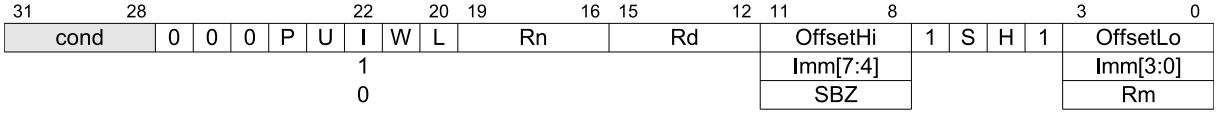
Adressanpassungen ARMv4 unterstützt keine unausgerichteten Zugriffe auf Worte im Speicher. Dennoch können, um auch Byte-Zugriffe zu ermöglichen, entsprechende Adressen aus Rd und Offset gebildet werden, sodass dieser Fall berücksichtigt werden muss. Falls ein Lesezugriff in Wortgröße erfolgt (LDR, LDRT), die Adresse aber keine Wortadresse ist, so ist das gelesene Wort so zu rotieren, dass das adressierte Byte – jede Nicht-Wortadresse kann als Byteadresse betrachtet werden – in der niederwertigsten Byteposition des Registers Rd abgelegt wird. Ein von Adresse 0x00000002 gelesenes Wort würde beispielsweise um zwei Byte rechtsrotiert.
Bei Schreibzugriffen in Wortgröße (STR, STRT) werden generell die beiden niederwertigsten Adressbits der errechneten Adresse maskiert (auf 00 gesetzt), sodass der Zugriff aus Sicht des Speichers ausgerichtet erscheint. Die Maskierung erfolgt lediglich für das Speichersystem, die aktualisierte Basisadresse wird ohne Maskierung nach Rn geschrieben.

Operandeneinschränkungen Falls Rd und Rn identisch sind und die modifizierte Basisadresse zurückgeschrieben wird, ist nicht spezifiziert, welcher Wert im Register erscheint. Die Verwendung von R15 für Rn mit gleichzeitigem Rückschreiben der Basisadresse ist nicht spezifiziert. Nicht spezifiziert ist auch das Laden eines Bytes nach R15. Das Schreiben des PCs in den Speicher ist hinsichtlich des Wertes des PCs als implementierungsabhängig (STR, STRT) bzw. unspezifiziert (STRB, STRBT) gekennzeichnet.

Transferbefehle für Halbworte und vorzeichenbehaftete Bytes

LDRH - *Load Register Halfword*
LDRSH - *Load Register Signed Halfword*
LDRSB - *Load Register Signed Byte*
STRH - *Store Register Halfword*

Aus dem Speicher geladene Halbworte oder Bytes müssen auf die 32 Bit eines Registers erweitert werden. Die unter 1.8.8 vorgestellten Instruktionen führen dazu eine Nullerweiterung bei Bytetransfers durch. Damit kann kein kleines, vorzeichenbehaftetes Datum



aus dem Speicher gelesen und unmittelbar verwendet werden. Die neuen Instruktionen ergänzen nun die Möglichkeit, vorzeichenbehaftete Bytes sowie Halbworte zu laden und Halbworte zu speichern. Da Schreibzugriffe auf den Speicher in Byte- und Halbwortquanta möglich sind, werden keine **Speicher**instruktionen mit Vorzeichenerweiterung benötigt.
Die Instruktionen sind in ihrer Wirkung jenen aus Abschnitt 1.8.8 ähnlich, jedoch deutlich anders codiert. Sie wurden erst mit ARMv4 eingeführt und besetzen eine bis dato offene Lücke in den Instruktionscodes, die nur bedingte Ähnlichkeit mit älteren Datentransferinstruktionen hat.
Rn, **Rd** und **L** entsprechen in ihrer Bedeutung den bereits vorgestellten Transferinstruktionen.

I

I = 0: Der Offset für die Adressberechnung ist der Inhalt des durch Rm codierten Registers.
I = 1: OffsetHi und OffsetLo bilden gemeinsam einen 8-Bit-Direktoperanden, der als vorzeichenloser Offset verwendet wird.

PU

P und U codieren die Adressierungsart ebenso wie in 1.8.8. Es gilt auch hier, dass bei P = 0 unbedingt das Basisregister aktualisiert wird.

W

W gibt für P = 1 an, ob das Basisregister aktualisiert wird (W = 1). Das Verhalten bei PW = 00 ist nicht spezifiziert.

S

S = 0: Nullerweiterung von gelesenen Halbworten/Bytes auf 32 Bit
S = 1: Vorzeichenerweiterung von gelesenen Halbworten/Bytes auf 32 Bit

H

H = 0: Bytetransfer
H = 1: Halbworttransfer

Die Wirkung der Kombination SH = 00 (Unsigned Byte) ist bereits durch die gewöhnlichen Bytetransferinstruktionen abgedeckt, daher zeigt die Kombination hier keine Transferinstruktion an. Stattdessen handelt es sich dann um eine der in 1.8.7 vorgestellten Multiplikationsinstruktionen.
Die Kombination LS = 01 (Store Signed) ist undefiniert und erzeugt eine entsprechende Ausnahme.

Hinweis Ab ARMv5TE unterliegen L, S und H einer neuen Interpretation und LS = 01 zeigt Load/Store Doubleword-Instruktion an.

Unausgerichtete Speicherzugriffe Das Verhalten bei unausgerichteten Halbwortzugriffen ist nicht spezifiziert.

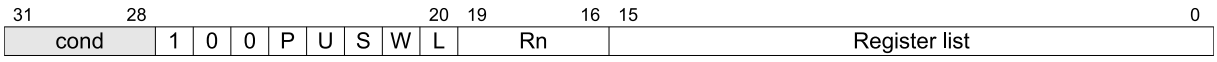
Operandeneinschränkungen Für alle vier Instruktionen gilt, dass das Verhalten nicht spezifiziert ist, wenn der PC für Rd verwendet wird, oder Rd und Rn auf dasselbe Register verweisen und zusätzlich das Basisregister aktualisiert werden soll. Darüber hinaus gelten für alle Instruktionen Einschränkungen und Vorgaben, die vom Wert der Bits I, P und W abhängen (Tabelle 1.9). Zu jeder Kombination ist eine Bezeichnung dieser speziellen Adressierungsart gemäß der ISA [1, A5.3] angegeben.

IPW	Adressierungsart	Einschränkungen/Vorgaben
110	Immediate offset	R15 darf als Rn verwendet werden, gelesen wird die Adresse der Instruktion + 8.
010	Register offset	R15 darf als Rn verwendet werden, gelesen wird die Adresse der Instruktion + 8. Die Verwendung von R15 als Rm ist nicht spezifiziert.
111	Immediate pre-indexed	Die Verwendung von R15 als Rn ist nicht spezifiziert.
011	Register pre-indexed	Die Verwendung von R15 als Rn oder die Verwendung identischer Register für Rn und Rm ist nicht spezifiziert.
101	Immediate post-indexed	Die Verwendung von R15 als Rn ist nicht spezifiziert.
001	Register post-indexed	Die Verwendung von R15 als Rn oder die Verwendung identischer Register für Rn und Rm ist nicht spezifiziert.

Tab. 1.9: Operandeneinschränkungen der Halbwort- und Signed Byte Transferinstruktionen.

Transferbefehle für mehrere Register

STM - *Store Multiple*
LDM - *Load Multiple*



Die Instruktionen für den Transfer mehrerer Register kopieren eine beliebige, nicht leere Teilmenge aller Register in den Speicher an aufeinanderfolgende Wortadressen oder 1 bis 16 Worte von hintereinanderliegenden Adressen aus dem Speicher in beliebige Register. Die Register können statt denen des aktuellen Betriebsmodus auch jene des User-Modus

sein. LDM-Instruktionen können daneben für die Rückkehr aus einer Ausnahmebehandlung verwendet werden.

Rn

Basisregister der ersten Zugriffsadresse

L

Unterscheidet zwischen Schreibzugriffen (L = 0, STM) und Lesezugriffen (L = 1, LDM)

Register list

Ein 16-Bit-Vektor. Jedes Bit ist einer der 16 ARM-Registeradressen zugeordnet, Bit 0 steht für Register 0, Bit 15 für den PC. Jedes Register, dessen Bit gesetzt ist, wird entweder mit einem Wort aus dem Speicher geladen oder in den Speicher geschrieben. Das Verhalten bei einer leeren Registerliste ist nicht spezifiziert, jede nicht leere Teilmenge aller Register darf verwendet werden. Das Register mit dem niedrigsten Index wird zuerst geladen oder in den Speicher kopiert, und dann mit aufsteigendem Index fortgefahren.

W

Mit W = 1 wird das Basisregister aktualisiert. Der nach Rn zurückgeschriebene Wert ist von der verwendeten Adressierungsart und der Menge der übertragenen Worte abhängig.

PU

P-Bit und U-Bit codieren die Adressierungsart. Anders als die Bezeichnungen der Adressierungsarten (*increment/decrement before/after*) vermuten lassen, wird die Speicheradresse nicht wahlweise vor oder nach jedem Speicherzugriff inkrementiert oder dekrementiert. Stattdessen ist die niedrigste Adresse zu ermitteln, die auftreten würde, wenn der Prozessor so verführe. An dieser Adresse findet der erste Speicherzugriff statt, für jeden weiteren Zugriff wird die Wortadresse inkrementiert.
Beispiel: In der Registerliste seien drei Bits gesetzt, es finden also drei Speicherzugriffe statt. Die Adressierungsart sei PU = `00` (*decrement after*). Bei üblichem Verständnis entspräche die erste Zugriffsadresse (dem Wert in) Rn, die zweite $Rn - 4$, die dritte $Rn - 8$. Nach dem letzten Speicherzugriff muss die Adresse noch einmal dekrementiert und dieser Wert gegebenenfalls nach Rn zurückgeschrieben werden. Es gilt also: $Rn* := Rn - 12$.
Da der erste Speicherzugriff aber an der niedrigsten Adresse stattfindet, wird er an $Rn - 8$ (anders ausgedrückt: $Rn - (4 \cdot 3) - 4$) durchgeführt. Es folgen Zugriffe an $Rn - 4$ und Rn. Die aktualisierte Basisadresse ist weiterhin $Rn - 12$ (anders ausgedrückt: $Rn - (4 \cdot 3)$).

Die Tabellen 1.10 und 1.11 geben für alle Adressierungsarten an, wie die erste Zugriffsadresse und die aktualisierte Basisadresse für eine beliebige Menge von Registern (#Register) zu ermitteln ist und welches die Adresse des letzten Speicherzugriffs bei der Ausführung der Instruktion ist.

S

Mit S = 0 ist die Registerliste auf Register des aktuellen Betriebsmodus bezogen. Mit S = 1 bezieht sie sich auf die Register des User-Modus, Rn aber weiterhin auf ein Register im aktuellen Betriebsmodus. Auf diese Weise können User-Register aus einem Ausnahmemodus heraus gesichert (oder geladen) werden. Die Wirkung von S = 1 in den Modi User und System ist nicht spezifiziert.

PU	Adressierungsart	erste Zugriffsadresse	letzte Zugriffsadresse
00	decrement after	Rn − (4 · #Register) + 4	Rn
10	decrement before	Rn − (4 · #Register)	Rn − 4
11	increment before	Rn + 4	Rn + (4 · #Register)
01	increment after	Rn	Rn + (4 · #Register) − 4

Tab. 1.10: Erste und letzte Speicheradresse in Abhängigkeit von der Adressierungsart bei STM- und LDM-Instruktionen.

PU	Adressierungsart	aktualisierte Basisadresse
00	decrement after	Rn* := Rn − (4 · #Register)
10	decrement before	Rn* := Rn − (4 · #Register)
11	increment before	Rn* := Rn + (4 · #Register)
01	increment after	Rn* := Rn + (4 · #Register)

Tab. 1.11: Bestimmung der aktualisierten Basisadresse in Abhängigkeit von der Adressierungsart bei STM- und LDM-Instruktionen.

Adressausrichtung und Operandeneinschränkungen Die beiden niederwertigsten Adressbits werden ignoriert, sodass auch Nicht-Wortadressen zu Wortzugriffen führen. Die Verwendung von PC als Rn ist nicht spezifiziert.

Für LDM und STM gelten diverse Einschränkungen und Seiteneffekte, die im Folgenden getrennt betrachtet werden.

STM Die Kombination SW = 11, also ein Zugriff auf die Register des User-Modus mit gleichzeitiger Aktualisierung des Basisregisters, ist nicht spezifiziert. Der Grund für diese Einschränkung liegt vermutlich darin, dass andernfalls unmittelbar hintereinander oder gar gleichzeitig Schreibzugriffe auf Register unterschiedlicher Betriebsmodi notwendig wären, was Prozessorsteuerung und Entwurf des Registerspeichers verkompliziert.
Auf STM-Instruktionen mit S = 1 folgt per Konvention keine Instruktion, die auf andere Register als R0 - R7 und R15 zugreift.
Wird R15 in der Registerliste angegeben, ist der gespeicherte Wert implementierungsabhängig. Ist Rn Teil der Registerliste aber nicht das Register mit dem niedrigsten Index

und wird gleichzeitig das Basisregister (gerade Rn) aktualisiert, ist nicht spezifiziert, welcher Wert für Rn in den Speicher geschrieben wird. Ist Rn in der Registerliste und das Register mit dem niedrigsten Index, so muss es sich um den unveränderten Wert von Rn handeln.

LDM Falls Rn aktualisiert werden soll und gleichzeitig Teil der Registerliste ist, ist nicht spezifiziert, ob das Datum aus dem Speicher oder die aktualisierte Basisadresse nach Rn geschrieben wird.
Die Verwendung von Bit 15 der Registerliste ggf. in Verbindung mit dem S-Bit steuert spezielle Effekte der LDM-Instruktion:

Wenn PC nicht Teil der Registerliste ist:

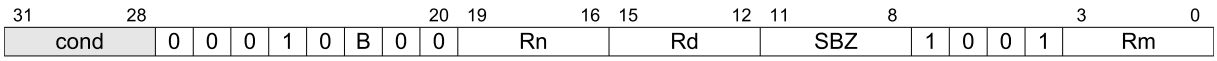
> Mit S = 1 bezieht sich die Registerliste (nicht aber Rn) auf die Register des User-Modus. Es folgt per Konvention keine Instruktion, die auf andere Register als R0-R7 und R15 zugreift. Das Verhalten bei gleichzeitig gesetztem W-Bit ist auch hier nicht spezifiziert, ebenso wenig das Verhalten bei gesetztem S-Bit im User-Modus.

Falls PC Teil der Registerliste ist:

> Ist das S-Bit nicht gesetzt, verursacht das Laden des letzten Wertes einen Sprung. Falls das S-Bit gesetzt ist, erfolgt ein Sprung und zusätzlich wird das SPSR des aktuellen Modus in das CPSR kopiert. Dieser Mechanismus dient dem Rücksprung aus einer Ausnahmebehandlung. Die Registerliste bezieht sich auf die Register des aktuellen Betriebsmodus und nicht auf den User-Modus. Das parallele Rückschreiben der Basisadresse ist in beiden Fällen möglich.

Austausch eines Registerinhalts und eines Speicherwortes oder -bytes

SWP - *Swap*
SWPB - *Swap Byte*



Die Swap-Instruktion dient dem nicht unterbrechbaren (*atomaren*) Austausch eines Registerinhalts mit einem Speicherwort in zwei aufeinanderfolgenden Speicherzugriffen. Der erste Speicherzugriff erfolgt lesend, der zweite schreibend. Die Instruktion dient damit der Realisierung von Semaphoren.

Rn
: Rn ist das Register, welches die Basisadresse beider Speicherzugriffe enthält.

Rd

Rd ist das Zielregister des Lesezugriffs.

Rm

Rm ist das Quellregister, aus dem der Operand für den Schreibzugriff entnommen wird.

B

Das B-Bit legt fest, ob Worttransfers (B = 0) oder Bytetransfers durchgeführt werden, und unterscheidet damit die SWP-Instruktion (B = 0) von der SWPB-Instruktion (B = 1). Das im Rahmen der SWPB-Instruktion aus dem Speicher gelesene Byte wird vor dem Ablegen in Rd nullerweitert.

Die beiden nacheinander durchgeführten Speicherzugriffe haben für B = 0 die Form:

Rd := Mem[Rn];
Mem[Rn] := Rm

Der Schreibzugriff auf den Speicher erfolgt nur dann, wenn der vorhergehende Lesezugriff erfolgreich war und keine Data Abort Exception verursacht hat. Das aus dem Speicher geladene Datum wird erst nach Rd geschrieben, wenn auch der zweite Speicherzugriff ohne Ausnahme abgeschlossen wurde.

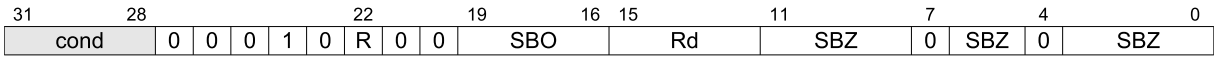
Einschränkungen und Adressausrichtung Rm und Rn dürfen auf dasselbe Register verweisen, müssen aber jeweils verschieden von Rd sein. Die Verwendung des PCs ist für keines der Register spezifiziert. Bei B = 0 gelten für nicht ausgerichtete Adressen dieselben Regeln wie für die Instruktionen STR und LDR.

1.8.9 Instruktionen für den Transfer des Statusregisterinhalts

Diese Instruktionen kopieren den Inhalt des Statusregisters in ein Register, den Inhalt eines Registers in das Statusregister oder einen Direktoperand in das Statusregister. Zwei Instruktionen gehören zu dieser Klasse.

Lesen des Statusregisters

MRS - *Move PSR to general-purpose register*



Die MRS-Instruktion kopiert das CPSR oder das SPSR des aktuellen Modus in ein Register.

Rd

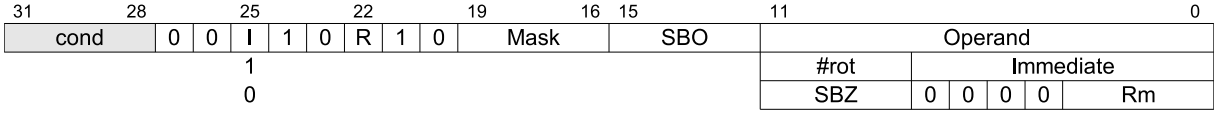
Rd ist das Zielregister, in das der Inhalt des Statusregisters kopiert werden soll. Die Verwendung von R15 ist nicht spezifiziert.

R

R = 0: Gelesen wird das CPSR.
R = 1: Gelesen wird das SPSR, das Verhalten des Prozessors in den Modi User und System ist für diesen Fall nicht spezifiziert.

Schreiben des Statusregisters

MSR - *Move to Status Register from ARM Register*



I

I = 1: In das Statusregister wird ein 8-Bit-Direktoperand geschrieben, der auf 32 Bit nullerweitert und um $2 \cdot$ #rot Stellen rechtsrotiert wird.
I = 0: In das Statusregister wird der Inhalt von Rm geschrieben.

R

Mit R = 0 erfolgt der Schreibzugriff in das CPSR, mit R = 1 in das SPSR des aktuellen Modus. Im User- oder System-Modus ist das Verhalten bei Zugriff auf das SPSR nicht spezifiziert, da diese Modi über kein SPSR verfügen.

Mask

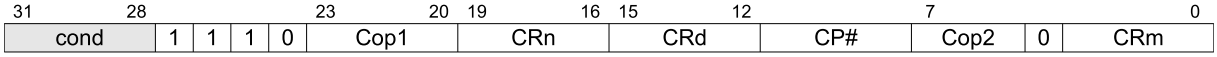
Jedes der vier Maskenbits ist einem Byte des Statusregisters zugeordnet, Maskenbit 3 (Bit 19 des Befehlswortes) Byte 3, Maskenbit 2 Byte 2 usw. Schreibzugriffe auf das CPSR/SPSR wirken nur auf die Bytes des Statusregisters, deren Maskenbit gesetzt ist. Im User-Modus werden die Maskenbits 2, 1 und 0 ignoriert, Schreibzugriffe sind nur auf das höchstwertige Byte des Statusregisters möglich. Schreibzugriffe auf alle in ARMv4 nicht verwendeten Bits innerhalb des Statusregisters werden in jedem Betriebsmodus ignoriert. Da die Bytes 1 und 2 innerhalb des Statusregisters keine definierten Informationen enthalten, sind auch die Maskenbits 1 und 2 de facto wirkungslos.

1.8.10 Coprozessorinstruktionen

Die Instruktionen dieser Klasse tauschen Inhalte von ARM-Registern mit Coprozessorregistern aus, verursachen Arbeitsspeicherzugriffe eines Coprozessors oder die Verknüpfung von Operanden aus Coprozessorregistern. ARMv4 enthält 5 Instruktionen dieser Klasse.

Datenverarbeitung im Coprozessor

CDP - *Coprocessor Data Processing*



CDP-Instrukionen steuern die Datenverarbeitung in Coprozessoren, es findet dabei kein Operanden- oder Ergebnisaustausch mit dem ARM-Kern und auch kein Speicherzugriff statt. Den Feldern Cop1, Cop2, CRn, CRd und CRm ist keine feste Bedeutung zugeordnet, sie werden vom ARM-Kern ignoriert. ARM empfiehlt die unten beschriebene Verwendung.

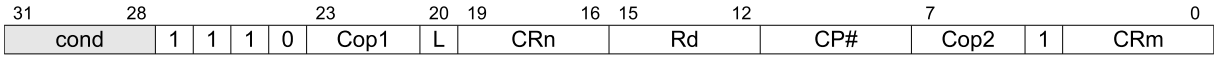
Cop1, Cop2
: Beide Felder gemeinsam codieren die Operation, die im Coprozessor ausgeführt werden soll.

CRn, CRm
: Adressen von Coprozessorregistern, die den ersten und zweiten Operanden enthalten.

CRd
: Zielregister der Operation innerhalb des Coprozessores.

Datentransfer zwischen Coprozessor und ARM-Kern

MRC - *Move ARM Register from Coprocessor*
MCR - *Move to Coprocessor from ARM Register*



Coprozessoren können lesend und schreibend auf ARM-Register zugreifen sowie den Condition-Code überschreiben.
Zwischen ARM-Kern und Coprozessor übertragene Daten dürfen im Coprozessor manipuliert werden, bevor sie in das Zielregister im Coprozessor oder im ARM-Kern geschrieben werden. Den Feldern Cop1, Cop2, CRn und CRm ist erneut keine feste Bedeutung zugeordnet. ARM empfiehlt die unten genannte Verwendung.

L
: L = 1: MRC-Instrukion, ein ARM-Register wird vom Coprozessor beschrieben.
L = 0: MCR-Instruktion, ein ARM-Registerinhalt wird zum Coprozessor übertragen.

Rd
: Rd ist das am Transfer beteiligte ARM-Register. Die Verwendung von R15 gemeinsam mit L = 0 ist nicht spezifiziert. Mit Rd = R15 und L = 1 wird nicht in den PC geschrieben und kein Sprung durchgeführt. Stattdessen werden die Bits 31:28 des an den ARM-Kern übertragenen Datums in den Condition-Code geschrieben.

CRn
: Das Quell- oder Zielregister des Datentransfers im Coprozessor.

CRm

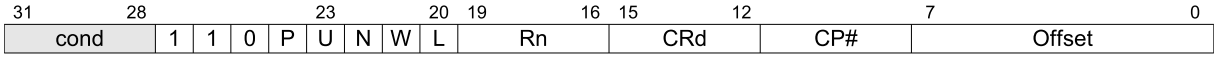
Ein weiteres Quell- oder Zielregister im Coprozessor. Beispielsweise ist es denkbar, erst die Inhalte von CRn und CRm im Coprozessor zu addieren und anschließend das Ergebnis nach Rd zu schreiben.

Cop1, Cop2

Beide Felder gemeinsam codieren eine Operation des Coprozessors, die dieser vor der Übertragung eines Datums zum ARM-Kern bzw. nach Übertragung aus dem ARM-Kern durchführen kann. Grundsätzlich dürfte die Operation auch vollständig unabhängig vom übertragenen Datum sein.

Datentransfer zwischen Coprozessor und Speicher

LDC - *Load Coprocessor*
STC - *Store Coprocessor*



Ein Coprozessor kann zwischen ein und 16 Worte hintereinander in den Speicher schreiben oder aus dem Speicher lesen. Die Zugriffsadressen werden dabei vom ARM-Kern erzeugt. Die erste Zugriffsadresse wird aus einem ARM-Register und einem Offset gebildet, weitere Zugriffe erfolgen durch permanentes Inkrementieren dieser Adresse um 4. Die Anzahl der Speicherzugriffe ist nicht in der Instruktion codiert, stattdessen signalisiert der Coprozessor das Ende des Speicherzugriffs. Wie die Worte im Coprozessor verarbeitet und in Registern abgelegt werden, ist nicht vorgeschrieben und hängt folglich von der Implementierung des Coprozessors ab. Es wäre prinzipiell möglich, alle gelesenen Worte hintereinander in einem einzigen Register abzulegen, denn Coprozessorregister dürfen beliebig breit ausfallen.

L

L = 0: STC-Instruktion, der Coprozessor schreibt in den Speicher Worte an aufeinanderfolgende Wortadressen.
L = 1: LDC-Instruktion, der Coprozessor liest Worte von aufeinanderfolgenden Wortadressen aus dem Speicher.

N

Das Bit ist coprozessorspezifisch und wird vom ARM-Kern ignoriert.

CRd

Das Feld wird vom ARM-Kern ignoriert. Die ISA empfiehlt, hier das Coprozessorzielregister bzw. Quellregister eines Transfers zu codieren.

Rn

Rn ist das Basisregister für die Berechnung der ersten Zugriffsadresse.

Offset

Ein Direktoperand, der um zwei Stellen nach links geschoben und dann, abhängig von der verwendeten Adressierungsart, auf den Inhalt von Rn addiert oder davon subtrahiert wird.

W

Das W-Bit spezifiziert, ob die durch den Offset modifizierte Basisadresse nach Rn zurückgeschrieben wird (W = 1) oder Rn unverändert bleibt.

PU

P und U codieren die Adressierungsart. Die Bildung der ersten Zugriffsadresse und der modifizierten Basisadresse entspricht Tabelle 1.8. Die modifizierte Basisadresse ist **nicht** von der Anzahl der Speicherzugriffe abhängig.

Ausnahmen und Adressausrichtung

P=0, U = 1, W=0: Das Offset-Feld wird weder zur Bildung der ersten Zugriffsadresse herangezogen noch die modifizierte Basisadresse zurückgeschrieben. Das Offset-Feld wird vom Prozessor ignoriert und kann für coprozessorspezifische Steuerinformationen verwendet werden.
P=0, U = 0, W =0: Das Verhalten bei dieser Kombination ist undefiniert. Entweder erkennt der Prozessor dies unmittelbar und erzeugt eine Undefined Exception oder aber Coprozessoren bearbeiten die Instruktion nicht, was ebenfalls eine Undefined Exception im ARM-Kern verursacht.
Rn = R15: Der verwendete Wert von R15 entspricht der Adresse der Instruktion + 8. Das Verhalten des Prozessors bei Rückschreiben der Basisadresse nach R15 durch W = 1 ist nicht spezifiziert.

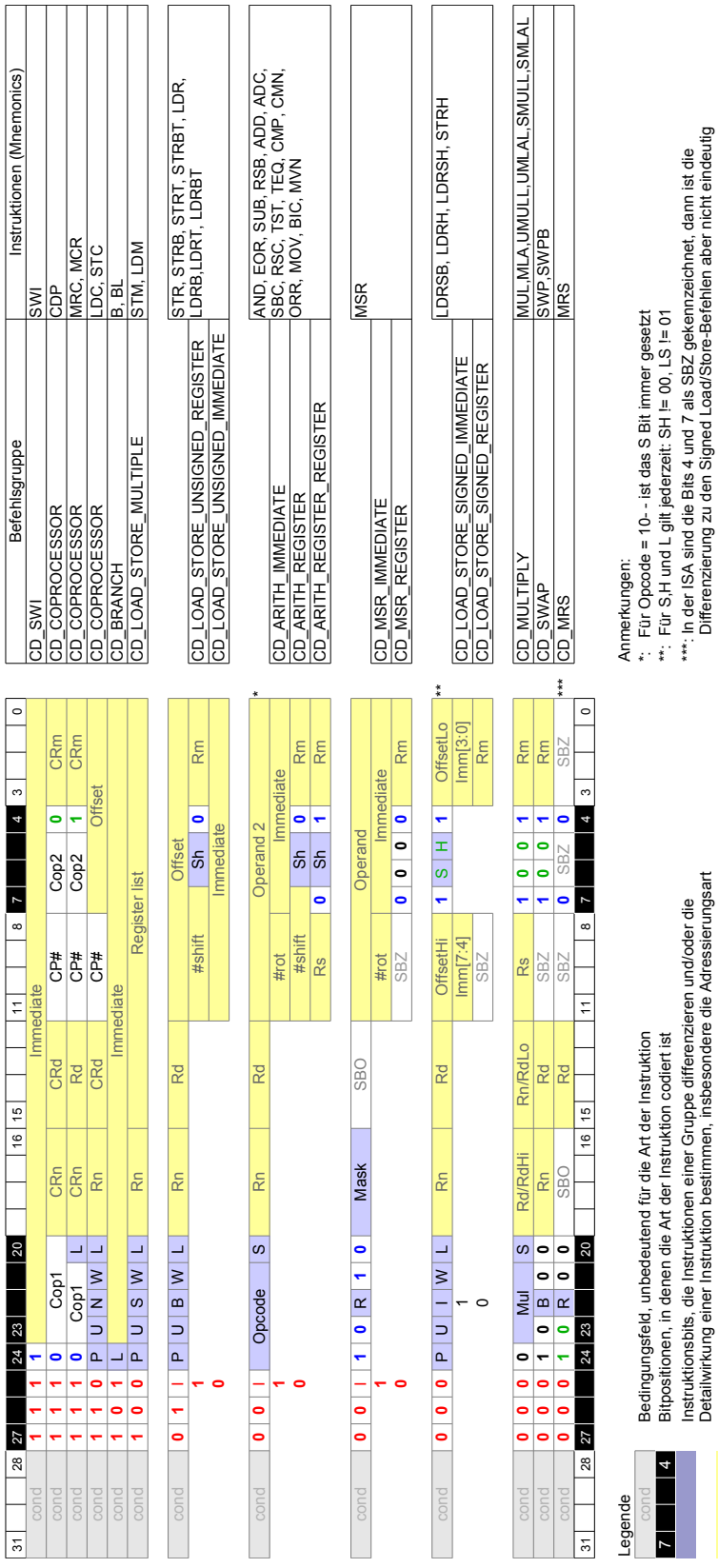
1.8.11 Zusammenfassung

Tabelle 1.12 fasst die Codierung aller definierten Instruktionen zusammen, dabei sind die verschiedenen Operand 2-Formate aufgeschlüsselt. Die Tabellenspalte **Befehlsgruppe** ist ein Vorgriff auf die Implementierung, die Gruppennamen gehören nicht zur Befehlssatzarchitektur. Die Instruktionen werden hierbei in 16 neue Gruppen unterteilt, welche nicht nur Instruktionen voneinander trennen, sondern auch die Formate von Operand 2 innerhalb der Instruktionen. Für dieses Vorgehen gibt es zwei Gründe:

1. Die Interpretation und Gültigkeit eines Befehlswortes hängt häufig von komplexen Bedingungen ab, die auch den Inhalt des Operand 2-Feldes betreffen. Während der Instruktionsdecodierung im Prozessor muss Operand 2 also ohnehin „entfaltet“ werden.

2. Die Operand 2-Formate wiederholen sich in verschiedenen Instruktionen und können dann instruktionsübergreifend gleich behandelt werden. Wenn während des Decodierens bereits das Operand 2-Format ermittelt wird, vereinfacht sich die Behandlung der Instruktionsbits 11:0 für viele Instruktionen stark.

Die in der Tabelle für fest vorgegebene Bits verwendeten Farben liefern einen Anhaltspunkt dafür, in welcher Reihenfolge diese Positionen sinnvoll ausgewertet werden können, um eine Instruktion eindeutig zu identifizieren. Ein Sprung kann beispielsweise eindeutig anhand der Kombination `101` in den Instruktionsbits 27:25 (rot) identifiziert werden. Wird Bit 24 (blau) hinzugezogen, steht `1111` eindeutig für den Software Interrupt. Durch Erweiterung auf Bit 4 (hier grün) kann eindeutig die CDP-Instruktion von MRC und MCR getrennt werden. Bits in Schwarz haben die niedrigste Priorität während der Auswertung. Die Erkennung der zahlreichen Ausnahmen im nicht sehr gleichmäßigen Instruktionssatz kann durch dieses einfache Farbschema allerdings nicht vollständig wiedergegeben werden.



Tab. 1.12: Übersicht der ARMv4-Instruktionen geordnet nach Gruppen und mit Hervorhebung der Bedeutung von Feldern innerhalb der Instruktionen.

Literaturverzeichnis

[1] *ARM Architecture Reference Manual*, 2005.

[2] *Procedure Call Standard for the ARM Architecture*, 2009. URL `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf`.

[3] Flik, Thomas, H. Liebig und M. Menge: *Mikroprozessortechnik: CISC, RISC, Systemaufbau, Assembler und C*. Springer Berlin, 6., neu bearb. Aufl. Auflage, 2001. ISBN 3540420428.

[4] Furber, Steve B.: *ARM System Architecture*. Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN 0201403528. URL `http://portal.acm.org/citation.cfm?id=525002`.

[5] Hennessy, John L. und David A. Patterson: *Computer Architecture. A Quantitative Approach*. Academic Press, 4th rev. ed. Auflage, September 2006. ISBN 0123704901.

[6] Menge, Matthias: *Moderne Prozessorarchitekturen. Prinzipien und ihre Realisierungen*. Springer, Berlin, 1 Auflage, März 2005. ISBN 3540243909.

[7] Patterson, David A. und John L. Hennessy: *Rechnerorganisation und -entwurf*. Spektrum Akademischer Verlag, 3. A. Auflage, September 2005. ISBN 3827415950.

[8] Sloss, Andrew, Dominic Symes und Chris Wright: *ARM System Developer's Guide. Designing and Optimizing System Software.: Designing and Optimizing System Software*. Morgan Kaufmann, Mai 2004. ISBN 1558608745.