PS Mikroprozessoren, Vortrag vom 16.05.2001

ARM: Advanced RISC Machine

Vortragende:Rene Werner, Tobias Kaempf

kommentierte Folien[1]

Teil 1

Allgemeines / Historie Instruction Set

Literatur:
"ARM system on chip architecture"; S. Furber (1)
"Tech. Informatik 2"; Schiffmann / Schmitz (2)
www.arm.com (3)

[1] sämtliche Abbildungen aus (1)

ARM: Historie

- *Oct. 1983 – Apr. 1985*:
  - Entwicklung der **A**corn **R**ISC **M**achine
  - durch Acorn Computers Limited, Cambridge

Acorn Limited hatte bereits vor 1984 aufgrund des Erfolges des BBC Microcomputers (mit 8 bit 6502 microprocessor) eine starke Marktposition bzgl. des PC-Marktes in UK. Im Sinne einer Weiterentwicklung entsprachen jedoch die erhältlichen CISC-Prozessoren nicht den Erwartungen der Acorn-Entwicklungen, infolge dessen diese sich zur Entwicklung eines eigenen Prozessors entschieden.
Dieser *working silicon* wurde am 26.4.1985 fertiggestellt und war der erste zur kommerziellen Nutzung erstellte RISC-Microprozessor.

- *1987*: Verwendung eines ARM-Proc. in PC-Bereich

Ein weiterer Schritt zur heutigen dominanten Stellung der ARM-Prozessoren war der Einsatz eines ARM als erster RISC-Processor überhaupt in einem (low cost) PC.

- *1990*: **A**dvanced **R**ISC **M**achines Limited eigenständig
- *1991*: Einführung des ARM6
- *1995*: Erweiterung der Proc. um Thumb architecture

Vergleiche hierzu Teil 2 (Autor: Tobias Kaempf). Thumb architecture erweitert den ARM-Befehlssatz um einen 16-bit Befehlssatz (z.B. SHIFT-Operationen) zur Erhöhung der Code-Density bzw. zur Energieeinsparung (auch in Verbindung mit 16bit-memory).

- *1996*: Zusammenarbeit mit Microsoft (Windows CE)
- *1999*: Zusammenarbeit mit Ericsson

ARM: Eigenschaften

- Reduced Instruction Set Computer
  (angelehnt an Berkeley RISC I)

Da Acorn Limited auf dem Segment der Prozessorentwicklung Neuling war, bot sich der durch die Einfachheit (Berkeley RISC II: 39 Befehle) u.a. seines Befehlssatzes auszeichnende Berkeley RISC I als Vorbild an. So sind Features wie Store-Load-Architecture oder auch 3-Adress-Format der Befehle mit diesem übereinstimmend, obgleich Features wie Register-Windows, Delayed Branches und auch die 5stage-Pipeline in Verwendung mit einem Instruction-Cache nicht nachimplementiert wurden, der Kosten aber vermutlich auch dem Streben nach Einfachheit der Konstruktion wegen.
Zu RISC vergleiche nächste Seite.

- "marketleader for low-power and cost-sensitive embedded applications"
- Lizenzvergabe an: Texas Instruments, Samsung, Digital, NEC, Rockwell, Philips, Hyundai, Sony, Seiko, HP, IBM, 3Com, Toshiba, Motorola, …

ARM Limited vergibt Lizenzen zur Integration und Produktion der Cores an die Systeme. Wie noch deutlich werden wird, zeichnen sich die ARM-Proc. durch Erweiterbarkeit (u.a. durch Programmierung des Co-Prozessors aus).

Entwicklung Mikroprozessor

	#Trans.	*Clock [MHz]*	*Vdd [V]*	*MIPS*	*Power [mW]*	*MIPS/W*	*pipeline*
ARM7TDMI	74209	66	3,3	60	87	690	3 stage
ARM9TDMI	111000	200	2,5	220	150	1500	5 stage
ARM10TDMI	??	>300	??	>420	??	??	6 stage

Tab 1

ARM: **A**dvanced **R**ISC **M**achine
7: Reihe
T: **T**humb-erweiterter Befehlssatz
D: **D**ebug support on chip
M: enhanced **M**ultiplier (u.a. 64bit Ergebnis)
I: Embedded **I**CE-hardware

Zu beachten ist, dass ein Kompromiss zwischen geringem Energieverbrauch (Einsatz z.B. in Handys, Palms,...) und ausreichender Rechenleistung zu finden ist. Bzgl. Pipelining / ARM wiederum der Verweis auf Teil2 (Tobias Kaempf).

RISC: **R**educed **I**nstruction **S**et **C**omputer

Def. RISC-Prozessor nach D. Tabak (1985):

1. ≥ 80% der Befehle über single-cycle execution
2. Alle Befehle in einem Maschinenwort codiert
3. ≤ 128 Maschinenbefehle

durch obige Tops soll die Einfachheit des Befehlssatzes gewährleiste bleiben.

4. ≤ 4 Befehlsformate
5. ≤ 4 Adressierungsarten
6. load-store Architektur
7. Register-Register Architektur
8. Mind. 32 Prozessorregister

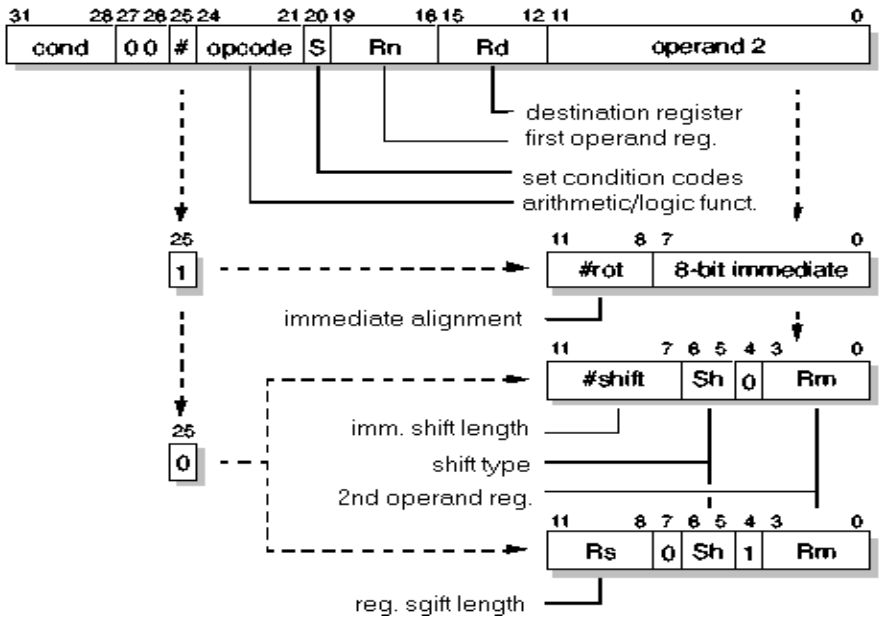
Umsetzung bei den ARM-Prozessoren:

zu 4.)

Befehlskategorien

- Data Processing Instructions

Data Processing Instructions haben in ARM-Architekturen 32bit breite Operanden:



Durch die Rotation um 0-32 bit (right-rotation) sind immediates im Bereich $[(0 \rightarrow 255) \times (2^2)^n]_{10}$ als 2. Operand möglich.

Es lassen sich unterteilen:

- Arithmetic Operations

  z.B.:

  ```
  ADD    r3, r3, #&a          ; r3 = r3 + 10 , immediate auch als dezimal schreibbar
  RSC    r0, r1, r2           ; r0 = r2 – r1 + C – 1
  ```

- Bit-wise logical Operations (z.B. AND, EOR, …)
- Register movement Operations (z.B. MOV, MVN, …)
- Comparison Operations

  z.B.:

  ```
  CMN    r1, r2               ; set cc on r1 + r2
  ```

- Shifting Operations

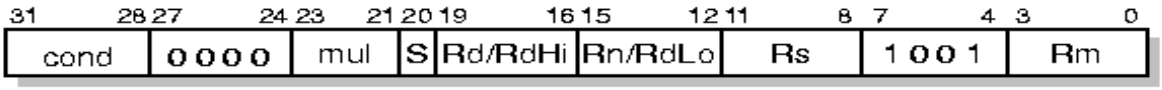
  z.B.:

  ```
  ADD    r5, r5, r3,      LSL r2   ; r5 = r5 + r3 × 2^r2
  ```

  Im eigentlichen (ohne Thumb-Erweiterung) ARM-Befehlssatz gibt es keine alleinigen Shift-Befehle. Diese werden nur in Verbindung mit anderen ALU-Befehlen durchgeführt, womit quasi 2 Bearbeitungsschritte zu einem zusammengefasst wurden (und in nur einem Taktzyklus ausgeführt werden).

- Multiplikation

  Die ARM-Multiplikation gestatte mehrere Modi. Wie o.g. sind die Operanden 32bit breit, folglich kann das resultierende Ergebnis 64bit breit sein. Diesem wird mit mehreren Modi wie folgt Rechnung getragen:



Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	Rd := (Rm * Rs) [31:0]
001	MLA	Multiply-accumulate (32-bit result)	Rd := (Rm * Rs + Rn) [31:0]
100	UMULL	Unsigned multiply long	RdHi:RdLo := Rm * Rs
101	UMLAL	Unsigned multiply-accumulate long	RdHi:RdLo += Rm * Rs
110	SMULL	Signed multiply long	RdHi:RdLo := Rm * Rs
111	SMLAL	Signed multiply-accumulate long	RdHi:RdLo += Rm * Rs

- # Data Transfer Instructions

Data Transfer Instructions dienen dem Datentransfer zwischen ARM-Registern und dem Speicher (Register-indirekte Adressierung). Es lassen sich unterscheiden:

- Single Register Load and Store Instructions

Es werden einzelne Datenobjekte zwischen Register und Speicher transferiert. Hierbei kann es sich um unsigned byte oder ein word oder aber ein halfword oder ein signed byte handeln.

- Multiple Register Load and Store Instructions

Es kann jedes der 16 sichtbaren Register bzw. jede beliebige Teilmenge abgespeichert werden. Hilfreich z.B. bei procedure entry oder exit.

- Single Register and Memory Instructions

Hierbei handelt es sich um den Austausch eines Registereinhalts mit einem Speicherinhalt, d.h. es werden zwei Befehle zusammengefasst (folglich danach atomar)

- # Control Flow Instructions

This third category of instructions neither processes data nor moves it around; it simply determines which instruction get executed next.[2]

Branch instructions:

- bedingte Sprungbefehle
- branch and link: Speichern der Adresse des auf den Branchbefehl folgenden Befehls im *link-reg* r14

```
BL      SUBR            ; branch to subroutine
```

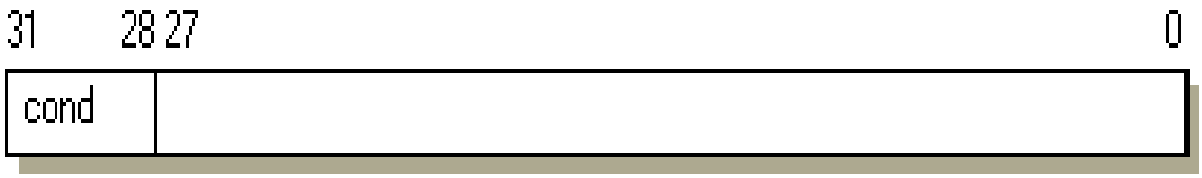
- subroutine return: Aktualisieren des pc durch

```
MOV     pc, r14         ; copy r14 into r15 and return
```

Des weiteren bietet der Befehlssatz die Möglichkeit eines *Supervisor calls*, d.h. die Möglichkeit, Funktionalitäten aufzurufen, die vom user-level program nicht erreichbar sind (z.B. Input / Output).
Man beachte, dass Arm auf *delayed branches* verzichtete (im Gegensatz zu vielen anderen Herstellern von RISC-Prozessoren) und die *every time conditional execution* die Möglichkeit bietet, auf eine Vielzahl von bedingten Sprüngen zu verzichten, und so denCode kurz zu halten →

[2] aus S.Furber, „ARM system on chip architecture“

Eine Besonderheit des ARM Architektur ist, das jeder Befehl als conditional instruction angesehen wird. Infolgedessen hat jeder Befehl umgesetzt auf Registerebene zunächst folgende Form:



Demnach ist der Konditionalteil 4bit breit und hat somit 16 verschiedene Kombinationsmöglichkeiten. Diese sind folgenden Tabellen zu entnehmen:

Opcode	State for execution
0000	Z set
0001	Z clear
0010	C set
0011	C clear
0100	N set
0101	N clear
0110	V set
0111	V clear

Opcode	State for execution
1000	C set and Z clear
1001	C clear or Z set
1010	N equals V
1011	N is not equal to V
1100	Z clear and N equals V
1101	Z set or N is not equal V
1110	any
1111	none

N: *negative*, letzte ALU-Operation (mit Flagänderung) verursachte negatives Ergebnis

Z: *zero*, letzte ALU-Operation (s.o.) ergab Null-Resultat

C: *carry*, letzte ALU-Operation (s.o.) generierte Carry-out

O: *overflow*, generierter Überfluss in Sign-bit durch ALU

Somit ergibt sich auch die Möglichkeit, Assembler-Code wie folgend zu optimieren:

Assemblerbeispiel: if (r0 != 5) {r1 = r1 + r0 – r2}

1. Möglichkeit (unoptimiert):

```
    CMP     r0, #5
    BEQ     BYPASS          ; if (...)
    ADD     r1, r1, r0              ; r1 = r1 + r0
    SUB     r1, r1, r2              ; r1 = r1 – r2
BYPASS ...
```

2. Möglichkeit (optimiert):

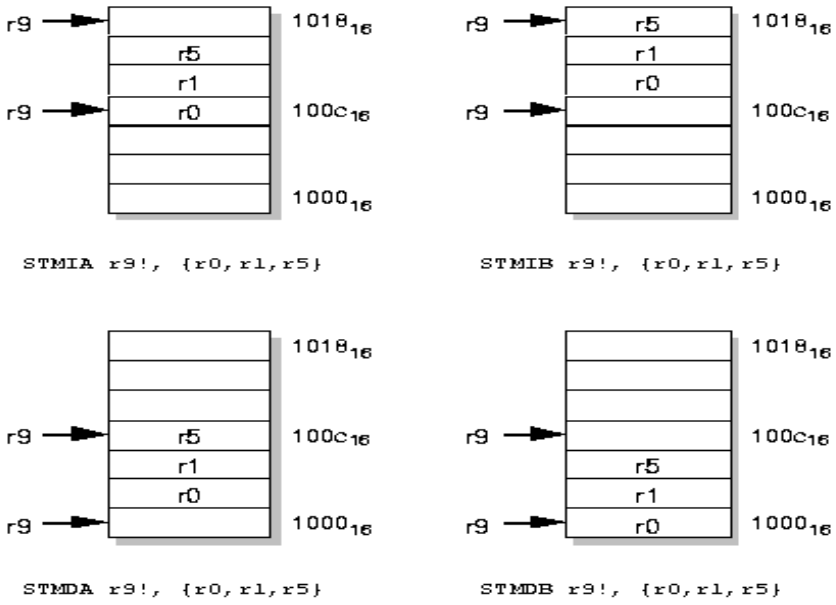
```
    CMP     r0, #5
    ADDNE   r1, r1, r0      ; r1 = r1 + r0 (if not equal)
    SUBNE   r1, r1, r2      ; r1 = r1 – r2
```

zu 5.) Adressierungsarten

Der ARM Datentransfer basiert auf Register-Indirekter Adressierung, wobei die Möglichkeit eines base-plus-offsets von bis zu 4kB und einer base-plus-index Adressierung gegeben ist. Die verschiedenen Modi der (interessanten) multiple data transfer operations entnehme man der unteren Skizze:

- Register-Indirekt (base-plus-offset und base-plus-index mgl.)

```
LDR     r0, [r1, #4]     ; r0 = mem32[r1+4]                    ( pre-indexing )
LDR     r0, [r1, #4]!    ; r0 = mem32[r1+4] and r1 = r1 + 4    ( auto-indexing )
LDR     r0, [r1], #4     ; r0 = mem32[r1] and r1 = r1 + 4      ( post-indexing )
```



- Stack-Adressierung:

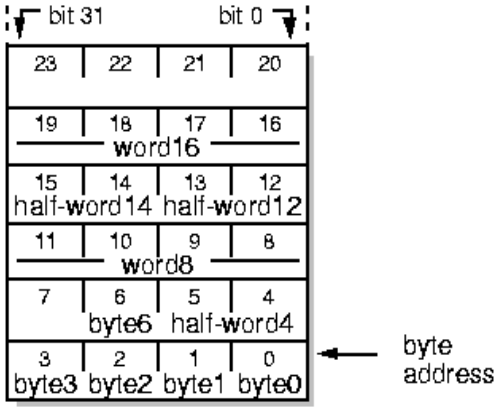
Nun gut, einen Stack gibt es natürlich auch (wobei auch hier Registerindirekte Adressierung gegeben ist). Dieser unterscheidet sich nicht wesentlich von bereits kennengelernter Architektur (z.B. T3-Praktikum).

ARM Speicher Organisation

adressierbar:

Bytes	(8 bit)
Half-word	(16 bit)
Word	(32 bit)

ARM-chips können mit Big- und Little-Endian Struktur arbeiten (konfigurierbar, gebräuchlich Little Endian)



Anbei sind genau dieses die 6 DT (jeweils signed / unsigned), welche eben durch data transfer instructions adressiert werden können.

zu 6.)

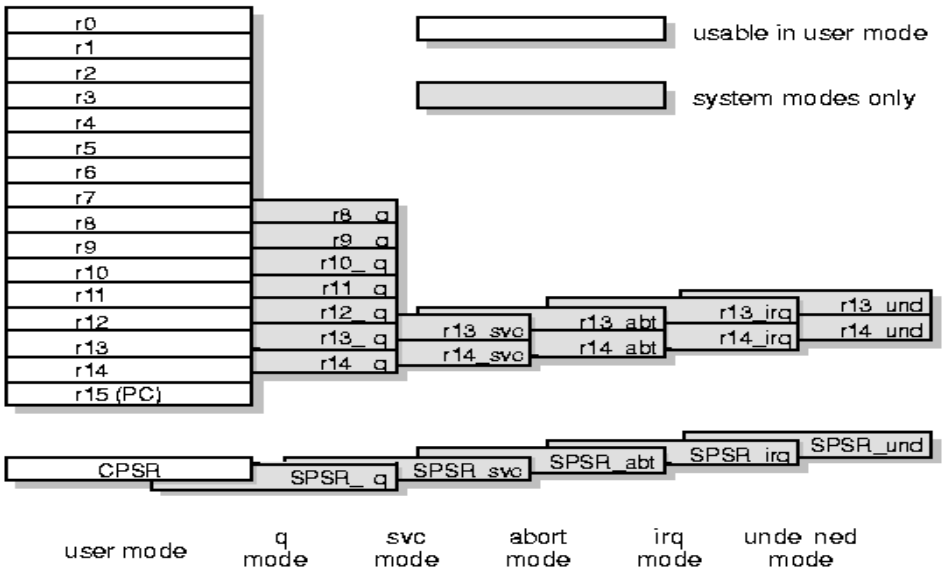
Load-Store Architecture

- Verarbeitung von Werten aus Registern (oder Imm.)
- Schreiben der Ergebnisse in Register
- Zugriff auf Speicher nur zum Kopieren von Werten

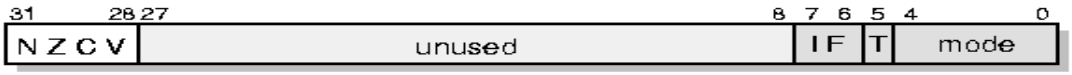
Durch die Load-Store-Architektur ist eine Abgrenzung zu den meisten CISC-Processoren, welche als Operanden von Data Process Instructions durchsaus auch Speicherinhalte akzeptieren.
Die klare Trennung von Datenverarbeitung (i.S. von data process, auf Registerebene) und dem Speicher sorgt für einen übersichtlicheren Befehlssatz bei allerdings abnehmender code density (welche jedoch auch nicht primäres Anliegen der RISC-Proc. ist).
ARM hat, wie bereits beschrieben, die load-store-Architektur umgesetzt.

zu 9.)

ARMs Register (-zugriffsmöglichkeiten)



ARM-Register sind jeweils 32bit breit. Sichtbar auf user-level sind 15 (r0 – r14) allgemeine, der pc (r15) und das current program status register:



N: *negative*, letzte ALU-Operation (mit Flagänderung) verursachte negatives Ergebnis
Z: *zero*, letzte ALU-Operation (s.o.) ergab Null-Resultat
C: *carry*, letzte ALU-Operation (s.o.) generierte Carry-out
O: *overflow*, generierter Überfluss in Sign-bit durch ALU

ARM
Advanced RISC Machine

Teil 2.

ARM Architektur und Anwendungsbeispiele

ARM : Advanced RISC Machine

2.Teil : Architektur und Anwendungsbeispiele

1. Architektur:

1.1. Pipeline bei ARM

1.1.1. Pipeline allgemein
1.1.2. 3-Stage Pipeline
1.1.3. 5-Stage Pipeline

1.2. Die Thumb - Architektur

1.2.1. Was ist Thumb?

2. Anwendungsbeispiele:

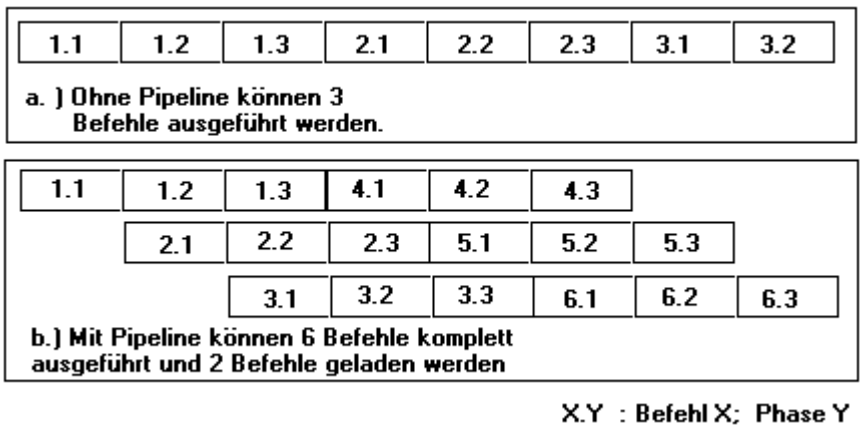
2.1. Mobiltelefon
2.2. Psion 5

1. Architektur bei ARM:

1.1. Pipeline bei ARM

1.1.1 Pipeline allgemein

In einem Prozessor (ohne Pipeline) wird immer nur ein kleiner Bereich gleichzeitig genutzt. So wird zum Beispiel während ein Befehl aus dem Speicher geladen wird, die ALU nicht benutzt. Während der Befehl decodiert wird, wird weder die ALU noch der Speicher benutzt und so weiter.
Die Abarbeitung eines Befehls kann also in verschiedene Phasen unterteilt werden, je nachdem welcher Bereich des Prozessors gerade benutzt wird.
Diese Unterteilung kann sowohl bei CISC, wie auch bei RISC Architekturen durchgeführt werden, aber die RISC Architekturen haben einen Vorteil, denn bei RISC-Architekturen benötigen die meisten Befehle (über 80% aller Befehle) nur einen einzigen Takt zum ausführen. Bei CISC benötigen die Befehle unterschiedlich lange um ausgeführt zu werden.
Durch diesen Vorteil können bei RISC- Architekturen über 80% der Befehle in Phasen unterteilt werden, die genau einen Takt dauern und verschiedene Bereiche des Prozessors benutzen. Da diese einzelnen Phasen auf unterschiedlichen Bereichen des Prozessors arbeiten, können die Phasen gleichzeitig nebeneinander ausgeführt werden. Dies führt zu einer deutlichen Steigerung der Prozessorgeschwindigkeit.
Auch bei CISC- Architekturen ist Pipelining durchführbar, aber durch die unterschiedliche Ausführungsdauer der einzelnen Befehle ist das Design einer CISC-Pipeline sehr viel komplexer und schwieriger.



Grafik 1.) Allgemeines Pipelineprinzip mit 3-Stage Pipeline

Problematisch an Pipelines sind aber Situationen wo ein Befehl ein Resultat eines vorherigen Befehls benötigt, wenn dieses Resultat noch nicht im Speicher ist. Bei solchen Situationen muss die Ausführung eines Befehls so lange angehalten werden, bis das gesuchte Resultat vorliegt.

1.1.2 3-Stage Pipeline bei ARM

Die 3-Stage Pipeline wurde zwischen 1990 und 1995 in den ARM6 und ARM7 Prozessoren eingesetzt. Die Grafik zeigt die 3-Stage Pipeline Organisation im Prozessor.

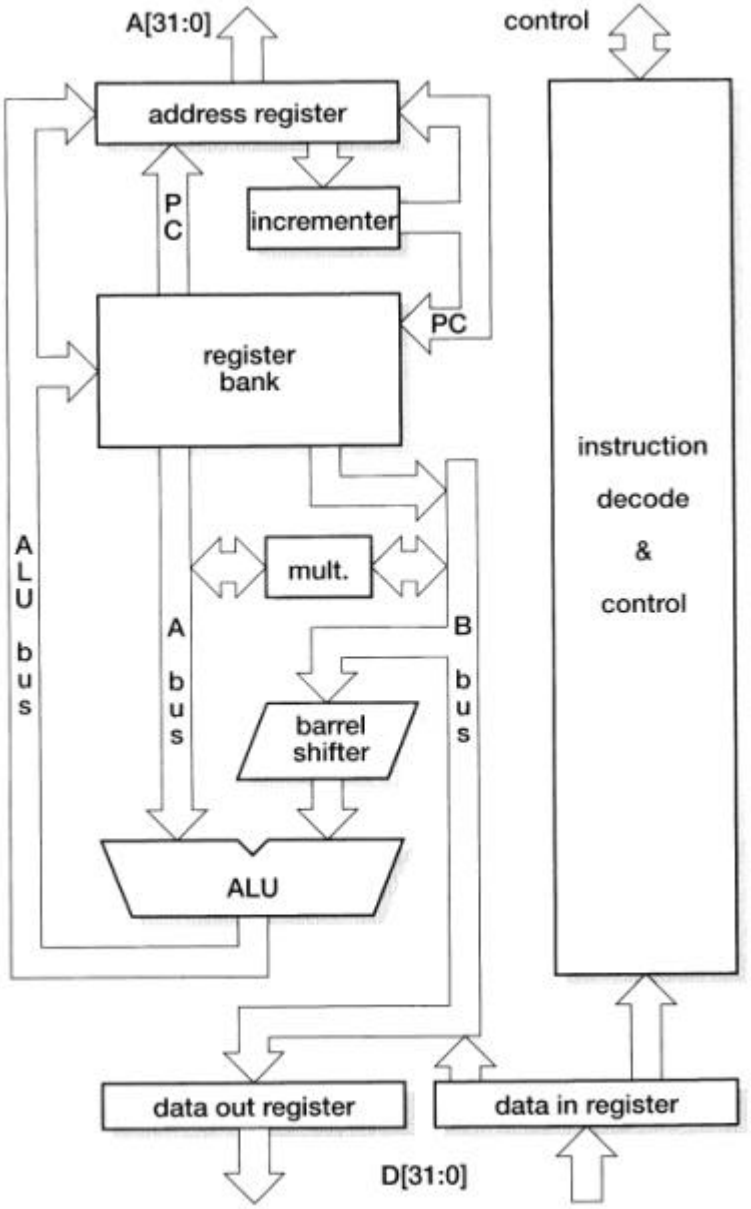


Figure 4.1 3-stage pipeline ARM organization.

Die einzelnen dargestellten Komponenten sind :

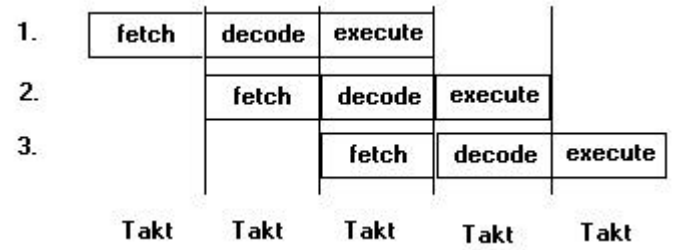
- register bank, sie speichert den Prozessor Status. Sie hat zwei Lese und einen Schreib Port, plus Lese und Schreib Port mit dem das r15 Register angesprochen werden kann, r15 enthält den Programm Zähler (PC)
- barrel shifter, hier kann ein operand um eine beliebige Anzahl von Bits rotiert oder verschoben werden
- ALU, hier werden die arithmetischen und logischen Funktionen durchgeführt

- address register and incrementer, hier werden die Adressen selektiert und alle Speicher Adressen gehalten. Es können auch sequentielle Adressen erzeugt werden, wenn benötigt.
- data registers, diese halten alle Daten die von und zu dem Speicher transportiert werden
- instruction decoder and assocoated control logic

Bei der 3-Stage Pipeline werden die Befehle, die nur einen Takt zum ausführen benötigen in 3 Phasen unterteilt :

- Fetch : Der Befehl wird aus dem Speicher geladen und in die Pipeline eingefügt
- Decode : Der Befehl wird decodiert
- Execute : Der Befehl wird ausgeführt, wobei die Register ausgelesen werden, dann die ALU die Daten kombiniert und in das Zielregister zurückschreibt

Die normale Ablaufreihenfolge single-cycle Befehlen :



Bei multi-cycle Befehlen wird der Ablauf kurz unterbrochen, wie diese Grafik zeigt (hier wird ein Store Befehl als multi-cycle Beispiel verwendet) :

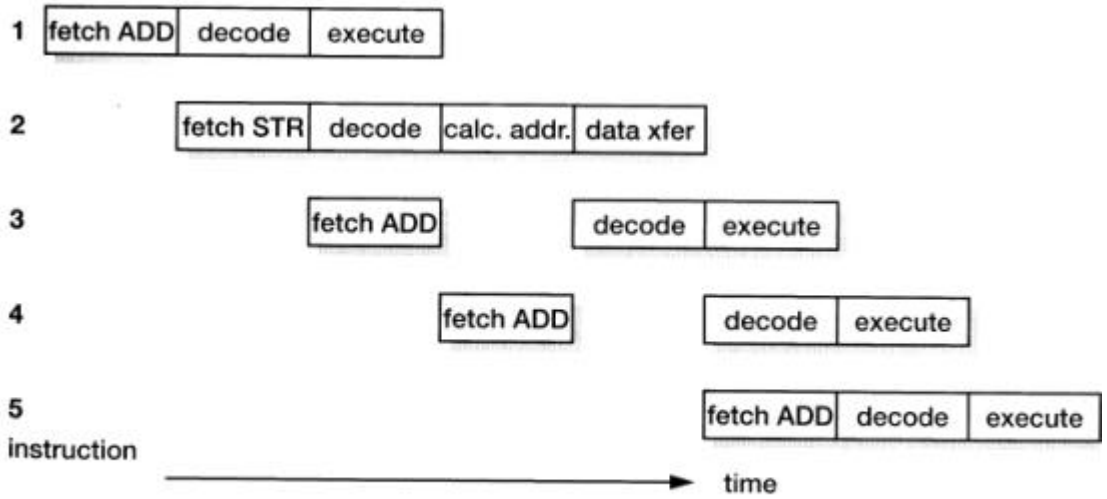


Figure 4.3 ARM multi-cycle instruction 3-stage pipeline operation.

Diese Befehlsausführung mit einer Pipeline hat zur Folge, das der Programm Zähler dem gerade ausführenden Befehl vorausläuft.

1.1.3 5-Stage Pipeline bei ARM

Durch die Nachfrage nach höherer Performance wurde die 3-Stage Pipeline nach dem ARM7 Prozessoren neu überdacht, um den Prozessor noch schneller zu machen. Dabei wurde auf einer Formel aufgebaut :

- $T_{prog} = (N_{inst} * CPI) / f_{clk}$

Wobei T_{prog} die Ausführungsdauer eines Programms, N_{inst} die Anzahl der Instruktionen in diesem Programm, CPI die durchschnittliche Anzahl von Takten pro Instruktion und f_{clk} die Taktrate des Prozessors ist.
Diese Formel besagt also, das die Dauer eines Programms die Anzahl der abzuarbeitenden Befehle mal die Anzahl der Takte pro Instruktion durch den Takt ist. Da N_{inst} für ein Programm festgelegt ist, kann das Programm also nur durch eine Verkleinerung der durchschnittlich benötigten Takte pro Befehl und / oder durch eine Erhöhung der Taktrate des Prozessors schneller ausgeführt werden.
Um die Taktrate erhöhen zu können muss die Logik in den Pipelineschritten vereinfacht werden und dafür muss die Anzahl der Schritte erhöht werden.
Um die Anzahl an benötigten Takten für die Instruktionen zu verkleinern muss jeder Befehl der in der 3-Stage Pipeline zwei oder mehr Takte brauchte um ausgeführt zu werden, so verändert wird das er nur noch einen Takt zum Ausführen benötigt. Aber auch Abhängigkeiten zwischen einzelnen Befehlen müssen aufgehoben und entschärft werden.
Innerhalb der 3-Stage Pipeline wird in nahezu jedem Takt auf den Speicher zugegriffen, entweder um einen Befehl zu laden, oder um einen Wert zu transportieren. Die durchschnittliche Anzahl an Takten für jeden Befehl kann durch diesen Umstand nicht über ein bestimmtes Maß hinaus verkleinert werden, außer der Speicher gibt in einem Takt mehr als einen Datenwert. Dies kann entweder durch einen Datenstrom von mehr als 32 Bits pro Takt oder durch getrennte Daten und Befehlsspeicher realisiert werden. Von diesen Überlegungen profitieren die schnelleren 5-Stage Pipeline Prozessoren von ARM. Diese Prozessoren teilen die Befehle in 5 Phasen auf und benutzen getrennte Daten und Befehlsspeicher. Durch die Aufteilung in 5 Phasen muss in jeder Phase weniger getan werden und dies erlaubt die Erhöhung der Taktrate. Durch die getrennten Speicher werden die Takte pro Befehl noch einmal stark gesenkt.

Die 5 Phasen sind :

- Fetch : Der Befehl wird vom Befehlsspeicher geladen
- Decode : Der Befehl wird decodiert und die Operanden werden aus den Registern gelesen. Drei Operand ReadPorts erlauben Zugriff in einem Takt
- Execute : Die ALU berechnet das Ergebnis, wenn der Befehl ein load oder store Befehl ist, berechnet die ALU die Speicher Adresse
- Buffer/Data : Operationen auf dem Datenspeicher finden statt, wenn keine benötigt werden, wird das Resultat der ALU einen Takt gepuffert
- Write-Back : Die Resultate des Befehls werden in die Register zurück geschrieben, auch alle Daten die vom Speicher gelesen worden sind.

Hier eine Grafik die den Aufbau einer 5-Stage Pipeline in einem ARM9 Prozessor beschreibt :

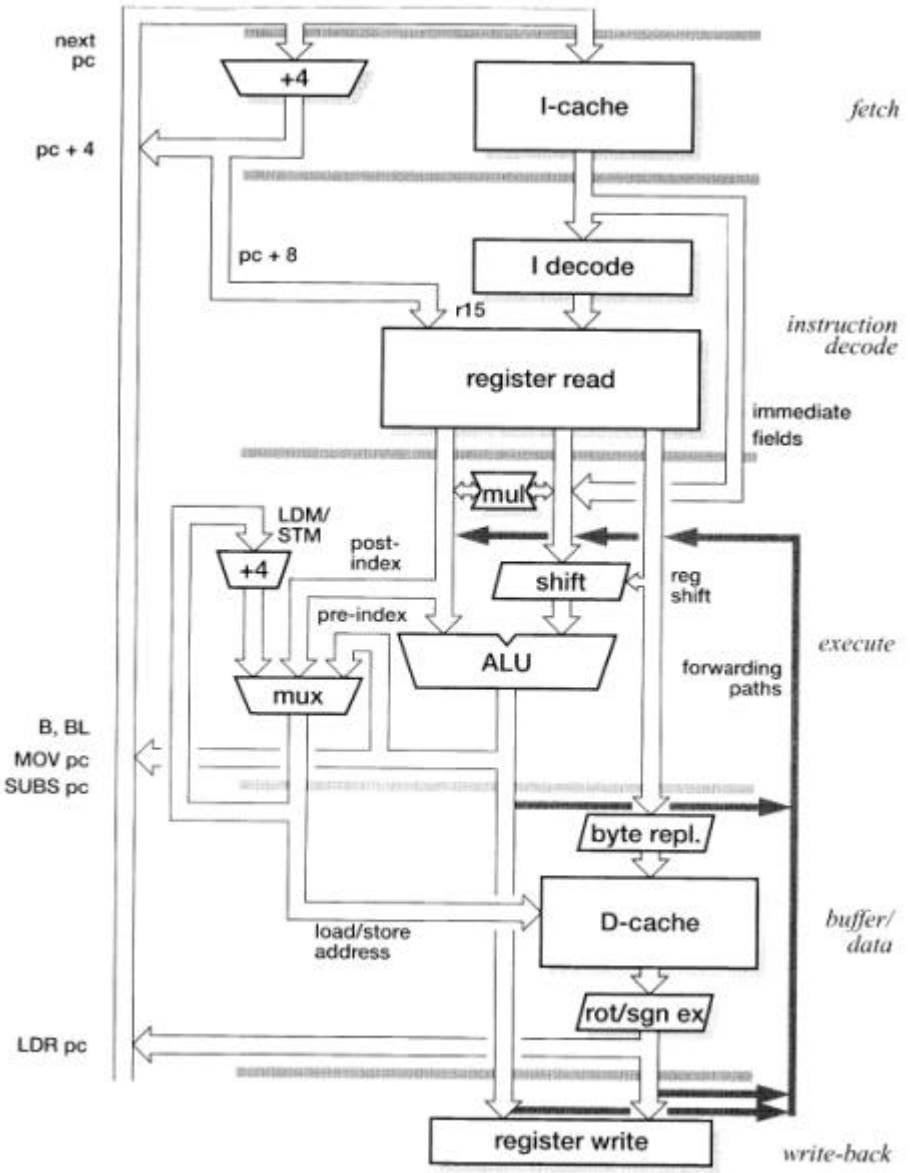


Figure 4.4 ARM9TDMI 5-stage pipeline organization.

Der 5-Stage Pipeline Ansatz ist der „klassische“ Ansatz eine Pipeline zu designen. Obwohl der ARM Befehlssatz nicht auf eine 5-Stage Pipeline ausgelegt war, konnte er gut an diese Struktur angepasst werden. Besonderheiten der ARM Pipeline zu anderen Pipeline Designs waren die drei Lese und zwei Schreib Ports am Registerblock, andere Pipelines hatten nur zwei Lese und einen Schreib Port. Diese Eigenschaften waren zur Anpassung an den ARM Befehlssatz eingefügt worden, genauso wie eine Adressen erhöhende Hardware in der Ausführungsphase der Pipeline um mehrfache Speicherzugriffe zu ermöglichen.
Um den Zeitverlust durch Datenabhängigkeiten (ein Befehl benötigt das Resultat des vorherigen Befehls als Eingabe) zu minimieren, wurde es ermöglicht aus jeder Phase die Ergebnisse der Befehle zu lesen. Dies nennt man „Data Forwarding“ und wird in der Grafik mit den immediate fields beschrieben. Durch „Data Forwarding“ kann nicht verhindert werden, das die Pipeline einen Befehl kurzzeitig aussetzt wenn der

Befehl einen Wert benutzen will, den sein direkter Vorgänger aus dem Speicher gelesen hat, aber alle anderen Zeitverluste können verhindert werden.

Hier nun eine Erklärung wie bei ARM die Befehle in einer Pipeline ausgeführt werden :

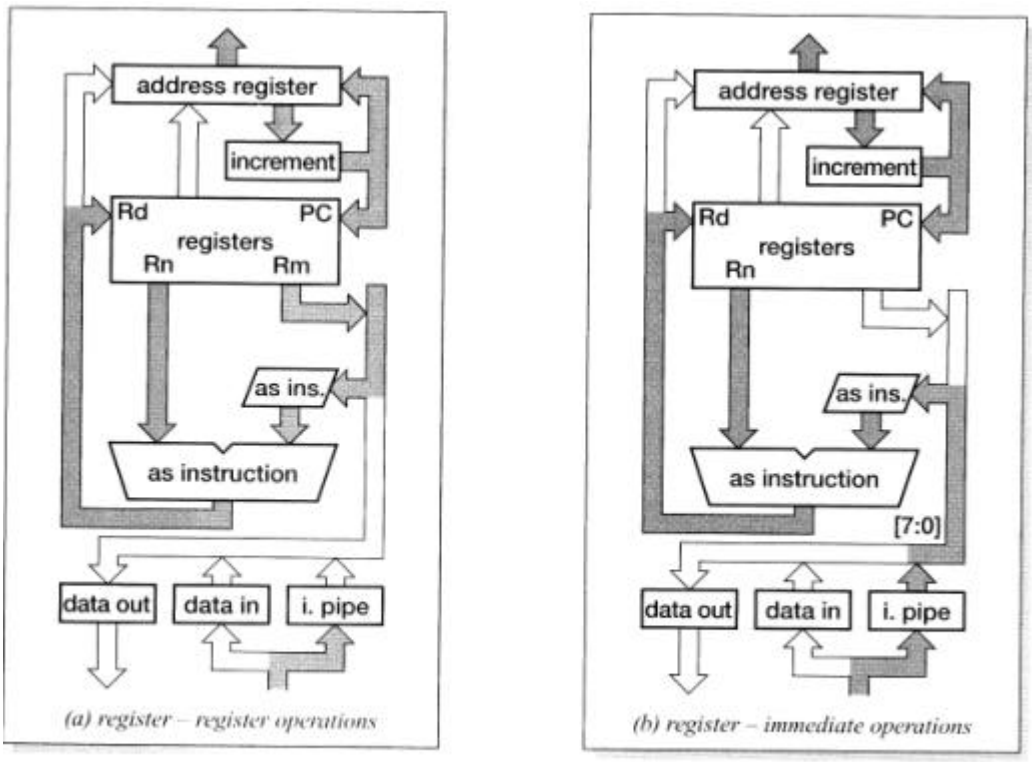


Figure 4.5 Data processing instruction datapath activity.

Dieses Beispiel ist ein datenverarbeitender Befehl, der zwei Operanden benötigt. Der eine Operand ist ein Register, der andere kann ebenfalls ein Register sein (a) oder ein im Befehl mit angegebener Wert (b).
Innerhalb eines Taktes wird der erste Operand aus dem Register genommen, der zweite Operand wird in den „barrel shifter“ geladen und dort einer shift-Operation unterzogen. Nach dieser Operation werden beide Operanden in der ALU miteinander kombiniert (mit dem zugehörigen ALU Befehl) und das Ergebnis wieder in das Zielregister zurückgeschrieben. Währenddessen wird der Programm Zähler erhöht und sowohl in den Adressblock als auch in r15 geschrieben. Der nächste Befehl wird auf den Boden der Instruktion Pipeline „i.pipe“ geladen. Der zweite Operand wird, wenn er als direkter Wert kommt, von dem Befehl an der Spitze der Instruktion Pipeline extrahiert. Der Operand ist dann in den letzen 8 Bits des Befehls.

Ein weiteres Beispiel für die Aktivitäten auf dem Datenpfad der ARM Befehlen

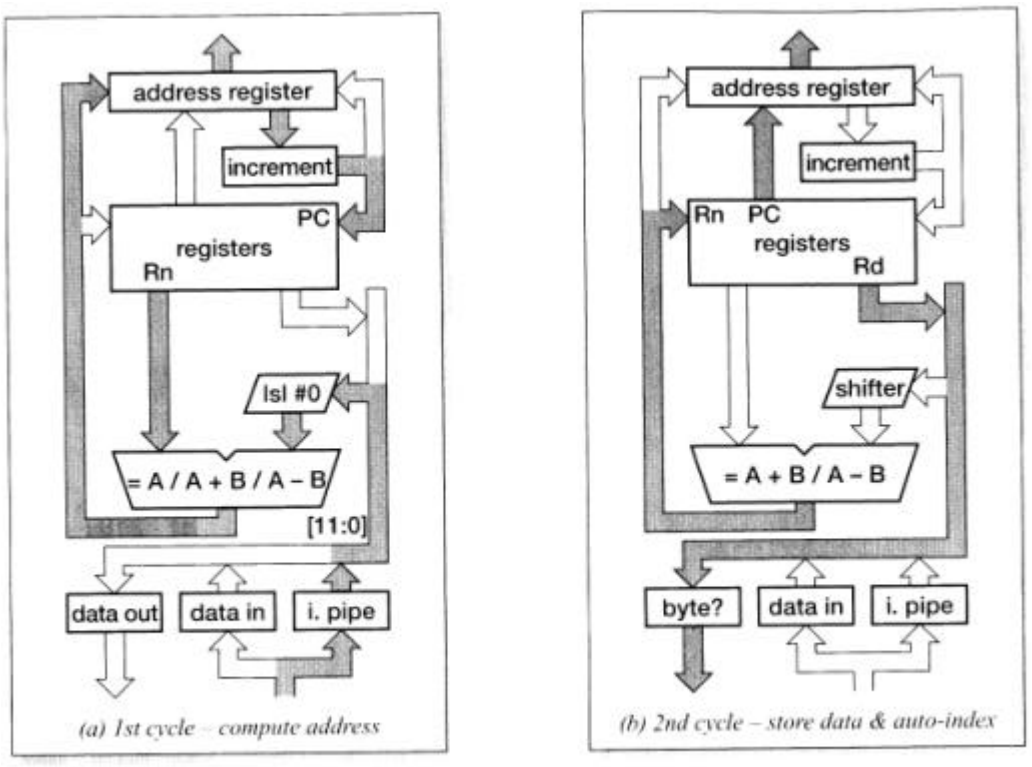


Figure 4.6 STR (store register) datapath activity.

Dieses Beispiel zeigt den Datenfluss während der zwei benötigten Takte zum Speichern eines Wertes in einen externen Speicher als Beispiel für Datentransfer Befehle.
Bei einem Store-Befehl wird zuerst aus einem Register, das als Basisadresse dient, ein Offset hinzuaddiert oder subtrahiert. Dieses Offset kann in einem weiteren Register oder in dem Befehl selbst stehen. Hier in dem Beispiel steht es im Befehl. Das Offset ist nicht wie bei Datenverarbeitenden Befehlen 8 Bit sondern 12 Bit lang, das ohne den shift-Befehl benutzt wird. Die so gewonnene Adresse wird in das Adressregister geschrieben und in dem zweiten Takt werden die Daten an diese Adresse geschrieben. Anstatt während des zweiten Taktes die ALU ungenutzt zu lassen wird eine auto-indexing modifikation durch die ALU an dem Base-Register vorgenommen. Wenn diese nicht benötigt wurde, wird die Änderung einfach nicht auf das Register geschrieben. Während der ersten Phase wird der Programmzähler in den Registerblock übertragen, damit das Adressregister frei für eingaben aus der ALU ist. Am Ende des zweiten Taktes wird der Programmzähler dann wieder in das Adressregister geschrieben um mit der Pipeline normal weiterarbeiten zu können.

Soviel zu Pipelines und deren Benutzung in ARM Prozessoren. Weitere Information zu diesem Thema kann man in der angegebenen Literatur finden.

1.2. Die Thumb-Architektur

1.2.1 Was ist Thumb ?

Die Befehlslänge im ARM Befehlssatz betrug 32 Bit, mit diesen 32 Bit konnten alle Befehle dargestellt und kodiert werden, sowie bei einigen direkte Werte mit angehängt werden. Einige Befehle benötigen dabei die kompletten 32 Bit, andere hingegen benötigen nicht alle 32 Bit Befehlslänge. Um diese „kleineren“ Befehle den 32 Bit Befehlslänge anzupassen, werden sie mit 32 Bit dargestellt und einige dieser Bits werden nicht interpretiert. Durch diese nicht interpretierten Bits ist die Code-Dichte des ARM Befehlssatzes nicht besonders gut.
Um die Code-Dichte zu erhöhen ist die Thumb- Technologie entwickelt worden. Thumb ist eine komprimierte Form des ARM Befehlssatzes der mit 16 Bit Befehlslänge auskommt. Thumb wird aber nicht als eigenständiger Befehlssatz implementiert, sondern er wird in eine ARM Befehlssatz Pipeline integriert indem die Thumb Befehle dynamisch innerhalb der ARM Pipeline decodiert und dann als normale ARM Befehle ausgeführt werden. Thumb kann wird nur von ARM-Prozessoren unterstützt die ein T in ihrem Namen haben, z.B. ARM7TDMI.

Gemeinsamkeiten zwischen ARM- und Thumb- Befehlssätzen sind :
- Die load-store Architektur mit Datenverarbeitung, Datentransfer und Flusskontroll Befehlen
- Die Unterstützung von 8-Bit byte, 16-bit half-word and 32-bit word Datentypen
- Ein 32-bit unsegmentierter Speicher

Folgende Unterschiede sind durch die Reduzierung auf 16-bit Befehle enstanden :
- Thumb Befehle werden ohne Bedingungen ausgeführt
  (Alle ARM Befehle werden nur mit Bedingungen ausgeführt)
- Die meisten Thumb Befehle benutzen ein 2-Adress Format
  (Die ARM Befehle benutzen ein 3-Adress Format)
- Thumb Befehle sind nicht so regulär wie ARM Befehle
  (Dies ist eine Folge der dichten Codierung)

Wenn ein Programmierer ein Programm für den Thumb- Befehlssatz schreiben will, hat er auf einige Besonderheiten gegenüber dem ARM- Befehlssatz zu achten. Der Programmierer hat nur noch unbeschränkten Zugriff auf die Register r0 bis r7, die Register r8 bis r12 werden als HI Register beschrieben und haben nur eingeschränkte Zugriffsmöglichkeiten, genauso wie der CPSR (Current Prozessor Status Register).
Die Register r13 bis r15 werden wie folgt behandelt :
r13 wird als Stack-Pointer verwendet
r14 wird als link Register verwendet und
r15 ist der Programm Zähler

Dies folgt stark den Konventionen des ARM Befehlssatzes, obwohl die Benutzung von r13 als Stack Pointer bei ARM –Befehlen nur Software Konvention war und bei Thumb schon fast Hardware gestützt.
Um vom ARM Modus in den Thumb Modus zu wechseln gibt es die Möglichkeit des Befehls BX (Branch and Exchange). Dieser Befehl setzt das T-Bit des CPSR wenn das unterste Bit des angegebenen Registers gesetzt ist und lässt den Programm Zähler auf die Adresse wechseln die im Register angegeben ist. Die Pipeline wird geflutet womit alle anderen schon geladenen Befehle abgebrochen werden.
Um von Thumb auf ARM wieder zurück zu schalten muss ein Thumb BX- Befehl benutzt werden.
Eine Besonderheit von dieser Regel bilden Exceptions, sobald eine Exception auftritt, wird der Thumb Modus abgebrochen und die Exception im ARM Befehlsmodus behandelt. Nach der Behandlung der Exception wird der Thumbmodus wiederhergestellt, da der CPSR komplett wiederhergestellt wird.
Als Beispiel für die Erhöhung der Code Dichte wird ein Sprungbefehl genauer unter die Lupe genommen :

Zuerst die Überlegung, wofür ein Sprungbefehl benutzt wird :

1. Mit einer Bedingung, zum Beispiel zum beenden einer Schleife
2. Ohne Bedingung über mittlere Entfernungen zum überspringen von Befehlen (goto)
3. Zum Aufrufen von Subroutinen

Diese möglichen Einsatzgebiete werden im ARM Befehlssatz durch ein und den selben 32 Bit Befehl durchgeführt. Dabei werden in dem zweiten Fall die Bits für die Bedingung und Offsetbits durch einen relativ kleinen Sprung verschwendet (bei ARM werden 24 Bit als Offset benutzt, diese werden nicht immer benötigt).
Der Thumb Befehlssatz hat für jede dieser Anwendungsmöglichkeiten einen eigenen Befehl, wie die Grafik zeigt :

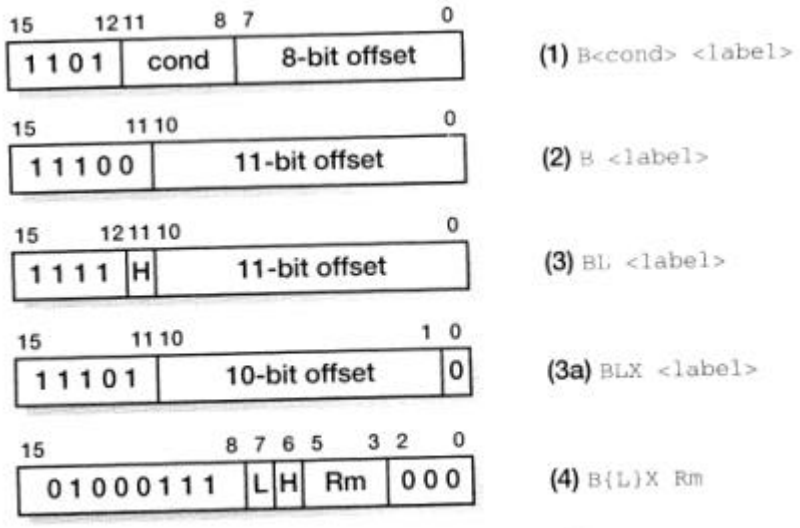


Figure 7.2 Thumb branch instruction binary encodings.

Der erste Befehl ist ein bedingter Sprungbefehl mit einem Offset von 8 Bit
Der zweiter Befehl ist ein Sprung ohne Bedingung mit einem Offset von 11 Bit
Der dritte Befehl wird für Sprünge mit einem Offset von über 11 Bit verwendet, dabei wird der Befehl zweimal ausgeführt. Beim ersten Mal (H=0) wird der Offset nach links um 12 Plätze verschoben und dann mit dem Programmzähler addiert in das Linkregister geschrieben. Beim zweiten Mal (H=1) wird der Offset einen Platz nach links verschoben und mit dem Linkregister addiert in den Programmzähler verschoben. Der Befehl 3a kommt nur in v5T Prozessoren vor und der vierte Befehl ist genau wie der BLX Befehl in dem ARM Befehlssatz.

Hier noch eine Liste von ARM- Befehlen und deren äquivalente Thumb Befehle :

```
ARM instruction                Thumb instruction
MOVS    Rd, #<#imm8>       ; MOV   Rd, #<#imm8>
MVNS    Rd, Rm             ; MVN   Rd, Rm
CMP     Rn, #<#imm8>       ; CMP   Rn, #<#imm8>
CMP     Rn, Rm             ; CMP   Rn, Rm
CMN     Rn, Rm             ; CMN   Rn, Rm
TST     Rn, Rm             ; TST   Rn, Rm
ADDS    Rd, Rn, #<#imm3>   ; ADD   Rd, Rn, #<#imm3>
ADDS    Rd, Rd, #<#imm8>   ; ADD   Rd, #<#imm8>
ADDS    Rd, Rn, Rm         ; ADD   Rd, Rn, Rm
ADCS    Rd, Rd, Rm         ; ADC   Rd, Rm
SUBS    Rd, Rn, #<#imm3>   ; SUB   Rd, Rn, #<#imm3>
SUBS    Rd, Rd, #<#imm8>   ; SUB   Rd, #<#imm8>
SUBS    Rd, Rn, Rm         ; SUB   Rd, Rn, Rm
SBCS    Rd, Rd, Rm         ; SBC   Rd, Rm
RSBS    Rd, Rn, #0         ; NEG   Rd, Rn
MOVS    Rd, Rm, LSL #<#sh> ; LSL   Rd, Rm, #<#sh>
```

Wenn weitere Fragen zum Thema Thumb-Architektur bestehen kann das Buch in der Literaturliste vielleicht weiterhelfen. Dies war nur eine kleine Einführung warum Thumb und wie Thumb zu ARM- Befehlen im allgemeinen steht. Auf den nächsten Seiten werden einige Anwendungsbeispiele dargestellt, wo überall ARM Prozessoren eingesetzt werden.

2. Anwendungsbeispiele

2.1. Mobiltelefon

Der OneC™VWS22100 GSM Chip wurde von VLSI Technology Inc. für GSM Handys entwickelt. Dieser Chip wird beispielsweise in im Samsung SGH2400 Dualband Handy eingesetzt. Der OneC™VWS22100 benutzt einen ARM7TDMI Prozessor der folgende Funktionen übernimmt :

- Die User Interface Software
- Den GSM Protokoll Stack
- Das Energiemanagement
- Ansteuern der externen Anschlüsse
- Einige der Datenapplikationen

Hier eine Grafik vom Aufbau eines GSM Handys um den Onec™VWS22100 :

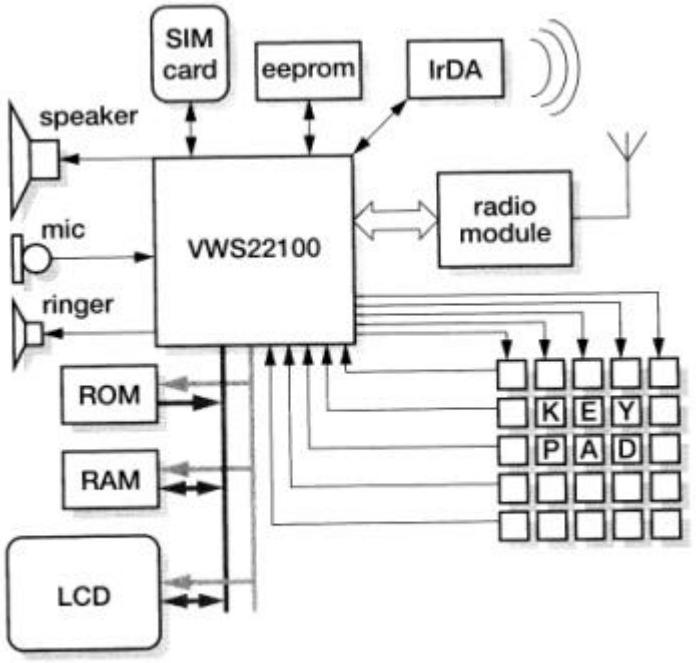


Figure 13.6 Typical GSM handset architecture.

Der OneC™VWS22100 wird auf der nächsten Seite dargestellt.

Hier eine Grafik des Aufbaus vom OneC™VWS22100 GSM Chip :

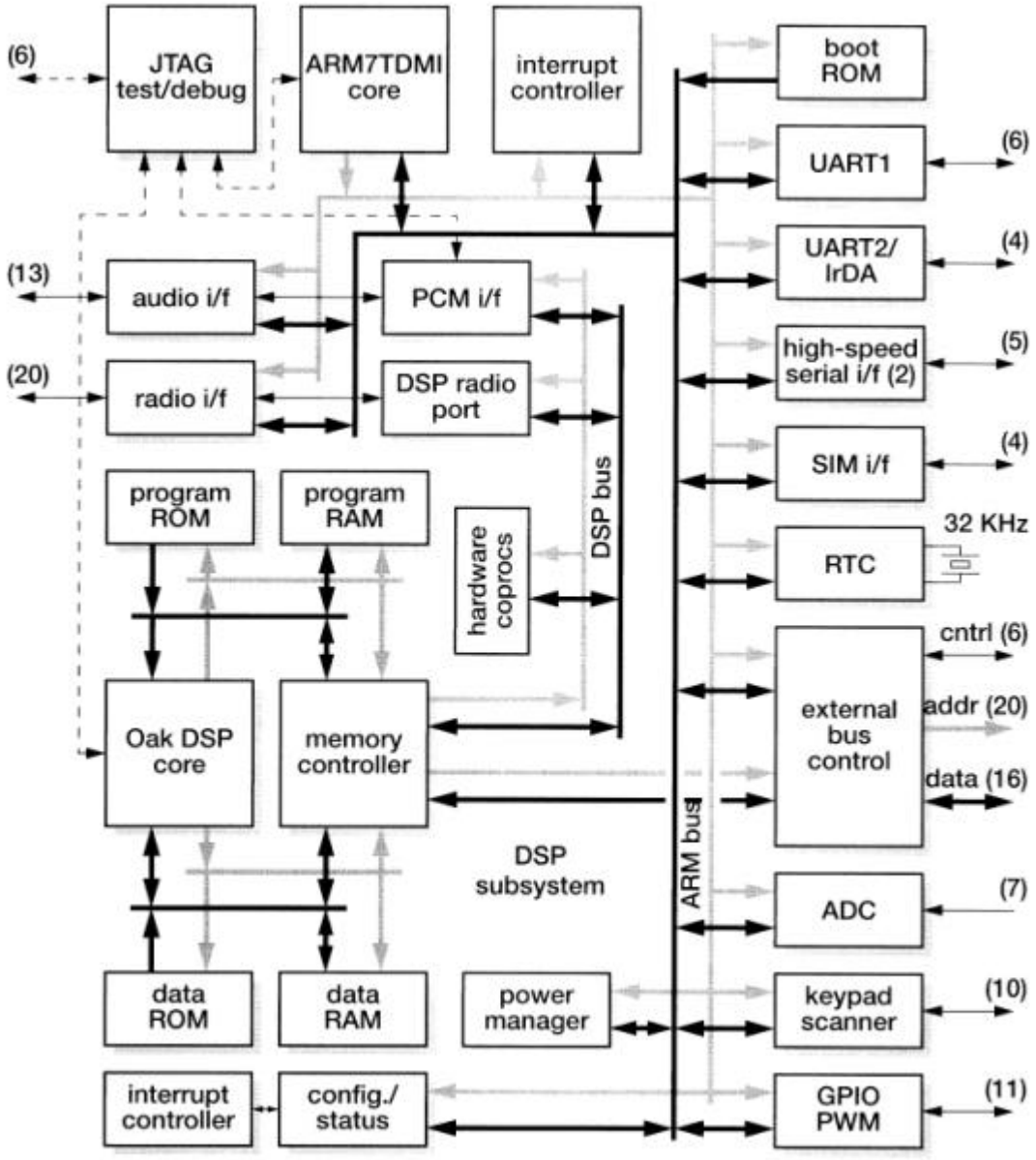


Figure 13.5 OneC VWS22100 GSM chip organization.

2.2. Psion 5

Der Psion 5 ist ein Palmtop Computer der den ARM7100 verwendet.
Die Architektur des ARM7100 ist in der folgenden Grafik dargestellt, er beinhaltet einen ARM7 Prozessor, ein 8 Kbyte 4fach assoziativen cache un einen 4Adress 8Daten word-write Puffer. Der Cache ist hauptsächlich für eine Verbesserung der Energie-Effizienz eingeführt worden, die durch eine Minimierung der Off-Chip Kommunikation erreicht wird.

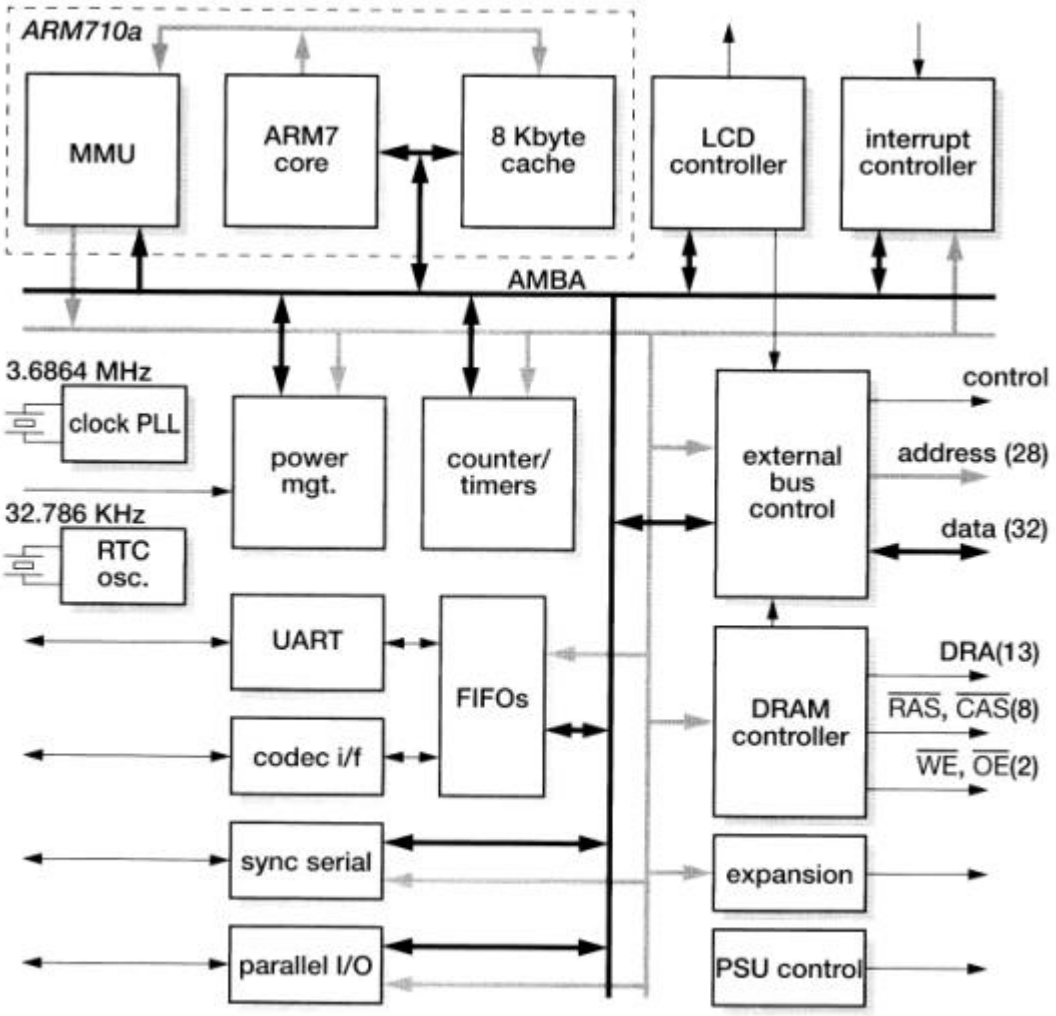


Figure 13.13 ARM7100 organization.

Auf der nächsten Seite eine Grafik wie der ARM7100 in dem Psion5 Palmtop eingebettet wurde.

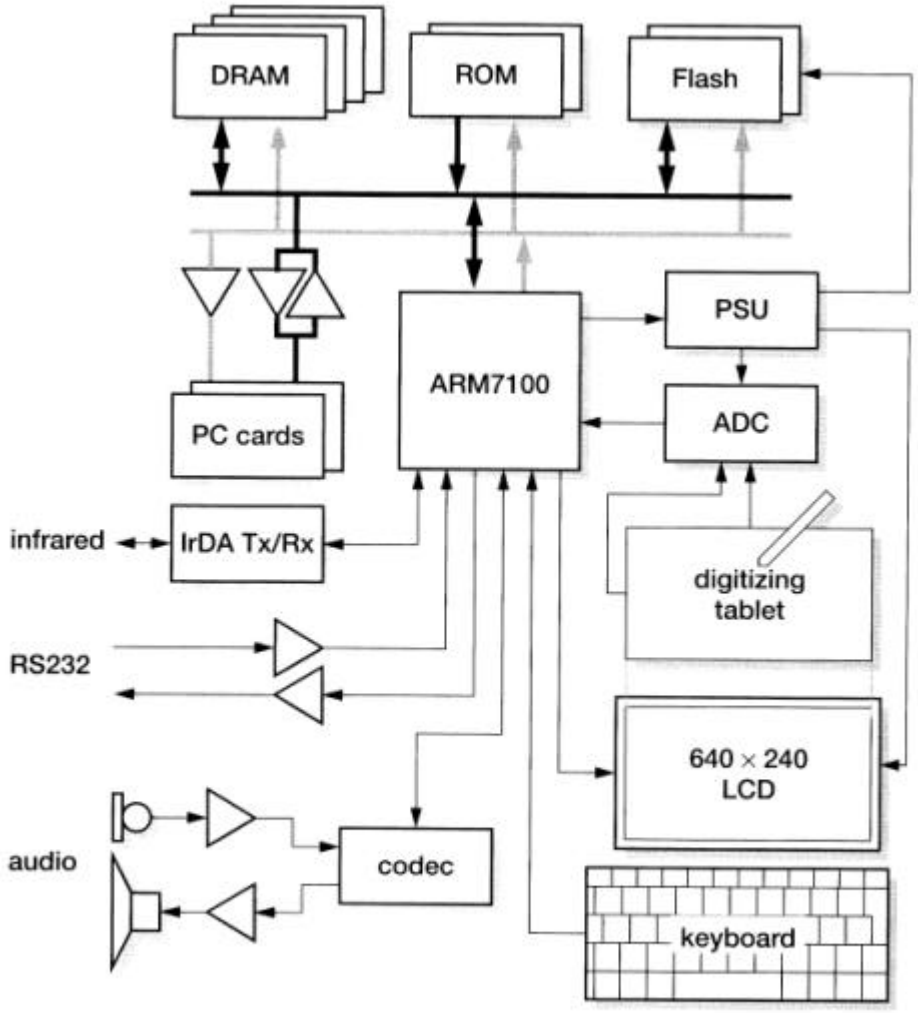


Figure 13.14 The Psion Series 5 hardware organization.

Weitere Anwendungen von ARM Prozessoren :

Außer den hier angesprochen Anwendungen werden ARM Komponenten in vielen weiteren Systemen eingesetzt :

- Einem ISDN-Subscriber, dem VLSI ISDN SUBSCRIBER PROCESSOR (VIP)
- Dem Ericsson-VLSI Bluetooth Baseband Controller
- Einem neuen Game Boy namens GameBoy Advance (siehe www.ARM.com)
- und vielen mehr.

Weiterhin gibt es viele Möglichkeiten ARM Prozessoren mit Co-Prozessoren auszustatten oder die Prozessoren zu verändern.
Zum Beispiel durch die Jazelle™ Technologie wird der ARM Prozessor zu einer Hardware Java Maschine die Java-Befehle ausführen und ByteCode lesen kann.

Für weitere Informationen zu Anwendungen der ARM Prozessoren siehe www.ARM.com