Inhaltsverzeichnis

Kapitel 1

Einleitung

Leistungsfähige und energiesparende Mikroprozessoren werden heutzutage immer mehr in eingebetteten Systemen eingesetzt. Zwei typische Anwendungsbeispiele für die Nutzung solcher Systeme sind die KFZ-Industrie und die Telekommunikationsbranche. So wird bei einem modernen Auto durch den Einsatz spezieller Regelungselektronik der Kraftstoffverbrauch auf eine optimale Menge reduziert. Zusätzliche Systeme unterstützen den Fahrer durch automatisches Eingreifen der Boardelektronik in kritischen Fahrsituationen. Im ruhenden Zustand werden Autos durch intelligente Alarmanlagen gesichert. Der Einsatz dieser Systeme erfordert ein hohes Maß an Energiekosten. Diese Kosten wachsen für jede neue Autogeneration kontinuierlich an.

Bei Mobiltelefonen der neueren Bauart gewinnen leistungsfähige Prozessoren ebenfalls immer mehr an Bedeutung. Durch die Integration von zusätzlichen Anwendungen wie z.B. Terminkalenderfunktionalität, Adressbuchverwaltung und neuerdings auch Internetzugang entwickeln sich Mobiltelefone immer mehr zu portablen Organizern weiter. Eine wichtige Anforderung bei solchen Systemen ist es, eine hohe stromunabhängige Bereitschaftszeit (Standby-Zeit) zu erzielen, da diese den heutigen Benutzeransprüchen nicht entspricht. Weiterhin können die von Herstellern angegebenen Standby-Zeiten nicht immer erfüllt werden. Verschiedene Gründe sind dafür verantwortlich. Einer davon liegt im Nutzungsverhalten des jeweiligen Anwenders. So wird bei einem Mobiltelefon bei regelmäßiger Verwendung des Terminkalenders oder dem täglichen Versenden von E-Mails und SMS-Nachrichten (Short Message Service) die Bereitschaftszeit des Telefons deutlich reduziert, da bei der Berechnung von Standby-Zeiten üblicherweise nur die Faktoren Telefonieren und Empfangsbereitschaft berücksichtigt werden.

Eine interessante Frage, die sich aus der Sicht der Energiebetrachtung stellt, ist, welche Komponenten in solchen Systemen wieviel Energie verbrauchen. Gelingt es den Energieverbrauch in hardware- und softwareabhängige Kosten aufzuteilen, können Optimierungen an beiden Stellen durchgeführt werden.

Unterstützend für den zweiten Fall, die Energieoptimierung von Software, wird an der Universität Dortmund am Fachbereich Informatik, Lehrstuhl 12 ein Compiler entwickelt, der Programme, implementiert in der Hochsprache C, in einem stromoptimierten Maschinencode überführen soll. Die Forschungen konzentrieren sich in der ersten Phase auf einen speziellen Prozessortyp mit RISC-Architektur. Dieser Prozessor zeichnet sich durch seinen geringen Energiebedarf aus und wird hauptsächlich in eingebetteten Systemen eingesetzt. Eine wichtige Anforderung für diesen 'Low Power'-Compiler ist es, zu wissen, welche Instruktionen wieviel Energie verbrauchen, um stromoptimierte Befehlssequenzen bilden zu können. Die vorliegende Diplomarbeit wird sich mit diesem Thema beschäftigen und den Einfluss unterschiedlicher Prozessorinstruktionen auf den Stromverbrauch von Prozessor und externen Arbeitsspeicher genauer untersuchen.

1.1 Motivation

Die Motivation zu dieser Diplomarbeit enstand dadurch, dass viele Hersteller ihre Prozessorarchitekturen nur schematisch anhand von Datenblättern veröffentlichen. Zum Schutz des eigenen Know-Hows werden Architekturbeschreibungen anhand einer hardwarenahen Beschreibungssprache wie VERILOG oder VHDL nicht veröffentlicht. Simulationen, die Energieverbrauchswerte für die verbesserte Codegenerierung von Compilern liefern und diese Beschreibung benötigen, sind daher nicht möglich. Somit können neue Optimierungsansätze von unabhängigen Forschungseinrichtungen nur bedingt in gängigen Prozessoren eingesetzt werden.
Aus dieser Überlegung heraus wurde versucht, auf der nächsthöheren Abstraktionsebene, dem Maschinencode eines Prozessor, Informationen darüber zu gewinnen, inwieweit Energieeinsparung durch das Ersetzen, Reorganisieren und Optimieren von Befehlssequenzen innerhalb eines Maschinenprogrammes vorgenommen werden können. Diese energiesparenden Ansätze sollen zukünftig verstärkt bei der Implementierung des Low-Power-Compilers berücksichtigt werden.

1.2 Ähnliche Arbeiten

Tiwari et. al. [TMW94a] [TMW94b] haben gezielt Forschungen auf dem Gebiet der Energiemessung von Prozessor-Instruktionen betrieben. Über den Kenntnisstand der Verbrauchsinformationen einzelner Instruktionen, gemessen an unterschiedlichen Rechnerarchitekturen, wurden Befehle und Befehlsequenzen in verschiedenen Programmen durch kostengünstigere ersetzt. Das Ergebnis dieser Ersetzung war, am Beispiel eines Sortierprogrammes (Heapsort), eine 40-prozentige Ersparnis an Energiekosten gegenüber dem Original-Programmcode [TMW94a].

Zwei wichtige Ergebnisse aus Tiwaris Forschungen:

1. Unterschiedliche Instruktionen generieren bei Ausführung im Prozessor verschiedene Schaltwechselfolgen. Die Anzahl der benötigten Schaltvorgänge zum Verarbeiten einer Instruktion ist dabei entscheidend für den Energieverbrauch des Prozessors.

2. Ein Energiemodell wurde entwickelt, das alle anfallenden Kosten des Prozessors beim Verarbeiten einer Instruktion berücksichtigt. Das Energiemodell benötigt keine genauen technischen Informationen über den internen Aufbau des eingesetzten Prozessors. Auf Basis dieses Modells lässt sich mit einem geringen messtechnischen Aufwand der Energiebedarf des Prozessors bei Ausführung einer Instruktion oder eines Programmes mit einer hohen Genauigkeit bestimmen.

Energiemessungen von Prozessor-Instruktionen sind weiterhin von Gebotys [GEB98] am Texas Instruments DSP TMS320C5x und von Russel [RUS98] am Intel i960 vorgenommen worden. Stouraitis [SIT99] hat in einem europäischen Forschungsprojekt Messungen von ARM7TDMI-Instruktionen durchgeführt, der auch Gegenstand dieser Diplomarbeit werden wird. Der ARM7TDMI stellt jedoch zwei Instruktionssätze zur Verfügung. Diese Diplomarbeit wird den noch nicht betrachteten Thumb-Befehlssatz untersuchen und dabei den Einfluss einer Instruktion auf den Prozessorstrom und parallel dazu den Stromverbrauch des externen Arbeitsspeichers betrachten.
Die meisten Arbeiten über Messungen von Prozessor-Instruktionen basieren auf Tiwaris Forschungsergebnissen und seinem aufgestellten Energiemodell, das auch Grundlage dieser Arbeit werden wird.

1.3 Ziele dieser Arbeit

Ein erstes Ziel dieser Arbeit besteht darin, Tiwaris Energiemodell funktional zu erweitern, da es nur den Einfluss einer Instruktion auf die Prozessorkosten beschreibt. Die Erweiterung sieht vor, anfallende Kosten des externen Arbeitspeichers mit zu berücksichtigen. Ein weiteres Ziel besteht in der Überführung des aufgestellten Energiemodells auf den ARM7TDMI-Prozessor. Für diese Überführung sind geeignete Testmuster zu entwerfen, die alle Kostenarten des Energiemodells abdecken. Ein letztes Ziel ist die Dokumentation und Bewertung der durchgeführten Messreihen für den ARM7TDMI-Prozessor und den dazugehörigen Arbeitsspeicher. Die Bewertung soll im Hinblick auf möglicher Energieoptimierungen durchgeführt werden.

1.4 Kapitelübersicht

In einer kurzen Kapitelübersicht wird der chronologische Aufbau dieser Arbeit beschrieben:

In Kapitel 2 wird eine Einführung in das Themengebiet 'Low Power' vorgenommen. Dazu werden typische Anwendungsgebiete für die Low-Power-Forschung vorgestellt. Anschließend erfolgt die Definition zweier physikalischer Größen, die Gegenstand jeder Energieoptimierung sind. Weiterhin werden Energiekosten von Halbleiterschaltungen betrachtet und Verfahren zur Energieoptimierung vorgestellt, die im Rahmen des Hardwareentwurfs und beim Compilerbau eingesetzt werden. In Kapitel 3 wird der für die Energiemessung verwendete ARM7TDMI-Prozessor und die dazugehörige Versuchsanordnung beschrieben.

Kapitel 4 stellt Tiwaris Energiemodell vor und erläutert die vorgenommene Erweiterung für die Betrachtung des externen Hauptspeichers. Weiterhin wird für die Anwendung des Energiemodells auf den ARM7TDMI-Prozessor eine Auswahl der verwendeten Testmuster gezeigt und die Messmethodik dargelegt. Kapitel 5 beschreibt die Umsetzung einer Messsoftware, in der alle Testmuster integriert worden sind. In Kapitel 6 erfolgt die Dokumentation und Bewertung der gemessenen Energiedaten. Abschließend fasst Kapitel 7 alle zentralen Punkte dieser Diplomarbeit noch einmal zusammen und gibt einen kurzen Ausblick auf zukünftige Forschungsansätze.

Kapitel 2

Einführung in Low Power

Das Forschungsgebiet 'Low Power' beschäftigt sich mit Methoden zur Energieoptimierung von Hardware- und Software-Systemen. Die erzielten Ergebnisse sind für viele Anwendungen interessant, bei denen die Reduktion von Energiekosten einen hohen Stellenwert besitzt. Dazu gehören Systeme mit folgenden Eigenschaften (angelehnt an [SW99]):

- **begrenzte Energieversorgung**

  Akkumulatorkapazitäten begrenzen mobile Systeme in ihrer Nutzungsdauer. Hohe Bereitschaftszeiten sind bei diesen Systemen wichtige Kriterien. Abbildung 2.1 zeigt jedoch, dass die Entwicklung von leistungsfähigen Akkumulatorspeichern nur langsam voranschreitet. In knapp dreißig Jahren hat sich die Nominalkapazität von Nickel-Cadmium-Akkumulatoren etwas mehr als verdoppelt. Zukunftsweisender erscheinen neuere Verfahren auf Basis von Nickel-Metal-Hydrid oder Lithium-Ionen, die seit Anfang der 90er Jahre im Einsatz sind.

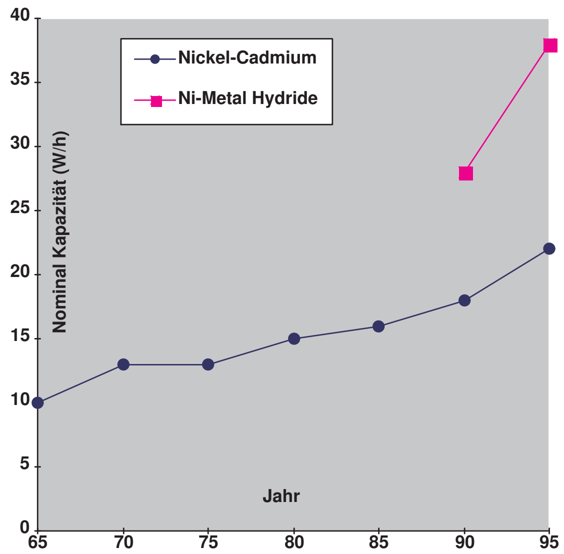

Abbildung 2.1: Entwicklung von Akkukapazitäten [TIW96]

- **kritische Wärmeentwicklung**

  Bei einem hohem Energieverbrauch können kritische thermische Effekte entstehen. Als Gegenmaßnahme muss die Wärmeentwicklung durch Kühlkörper reduziert werden. Platzmangel gerade in eingebetteten Systemen, schließt eine separate Kühlung oft aus. Weiterhin haben aktive Kühlkörper selbst einen hohen Energieverbrauch.

- **erhöhte Zuverlässigkeit**

  Durch eine geringe Leistungsaufnahme wird die Lebensdauer und die Zuverlässigkeit von Halbleiterbausteinen erhöht.

2.1 Physikalische Grundlagen

Im folgenden Abschnitt werden die Begriffe Energie und Leistung definiert. Beide physikalischen Größen werden in der Low-Power-Forschung als Grundlage für unterschiedliche Optimierungsansätze verwendet. Die folgenden Definitionen sind aus [PRE85] [TIW96] entnommen worden.

2.1.1 Leistung

Die Leistung P ist definiert als Produkt aus der Spannung U und dem Strom I.

$$P = U \cdot I$$

P hat die Einheit 1W (Watt)= 1V (Volt) $\cdot$ 1A (Ampere). Die Leistung eines Prozessors beim Ausführen eines Programmes ist dementsprechend

$$P = V_{dd} \cdot I$$

wobei $V_{dd} = U$ entspricht, und die Spannungsversorgung des Prozessors darstellt. Der Strom I stellt den durchschnittlichen Strombedarf dar.

2.1.2 Energie

Wird die Leistung P mit der Zeit T multipliziert, erhält man die Energie E.

$$E = P \cdot T$$

E hat die Einheit 1J (Joule)= 1Ws = 1VAs. Die Zeiteinheit T in Sekunden entspricht bei Betrachtung des Energieverbrauchs am Prozessor der Gesamtausführungszeit eines Programmes. Die Zeit T kann weiter aufgeteilt werden in

$$T = N \cdot t$$

N entspricht der Anzahl aller benötigten Taktzyklen eines Programmes und t der Taktperiode des Systems. Demnach ist der Energiebedarf eines Programms vollständig definiert als

$$E = V_{dd} \cdot I \cdot N \cdot t$$

Geht man davon aus, dass beim Ablauf eines Programmes die Versorgungsspannung V_{dd} und die Taktperiode t konstante Größen sind, folgt, dass der Energiebedarf eines Programmes proportional mit der Anzahl der Taktzyklen N und dem Stromverbrauch I für die Abarbeitung der einzelner Prozessorinstruktion wächst.

2.2 Sichtweise des Begriffs Energie

Die Unterscheidung zwischen verbrauchter Energie und Leistung bei Ablauf eines Programmes wird an folgenden Programmbeispiel [TIW96] deutlich:

```
MOV DX, [BX]
MOV AX, CX
ADD AX, DX


Leistung = 1.15 Watt
Energie  = 8.6 x 10^-8 Joule

      (a)
```

```
NOP
MOV DX, [BX]
NOP
NOP
MOV AX, CX
NOP
NOP
ADD AX, DX
NOP

Leistung = 0.99 Watt
Energie  = 22.3 x 10^-8 Joule

      (b)
```

Die zwei Assemblersequenzen für einen Intel486-Prozessor sind, bis auf die eingestreuten NOP-Befehle in der Sequenz (b), identisch. Dies hat zur Folge, dass die durchschnittliche Leistung dieser Sequenz (b) gegenüber (a) 14 Prozent günstiger bewertet wird, obwohl das Programm länger ist und mehr Taktzyklen benötigt. Der Grund hierfür liegt in der Verwendung der NOP-Befehle. Im Gegensatz zu den anderen Instruktionen ist der Stromverbrauch des Prozessors bei Abarbeitung eines NOP-Befehls sehr gering, da keine Datenoperationen durchgeführt werden. Bei der Berechnung des durchschnittlichen Leistungsverbrauchs eines Programmes helfen stromsparende Instruktionen, um Leistungsspitzen in einem bestimmten Zeitintervall zu glätten. Der tatsächliche Energiebedarf von (b) ist jedoch mehr als doppelt so hoch wie der von (a), da für die Energiebetrachtung, aufgrund der höheren Zyklenzahl von (b), die längere Bearbeitungszeit mitberücksichtigt werden muss.

Für Low-Power-Anwendungen ist in der Regel die Energiebetrachtung wichtiger als die Leistungsbetrachtung, da durch die Begrenzung von Energieressourcen gerade diese optimiert werden sollen. Das Vermeiden von Leistungsspitzen ist z.B. in Systemen wichtig, wo kritische Wärmeentwicklungen vermieden werden sollen.

2.3 Energiekosten bei aktiven Schaltungen

Betrachtet man innerhalb eines Prozessors eine Halbleiterschaltung in CMOS-Technologie, ergeben sich drei technische Ursachen [SYN96] für den Energieverbrauch:

- **Switching Power**

  Laden und Entladen von Lastkapazitäten am Ausgang einer Zelle haben bei aktiven CMOS-Schaltungen einen Anteil von 70 bis 90 Prozent am Gesamtenergieverbrauch. Der Energiebedarf P_{sw} einer Zelle ist dabei definiert als:

  $$P_{sw} = \frac{V_{dd}^2}{2} \sum_{nets(i)} (C_{Load_i} \cdot TR_i)$$

  wobei V_{dd} der Versorgungsspannung entspricht. C_{Load_i} entspricht den Lastkapazitäten einer Zelle und TR_i der Wechselrate beim Laden und Entladen der Kapazität.

- **Short Circuit Power**

  Beim Schalten von Transistoren treten kurzfristig Kurzschlussströme auf. Bei Halbleitertechnologien mit langsamen Transistorschaltzeiten können diese bis zu 30 Prozent am Gesamtenergieverbrauch ausmachen.

  $$P_{sc} = \sum_{cell(j)} f(C_{Load_j}, TransTime_{input}) \cdot TR_j$$

  $TransTime_{input}$ entspricht der Schaltzeit eines Transistors.

- **Leakage Power**

  Im inaktiven Zustand, d.h. wenn der Transistor nicht schaltet, wird ebenfalls Energie verbraucht. Jedoch ist der Wert mit unter 1 Prozent am Gesamtenergieverbrauch minimal.

  $$P_{lk} = \sum_{cells(i)} P_{CellLeakage_i}$$

  $P_{CellLeakage_i}$ entspricht dem Energieverbrauch einer Zelle in einem inaktiven Zustand.

Die Abbildung 2.2 zeigt ein Beispiel für switching-, short-circuit- und leakage-power bei einer Inverterschaltung in CMOS-Technologie. Das oberste Ziel, das Laden und Entladen von Lastkapazitäten zu reduzieren, wird beim Entwurf durch Einsatz spezieller Optimierungsverfahren erzielt. Zu nennen sind Verfahren wie Clock Gating zur Reduktion der Schalthäufigkeit von Gattern oder Bus Encoding zur Senkung von Schaltwechselfolgen innerhalb des Bussystems [SS00].

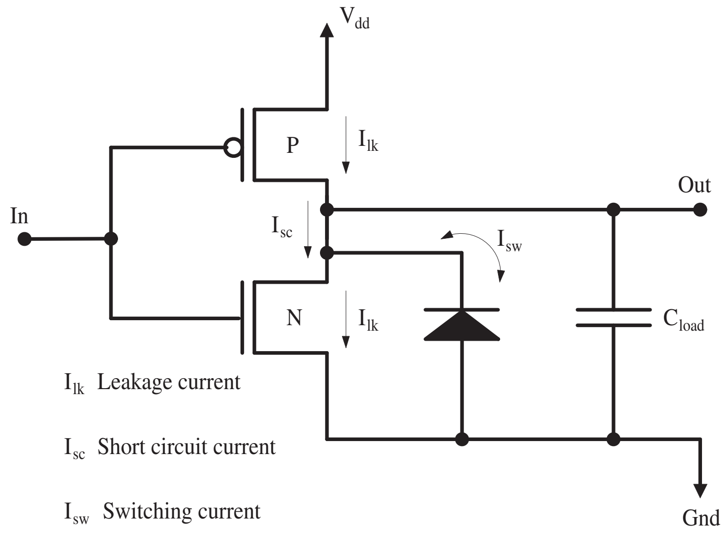

Abbildung 2.2: Inverter in CMOS-Technologie [SYN96]

Weitere Reduktionsmöglichkeiten zum Absenken der Energiekosten aktiver Schaltungen werden durch zwei weitere Techniken motiviert:

- **Reduktion der Versorgungsspannung am Prozessor**

  Ein neuerer Ansatz ist das Herabsetzen der Versorgungsspannung des Prozessors bei gleichzeitiger Senkung der Taktgeschwindigkeit, falls dieser nicht mit voller Rechenleistung arbeiten muss. Dieser Ansatz wird motiviert durch das ungleiche Verhältnis von Versorgungsspannung zu Energiekosten. Energiekosten steigen proportional zum Quadrat der Leistung [YAS00].

- **Abschaltung der Peripherie**

  Intelligente Power-Management-Systeme schalten nach einer festgelegten Zeitvorgabe automatisch nicht verwendete Peripherie in einen energiegünstigeren Standby-Modus zurück. Bei einem Zugriff erfolgt ein automatisches Zurückschalten in den normalen Betriebsmodus. Ein Nachteil dieser Technik ist jedoch die Verzögerungszeit, die auftritt, bis die Peripherie nach dem Zurückschalten wieder betriebsbereit ist.

2.4 Techniken im Compilerbau

Es folgt eine Auswahl an Möglichkeiten, um den Energieverbrauch über Methoden der Softwareoptimierung zu senken. Diese Methoden werden im Rahmen des Compilerbaus eingesetzt. Die Grundidee ist hierbei, andere Befehlssequenzen zu verwenden, die schneller sind oder weniger Energie benötigen.

- **Reduced Memory Access**

  Externe Speicherzugriffe verursachen hohe Energiekosten. Compiler sollten den Zugriff auf den externen Speicher nach Möglichkeit vermeiden oder alternativ kostengünstigere Speicherbereiche nutzen, wie zum Beispiel den Scratch-Pad-Speicher innerhalb eines Prozessors, falls dieser vorhanden ist. Ein Optimierungsverfahren zur Reduktion von Speicherzugriffen ist Registerpipelining [SS00].

- **Strength Reduction**

  Teure Befehle werden bei dieser Technik durch adäquate Kostengünstigere ersetzt. Ein typisches Beispiel für Strength-Reduction ist das Ersetzen einer teuren mehrzyklischen Multiplikationsoperation $i = i \cdot 2$, durch eine billigere Shift-Left $i = i \ll 1$ Anweisung [SIT99].

- **Instruction Reordering**

  Befehle und Befehlssequenzen werden so umgruppiert, dass Schaltkreisaktivitäten minimiert werden. Die logische Abhängigkeit der Befehlssequenzen untereinander und die Datenabhängigkeit darf nach der Reorganisation nicht verletzt werden. Durch Instruction-Reordering werden teure externe Datenzugriffe bei Zugriffsfehlern auf Cache-Speichersysteme (Cache-Misses) und Wartezustände des Fließbands (Pipeline-Stalls) ebenfalls reduziert [SIT99].

- **Data Regeneration**

  Daten werden bei diesem Verfahren nicht über teure Speicherzugriffe erneut geladen, sondern kostengünstiger neu berechnet.

Kapitel 3

ARM7TDMI-Prozessor

Für die Energiemessungen der Befehlsinstruktionen wurde der kurz in der Einleitung erwähnte ARM7TDMI-Prozessor ausgewählt. Der ARM7TDMI ist ein 32-Bit-RISC-Prozessor, der aufgrund seines niedrigen Energieverbrauchs und der schnellen spezifischen Verarbeitungsgeschwindigkeit von 117 Mips/Watt [ARM95a] seinen Einsatz in eingebetteten Systemen, Mobiltelefonen und Kleinstcomputern findet. Weitere Besonderheiten des ARM7TDMI-Prozessors sind seine zwei Befehlssätze. Die Befehlssätze unterscheiden sich in ihrer Wortbreite und der Anzahl ihrer Instruktionen.
Der erste Befehlssatz, genannt ARM-Instruction-Set, hat eine Instruktionswortbreite von 32 Bit und zeichnet sich durch seinen großen Befehlssatz für RISC-Prozessoren und seiner Schnelligkeit aus. Der ARM-Befehlssatz beinhaltet insgesamt 80 Kernbefehle.
Der zweite Befehlsatz ist der Thumb-Instruction-Set. Thumb-Instruktionen haben eine Wortbreite von 16 Bit und bieten den Vorteil einer hohen Codedichte sowie eines geringeren Energieverbrauchs gegenüber ARM-Instruktionen. Der Thumb-Instruction-Set stellt als Untermenge vom ARM-Instruction-Set nur 36 Befehle zur Verfügung. Bedingt dadurch, dass die Wortbreite einer Instruktion im Thumb-Modus auf 16 Bit reduziert ist, sind viele Adressierungsarten, die im ARM-Modus möglich sind, aus Platzgründen innerhalb des Befehlswortes ebenfalls eingeschränkt.

3.1 Eigenschaften des ARM7TDMI

Der ARM7TDMI-Prozessor zeichnet sich durch folgende Eigenschaften aus:

- 32-BIT-RISC
- 31 $\times$ 32-BIT-Register sowie 6 zusätzliche Status Register
- klassische Load-/Store-Architektur
- dreistufige Instruction-Pipeline
- 32-BIT-ALU

- separater Barrelshifter
- 32-BIT-Multiplizierwerk
- 32-BIT-Adress- und Datenbus

Der Datenpfad des ARM7TDMI-Prozessors ist in Abbildung 3.1 dargestellt.

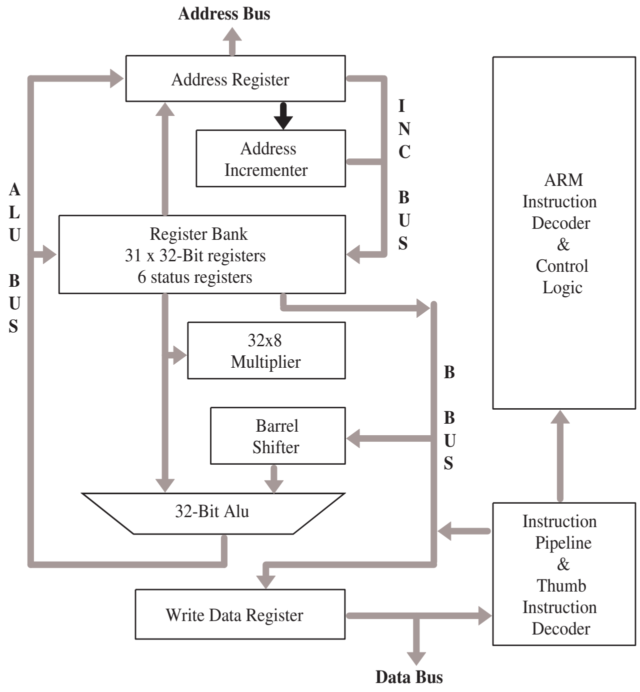

Abbildung 3.1: Datenpfad des ARM7TDMI-Prozessors [ARM95a]

3.1.1 Registerorganisation

Der ARM7TDMI-Prozessor stellt 31 Register mit einer Wortbreite von 32 Bit zur Verfügung, davon sind 16 Register (R0-R15) für den ARM-Instruction-Set im Benutzermodus (Usermode) frei verfügbar. Die übrigen Register sind für Ausnahmebehandlungen (Exceptions) reserviert.
Im Thumb-Modus ist die Registermenge eingeschränkt. Es stehen hier 8 frei verfügbare Register (R0-R7) in der unteren Registerbank zur Verfügung. 5 weitere Register (R8-12) in der oberen Bank sind nur über spezielle Thumb-Instruktionen ansprechbar. Register R13-R15 sowie die Statusregister CPSR und SPSR haben spezielle Aufgaben:
In Register R13 wird der Stackpointer abgelegt. Register R14 speichert bei 'Branch and Link'-Anweisungen eine Kopie vom aktuellen Program-Counter, um die Rücksprungadresse zu sichern. Auch dient dieses Register als Ablageort für Rücksprungadressen bei ausgelösten Ausnahmebehandlungen durch Interrupts. Register R15 ist für den Program-Counter bestimmt. Im CPSR (Current Program Status Register) und SPSR (Saved Process Status Register) werden

Condition-Flags sowie aktuelle Zustand-Bits abgespeichert. Abbildung 3.2 zeigt die Thumb-Registerbank im Überblick.

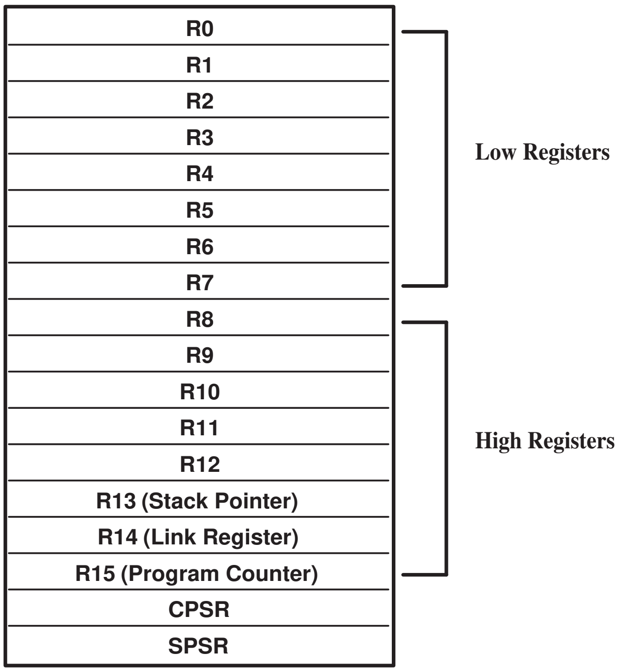

Abbildung 3.2: Registerbank des ARM7TDMI [ARM95a]

3.1.2 Unterstützte Wortgrößen

Der ARM7TDMI unterstützt 8-Bit-Byte-, 16-Bit-Halfword- sowie 32-Bit-Word-Speicherzugriffe. Der Speicher ist dabei linear angeordnet, wobei Byte 0 bis 3 das erste Wort, und Byte 4 bis 7 das zweite Wort hält. Der ARM7TDMI unterstützt das Ablegen von Daten im Big-Endian- und Little-Endian-Format.

3.1.3 Instruction-Pipeline

Der ARM7TDMI besitzt eine dreistufige Befehls-Pipeline, wie Abbildung 3.3 zeigt. Die Stufen der Pipeline bestehen aus einer Instruction-Fetch-, Instruction-Decode- und einer Instruction-Execute-Phase. Eine Besonderheit innerhalb der Instruction-Decode-Phase ist die Behandlungsweise von Thumb-Instruktionen. Beim Dekodieren erfogt ein Umsetzen (decompress) eines Thumb-Befehls auf den ARM-Befehlsatz, bevor dieser ausgeführt wird. Abbildung 3.4 zeigt ein Beispiel für diese Umsetzung anhand der Dekodierung einer ADD-Immediate-Instruktion von Thumb- nach ARM-Code (Opcode-Darstellung). Die Effizienz der Pipeline zeigt sich bei näherer Betrachtung in der Zyklenzeit beim Abarbeiten eines Befehls in der Decode-Phase. In nur einem Taktzyklus wird ein Befehl von Thumb- nach ARM-Code dekodiert. Abbildung 3.5 zeigt dazu ein Blockschaltbild des ARM-Dekodierers.

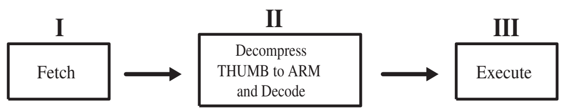

Abbildung 3.3: dreistufige Instruction-Pipeline des ARM7TDMI

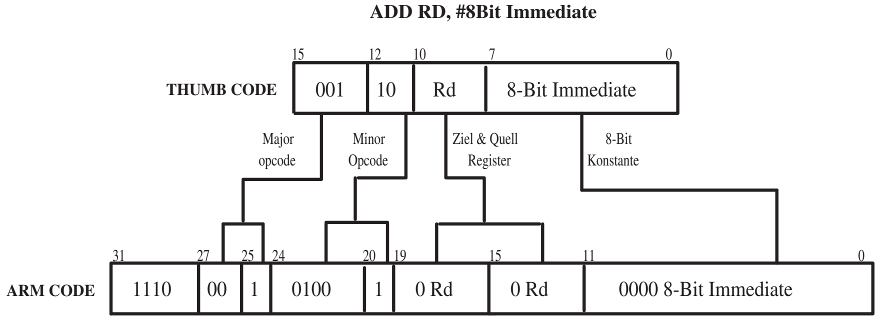

Abbildung 3.4: Umsetzung einer Thumb-ADD-Instruktion in ARM-Code

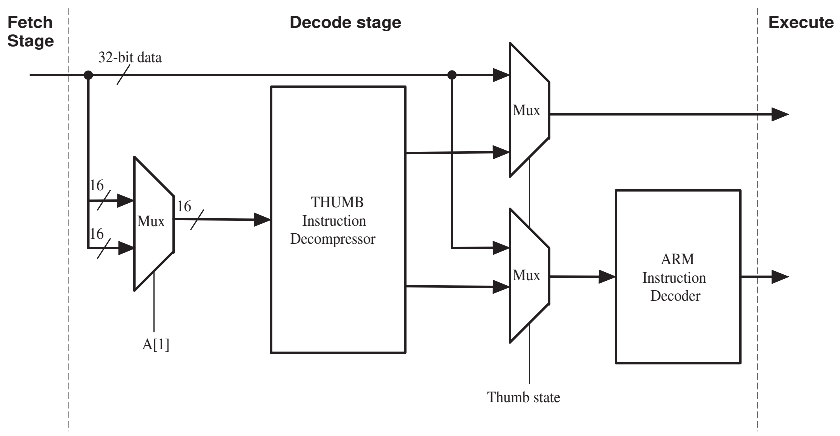

Abbildung 3.5: Blockschaltbild des ARM7TDMI-Dekodierers [ARM95a]

3.1.4 Thumb- vs. ARM-Instruktionen

Bedingt dadurch, dass Thumb-Instruktionen nur 16 Bit breit sind, lässt sich eine bis zu 30 Prozent höhere Codedichte beim Programmentwurf gegenüber ARM-Instruktionen erreichen. Dies wurde in verschiedenen Benchmark-Untersuchungen [ARM95a] bestätigt. Thumb- und ARM-Instruktionen können wahlweise abwechselnd im selben Programm benutzt werden. Durch das Setzen einer BX-Direktive (Branch and Exchange) wird der Wechsel von einem Modus in den anderen erreicht.
Die hohe Codedichte von Thumb-Instruktionen wird an einem Beispiel deutlich. Es stellt einen Binär- nach Hex-Konverter dar, jeweils implementiert für beide Befehlssätze.

```
ARM CODE                      THUMB CODE


MOV r1, r0                    MOV r1, r0
MOV r2, #8                    MOV r2, #8

Loop                          Loop1
MOV r1, r1, ROR #28           LSR r0, r1, #28
AND r0, r1, #15               LSL r1, r1, #4
CMP r0, #10                   CMP r0, #10
ADDLT r0, r0, #'0'            BLT Loop2
ADDGE r0, r0, #'A'            ADD r0, #'A'-'0'-10

                              Loop2
SWI 0                         ADD r0, #'0'
SUBS r2, r2, #1               SWI 0
BGT Loop                      SUB r2, #1
MOV pc, lr                    BNE Loop1
                              MOV pc, lr

   (a)                           (b)
```

Der ARM-Code (a) des Konverters hat insgesamt eine Programmlänge von 44 Bytes bei 11 Instruktionen. Das gleiche Programm im Thumb-Modus (b) implementiert hat 12 Befehle, ist aber nur 24 Bytes lang. Die Längenersparnis beträgt in diesem Beispiel ca. 45 Prozent.

3.1.5 Thumb-Befehlsübersicht

Der Thumb-Befehlssatz beinhaltet insgesamt 36 Kerninstruktionen und wird in diesem Abschnitt genauer vorgestellt, da er für die Messungen der Prozessorinstruktionen verwendet wird.

- **Registertransfer- und ALU-Operationen**

  Der Thumb-Befehlssatz unterstützt Transferoperationen innerhalb der Register durch Move-Anweisungen und stellt ALU-typische arithmetische und logische Operationen zur Verfügung. Bedingt durch die Load/Store-Architektur des ARM7TDMI-Prozessors werden alle Move- und ALU-Operationen ausschließlich auf den Registern durchgeführt. Die Bereitstellung von Daten aus dem Speicher erfolgt deshalb immer unter Zuhilfenahme einer zusätzlichen Load-Instruktion.

- **Branch-Operationen**

  Branch-Befehle können unkonditional und in Abhängigkeit vom CPSR-Register konditional ausgeführt werden. Die Sprungweiten eines Branch-Conditional-Befehls sind auf 256 Byte begrenzt. Ein unkonditionaler Branch hat eine Sprungweite von 2K-Byte, als Long Branch 4M-Byte.

- **Load- und Store-Operationen**

  Neben den verschiedenen Load- und Store-Instruktionsarten unterstützt der ARM7TDMI-Prozessor 'Multiple Data Instructions'. Dadurch kann diese Instruktion mit bis zu 8 Registern gleichzeitig ausgeführt werden.

- **Stack-Operationen**

  Der Stack ist linear absteigend organisiert und wird mit Push- und Pop-Instruktionen angesprochen. Beide Befehle unterstützen ebenfalls Multiple-Operationen. Diese werden in diesem Fall dazu verwendet, um bis zu 8 Register gleichzeitig auf den Stack abzulegen oder wieder einzulesen.

Tabelle 3.1 zeigt den Thumb-Kernbefehlssatz, mit Taktzyklenangaben zu jedem Befehl und einer kurzen formalen Beschreibung. Aus Gründen der Übersicht sind nicht alle möglichen Adressierungsarten angegeben. Im Anhang A ist der Thumb-Befehlssatz vollständig aufgelistet.

Instruktion	Instruktionsart	Zyklen	Aktion
ADC (Op1, Op2)	Add with Carry	1	Op1:= Op1 + Op2 + C-bit
ADD (Op1, Op2)	Add	1	Op1:= Op1 + Op2
AND (Op1, Op2)	AND	2	Op1:= Op1 AND Op2
ASR (Op1, Op2)	Arithmetic Shift Right	1	Op1:= Op1 ASR Op2
BIC (Op1, Op2)	Bit Clear	1	Op1:= Op1 AND NOT Op2
CMN (Op1, Op2)	Compare Negative	1	CPSR Flags:= Op1 + Op2
CMP (Op1, Op2)	Compare	1	CPSR Flags:= Op1 - Op2
EOR (Op1, Op2)	EOR	1	Op1:= Op1 EOR Op2
LSL (Op1, Op2)	Logical Shift Left	2	Op1:= Op1 $\ll$ Op2
LSR (Op1, Op2)	Logical Shift Right	2	Op1:= Op1 $\gg$ Op2
MUL (Op1, Op2)	Multiply	2-5	Op1:= Op1 $\cdot$ Op2
NEG (Op1, Op2)	Negate	1	Op1:= - Op2
ORR (Op1, Op2)	OR	1	Op1:= Op1 OR Op2
ROR (Op1, Op2)	Rotate Right	1	Op1:= Op1 ROR Op2
SBC (Op1, Op2)	Subtract with Carry	1	Op1:= Op1 - Op2 - NOT C-Bit
SUB (Op1, Op2)	Subtract	1	Op1:= Op1 - Op2
B (Op1)	Unconditional Branch	3	PC:= Op1
BCC (Op1)	Conditional Branch	3	PC:= Op1
BL (Op1)	Branch and Link	3	LR:= PC - 2; PC:= Op1
BX (Op1)	Branch and Exchange	3	PC:= Op1
LDMIA (Op1!, OpN)	Load Multiple	3-10	OpN := [Op1]; Op1:=Op1 + 4$\cdot$N
LDR (Op1, Op2)	Load Word	3	Op1:= [Op2]
LDRB (Op1, Op2)	Load Byte	3	Op1:= [Op2]
LDRH (Op1, Op2)	Load Halfword	3	Op1:= [Op2]
LDRSB (Op1, Op2)	Load signed Byte	3	Op1:= [Op2]
LDRSH (Op1, Op2)	Load signed Halfword	3	Op1:= [Op2]
MOV (Op1, Op2)	Move Register	1	Op1:= Op2
STMIA (Op1!, OpN)	Store Multiple	2-9	[OpN] := Op1; Op1:=Op1 + 4$\cdot$N
STR (Op1, Op2)	Store Word	2	[Op2]:= Op1
STRB (Op1, Op2)	Store Byte	2	[Op2]:= Op1
STRH (Op1, Op2)	Store Halfword	2	[Op2]:= Op1
POP (OpN)	Pop Multiple	3-10	OpN:= [SP]; SP:=SP - 4$\cdot$N
PUSH (OpN)	Push Multiple	2-9	[SP]:= OpN; SP:=SP + 4$\cdot$N
SWI (Op1)	Software Interrupt	2	LR:= PC - 2; PC:= Op1

Tabelle 3.1: Thumb-Kernbefehlssatz

3.2 Messaufbau

Zur Untersuchung der Energieeigenschaften aller Thumb-Instruktionen sind Strommessungen unter Zuhilfenahme eines Evaluationboard mit ARM7TDMI-Prozessor, eines Digitalmultimeters sowie eines Host-PC durchgeführt worden. Die Abbildung 3.6 zeigt den verwendeten Versuchsaufbau. Über den Host-PC wird das Evaluationboard angesteuert. Somit lassen sich Programme zum Board übertragen und ausführen. Das Digitalmultimeter wird zur Strommessung wahlweise am Prozessor oder am externen Speicher angeschlossen.

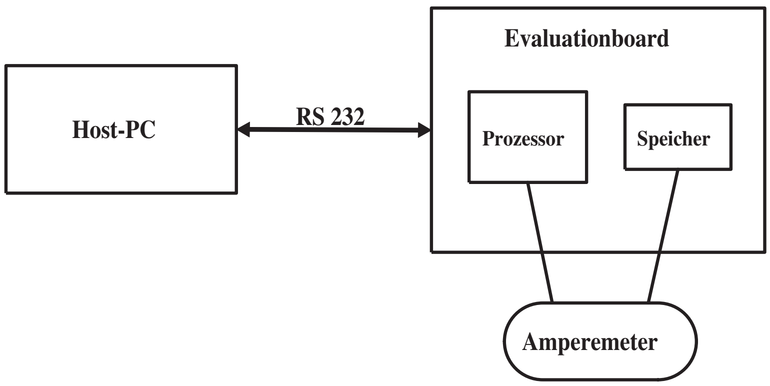

Abbildung 3.6: Versuchsanordnung

3.3 ATMEL-Evaluationboard

Das eingesetzte Evaluationboard ist von der Firma ATMEL-Corporation und trägt die Bezeichnung AT91EB01. Das Board zeichnet sich durch folgende Eigenschaften aus:

- M40400 Microcontroller mit ARM7TDMI Core
- 512K-Byte 16-Bit-SRAM
- 128K-Byte 16-Bit-FLASH, 64K-Byte davon frei verfügbar
- 16-Bit-Datenbus und 24-Bit-Adressbus
- 2 serielle Schnittstellen
- JTAG ICE Debug Interface
- Angel Debug Monitor

Die Versorgungsspannung beträgt 3,3 Volt. Die Systemtaktfrequenz für das Bussystem und den Prozessor ist auf 32,768 MHz eingestellt und kann bei Bedarf heruntergetaktet werden. Über die seriellen Schnittstellen oder über das integrierte JTAG-Interface erfolgt die externe Ansteuerung des Boards über den Host-PC. Abbildung 3.7 zeigt ein Blockdiagramm des Evaluationboards.

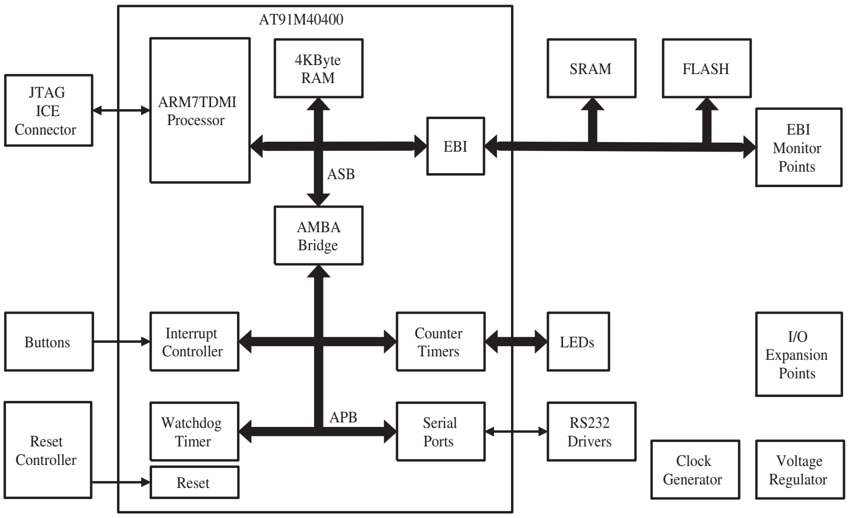

Abbildung 3.7: Evaluationboard AT91EB01 [ATM98b]

3.3.1 Microcontroller AT91M40400

Der eingesetzte Microcontroller AT91M40400 auf dem Evaluationboard besitzt neben dem ARM7TDMI-Prozessorcore einen 4K-Byte grossen On-Chip-Speicher sowie zusätzliche Peripherie. Der On-Chip-Speicher ist für Programm-Code und Daten frei verfügbar, da er vom Prozessor nicht als Cache verwendet wird. Abbildung 3.8 zeigt den ATMEL-Prozessor in einer Detailansicht.

Strommessung am Prozessor

Um Strommessungen am Prozessor durchzuführen, bietet das Board von sich aus eine entsprechende Abgriffmöglichkeit für ein Messgerät an. Jedoch sind isolierte Messungen nur vom ARM7TDMI-Core ohne Betrachtung zusätzlicher Prozessorperipherie wie z.B. des On-Chip-Speichers nicht möglich, da beide Komponenten feste Bestandteile des ATMEL-Chips sind.

3.3.2 Hauptspeicher

Der Hauptspeicher ist in 4 Speicherbänken zu je 128K-Byte SRAM organisiert. Abbildung 3.9 zeigt einen der vier ATMEL-Speicherbausteine mit der Typenbezeichnung 'IDT71V124SA CMOS Static Ram'.

Strommessung am Speicher

Bedingt dadurch, dass Messungen des Speicherstroms für dieses Evaluationboard vom Hersteller nicht vorgesehen sind, wurde zusätzlich an einem der vier Speicherbausteine eine entsprechende Abgriffmöglichkeit angelötet. Anschließend konnte zwischen V_{dd} und GND des Speicherbausteins das Messgerät angeschlossen werden.

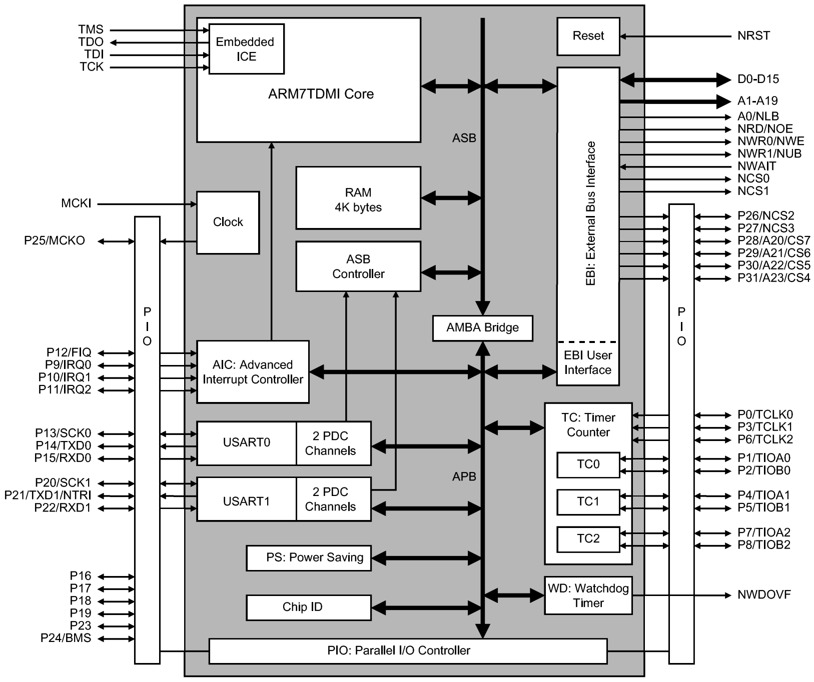

Abbildung 3.8: Detailansicht Microcontroller AT91M40400 [ATM98a]

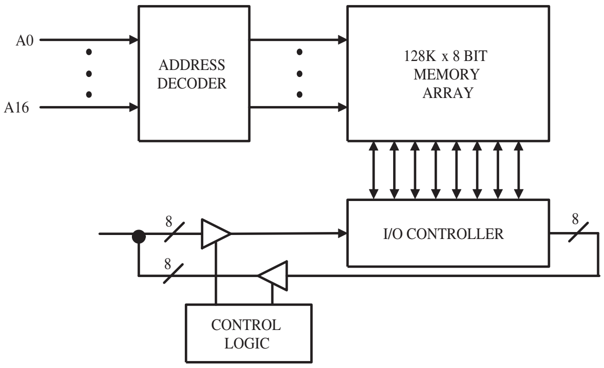

Abbildung 3.9: 128K-Byte SRAM-Baustein in CMOS-Technologie [IDT99]

3.3.3 Memory-Map

Die vom Hersteller initial eingestellten Adressbereiche für die ATMEL-Peripherie sind in Tabelle 3.2 aufgelistet. Aus diesen Informationen resultiert, dass Programm-Code und Daten in folgenden Adressbereichen abgelegt werden können:

1. **Off-Chip**

   Für den externen Arbeitspeicher entspricht der Adressbereich:

   0x02000000 - 0x02080000

   einer Größe von 512K-Byte. Innerhalb dieses Bereiches haben Messreihen im Vorfeld ergeben, dass, wenn Instruktionen oder Daten im oberen Adressbereich:

   0x02040000 - 0x2080000

   liegen, vom Amperemeter erfasst werden. Der angegebene Adressbereich, auf dem das Messgerät reagiert, umfasst jedoch 256K-Byte statt erwarteten 128K-Byte für einen Speicherbaustein. Der Grund hierfür liegt in der Speicherorganisation der Ram-Bausteine. Für das Lesen oder Schreiben von Programm-Code und Daten sind immer zwei Ram-Bausteine beteiligt. Jeweils ein Ram-Baustein hält die Daten für gerade oder ungerade Adressen. Speicherzugriffe werden vom ARM7TDMI aus Gründen der Effizienz immer im Aligned-Modus ausgeführt. Dies bedeutet, dass bis auf Store-Byte-Operationen immer zwei Ram-Bausteine bei einem Speicherzugriff beteiligt sind.

2. **On-Chip**

   Der im Prozessor liegende 4K-Byte große On-Chip-Speicher wird nach einem Reboot des Evaluationboards auf den Adressbereich:

   0x00000000 - 0x00000FFF

   gemappt. Dieser Adressbereich ist anschließend ebenfalls für Programm-Code und Daten frei verfügbar.

3. **Flash-Rom**

   Für den Flash-Rom-Speicher entspricht der Adressbereich:

   0x01000000 - 0x0101FFFF

   einer Speichergröße von 128K-Byte. Jedoch ist nur die Verwendung der oberen 64K-Byte für eigene Programme gestattet, da im unteren Bereich notwendige Applikationen des Betriebssystems abgelegt sind.

Speicherwahl

Im Rahmen dieser Diplomarbeit ist für die Speicherwahl von Programm-Code und Daten der externe Arbeitspeicher sowie der prozessorinterne On-Chip-Speicher verwendet worden.

Name	**Address**	**Size**
Peripheral Devices	HFFFFFFFF HFFD00000	3M-Byte
undefined		
External Memory [7]	H7FFFFFFF H70000000	1,4, 16 or 64M-Byte - not used
undefined		
External Memory [6]	H6FFFFFFF H60000000	1,4, 16 or 64M-Byte - not used
undefined		
External Memory [5]	H5FFFFFFF H50000000	1,4, 16 or 64M-Byte - not used
undefined		
External Memory [4]	H4FFFFFFF H40000000	1,4, 16 or 64M-Byte - not used
undefined		
External Memory [3]	H3FFFFFFF H30000000	1,4, 16 or 64M-Byte - not used
undefined		
External Memory [2]	H2FFFFFFF H20000000	1,4, 16 or 64M-Byte - not used
undefined		
External Memory [1]	H0FFFFFFF H02000000	256K-Byte to 2048K-byte 16-bit SRAM (4M-Byte page size)
undefined		
External Memory [0]	H01FFFFFF H01000000	128K-Byte 16-Bit Flashrom (1M-Byte page size)
undefined		
ON-CHIP RAM (reboot)	H00300FFF H00300000	4K-Byte
Reserved ON-CHIP device	H002FFFFF H00100000	2M-Byte
undefined		
ON-CHIP RAM (normal)	H00000FFF H00000000	4K-Byte

Tabelle 3.2: Memory-Map der ATMEL-Peripherie [ATM98b]

3.3.4 Angel-Debugger

Die Kommunikation zwischen Host-PC und Board wird durch ein System ermöglicht, dass Angel genannt wird. Angel ist ein Programm, das die Ausführung und das Debugging von Programmen auf dem Evaluationboard ermöglicht [ATM98b]. Das Angel-System besteht aus zwei Komponenten, dem Angel-Kernel auf dem Board selbst und einem externen Debugger auf dem Host-PC mit einer standardisierten Protokollschnittstelle für Angel-Requests-Aufrufe. Das Angel-System ist in Abbildung 3.10 dargestellt.

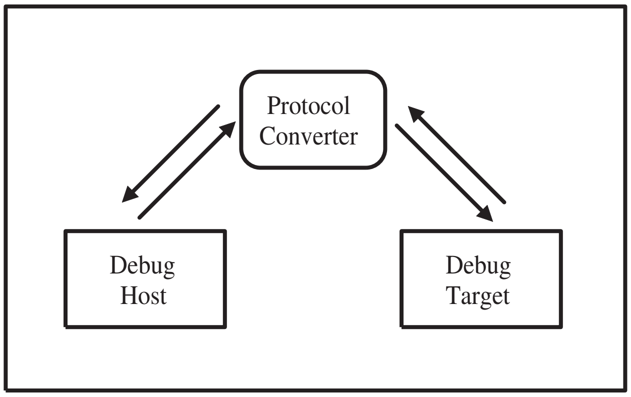

Abbildung 3.10: Angel Debug System

Angel-Kernel

Der Angel-Kernel ist im Laufzeitsystem des Evaluationboard integriert. Nach einem Board-Reset wird Angel aus dem Flash-Rom-Speicher in den schnelleren externen Hauptspeicher kopiert und wird über eine Interruptbehandlung permanent vom Laufzeitsystem angesprochen. Dabei hat Angel folgende Aufgaben durchzuführen:

Überwachung der Kommunikationsleitungen

Dazu gehören die beiden seriellen Schnittstellen, nicht jedoch das JTAG-Interface innerhalb des Prozessors.

Annahme von Host-Requests

Folgende Debugsequenzen werden von Angel interpretiert:

- Ausführbare Programme vom Host laden
- Starten und Stoppen der Programme
- Setzen von Breakpoints
- Tracen von Programmen
- Auslesen und Modifizieren von Prozessorregistern
- Auslesen und Modifizieren von Speicheradressen

Control-Monitoring

STDIO-Ausgaben von Programmen werden zur Konsole des Debug-Hosts zurückgeschickt. STDIO-Eingaben können über die Tastatur zum Board gesendet werden. Somit ist eine Interaktion während eines Programmlaufs möglich.

3.4 Multimeter Escort 95

Die Messungen erfolgten mit einem Digitalmultimeter mit einer vom Hersteller angegebenen Genauigkeit von unter 1 Prozent. Über die serielle Schnittstelle erlaubt dieses Messgerät alle Messdaten zusätzlich zur Displayanzeige auf dem Debug-Host mitzuprotokollieren. Eine Zusammenstellung der wichtigsten technischen Daten aus [COS00] ist in Tabelle 3.3 dargestellt.

Grundgenauigkeit	0,06 %
DC V	$10\,\mu$V - 1000 V
AC V	$10\,\mu$V - 750 V
DC A	100 nA - 10 A
AC A	100 nA - 10 A
Widerstand	$0{,}1\,\Omega$ - $40\,\mathrm{M}\Omega$
Kapazität	1 pF - 2 mF
Messinterval	20 Mess./Sec
Schnittstelle	RS-232C

Tabelle 3.3: Technische Daten zum eingesetzten Multimeter ESCORT 95

Kapitel 4

Energiemodell

Tiwari hat in seinen Arbeiten [TMW94a] [TIW96] ein allgemeines Energiemodell zur Messung von Prozessor-Instruktionen vorgestellt. Dieses Modell wird für die anschließenden Messungen als Grundlage dienen. Das Energiemodell wurde im Rahmen dieser Diplomarbeit erweitert, da Tiwaris Modell den Einfluss einer Instruktion auf den externen Speicher nicht berücksichtigt.

4.1 Prozessorkosten

Die Abbildung 4.1 zeigt drei Energiekostenarten, die Instruktionen bei Bearbeitung in einem Prozessor produzieren.

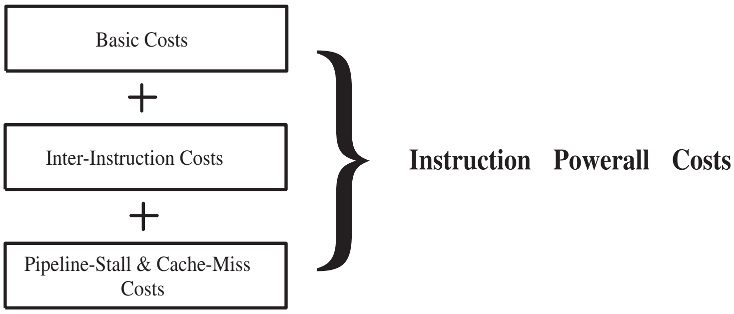

Abbildung 4.1: Energiemodell nach Tiwari

4.1.1 Basiskosten

Die Basiskosten definieren die Energiegrundkosten einer Instruktion. Jede Instruktion bewirkt beim Durchlaufen innerhalb eines Programmes Zustandsveränderungen am Prozessor. Die Auswirkungen einer Instruktion auf den Prozessor sind eine erhöhte Schaltkreisaktivität, um den Befehl zu verarbeiten.

$$E_{Base} = \sum_i (B_i \cdot N_i)$$

Die Basiskosten einer Instruktion B_i multipliziert mit der Anzahl ihrer Aufrufe N_i in einem Programm bilden die erste Komponente im Energiemodell.

4.1.2 Inter-Instruction-Seiteneffekte

Zur genaueren Berechnung des Energieverbrauchs eines Programms reicht die Basiskostenbestimmung einer Instruktion nicht aus. Ein Assemblerprogramm besteht in der Regel aus unterschiedlichen aufeinanderfolgenden Instruktionen und sich wiederholenden Instruktionssequenzen. Beim Wechsel von einer Instruktion zur nächsten werden im Prozessor unterschiedliche Schaltkreiszustände aktiviert, da verschiedene Befehle unterschiedliche Prozessorressourcen benötigen. Abbildung 4.2 zeigt an einem Beispiel die beteiligten Prozessorressourcen beim Durchlauf von drei unterschiedlichen Instruktionen innerhalb des ARM7TDMI-Datenpfades.

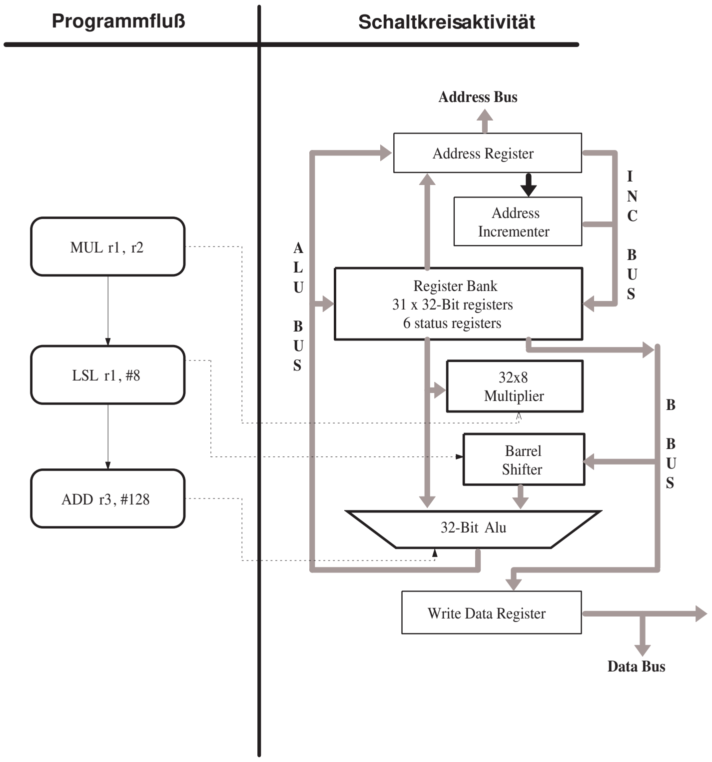

Abbildung 4.2: Instruktionsbedingte Ressourcenwechsel

Die zusätzlichen Kosten des Prozessors, um von einem Schaltkreiszustand in den nächsten zu gelangen, werden als Inter-Instruction-Seiteneffekte bezeichnet. Diese Seiteneffekte können zwischen zwei Befehlen unterschiedlich stark ausgeprägt sein. Dies erklärt sich dadurch, dass Instruktionen mit ähnlicher Funktionalität, z.B. die Ausführung zweier aufeinanderfolgender Addieroperationen ADD und ADC, hauptsächlich Schaltkreise der ALU beanspruchen. Jedoch erzeugt eine Addieroperation gefolgt von einer Multiplikation am Beispiel der ARM7TDMI-Architektur eine höhere Schaltwechselfolge, da in diesem Fall erst die ALU und anschließend das separate Multiplizierwerk angesprochen werden muss.
Ein weiterer Inter-Instruction-Effekt zeigt sich bei Betrachtung des Adress- und Datenbusses bei einem Instruktionswechsel. Beim Übergang von einer Instruk-

tion zur nächsten erfolgt eine Neubelegung des Bussystems mit Nullen und Einsen. Unterscheidet sich dabei der Opcode der nachfolgenden Instruktion deutlich vom Vorgänger, müssen auf dem Bussystem mehr Bits umgelegt werden als bei ähnlichen Instruktionen. Die Anzahl der 'gekippten' Bits wird als Hamming-Distanz bezeichnet. Je größer die Hamming-Distanz zwischen zwei Instruktionen, desto höher ist der Inter-Instruction-Effekt [NKH00]. Die Kosten für Inter-Instruction-Seiteneffekte werden definiert als:

$$E_{InterInstruction} = \sum_{i,j} (O_{i,j} \cdot N_{i,j})$$

wobei $O_{i,j}$ die zusätzlichen Energiekosten eines Instruktionspaares ij entspricht, multipliziert mit der Anzahl ihres Auftreten $N_{i,j}$ in einem Programm.

4.1.3 Pipeline-Stalls und Cache-Misses

Beim Durchlaufen eines Programmes sind dynamische Einflüsse der Pipeline und des Cache-Speichers ebenfalls zu berücksichtigen. Ein Cache-Miss generiert einen teuren externen Speicherzugriff, da das fehlende Datum im lokalen Cache-Speicher nicht mehr verfügbar ist. Bei einem Pipeline-Stall wird die Fließbandverarbeitung solange angehalten, bis der Stall bereinigt ist. Für Pipeline-Stall-Effekte sind drei Gründe verantwortlich:

1. **Resource Hazard**

   Bei Zugriff von Instruktionen auf gleiche Systemressourcen wird ein Halten der Pipeline erzwungen, da nur sequentiell auf bestimmte Ressourcen zugegriffen werden kann.

2. **Data Hazard**

   Bei Befehlen, die nur in Abhängigkeit von Ergebnissen vom Vorgänger ausgeführt werden können, wird ebenfalls ein Stall generiert.

3. **Control Hazard**

   Control-Hazards treten in der Regel bei konditionalen und unkonditionalen Sprüngen auf. Der Grund hierfür liegt in der Prefetch-Strategie der Pipeline, die in aller Regel nur sequentiell arbeitet und Verzweigungen nicht berücksichtigt.

Stall-Effekte haben zur Folge, dass interne Wartebefehle in die Pipeline eingefügt werden müssen, und Befehle in der Pipelineverarbeitung auf der Ebene verbleiben, bis der Stall aufgehoben ist.

$$E_{PipelineCache} = \sum_{k} (E_k)$$

Die Energiekosten E_k werden bei jedem Auftreten von Pipeline-Stalls und Cache-Misses berücksichtigt.

4.1.4 Energiemodell Prozessor

Die Aufaddierung der einzelnen Kostenarten ergibt ein vollständiges Modell zur Bestimmung des Stromverbrauchs einer Prozessor-Instruktion beim Durchlaufen innerhalb eines Programmes.

$$E_{cpu} = \sum_i (B_i \cdot N_i) + \sum_{i,j} (O_{i,j} \cdot N_{i,j}) + \sum_k (E_k)$$

4.2 Speicherkosten

Eine Instruktion hat im Programmablauf nicht nur Auswirkung auf den Prozessorstrom, sondern auch auf den Peripheriespeicherstrom. Bei lesenden oder schreibenden Zugriffen auf den Speicher erfolgt eine Erhöhung des Energieverbrauchs der angesprochenen Speicherbausteine. Das definierte Energiemodell für den Prozessorstrom kann in abgewandelter Weise auch für Messungen des Speicherstroms benutzt werden. Beim Speicherstrom treten zwei Kostenarten auf:

- **Zugriffskosten beim Holen einer Instruktion**

  Die Zugriffskosten sind definiert als Energiekosten beim lesenden Zugriff des Prozessors auf eine im externen Hauptspeicher liegende Instruktion.

- **Schreib- und Lesekosten für ein Datum**

  Instruktionen, die von sich aus auf den Speicher zugreifen, generieren zusätzliche Kosten, da ein Datenwort aus dem Speicher in den Prozessor eingelesen (Load), oder im umgekehrten Fall, in den Speicher zurückgeschrieben (Store) werden muss.

4.2.1 Energiemodell Speicher

Diese beiden Kostenarten für Speicherenergiekosten werden in E_{mem} festgehalten.

$$E_{mem} = \sum_i ((D_i + Z_i) \cdot N_i)$$

D_i gibt die Zugriffskosten für das Holen einer Instruktion (Instruction-Fetch), multipliziert mit der Anzahl ihrer Aufrufe N_i innerhalb eines Programmes, an. Initiiert die Instruktion von sich aus einen zusätzlichen lesenden oder schreibenden Zugriff Z_i auf den Speicher, werden diese Kosten mitberücksichtigt.

4.3 Gesamtenergieverbrauch einer Instruktion

Die Wirkung einer Instruktion auf den Gesamtstromverbrauch von Prozessor und Speicher wird festgehalten als:

$$E_{total} = E_{cpu} + E_{mem}$$

4.4 Energiebedarf eines Programmes

Das aufgestellte Energiemodell kann in einer ersten Anwendung dazu benutzt werden, um den Gesamtenergiebedarf eines Programmes zu ermitteln. Folgende Phasen sind dabei zu durchlaufen:

1. **Generierung Maschinencode**

   Das zu untersuchende Programm wird, falls in einer Hochsprache implementiert, mit Hilfe eines Compilers und Linkers in Maschinencode übersetzt.

2. **Basisblockanalyse**

   Als nächster Schritt erfolgt eine Unterteilung von verschiedenen Befehlssequenzen in Basisblöcke. Basisblöcke sind zusammenhängende Instruktionssequenzen, die immer gemeinsam durchlaufen werden.

3. **Ermittlung des Energiebedarfs pro Basisblock**

   Für jeden Basisblock wird der Energieverbrauch ermittelt. Die Energiekosten für jede Instruktion liegen in einer Datensenke. Zusätzlich ermittelte Energiekosten, wie Inter-Instruction- und Stall-Effekte werden ebenso berücksichtigt, wie anfallenende Speicherkosten für den gesamten Funktionsblock.

4. **Profiler-Analyse**

   Mit Hilfe eines Profilers wird die Anzahl der Durchläufe pro Funktionsblock ermittelt.

5. **Cache-Analyse**

   Mit einem Cache-Simulator wird die Anzahl der Seitenzugriffsfehler notiert. Jeder Fehler erhöht die aktuellen Energiekosten des Basisblocks und wird deshalb als zusätzlicher Kostenfaktor mitberücksichtigt.

6. **Berechnung des Gesamtverbrauches**

   Der Energiebedarf für jeden Basisblock wird mit der Anzahl seiner Aufrufe im Gesamtprogramm multipliziert. Die Summe aller Basisblöcke ergibt den Gesamtenergiebedarf eines Programmes.

Die Abbildung 4.3 zeigt die einzelnen Phasen zur Bestimmung des Gesamtenergieverbrauchs eines Programmes in einem Ablaufdiagramm.

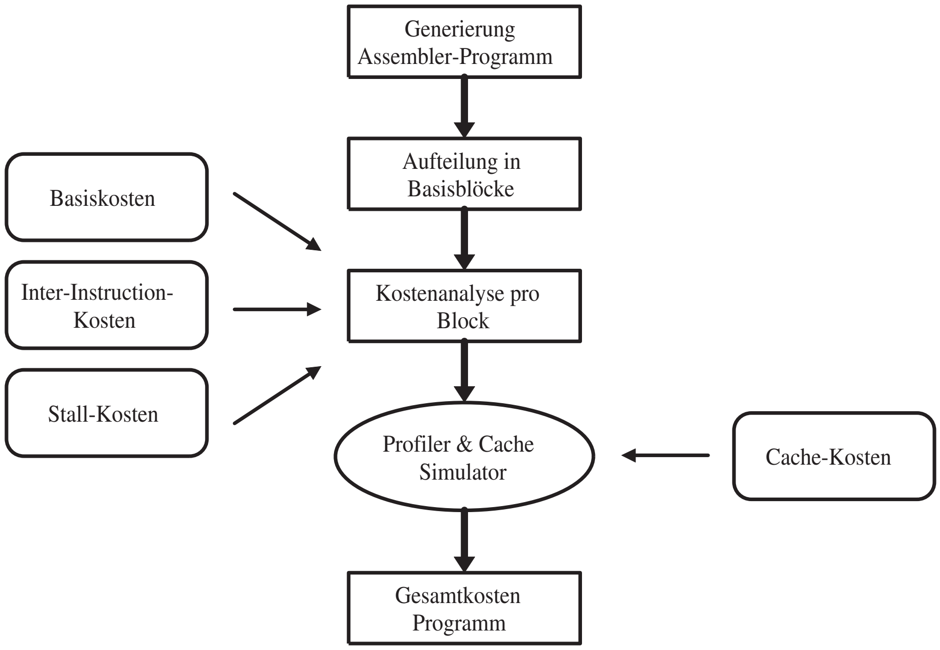

Abbildung 4.3: Energiekostenberechung eines Programmes

Eine Vereinfachung des Ablaufs kann dadurch erreicht werden, dass auf die Basisblockanalyse verzichtet wird. Die anfallenden Energiekosten für jede Instruktion werden alternativ einzeln bewertet. Für diesen Fall ist am Fachbereich Informatik, Lehrstuhl 12 im Rahmen eines Low-Power-Projektes ein Sourcetrace-Analyzer implementiert worden, der die Gesamtenergiekosten eines Programmes errechnet. Die genaue Arbeitsweise des Trace-Analyzers wird am Ende dieses Kapitels erläutert.

4.5 Anwendung des Energiemodells für den ARM7TDMI

Das aufgestellte Energiemodell wird im nächsten Schritt für den ARM7TDMI-Prozessor unter Berücksichtigung der Speicherhierarchie des ATMEL-Boards überführt. Die Überführung sieht vor, geeignete Testmuster für die Messung der Prozessor-Instruktionen zu erzeugen. Alle Kostenarten des Energiemodells sind dabei zu berücksichtigen.

4.5.1 Basiskosten

Die Energiebasiskosten aller Thumb-Instruktionen werden durch n-maliges Wiederholen eines Befehls in einer Schleife gemessen. Die Wiederholung einer Prozessor-Instruktion ist insofern wichtig, als das angeschlossene Messgerät für eine Messung ein Zeitfenster von mindestens 50 ms benötigt, um einen stabilen Wert anzuzeigen. Die Anzahl der Instruktion hintereinander ist geeignet zu wählen, damit der abschließende Rücksprung (Branch Loop) zum Schleifenanfang keine signifikanten Auswirkungen auf den Stromverbrauch hat. Die Tabelle 4.1 zeigt das Verfahren am Beispiel einer Move-Instruktion.

Programmzähler		**Messmodul**
1	Loop	MOV r0, #10
2		MOV r0, #10
.		.
.		.
.		.
100		B Loop

Tabelle 4.1: Basiskostenberechnung

Durch die Wiederholung einer Instruktion innerhalb einer Schleife wird auf dem Messgerät die Stromstärke in mA angezeigt. Für eine Energiebestimmung ist der angezeigte Wert zusätzlich mit der Anzahl der benötigten Taktzyklen, der Taktperiode und der Versorgungsspannung zu multiplizieren.
Die Basiskosten einer Instruktion hängen von weiteren Faktoren ab, die näher untersucht werden müssen. Zu nennen sind Einflüsse von Operanden- und Registerwahl sowie unterschiedliche Adressbereiche von Programm-Code und Daten.

Operandenwahl

Die Operandeninhalte einer Instruktion sind variabel und können sich beim Ablauf eines Programmes ändern. Der Energieverbrauch einer Instruktion kann bei unterschiedlichen Operandeninhalten variieren. Um dies zu berücksichtigen, müssen Messungen der Basiskosten mit unterschiedlichen Operandeninhalten durchgeführt werden. Die Auswahl der gewählten Daten erfolgt dabei im erlaubten Gültigkeitsbereich der zu messenden Instruktion. Aus den gesammelten Einzelmessungen wird im Anschluss ein Durchschnittswert für den Energieverbrauch dieser Instruktion gebildet. Tabelle 4.2 zeigt für diese Untersuchung verwendete Initialisierungsdaten. Die Operandenwerte sind so ausgewählt, dass Immediate- und Registerwerte jeweils mit minimalen und maximalen Werten vorbelegt werden, um den höchstmöglichen Einfluss von Datenabhängigkeiten zu berücksichtigen. Zusätzlich ist zu beachten, dass spezielle Thumb-Instruktionen, die Speicher-Offset-Berechnungen durchführen, nur die Verwendung von Aligned-Werten erlauben.

Operandenart	Min	Med	Max
Reg 32 Bit	0x0	0xAAAAAAAA	0xFFFFFFFF
Imm 3 Bit	0x0	0x1	0x3
Imm 5 Bit byte aligned	0x0	0x15	0x1F
Imm 6 Bit halfword aligned	0x0	0x2A	0x3E
Imm 7 Bit word aligned	0x0	0x38	0x7c
Imm -7 Bit word aligned	0x0	-0x38	-0x7c
Imm 8 Bit	0x0	0xAA	0xFF
Imm 10 Bit word aligned	0x0	0x2a8	0x3FC

Tabelle 4.2: Dateninhalte für Operanden

Registerwahl

Alle Instruktionen mit Ausnahme von Branch-Befehlen verwenden für Quelle und Ziel mindestens ein Register. Es soll untersucht werden, ob die Wahl bestimmter Register zusätzlichen Einfluss auf den Energieverbrauch hat.

Sprungweiten

Für Branch-Instruktionen wird zusätzlich betrachtet, ob die Energiekosten für diese Befehle mit der Sprungweite oder der verwendeten Start- und Zieladdressen ansteigen. Tabelle 4.3 zeigt zu untersuchende Sprungweiten. Sprungweite 0 bedeutet, dass der Branch sich selber aufruft, um die Basiskosten dieser Instruktion zu ermitteln.

Instruktion	Min (Bytes)	Med (Bytes)	Max (Bytes)
BRANCH CONDITIONAL	0	124	256
BRANCH UNCONDITIONAL	0	512	1024
BRANCH LONG WITH LINK	0	1024	2048

Tabelle 4.3: Sprungweiten bei Branch-Befehlen

Speicherkombinationen

Innerhalb der Speicherhierarchie des ATMEL-Boards wird der externe Arbeitsspeicher (Off-Chip-Speicher) und zusätzlich der interne Prozessorspeicher (On-Chip-Speicher) betrachtet. Für die Untersuchung von Energiekosten bei Verwendung unterschiedlicher Speicherkombinationen werden in einer ersten Messreihe alle Instruktionen auf den internen On-Chip-Speicher des Prozessors und anschließend in einer zweiten Messreihe auf dem externen Off-Chip-Speicher positioniert. Bei Speicherzugriffen von Instruktionen werden weiterhin alle möglichen Speicherkombinationen von Instruktionen und Daten betrachtet, wie Tabelle 4.4 zeigt.

Instruktion	Daten
OFF-CHIP	OFF-CHIP
OFF-CHIP	ON-CHIP
ON-CHIP	OFF-CHP
ON-CHIP	ON-CHIP

Tabelle 4.4: Speicherkombinationen von Programm-Code und Daten

Unterschiedliche Adressbereiche

Innerhalb der beiden Speicherbereiche wird weiterhin untersucht, ob die Wahl unterschiedlicher Adressbereiche für Instruktionen, Daten und Stack Auswirkungen auf den Energieverbrauch hat.

4.5.2 Inter-Instruction-Seiteneffekte

Für die Betrachtung von Inter-Instruction-Effekten werden in einem ersten Schritt verschiedene Befehlsklassen aufgestellt und anschließend alle Thumb-Instruktionen darunter eingruppiert. Die Befehlsklassen werden so ausgewählt, dass ähnliche Instruktionen möglichst gleiche Funktionseinheiten des ARM7TDMI-Prozessors beanspruchen. Die Idee zu dieser Einteilung stammt aus [SIT99]. Folgende Befehlsklassen werden für den Thumb-Befehlssatz aufgestellt:

1. **Registertransfer-Operationen**

   Dazu gehören alle Move-Operationen. Diese Instruktionen verändern ausschließlich Daten der Registerbank.

2. **ALU-Operationen**

   Alle Operationen, die innerhalb der ALU durchgeführt werden. Zu nennen sind alle arithmetischen und logischen Instruktionen.

3. **Barrelshifter-Operationen**

   Die beiden ALU-Instruktionen Shift und Rotate werden beim ARM7TDMI in der Funktionseinheit Barrelshifter verarbeitet.

4. **Multiplier-Operationen**

   Die Multiplikations-Instruktion wird separat im Multiplier verarbeitet.

5. **Speicherzugriff-Operationen**

   Dazu gehören alle Instruktionen, die Speicherzugriffe durchführen. Zu nennen sind alle Load-/Store- sowie Push-/Pop-Instruktionen.

6. **Branch-Operationen**

   Branch-Anweisungen, konditional und unkonditional, benötigen für die Sprungvorhersageberechnung Funktionseinheiten der ALU und das Bussystem zum Laden des Sprungziels.

Anschließend erfolgt die Bildung von Instruktionstupeln nach folgenden zwei Regeln:

- **Inter-Instruction-Effekte innerhalb einer Befehlsklasse**

  Befehle innerhalb einer Klasse bilden mit Befehlen derselben Klasse ein Instruktionstupel.

- **Inter-Instruction-Effekte zwischen verschiedenen Befehlsklassen**

  Ein Befehl innerhalb einer Klasse bildet in Kombination mit einem Befehl einer anderen Klasse ein Instruktionstupel.

Es soll untersucht werden, ob Inter-Instruction-Effekte bei Instruktionstupeln innerhalb einer Klasse geringer ausfallen als bei Tupeln gebildet aus zwei unterschiedlichen Klassen. Da eine Messung aller Kombinationen von Instruktionstupeln ausgehend von der Anzahl aller Thumb-Instruktionen 36^2 Messungen erfordern würde, wird die Anzahl dadurch reduziert, dass aus jeder Befehlsklasse genau zwei Instruktionen ausgewählt werden, die ein Tupel bilden. Weiterhin bilden je zwei Instruktionen aus unterschiedlichen Klassen ein Tupel. Die Gesamtanzahl an Instruktionstupeln wird durch dieses Vorgehen deutlich reduziert. Werden während der Messung größere Inter-Instruction-Effekte innerhalb einer Klasse festgestellt, ist die Klasseneinteilung neu zu überdenken und gegebenfalls zu verfeinern.
Einflüsse von unterschiedlichen Hamming-Distanzen zwischen zwei Instruktionen werden dadurch untersucht, dass verschiedene Instruktionstupel, sortiert nach ihrer Distanzgröße, gebildet werden.

Messverfahren zur Berechnung von Inter-Instruction-Effekten

Die Instruktionstupel werden mit demselben Verfahren wie bei der Basiskostenberechnung einer einzelnen Instruktion durch n-maliges Wiederholen eines Tupels in einer Schleife gemessen. Bevor jedoch Untersuchungen von Inter-Instruction-Effekten durchgeführt werden können, müssen im Vorfeld beide Instruktionen des Messtupels einzeln gemessen werden. Aus den Einzelmessungen wird anschließend der durchschnittlich erwartete Stromwert für das Instruktionstupel bestimmt. Der erwartete Stromwert zweier aufeinanderfolgender Befehle wird errechnet durch:

$$I_{avg} = \frac{(Instr_1 \cdot n) + (Instr_2 \cdot m)}{n + m}$$

Dabei entspricht $Instr_1$ und $Instr_2$ dem gemessenen Strom der beiden Instruktionen, mit n,m als zugehörige Taktzyklen. Die Subtraktion zwischen gemessenem Wert des Instruktionstupels und dem erwarteten Wert I_{avg} ergibt den Inter-Instruction-Effekt.

$$I_{inter-inst} = Instr_{1,2} - I_{avg}$$

$Instr_{1,2}$ entspricht dabei dem gemessenen Strom für das Instruktionstupel.

Dazu folgendes Beispiel:

```
   Instruktion        Zyklenzahl      Strom (mA)

1  MOV r1, r2             1              41,3
2  LDR r3, [r1,#0]        3              48,7


   tatsaechlich gemessener Strom (mA): 49,3
```

Der erwartete Wert beträgt demnach 46,8 mA berechnet aus:

$$((41,3 \cdot 1) + (48,7 \cdot 3)) \div (1 + 3) \tag{4.1}$$

Wird der erwartete Wert vom tatsächlich gemessenen Wert subtrahiert, beträgt der Inter-Instruction-Effekt in diesem Fall 2,5 mA. Für die Leistungsbetrachtung wird dieser Wert abschließend mit V_{dd}=3,3 V multipliziert und ergibt somit 8,25 mW.

4.5.3 Pipeline-Stalls und Cache-Misses

Cache-Miss-Effekte sind dynamisch vom Programmablauf abhängige Kosten und werden im Energiemodell mit den Kosten eines zusätzlichen Speicherzugriffs bestraft. Diese Effekte treten jedoch beim ARM7TDMI nicht auf, da dieser Prozessor keinen Cache-Speicher besitzt. Dies ist nicht untypisch bei Prozessoren dieser Bauart. Spezielle Anwendungen erfordern Systeme ohne Cache, weil nur so das exakte Timing für zeitkrititsche Operationen genau berechnet werden kann. Ein wichtiger Anwendungsfall ist z.B. eine Airbagsteuerung. Da der ARM7TDMI Daten nicht 'cached', wird jeder Speicherzugriff mit zusätzlichen Energiekosten bestraft.

Pipeline-Stall-Effekte haben keine Auswirkung auf die Leistung einer Instruktion, jedoch müssen bei einem Hazard die Mehrkosten von zusätzlichen Wartezyklen für die Energiekostenbestimmung mitberücksichtigt werden. Resource- und Data-Hazards sind dabei zu vernachlässigen, da diese durch typische Compileroptimierungen wie Instruction-Scheduling [MAR00] vermieden werden. Control-Hazards treten beim ARM7TDMI für Branch-Instruktionen auf. Die Dauer des Branch-Delays für das erneute auffüllen (Refilling) der Pipeline beträgt drei Taktzyklen.

4.5.4 Speicherkosten

Speicherkosten werden konform nach der Methode der Berechnung der Basiskosten innerhalb einer Messchleife durchgeführt. Bei der Messung des Speichers wird folgendes Verfahren angewandt:

1. **Zugriffskosten beim Instruction-Fetch**

   Um die Kosten beim Holen einer Instruktion zu messen, werden alle Befehle in einen Speicherbereich des externen Speichers gelegt, der vom Messgerät erfasst werden kann. Bei Instruktionen, die auf den Speicher zugreifen, werden die angesprochenen Speicheradressen ausserhalb des Messbereiches vom Amperemeter gelegt. Abbildung 4.4 zeigt dies am Beispiel einer Load-Instruktion.

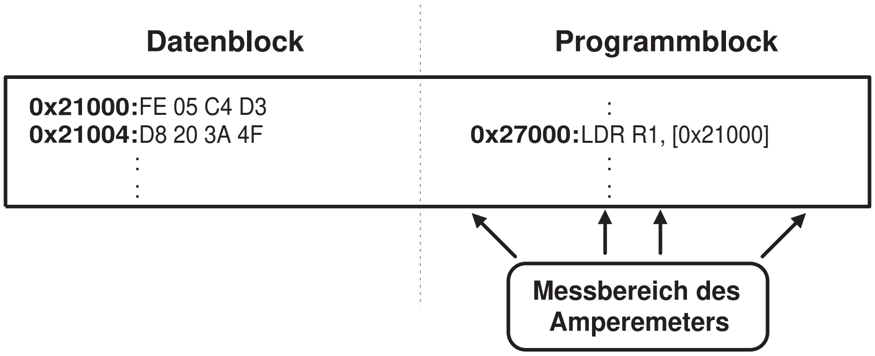

Abbildung 4.4: Messung des Instruction-Fetch

2. **Schreib- und Lesekosten für verschiedene Datenzugriffe**

   Initiiert eine Instruktion einen Speicherzugriff, werden diese Kosten gesondert behandelt. Das Verfahren sieht vor, eine Instruktion in einen vom Amperemeter nicht erfassten Speicherbereich auszulagern und die angesprochene Speicherzelle innerhalb des Messbereiches zu legen, wie Abbildung 4.5 zeigt. Dadurch werden gezielte Untersuchungen von Word-, Halfword- und Byte-Zugriffen ermöglicht.

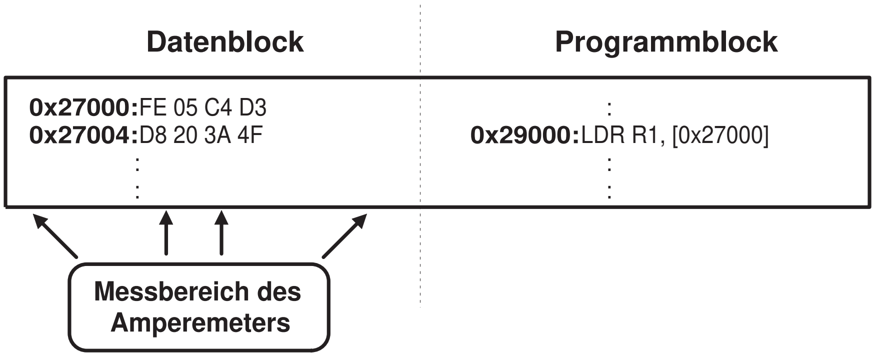

Abbildung 4.5: Messung verschiedener Datenzugriffe

4.5.5 Waitstate-Zyklen des Speichers

Der abgelesene Messwert am Amperemeter beim Durchlaufen einer Instruktion in einer Schleife zeigt den benötigten Strom des Prozessors oder Speichers an. Zur Berechnung der Energiekosten einer Instruktion ist jedoch die vollständige Anzahl der benötigten Taktzyklen und die Versorgungsspannung zu berücksichtigen. In den ARM7TDMI-Datenblättern [ARM95b] sind Taktzyklenangaben zu allen Instruktionen angegeben. Die angegebenen Zyklenzahlen gehen jedoch von idealisierten Bedingungen aus. Zusätzliche Waitstate-Zyklen des externen Speichers werden nicht berücksichtigt, da diese herstellerspezifisch unterschiedlich sein können. Um die genaue Zyklenzahl einer Instruktion zu bestimmen, wird mit Hilfe eines Timers die reale Zeit gemessen, die eine Instruktion bis zur vollständigen Verarbeitung benötigt. Das Verfahren sieht vor, identisch zur

Basiskostenberechnung eine Instruktion n-mal in einer Schleife durchlaufen zu lassen. Die Schleifenwiederholrate wird über einen Zähler eingestellt. Der Zähler ist auf einen Wert einzustellen, der die Schleife in einem Bereich von mindestens einer Minute durchlaufen lässt. Dadurch werden die Taktzyklen des rückspringenden Branch-Befehls zum Schleifenanfang des Instruktionsblocks vernachlässigt. Aus der Anzahl der Schleifendurchläufe und der gemessenen Zeit für die vollständige Bearbeitung der Schleife kann anschließend für jede Instruktion die genaue Zyklenzahl inklusive Waitstate-Zyklen bestimmt werden. Das Verfahren wird detailliert im anschließenden Kapitel 5 am Beispiel einer Load-Word-Instruktion vorgestellt.

4.5.6 Berechnung der Gesamtenergiekosten

Für die Berechnung der Gesamtenergiekosten einer Thumb-Instruktion für den ARM7TDMI-Prozessor mit dazugehörigem Arbeitsspeicher wird im folgenden eine Zusammenfassung der erlangten Kenntnisse zusammengetragen, um anschließend ein Gesamtenergiemodell für dieses System bilden zu können.

Der Energieverbrauch einer Instruktion ist wie in Kapitel 2.1.2 definiert durch:

$$E = V_{dd} \cdot I \cdot N \cdot t$$

Für das ATMEL-Board gilt, dass die Versorgungsspannung V_{dd} 3,3 Volt entspricht. Der Gesamtstromverbrauch I einer Instruktion für Prozessor und Speicher wird weiter unterteilt in:

$$I = (I_{cpu} + I_{cpu_{inter}} + I_{mem})$$

I_{cpu} und I_{mem} entsprechen den Stromwerten, die mit Hilfe der Basiskostenbestimmung am Messgerät abgelesen werden. Für die Energiekostenbetrachtung eines Programmes wird für jede Instruktion ein zusätzlicher Inter-Instruction-Stromwert $I_{cpu_{inter}}$ berücksichtigt. Dieser Stromwert ist eine Konstante, errechnet als Durchschnittswert aus den durchzuführenden Inter-Instruction-Messreihen und wird für jede Instruktion I hinzuaddiert. Die Verwendung eines konstanten Stromwertes für Inter-Instruction-Effekte ist notwendig, da wegen des hohen zeitlichen Aufwandes nicht für alle Kombinationen von Instruktionen Inter-Instruction-Werte gebildet werden können.
Die Taktfrequenz auf dem ATMEL-Board ist auf 32,768 MHz eingestellt. Die Taktperiode t errechnet sich als Kehrwert zur Taktfrequenz:

$$t = \frac{1}{32,768 \cdot 10^6}\ s$$

Die Anzahl der benötigten Taktzyklen N einer Instruktion werden aufgespalten in:

$$N = n + m$$

Dabei entspricht n den instruktionsabhängigen CPU-Zyklen und m den zusätzlichen Waitstate-Zyklen bei Zugriff auf den Arbeitsspeicher. Der resultierende Gesamtenergieverbrauch einer Instruktion für Prozessor und Arbeitspeicher auf dem ATMEL-Board wird durch Einsetzen aller Komponenten in die Energiegleichung bestimmt:

$$E_{instr} = \frac{3,3 \cdot (I_{cpu} + I_{cpu_{inter}} + I_{mem}) \cdot (n + m)}{32,768 \cdot 10^6} \; V s$$

Für die Energiebetrachtung einer einzelnen Instruktion ist der Inter-Instruction-Stromwert $I_{cpu_{inter}}$ zu vernachlässigen und deshalb auf 0 zu setzen. Für die Ermittlung des Energieverbrauchs eines Programms werden in einer abschließenden Gleichung die Energiekosten E_{instr} aufsummiert:

$$E_{prg} = \sum_i (E_{instr} \cdot Z_i)$$

Z_i gibt die Anzahl der Aufrufe einer Instruktion in einem Programm an.

4.6 Integration des Energiemodells in ARM12CC-Umgebung

Die Energiedaten für den ARM7TDMI-Prozessor und den externen Speicher fliessen in der in Abbildung 4.6 gezeigten Entwicklungsumgebung ein. Diese ist im Rahmen der Low-Power-Forschungsaktivitäten am Fachbereich Informatik, Lehrstuhl 12 entstanden und wird für die Energieoptimierung von Software eingesetzt.

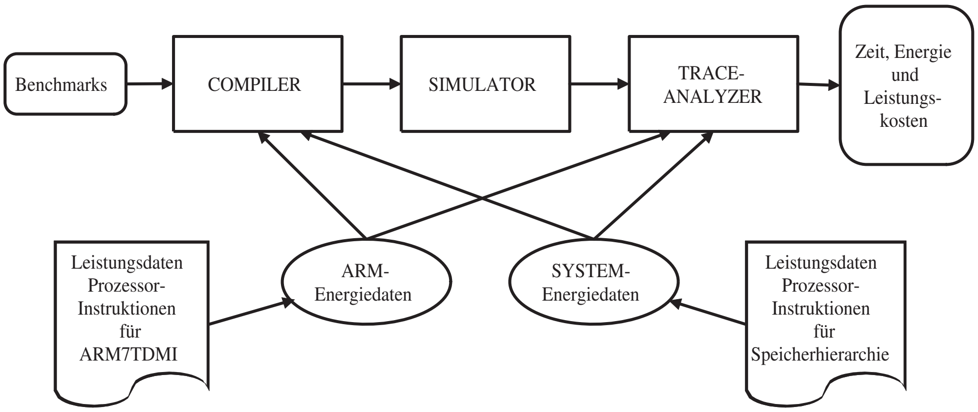

Abbildung 4.6: Entwicklungsumgebung für den ARM12CC-Compiler

Die Energiedaten aller Thumb-Instruktionen werden innerhalb der Entwicklungsumgebung an zwei Stellen benötigt:

1. **ARM12CC-Compiler**

   Im Compiler selbst wird bei Entscheidungsfreiheit eine Instruktion oder Instruktionssequenz ausgewählt, die einen geringeren Energieverbrauch hat. Weiterhin wird überprüft, ob der Einsatz verschiedener Optimierungsverfahren die Energieeigenschaften eines Programmes verbessert. Zur Validierung der Energieeigenschaften eines Programmes wird der im Anschluß beschriebene Trace-Analyzer verwendet.

2. **Trace-Analyzer**

   Hauptfunktion des Trace-Analyzers ist, den vom Simulator generierten Source-Trace mit entsprechenden Energiedaten zu versorgen. Das Ergebnis ist der Gesamtstromverbrauch eines Programmes, jeweils getrennt errechnet für Prozessor und Speicher.

4.6.1 Aufbau der Energiedatentabelle

Während der Entwicklungsarbeiten zu diesem Projekt sind zwei Konfigurationstabellen entstanden, um die Energiedaten der einzelnen Instruktionen zu erfassen:
Die erste Tabelle `ARM7TDMI.DAT` listet den gesamten Thumb-Befehlssatz mit Angaben zum Maschinencode, Wortgröße und Taktzyklenanzahl einer Instruktion auf. Weiterhin existiert für jede Instruktion eine Angabe der benötigten Prozessorleistung.
Eine zweite Tabelle `BOARDCONFIG.DAT` gibt zusätzliche Informationen zum eingesetzen Systemspeicher an. In Abhängigkeit vom gewählten Adressbereich und der Wortgröße bei einem lesenden oder schreibenden Datenzugriff wird die benötigte Energie aufgelistet. Weiterhin wird für jede verwendete Speicherhierarchie die Anzahl der benötigten Waitstate-Zyklen berücksichtigt.

4.7 Übertragbarkeit auf andere Architekturen

Die Übertragung des Energiemodells auf andere Rechnerarchitekturen ist mit geringem Aufwand realisierbar. Die Basiskostenberechnung einer Instruktion ist auf jede Von-Neumann-Architektur anwendbar. Für Load-/Store-Operationen müssen bei Systemen mit Cache-Speicher gesonderte Kostenbetrachtungen mit und ohne Cache-Misses durchgeführt werden. Für Pipeline-Stall-Effekte sind zusätzliche Wartezyklen bei einem Control-Hazard zu berücksichtigen. Für Messungen der Inter-Instruction-Effekte sollte, wie es für den ARM7TDMI-Instruktionssatz durchgeführt worden ist, eine Eingruppierung der Befehle in verschiedene Befehlsklassen erfolgen, um die Tupelmenge einzuschränken. Bei Untersuchungen der Kosten externer Speicher sind gegebenfalls systemabhängige Waitstate-Zyklen mit zu berücksichtigen.

Kapitel 5

Implementierung einer Messsoftware

Unter Berücksichtigung der verwendeten Testmuster für das ARM7TDMI-Energiemodell wurde im Rahmen dieser Diplomarbeit eine Messsoftware entwickelt, mit deren Hilfe der Stromverbrauch aller Thumb-Instruktionen gemessen worden ist. Diese Messsoftware besteht aus einer Sammlung verschiedener Messmodule für die Berechnung von Basis- und Inter-Instruction-Kosten, bei zusätzlicher Betrachtung von Daten- und Speicherabhängigkeiten. Für jede Instruktion existiert mindestens ein Messmodul.

5.1 Aufbau eines Messmoduls

Ein Messmodul besteht aus zwei Konfigurationsabschnitten `InitMem` und `InitData`, sowie einem Messabschnitt `InstLoop`. Innerhalb der drei Abschnitte werden folgende Aufgaben durchgeführt:

InitMem

Für Instruktionen, die Datenzugriffe ausführen, werden in diesem Abschnitt die verwendeten Speicherbereiche eingestellt. Zu nennen sind Speicherbereiche des On-Chip- oder Off-Chip-Speichers.

InitData

Innerhalb der Initialisierung erfolgt eine instruktionsabhängige Datenvorbelegung mit Testmustern für die verwendeten Operanden. Datenabhängigkeiten des Energieverbrauchs einer Instruktion können somit untersucht werden.

InstLoop

Gemäß Energiemodell wird jede Instruktion in einer Schleife durchlaufen. Genau 100 Wiederholungen eines Befehls werden ausgeführt, mit anschließendem Rücksprung an den Schleifenanfang. Der Rücksprungbefehl wird als Branch-Conditional implementiert, um die Möglichkeit eines kontrollierten Austritts aus der Schleife zu erzwingen. Dies ist insofern wichtig, falls Messmodule von übergreifenden Instanzen aufgerufen werden sollen, z.B. von einem Timer zur Berechnung von Waitstate-Zyklen des Speichers.

Die Abbildung 5.1 zeigt den Programmfluss beim Durchlaufen der einzelnen Abschnitte. Während des `InstLoop`-Abschnittes erfolgt die eigentliche Strommessung einer Instruktion.

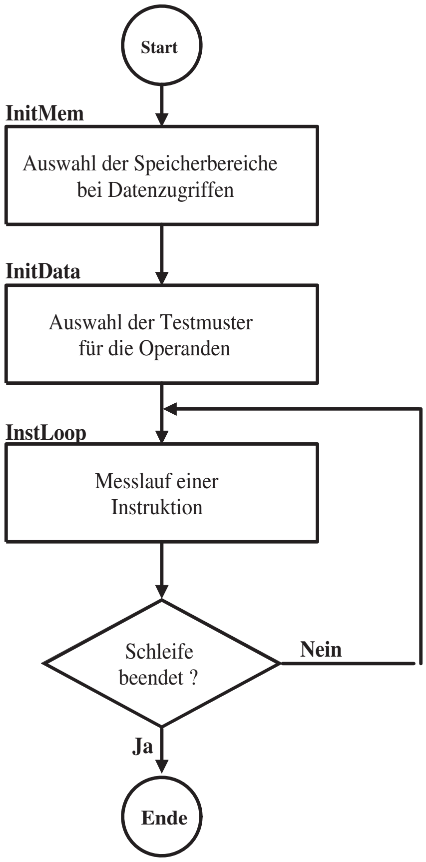

Abbildung 5.1: Programmfluss bei einem Messmodul

5.1.1 Auswahl der Schleifengröße

Die Wiederholhäufigkeit von 100 Instruktionen im `InstLoop`-Abschnitt erwies sich als geeignet, um stabile Werte auf dem Amperemeter anzuzeigen. Der Energieverbrauch des Branch-Befehls am Ende des Instruktionsblocks zeigte keine Auswirkung auf dem Messgerät. Dies wurde durch vorher ausgeführte Messreihen bestätigt, in dem verschiedene Instruktionsblöcke mit 1000 statt 100 Befehlen durchlaufen worden sind. Das Ergebnis auf dem Messgerät war identisch. Bei kleineren Schleifendurchläufe mit 20 Befehlen und weniger konnten jedoch minimale Schwankungen auf dem Messgerät festgestellt worden.

5.1.2 Inter-Instruction-Messungen

Für Messungen von Inter-Instruction-Effekten wird im `InstLoop`-Block die zu untersuchende Instruktion durch ein Instruktionstupel ersetzt. Eine Wiederholfrequenz von 100 Tupel erwies sich, wie für die Einzelmessung einer Instruktion, als geeignet.

5.1.3 Auszug aus der Implementierung

Im folgenden Abschnitt werden Ausschnitte aus der Implementierung am Beispiel von drei Messmodulen gezeigt. Die Messmodule sind gemäss des oben beschriebenen Aufbaus umgesetzt worden und zeigen einige instruktionsabhängige Besonderheiten, die zu beachten sind.

Beispiel 1 (LOAD WORD)

Des erste Beispiel zeigt den Source-Code eines Messmoduls zur Untersuchung einer 'Load Word with Register Offset'-Instruktion.

```
        AREA    Benchlib, CODE, READONLY
        ENTRY

InitMem
        AdrOnchip    EQU 0x500      ; Speicherbereiche definieren
        AdrOffchip   EQU 0x02050000
        AdrOffset    EQU 0x50
        AdrStack     EQU 0x02030000

        LDR     sp, =AdrStack       ; Stackpointer initialisieren
        LDR     r2, =AdrOffchip     ; Datenbereich Off-Chip
        LDR     r3, =AdrOffset      ; Offsetadresse LDR-Befehl
        ADR     r0, init +1         ; Thumb-Modus vorbereiten
        BX      r0                  ; Umschalten nach Thumb
        CODE16

InitData
        LDR     r0, =0xFFFFFFFF     ; Schleifenzaehler
        LDR     r1, =0xAA           ; Verwendetes Testmuster
        STR     r1, [r2,r3]         ; Testmuster abspeichern

InstLoop
        LDR     r1, [r2,r3]         ; Basiskostenbestimmung
        LDR     r1, [r2,r3]         ; einer LDR-Instruktion
                .
                .
        LDR     r1, [r2,r3]         ; 100 Instruktionen
        SUB     r0, #1
        BNE     Instloop
```

Der Adressbereich für Datenzugriffe dieser Instruktion wird in diesem Beispiel auf den Off-Chip-Speicher gelegt. Weiterhin erfolgt während der Initialisierung die Auswahl und das Speichern eines Testmusters im gewählten Adressbereich. Innerhalb der Messschleife wird das abgelegte Testmuster von der Load-Instruktion in einer sich wiederholenden Schleife permanent eingelesen.

Beispiel 2 (PUSH REGLIST)

Das zweite Beispiel zeigt die Besonderheiten bei der Implementierung von Stackmodulen. Eine ständige Wiederholung von Stackpointer-Operationen in einer Schleife würde bei entsprechender Anzahl von Durchläufen einen Stack-Overflow generieren. Deshalb wird der aktuelle Stackpointer im `InitData`-Abschnitt gesichert und vor dem Rücksprung des Branchbefehls zum Schleifenanfang `InstLoop` zurückgesetzt, wie folgender Programmauszug für ein Push-Modul zeigt:

```
InitData
        LDR     r0, =0xFFFFFFFF     ; Schleifenzaehler
        LDR     r2, =0xAA           ; erstes  Testmuster
        LDR     r3, =0xFFFF         ; zweites Testmuster
        MOV     r1, SP              ; Stack sichern

Instloop
        PUSH    {r2,r3}
        PUSH    {r2,r3}
          .
          .
        PUSH    {r2,r3}             ; 100 Instruktionen
        MOV     SP, r1              ; Stack auf alte Position setzen
        SUB     r0, #1
        BNE     Instloop
```

Zur Untersuchung von POP-Instruktionen ist es zusätzlich notwendig, innerhalb der Initialisierung den Stack mit PUSH-Anweisungen aufzufüllen bevor die eigentliche POP-Instruktion untersucht werden kann. Zu beachten ist dabei, dass mindestens genausoviele Daten auf dem Stack abgelegt wie später innerhalb der Schleife wieder abgeholt werden. Ansonsten würde ein Stack-Underflow entstehen.
Die Energiekosten des abschließenden MOV-Befehls, um den Stack auf seine alte Position zurückzusetzen, ist bei der hohen Wiederholhäufigkeit von 100 aufeinanderfolgenden Stack-Instruktionen zu vernachlässigen, da die Auflösung des verwendeten Messgerätes diesen Effekt nicht mehr erfassen kann.

Beispiel 3 (BRANCH)

Bei Branch-Befehlen ist es interessant zu wissen, ob Energiekosten von unterschiedlichen Sprungweiten abhängig sind. Für diesen Fall sind im Loop-Teil des Messmoduls zwei sich gegenseitig aufrufende Branch-Befehle eingefügt worden, die in unterschiedlichen Messreihen durch verschieden lange NOP-Sequenzen getrennt werden. Dazu folgendes Beispiel:

```
Instloop1
        B       Instloop2
        MOV     r8, r8              ; NOP
         .
         .
        MOV     r8, r8              ; NOP
Instloop2
        B       Instloop1
```

Für die Basiskostenbestimmung einer Instruktion werden demnach Branch-Befehle nicht wie die restlichen Thumb-Befehle in einer sich wiederholenden Schleife gemessen, sondern während ihres gegenseitigem Aufrufes.

5.2 Einsatzmöglichkeit der Messmodule

Die Messmodule können in zwei verschiedenen Varianten eingesetzt werden:

- **als eigenständige Programme**

  Die Messmodule sind kleine relokierbare Assemblerprogramme, die auch in den 4K-Byte grossen On-Chip-Speicher des Prozessors gelegt werden können. Die Relokierung innerhalb des Speichers erfolgt über eine entsprechende Anweisung im Linker. Das Verändern der Dateninhalte von Register und Speicherzellen zur Untersuchung von Datenabhängigkeiten einer Instruktion wird im Initialisierungsabschnitt vorgenommen, oder alternativ dazu, mit Hilfe des ARM-Debuggers, während des Programmlaufs. Dazu wird das laufende Programm durch den Debug-Host angehalten, die Registerwerte oder Speicherzellen verändert und anschließend das Programm erneut gestartet. Für Befehle mit einem Immediate-Anteil ist dieses Vorgehen im Debugger nicht möglich, da der Immediate-Operand fester Bestandteil der Instruktion ist. Mit einem Editor müssen deshalb innerhalb der `InstLoop`-Schleife, die Immediate-Werte ausgetauscht und anschließend neu assembliert werden, um auch die Auswirkungen dieser Operanden näher untersuchen zu können.

- **integriert innerhalb eines Hochsprachenprogramms**

  Innerhalb eines Programmes werden die verschiedenen Messmodule als einzelne Messfunktionen eingesetzt, die von einer Hauptinstanz nacheinander aufgerufen werden können. Durch dieses Vorgehen wird eine automatisierte Messung aller Instruktionen in einem einzigen Durchlauf ermöglicht. Für die Überführung eines Messmoduls zu einer Funktion muss der Header erweitert werden. Jedes Messmodul erhält eine Parameterschnittstelle, um die Versorgung mit instruktionsabhängigen Konfigurationsdaten zu berücksichtigen. Die Verwendung dieses Methode unterliegt jedoch zwei Einschränkungen:

1. Dateninhalte bei Instruktionen mit Immediate-Anteil können wie bei der ersten Variante nicht parametrisiert werden. Alternativ dazu muss für Untersuchungen von Datenabhängigkeiten jede betroffene Messfunktion dupliziert und anschließend modifiziert werden. Die Anzahl der Messfunktionen wird dadurch deutlich erhöht.
2. Ein weitaus größerer Nachteil der automatischen Messung sind die eingeschränkten Möglichkeiten, Untersuchungen innerhalb unterschiedlicher Speicherhierarchien durchzuführen. Die Übersetzung aller Messfunktionen zu einem Gesamtprogramm generiert ein sehr großes Image-File, das nicht in den On-Chip-Speicher des Prozessors platziert werden kann.

Für die Messungen des Prozessor- und Speicherstroms hat sich im Laufe dieser Arbeit die erste Variante, das manuelle Messen mit kleinen Stand-Alone-Programmen als sinnvoller erwiesen, als die automatische Messung. Massgebliche Gründe hierfür sind die uneingeschränkten Relokierungsmöglichkeiten von kleinen Assemblerprogrammen innerhalb der verschiedenen Speicherhierarchien des ATMEL-Boards und die Möglichkeit, mit Hilfe des ARM-Debuggers Registerdaten und Speicherzellen zur Programmlaufzeit sofort verändern zu können, falls während der Messungen Auffälligkeiten im Stromverlauf einzelner Instruktionen auftreten.

5.3 Nicht untersuchte Befehle

Die Leistung folgender Befehle kann aufgrund spezieller Befehlseigenschaften mit der Methode der Basiskostenbestimmung nicht ermittelt werden und wird deshalb aus Gründen der Vollständigkeit mit Leistungswerten ähnlicher Instruktionen abgeschätzt:

1. **SWI 8BIT-IMMEDIATE**

   Sprunganweisungen in Interruptroutinen werden mit den Kosten eines Branch-Befehls abgeschätzt.

2. **POP REGLIST, PC**

   Für diese Stack-Instruktion wird eine Abschätzung gemäß 'POP Reglist' durchgeführt. Die Registerliste beinhaltet dabei ein zusätzliches Register als Ersatz für das Program-Counter-Register R15.

3. **BX LABEL**

   Für 'Branch and Exchange'-Anweisung wird eine Abschätzung gemäß 'B Label' durchgeführt.

5.4 Timer für Zyklenberechnung

Für die Errechnung der Gesamtzyklenanzahl einer Instruktion ist eine Timermessung durchzuführen. Der folgende Sourcecode in ANSI-C zeigt die dazu verwendete Timerimplementierung am Beispiel des Aufrufes eines Load-Messmoduls:

```
Time = clock();
Call_Load_Reg_Word_Offset (Loop, Addr, Offset, Data);
Time = clock() - Time;
TimeSec = (double)Time / CLOCKS_PER_SEC
```

Der Timer misst die Ausführungszeit einer beliebigen Funktion in Sekunden. Die aufzurufende Funktion `CallLoadRegWordOffset` wird durch die Parameter `Loop`, `Addr`, `Offset` und `Data` parametrisiert. `Loop` entspricht der Anzahl der Schleifendurchläufe, `Addr` und `Offset` der Zieladresse für das Testmuster `Data`. Die gemessene Zeit `TimeSec` wird im Anschluss zur Zyklenzahlbestimmung in die in Kapitel 2 definierte Formel $T = n \cdot t$ eingesetzt. Eine Umstellung der Gleichung nach n gibt anschließend die Anzahl aller Taktzyklen für den Schleifendurchlauf aus. Teilt man das Ergebnis abschließend durch die Anzahl der Schleifenwiederholungen `Loop` und die Instruktionswiederholrate (100 Wiederholungen), ergibt sich bei einer Taktfrequenz von $t = 32,768\,\mathrm{MHz}$ folgende Formel zur Zyklenzahlbestimmung einer Instruktion:

$$n = \frac{TIME \cdot 32,768 \cdot 10^6}{Loop \cdot 100}$$

5.5 Entwicklungsumgebung ARM-SDT

Die einzelnen Messmodule wurden mit dem ARM-SDT 2.50 (ARM Software Development Toolkit) entworfen. Unter der grafischen Benutzeroberfläche APM (ARM Project Manager) sind eine Vielzahl von Entwicklungstools integriert, die zur Programmentwicklung von ARM7TDMI-Code notwendig sind. Die Abbildung 5.2 zeigt alle Komponenten des ARM-SDT im Gesamtüberblick.

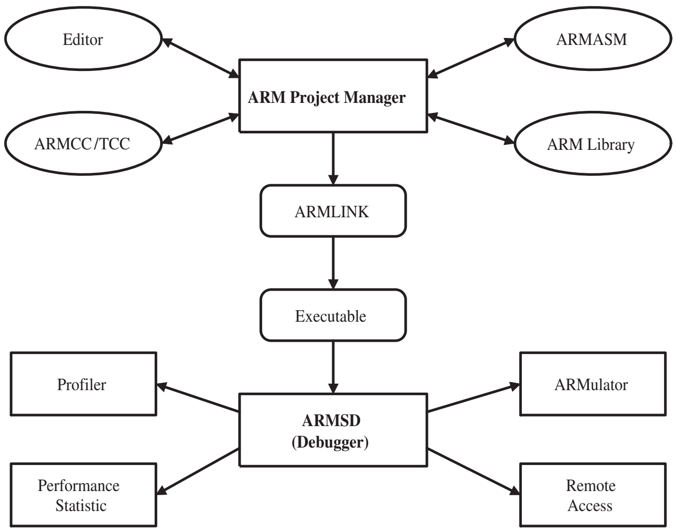

Abbildung 5.2: Entwicklungsumgebung ARM-SDT [ARM95b]

ARMCC und TCC

Die beiden C-Compiler für ARM7TDMI-Quellcode. Der ARMCC-Compiler übersetzt ANSI-C konformen Sourcecode in 32Bit-ARM-Maschinencode. Der TCC-Compiler übersetzt den gleichen Code in 16Bit-Thumb-Maschinencode.

ARMASM

Der ARM-Assembler ist in der Lage, ARM- und Thumb-Code in linkfähigen Objektcode zu übersetzen. Die Dualität des Assemblers ist erforderlich, da gemischter ARM- und Thumb-Code vom Prozessor akzeptiert wird.

ARMLINK

Der ARM-Linker erzeugt aus dem generierten Objektcode lauffähige Programme für den ARM7TDMI.

ARMSD

Der ARM-Debugger innerhalb des ARM-SDT. Er wird für folgende Anwendung benötigt:

- **Armulator**
  Der Armulator emuliert Prozessor und Arbeitsspeicher eines ARM7TDMI-Prozessors. Er dient als Testsystem, bevor Programme auf ein Echtzeitsystem übertragen und ausgeführt werden. Der Armulator unterstützt die beiden Befehlssätze des ARM7TDMI-Prozessors.
- **Remote Access**
  Über Remote-Access erfolgt die Übertragung und das Debugging von ARM7TDMI-Executables auf entfernte ARM7TDMI-Laufzeitsysteme. Innerhalb von Remote-Access wird das Kommunikationsprotokoll für Angel-Requests unterstützt.
- **Profiler**
  Die Anzahl der durchlaufenen Funktionsblöcke werden vom Profiler protokolliert und statistisch ausgewertet.

Kapitel 6

Energiemessung und Auswertung

In diesem Kapitel wird anhand von ausgewählten Messreihen der Einfluss einer Instruktion auf den Energieverbrauch von Prozessor und Speicher gezeigt. Eine vollständige Energietabelle aller gemessenen Thumb-Instruktionen ist im Anhang A zu finden.

6.1 Errechnete Zyklenzahl

Die Anzahl benötigter Taktzyklen und somit die benötigte Zeit zum Verarbeiten einer Instruktion im Prozessor ist von der gewählten Speicherkombination für Instruktionen und Daten abhängig. Liegen Instruktionen und Daten auf dem On-Chip-Speicher, sind die in den Datenblättern angegebenen Zyklenzahlen bestätigt worden. Daraus folgt, dass keine zusätzlichen Zyklen für Speicher-Waitstates bei Zugriff auf den On-Chip-Speicher anfallen. Für alle anderen Speicherkombinationen sind die in Tabelle 6.1 angegebenen Zyklenangaben bestimmt worden.

Instruktion	Daten	Zyklen	Instr.-Fetch	Data 32	Data 16	Data 8
OFF-CHIP	OFF-CHIP	m	1	$n\times 3$	1	1
OFF-CHIP	ON-CHIP	m	1	0	0	0
ON-CHIP	OFF-CHP	m	0	$n\times 3$	1	1
ON-CHIP	ON-CHIP	m	0	0	0	0

Tabelle 6.1: Taktzyklen für verschiedene Speicherkombinationen

Die Zyklen m sind dabei instruktionsabhängig. Zusätzliche Waitstate-Zyklen, die beim Zugreifen auf dem Speicher entstehen, sind unterteilt in:

Instruction-Fetch

Liegt eine Instruktion im externen Speicher, wird für 16-Bit-Thumb-Instruktionen genau ein zusätzlicher Zyklus benötigt, um die Instruktion aus dem Speicher auszulesen.

Datenzugriffe

Führt die Instruktion von sich aus Lese- oder Schreiboperationen auf dem externen Speicher aus, sind für jeden 32-Bit-Speicherzugriff drei zusätzliche Zyklen, für 16-Bit- und 8-Bit-Speicherzugriffe ein zusätzlicher Zyklus errechnet worden. Die Variable n gibt dabei an wie oft eine Instruktion auf den Speicher zugreift. Für Multiple-Instruktionen entspricht n der Anzahl der verwendeten Register, in allen anderen Fällen ist $n = 1$.

Tabelle 6.2 zeigt am Beispiel einer Load-Word-Instruktion unterschiedliche Zyklenzahlen bei der Positionierung von Instruktion und Daten in verschiedenen Speicherkombinationen. Die Kombination Instruktion- und Daten-Off-Chip ergibt dabei die höchste Zyklenzahl.

Load Word	Daten	Zyklen	Instr.-Fetch	Data 32	Total
OFF-CHIP	OFF-CHIP	3	1	3	7
OFF-CHIP	ON-CHIP	3	1	0	4
ON-CHIP	OFF-CHP	3	0	3	6
ON-CHIP	ON-CHIP	3	0	0	3

Tabelle 6.2: Taktzyklen einer Load-Word-Instruktion

6.2 Prozessorstrom

6.2.1 Basiskosten einer Instruktion

Die Abbildung 6.1 zeigt in einer ersten Übersicht ermittelte Stromwerte für den Thumb-Instruktionssatz. Instruktionen, Daten und Stack liegen dabei auf dem externen Speicher. Unterschiedliche Balkenlängen zeigen an, inwieweit eine Instruktion, abhängig von den gewählten Operanden, den Prozessorstrom beeinflussen kann.

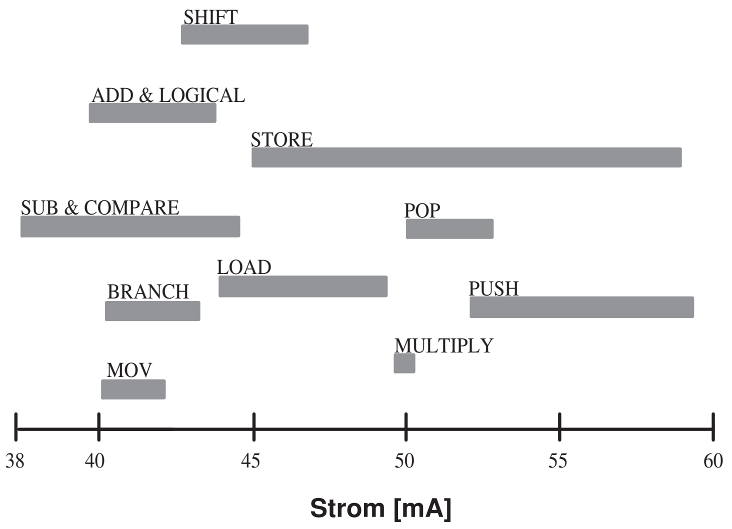

Abbildung 6.1: Stromwerte von Thumb-Instruktionen

Für eine erste Betrachtung wird folgendes festgestellt:

1. Instruktionen, die innerhalb der ALU ausgeführt werden, erzeugen annähernd dieselben Stromwerte am Prozessor. Dazu gehören SUB-, ADD- und Compare-Operationen sowie alle logischen Bitverknüpfungen.

2. Registertransfer- und Sprungbefehle, dazu gehören Move- und Branch-Instruktionen, zeigen ebenfalls einen ähnlichen Stromverlauf wie ALU-Instruktionen.

3. Shift-Operationen werden innerhalb des Barrelshifters verarbeitet. Die Verwendung dieser Ressource erhöht den Prozessorstrom. Barrelshifter-Operationen sind teurer als ALU-Operationen.

4. Multiplikationen, die im separaten Multiplier verarbeitet werden, sind die teuersten arithmetischen Operationen. Auch hier gilt, dass der Multiplier höhere Schaltwechselfolgen generiert als die ALU oder der Barrelshifter.

5. Speicherzugriffsbefehle, wie Load-, Store- und Stack-Instruktionen erzeugen im Durchschnitt die höchsten Stromwerte am Prozessor. Weiterhin ist der Stromverlauf dieser Befehle maßgeblich von der verwendeten Speicherkombination für Instruktion und Daten abhängig, wie spätere Untersuchungen zeigen werden.

6.2.2 Off-Chip vs. On-Chip

In diesem Abschnitt wird der Prozessorstrom in Abhängigkeit von der Positionierung von Instruktion und Daten in den beiden Speicherbereichen des Off-Chip- und On-Chip-Speichers genauer untersucht.

Registertransfer-, ALU- und Sprungbefehle

Tabelle 6.3 zeigt gemessene Stromwerte von Registertransfer-, ALU- und Sprungbefehlen. Diese Instruktionen erzeugen, bei Positionierung auf dem Off-Chip-Speicher, einen durchschnittlichen Strom von 42,6 mA. Liegt eine Instruktion auf dem On-Chip-Speicher, erhöht sich der Prozessorstrom, abhängig von der gewählten Instruktionsart, um 5-8 mA. Eine erhöhte Schaltkreisaktivität des On-Chip-Speichers bewirkt diesen Anstieg. Variationen bis 3 mA kommen dadurch zustande, dass die Opcodes der Instruktionen unterschiedlich sind. Je mehr Einsen im Opcode einer Instruktion auftauchen, desto höher ist der Prozessorstrom bei Verwendung des On-Chip-Speichers.

Instruktion	OFF/OFF (mA)	ON/ON (mA)	Diff
MOV 8 BIT IMM	41,3	47,1	5,8
MOV HI TO LO	42,4	49,1	6,7
ADD 8 BIT IMM	41,2	46,9	5,7
ADD LO AND LO	42,2	48,8	6,6
SUB WITH CARRY	40,3	45,2	4,9
CMP HI AND HI	41,0	46,4	5,4
MUL 32 x 32	49,8	56,4	6,6
LOGICAL AND	41,9	48,1	6,2
LOGICAL OR	42,4	49,3	6,9
SHIFT LEFT 5 BIT IMM	43,7	51,7	8,0
SHIFT RIGHT 5 BIT IMM	43,7	51,8	8,1
BRANCH IF Z CLEAR	41,7	46,8	5,1
BRANCH UNCONDITIONAL	42,9	48,2	5,3
BRANCH LONG	42,3	49,7	7,4
Durchschnitt	42,6	48,9	+6,3

Tabelle 6.3: Prozessorstrom für Registertransfer-, ALU- und Sprungbefehle

Speicherzugriffsbefehle

Für Load-, Store- und Stack-Instruktionen ist der Stromverlauf des Prozessors davon abhängig, in welcher Speicherkombination sich Instruktion und Daten befinden. Tabelle 6.4 zeigt dazu gemessene Stromwerte. Die Durchschnittswerte dieser Messung sind in Abbildung 6.2 als Balkendiagramm visualisiert. Der Strom variiert für diese Instruktionen in einem Bereich von 44,2-57,5 mA. Aus der Sicht einer Leistungsoptimierung ist die Kombination Instruktion-On-Chip und Daten-Off-Chip für diese Befehle die günstigste Wahl, da in dieser Kombination die Stromaufnahme des Prozessors am geringsten ist. Wird eine Energieoptimierung angestrebt, ist die Kombination Instruktion- und Daten-On-Chip vorzuziehen, weil nur in dieser Kombination der Prozessor die geringste Anzahl an Taktzyklen benötigt und damit das Produkt aus Leistung und Zeit minimal wird.

Instruktion	OFF/OFF (mA)	OFF/ON (mA)	ON/OFF (mA)	ON/ON (mA)
LOAD WORD	44,2	47,8	42,1	50,2
LOAD HALFWORD	47,9	46,3	45,1	47,4
LOAD BYTE	47,9	46,3	45,2	47,0
Durchschnitt Load	46,6	46,8	44,1	48,2
STORE WORD	49,1	51,3	45,5	56,0
STORE HALFWORD	55,4	49,8	51,3	53,6
STORE BYTE REG	55,3	49,7	50,5	53,1
Durchschnitt Store	53,2	50,2	49,1	54,2
PUSH 1 REG	57,1	53,1	47,4	57,5
POP 1 REG	51,9	51,1	44,9	51,5
Durchschnitt Stack	54,5	52,1	46,1	54,5

Tabelle 6.4: Prozessorstrom für Speicherzugriffsbefehle

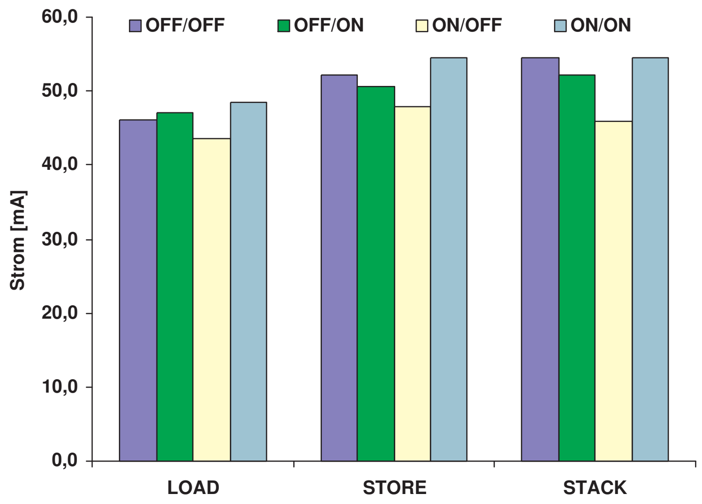

Abbildung 6.2: Prozessorstrom von Speicherzugriffsbefehle in Abhängigkeit von verschiedenen Speicherkombinationen

Für den Stromverlauf von Multiple-Instruktionen stellt sich ein ähnliches Bild dar, wie Tabelle 6.5 an weiteren Messwerten zeigt. Für diese Instruktionen gilt weiterhin, dass für jedes zusätzliche Register der Prozessorstrom geringfügig abnimmt. Der Grund hierfür liegt in einer steigenden Taktzyklenzahl bei Erhöhung der Registerzahl und der damit verbundenen längeren Bearbeitungszeit des Prozessors, um die Instruktion auszuführen. Der Instruction-Fetch für das Holen der nächsten Instruktion verzögert sich dadurch geringfügig. Aus dem Absinken des Prozessorstroms bei Erhöhung der Registerzahl kann gefolgert werden, dass der Instruction-Fetch teurer ist als die eigentliche Befehlsverarbeitung. Abbildung 6.3 verdeutlicht diesen Effekt des Stromabfalls anhand einer Load-Multiple-Instruktion bei Variation von 8 Registern.

Instruktion	OFF/OFF (mA)	OFF/ON (mA)	ON/OFF (mA)	ON/ON (mA)
LOAD MULTIPLE				
LOAD MULTIPLE 1 REG	45,6	48,7	44,6	50,8
LOAD MULTIPLE 2 REG	43,8	48,3	42,9	50,5
LOAD MULTIPLE 3 REG	43,1	48,1	42,0	50,3
LOAD MULTIPLE 4 REG	42,6	47,8	41,6	50,0
STORE MULTIPLE				
STORE MULTIPLE 1 REG	47,4	52,7	47,3	54,7
STORE MULTIPLE 2 REG	45,5	52,2	46,6	53,1
STORE MULTIPLE 3 REG	43,3	51,6	45,8	52,4
STORE MULTIPLE 4 REG	43,1	51,2	45,2	52,2

Tabelle 6.5: Prozessorstrom von Multiple-Instruktionen

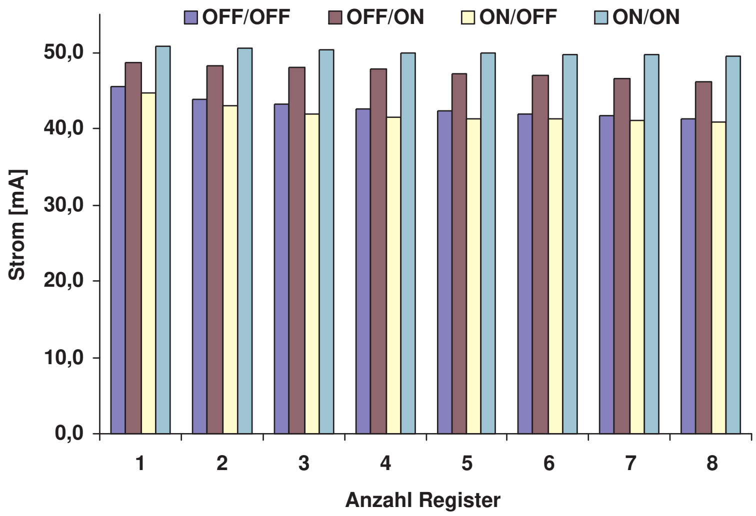

Abbildung 6.3: Stromabfall am Prozessor an einer LDMIA-Instruktion bei Erhöhung der Registeranzahl

6.2.3 Datenabhängigkeiten

Instruktionen zeigen bei Wahl unterschiedlicher Operandeninhalte Variationen in ihrem Stromverlauf auf. Messwerte einer Add-Instruktion 'ADD R1,R2,R3' mit verschiedenen Registerinhalten sind in Tabelle 6.6 dargestellt. Die Anzahl der Einsen innerhalb der Operanden ist für den Stromverlauf entscheidend. Je mehr Einsen in den Registern R2 und R3 auftreten, umso höher die Stromaufnahme des Prozessors. Das Ergebniss der Addition in Register R1 spielt dabei keine Rolle. Abbildung 6.4 visualisiert diesen Effekt. Der Stromanstieg am Prozessor bei Erhöhung der Anzahl von Einsen ist für jede Instruktion aufgetreten. Tabelle 6.7 zeigt maximale Stromunterschiede verschiedener Instruktionen bei Variation von Registerinhalten und Immediate-Operanden. Die Stromunterschiede sind dabei unterschiedlich stark ausgeprägt. So variiert der Strom für eine Load-Word-Anweisung in der Speicherkombination Instruktion- und Daten-Off-Chip maximal um 1,1 mA. Im Gegensatz dazu bei einer Store-Word-Anweisung bei gleicher Datenwahl und gleicher Speicherkombination um bis zu 9,3 mA. Es zeigt sich jedoch, dass für ALU- und Registertransferbefehle mit Immediate-Operanden der Prozessorstrom nicht so stark variiert wie bei ähnlichen Befehlen mit Register-Operanden. Dies erklärt sich dadurch, dass Immediate-Werte im Thumb-Befehlssatz im Gegensatz zu Registerwerten, nur eine Größe von maximal 10 Bit erreichen und somit weniger Einsen speichern können.

Reg 2	Reg 3	Anzahl Einsen	OFF-CHIP (mA)	ON-CHIP (mA)
0x0	0x0	0	40,3	45,1
0x0	0x1	1	40,4	45,3
0x5	0x7	5	40,8	46,0
0xAA	0xFF	12	41,1	46,7
0xAAAA	0xFFFF	24	41,6	47,6
0xAAAAAA	0xFFFFFF	36	42,0	48,6
0xAAAAAAAA	0xFFFFFFFF	48	42,7	49,9
0xFFFFFFFF	0xFFFFFFFF	64	44,0	52,6

Tabelle 6.6: Prozessorstrom einer ADD-Instruktion bei Variation der Registerinhalte

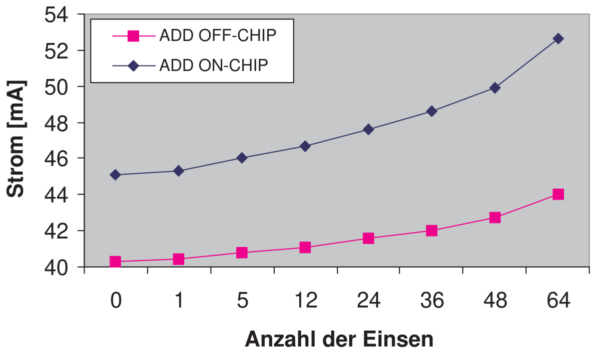

Abbildung 6.4: Stromverlauf einer ADD-Operation

Instruktion	OFF/OFF (mA)	OFF/ON (mA)	ON/OFF (mA)	ON/ON (mA)
MOV 8 BIT IMM	1,2	-	2,3	-
MOV HI TO LO	3,3	-	6,8	-
ADD 8 BIT IMM	2	-	4,1	-
ADD LO AND LO	3,8	-	5,2	-
SUB WITH CARRY	5,2	-	10,9	-
CMP HI and HI	7	-	13,3	-
LOGICAL AND	4,7	-	2,5	-
SHIFT LEFT 5 BIT IMM	1,1	-	2,5	-
LOAD WORD	1,1	1,0	1,2	3,7
LOAD HALFWORD	0,5	0,5	0,6	1,9
LOAD BYTE	0,3	0,4	0,3	1,0
STORE WORD	7,8	1,9	2,3	3,8
STORE HALFWORD	9,3	1,7	3,5	3,1
STORE BYTE	9,2	1,7	2,0	3,0
PUSH 1 REG	8,5	2,9	2,8	4.5
POP 1 REG	3	3,9	1,9	3,7

Tabelle 6.7: maximale Stromunterschiede von Instruktionen bei Variation der Registerinhalte und Immediate-Operanden

6.2.4 Unterschiedliche Wortgrößen

Speicherzugriffe bei Verwendung unterschiedlicher Wortgrößen beeinflussen ebenfalls den Prozessorstrom. Entscheidend ist auch hier die gewählte Speicherkombination für Instruktion und Daten. So ist ein 32-Bit-Word-Zugriff nicht immer die teuerste Wahl, wie Abbildung 6.5 zeigt. Die drei Store-Operationen in der Kombination Instruktion- und Daten-Off-Chip speichern dabei verschieden große 8-Bit-, 16-Bit- und 32-Bit-Wörter ab. Store-Halfword und Store-Byte-Instruktionen erzielen bei dieser Speicherkombination höhere Stromwerte als eine Store-Word-Anweisung. Dieser Effekt tritt ebenfalls auf, falls die Instruktion im On-Chip-Speicher liegt. Erst bei einer Datenbeschaffung aus dem On-Chip-Speicher ist eine Store-Word-Operation die teuerste Wahl. Das selbe Verhalten tritt in diesem Zusammenhang auch bei Load-Instruktionen auf.

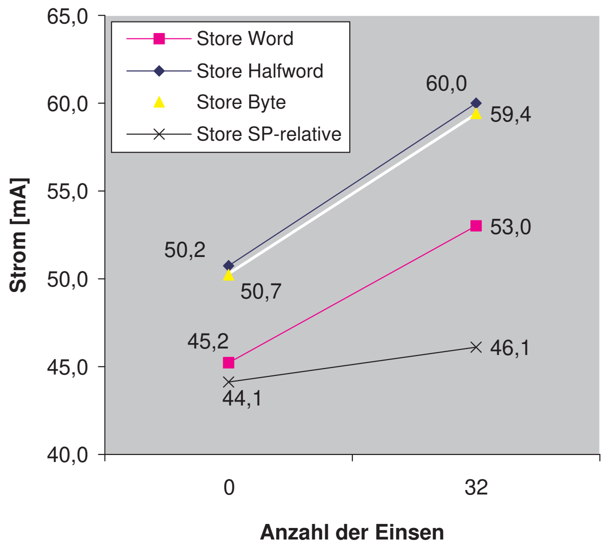

Abbildung 6.5: Prozessorstrom unterschiedlicher Store-Zugriffe

Bei Betrachtung der entsprechenden Zyklenzahl für verschiedene Store-Operationen, abgebildet in Tabelle 6.8, zeigt sich, dass Store-Word-Instruktionen bei Zugriff auf den externen Speicher die meisten Zyklen benötigen. Die Begründung für die geringere Leistung ist identisch zu der bei Multiple-Instruktionen. Bedingt durch die niedrige Zyklenzahl von Halfword- und Byte-Operationen, erfolgt der nächste Instruction-Fetch bei diesen Operationen eher als bei Word-Operationen. Die Kosten für den nächsten Instruction-Fetch sind höher als die Kosten für die eigentliche Bearbeitung der aktuellen Instruktion. Bei Datenzugriffen auf den On-Chip-Speicher ist die Taktzyklenzahl für alle drei Zugriffsarten identisch, und somit ist ein Word-Zugriff die teuerste Wahl, da der entscheidene Faktor Wortgröße eines Datums dort wieder dominiert.

Instruktion	Daten	Zyklen Store 32	Zyklen Store 16	Zyklen Store 8
OFF-CHIP	OFF-CHIP	**6**	4	4
OFF-CHIP	ON-CHIP	3	3	3
ON-CHIP	OFF-CHP	**5**	3	3
ON-CHIP	ON-CHIP	2	2	2

Tabelle 6.8: Benötigte Taktzyklen verschiedener Store-Operationen

6.2.5 Sprungweiten

Unterschiedliche Sprungweiten von Branch-Instruktionen haben ebenfalls eine Wirkung auf den Prozessorstrom, wie Tabelle 6.9 zeigt. Dabei ist jedoch nicht die eigentliche Sprungweite oder die Adresse des Sprungziels entscheidend, sondern wie bei den untersuchten Datenabhängigkeiten die Anzahl der Einsen, die für die Sprungvorhersage-Berechnung in der ALU für das Register R15 (Program Counter) berechnet werden müssen. So ist ein 500-Byte-Sprung (Binar: 111110100) mit 6 Einsen 2,4 mA teurer als ein 1K-Byte-Sprung, da dieser Sprungwert nur eine Eins enthält. Abbildung 6.6 visualisiert diesen Effekt der Stromaufnahme bei Erhöhung der Einsen. Ein nicht ausgeführter Branch-Conditional-Befehl erzeugt demnach im Prozessor den geringsten Strom, weil keine teuren Offset-Berechnungen des Sprungziels durchgeführt werden müssen.

Instruktion	Min-Strom (mA)	Max-Strom (mA)	Diff (mA)
OFF-CHIP			
BRANCH CONDITIONAL	41,1	42,5	1,4
BRANCH UNCONDITIONAL	41,3	44,5	3,2
BRANCH LONG WITH LINK	41,7	42,8	1,1
ON-CHIP			
BRANCH CONDITIONAL	46,0	47,5	1,5
BRANCH UNCONDITIONAL	47,3	49,1	1,8
BRANCH LONG WITH LINK	49,4	50,0	0,6

Tabelle 6.9: Prozessorstrom für Branchbefehle

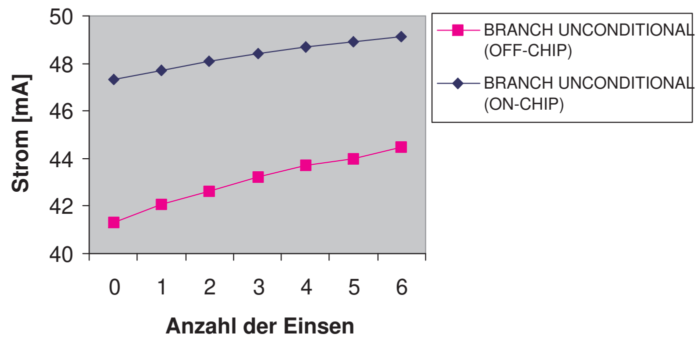

Abbildung 6.6: Stromverlauf einer Branch-Unconditional-Instruktion bei Variation der Einsen für den Sprungwert

6.2.6 Inter-Instruction-Effekte

Die Ergebnisse teilen sich auf in Inter-Instruction-Effekte zwischen Befehlen gleicher Klasse, abgebildet in Tabelle 6.10, und zwischen Befehlen verschiedener Klassen, dargestellt in Tabelle 6.11. Inter-Instruction-Effekte zwischen Instruktionstupeln einer Klasse sind geringer, mit durchschnittlich 0,4 mA, als bei Instruktionstupeln, die zu verschiedenen Klassen gehören. Hier variiert der Strom zwischen 2-4 mA. Der durchschnittliche Inter-Instruction-Effekt über alle durchgeführten Messreihen für den ARM7TDMI beträgt 2 mA, unabhängig davon, ob die Instruktion auf dem On-Chip- oder Off-Chip-Speicher positioniert ist.

Instruktion 1	Instruktion 2	Erwartet (mA)	Gemessen (mA)	Diff (mA)
SUB R1,R2,R3	ADD R1,R2,R3	41,7	42,4	0,7
LDR R1,R2,R3	STR R1,R2,R3	46,2	46,6	0,4
BNE LABEL	BCS LABEL	41,8	42,0	0,2
LSL R1,R2	LSR R1,R2	47,4	48,0	0,6
LDR R1,R2,R3	STR R1,R2,R3	46,2	46,6	0,4

Tabelle 6.10: Inter-Instruction-Effekte zwischen Befehlen gleicher Klasse

Instruktion 1	Instruktion 2	Erwartet (mA)	Gemessen (mA)	Diff (mA)
MOV R1,R8	ADD R1,R2,R3	42,0	45,8	3,8
MOV R1,R8	MUL R1,R2	48,3	50,4	2,1
CMP R1,R2	BNE LABEL	41,6	45,5	3,9
ADD R1,R2,R3	POP R1	49,4	52,1	2,7

Tabelle 6.11: Inter-Instruction-Effekte zwischen Befehlen fremder Klasse

Der Einfluss verschiedener Hamming-Distanzen, zweier aufeinanderfolgender Instruktionen wird anhand eines Instruktionstupels mit ADD-Immediate-Instruktionen untersucht. Der Immediate-Operand ist für solche Instruktionen Bestandteil des Opcodes, und kann daher leicht variiert werden. Die verwendeten Immediate-Werte sind in Tabelle 6.12 dargestellt. Die errechneten Inter-Instruction-Effekte dieser Messung, visualisiert in Abbildung 6.7, zeigen, dass bei einer Erhöhung der Hamming-Distanz zweier Instruktionen der Prozessorstrom ansteigt.

Immediate-Wert 1	Immediate-Wert 2	Hamming-Distanz
0x1	0x3	1
0x1	0x7	2
0x1	0xf	3
0x1	0x1f	4
0x1	0x3f	5
0x1	0x7f	6
0x1	0xff	7

Tabelle 6.12: Verwendete Immediate-Operanden für ADD-Instruktionstupel

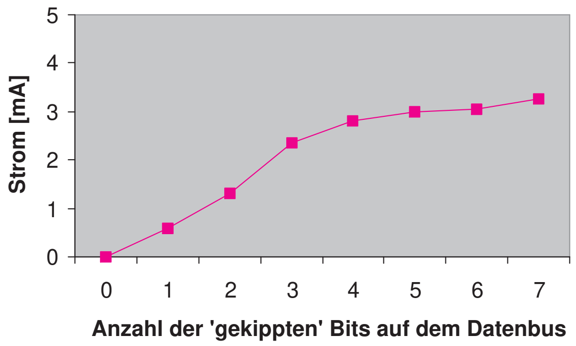

Abbildung 6.7: Inter-Instruction-Effekte eines Instruktionstupels mit unterschiedlicher Hamming-Distanz

Eine Reduzierung des Hamming-Abstandes kann aus diesem Grunde die Stromaufnahme des Prozessors senken. Dazu ein Beispiel an zwei einfachen Instruktionssequenzen mit gleichen Befehlen, nur unterschiedlich sortiert:

```
ADD R1, #0xff                    ADD R1, #0xff
MOV R2, #0x1                     ADD R1, #0xff
ADD R1, #0xff                    MOV R2, #0x1
MOV R2, #0x1                     MOV R2, #0x1


Hamming-Distanz = 30             Hamming-Distanz = 10
Prozessorstrom  = 43,8 mA        Prozessorstrom  = 42,6 mA

        (a)                              (b)
```

Wie Abbildung 6.8 zeigt, ist die Hamming-Distanz '10' für die beiden Instruktionen ADD und MOV. Die Anzahl der gekippten Bits auf dem Datenbus bei einem Instruction-Fetch entspricht für die vollständige Sequenz (a) 30 Bits, im Gegensatz zu 10 Bits für Sequenz (b). Durch die Reduzierung der Hamming-Distanz in Sequenz (a) wird die Stromaufnahme des Prozessors um 1,2 mA gesenkt.

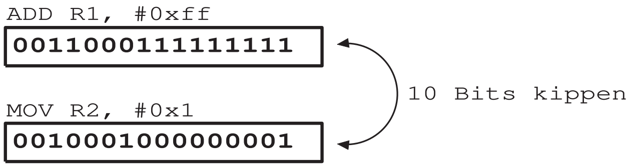

Abbildung 6.8: Hamming-Distanz zweier Instruktionen

Optimierungen durch Reduzierungen der Hamming-Distanz zwischen zwei Instruktionen erzielen jedoch nicht für alle Instruktionstupel den gewünschten

Effekt einer reduzierten Stromaufnahme des Prozessors. Bei einer zweiten Betrachtung der Inter-Instruction-Effekte für Tupel gebildet aus Instruktionen verschiedener Klassen in Tabelle 6.11 zeigt sich, dass trotz der geringen Hamming-Distanz von '2' für das Tupel 'MOV R1,R8' und 'ADD R1,[R2,R3]', der Inter-Instruction-Effekt 3,8 mA beträgt. Erhöhungen des Prozessorstroms durch den ständigen Wechsel von prozessorinternen Ressourcen bewirken diesen Anstieg und sind deshalb ebenso zu berücksichtigen.

6.2.7 Adressenwahl

Innerhalb des On-/Off-Chip-Speichers sind Instruktionen und Speicherzugriffe an unterschiedlichen Adressen plaziert worden. Für den On-Chip-Speicher zeigt dieses Vorgehen keine signifikanten Auswirkungen auf dem Prozessorstrom. Für Datenzugriffe auf dem Off-Chip-Speicher sind Stromunterschiede bei Verwendung unterschiedlicher Adressen festgestellt worden. Tabelle 6.13 zeigt dazu gemessene Werte einer Load-Instruktion bei einem 32-Bit-Datenzugriff. Bei steigender Anzahl von Einsen auf dem Adressbus erhöht sich die Stromaufnahme des Prozessors.

Speicheradresse des Datums	Anzahl der Einsen innerhalb der Adresse	Load-Word-Zugriff (mA)
0x02050000	3	43,1
0x02050002	4	43,5
0x02050006	5	43,6
0x02052220	6	45,2
0x020522F0	9	45,5
0x0205FFF0	15	49,4

Tabelle 6.13: unterschiedliche Adressen für Speicherzugriffe

Dieser Effekt kann für eine Stromoptimierung dadurch genutzt werden, dass für die Positionierung von Instruktionen-, Stackpointer- und Daten möglichst Adressen mit wenig Einsen verwendet werden. Dazu ein kurzes Programm-Beispiel, wo nur durch die Neupositionierung der Stackpointeradresse der Prozessorstrom um 2,1 mA abgesenkt wird.

```
Loop    ADD r1,SP,#32
        MOV r0,#0
LL1     LSL r4,r0,#2
        MOV r3,#7
        STR r3, [r1, r4]
        ADD r0,r0,#1
        CMP r0,#20
        BLT LL1
        B   Loop

Prozessorstrom  = 46,7 mA    (Stackpointer-Adresse: 0x0203FF00)
Prozessorstrom  = 44,6 mA    (Stackpointer-Adresse: 0x02030000)
```

Das Beispiel entspricht einem häufig angesprochenen Basisblock im Kernelbenchmark 'biquad_N_sections' aus dem DSPstone-Projekt [ZIV94].

6.2.8 Registerwahl

Für alle Instruktionen ist bei Wahl unterschiedlicher Register keine Veränderung des Prozessorstroms festgestellt worden. Der Grund dafür ist das homogene Registerfile des ARM7TDMI-Prozessors. Es stellt nur Register mit einer Wortbreite von 32 Bit zur Verfügung.

6.2.9 Ressourcenzuordnung

Auf Basis der gemessenen Stromwerte für die einzelnen Instruktionen lassen sich Instruktionskosten zu bestimmten Prozessorressourcen zuordnen. Errechnete Durchschnittswerte sind in Tabelle 6.14 zu finden. Der Prozessor selbst hat dabei eine Stromaufnahme von 41,0 mA. Branch-Befehle werden der Funktionseinheit ALU zugeordnet, da für die Sprungberechnung hauptsächlich ALU-Operationen benötigt werden und sich die Stromwerte dieser Instruktionen denen von ALU-Instruktionen ähneln. Für den On-Chip-Speicher sind Speicherkombinationen von Instruktionen und Daten zu berücksichtigen. Die beiden Abbildungen 6.9 und 6.10 zeigen zusätzliche Stromkosten des Prozessors bei einer Neupositionierung von Instruktionen und Daten von Off- nach On-Chip.

Funktionseinheit	CPU (mA)	Instruktionen
REGISTERBANK	+0,8	MOV
ALU	+1,0	B, BCC, BL, BX, ADC, ADD, AND, ASR, BIC, CMN, CMP, EOR, MVN, NEG, OR, SBC, SUB, TST
MULTIPIER	+8,7	MUL
BARRELSHIFTER	+4,8	LSL, LSR, ROR
ADRESS-/DATENBUS LESEND	+6,2	LDMIA, LDR, LDRB, LDRH, LDRSB, LDRSH, POP
ADRESS-/DATENBUS SCHREIBEND	+9,6	STMIA, STR, STRB, STRH, PUSH
ON-CHIP SPEICHER	-1,3 bis +9,5	ALLE INSTRUKTIONEN

Tabelle 6.14: Zuordnung von Ressourcen

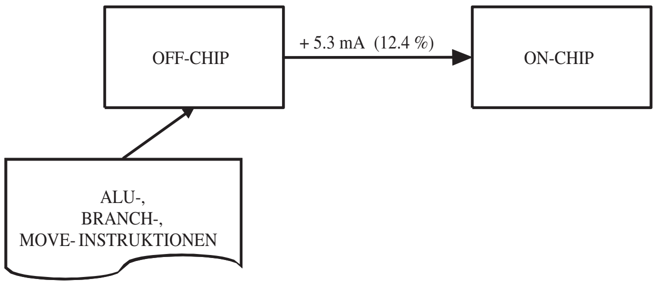

Abbildung 6.9: Verlegekosten für Registertransfer-, ALU- und Sprungbefehle von Off-Chip nach On-Chip

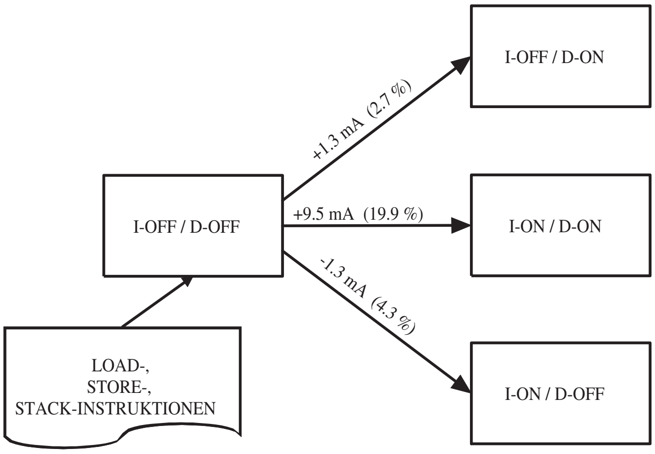

Abbildung 6.10: Verlegekosten für Speicherzugriffsbefehle von Off-Chip nach On-Chip

6.2.10 Zusammenfassung und Bewertung der Ergebnisse

Die durchschnittlich gemessenen Stromwerte beim ARM7TDMI bei Ausführung unterschiedlicher Thumb-Instruktionen variieren in einem Bereich von 38-58 mA. Eine weitere Anhebung des Prozessorstroms bis auf 65 mA wird durch den Einfluss verschiedener Seiteneffekte begünstigt, wie die folgende Aufstellung zeigt:

1. **Wahl der Instruktion**

   Abhängig von der gewählten Instruktion zeigen sich deutliche Unterschiede bei der Stromaufnahme des Prozessors. So haben Instruktionen die auschließlich in der ALU ausgewertet werden, einen geringeren Stromverbrauch, als Instruktionen die innerhalb des Barrelshifter oder im Multiplier berechnet werden. Load-, Store- und Stack-Operationen sind aufgrund ihrer zusätzlichen Verwendung des Daten- und Adressbusses am teuersten und sollten deshalb für leistungs- und energieoptimierte Programme so oft wie möglich vermieden werden.

2. **Datenabhängigkeiten**

   Variationen der Dateninhalte in den Operanden einer Instruktion können bis zu 15 Prozent Mehrkosten an Strom beim ARM7TDMI ausmachen. Dabei ist die Anzahl der Einsen entscheidend. Je mehr Einsen innerhalb der Operanden auftreten, umso höher ist der Prozessorstrom.

3. **Speicherwahl On-/Off-Chip**

   Liegen Registertransfer-, ALU- und Sprungbefehle auf dem internen On-Chip-Speicher des Prozessors, erhöht sich der Strom um durchschnittlich 12 Prozent im Vergleich zum Off-Chip-Speicher. Jedoch ist für eine Energiebetrachtung der On-Chip-Speicher für diese Befehle immer vorzuziehen, da keine zusätzlichen Kosten des externen Speichers anfallen.

Für Speicherzugriffsbefehle ist die Erhöhung des Prozessorstroms von der gewählten Speicherkombination für Instruktion und Daten abhängig. Die nachfolgende Auflistung der einzelnen Speicherkombinationen ist sortiert nach den günstigsten Leistungs- und Energiekosten für diese Befehle:

Leistungsoptimiert nach Kriterium: Prozessorstrom

(a) Instruktion ON-CHIP und Daten OFF-CHIP
(b) Instruktion OFF-CHIP und Daten ON-CHIP
(c) Instruktion OFF-CHIP und Daten OFF-CHIP
(d) Instruktion ON-CHIP und Daten ON-CHIP

Energieoptimiert nach Kriterium: Taktzyklen

(a) Instruktion ON-CHIP und Daten ON-CHIP
(b) Instruktion OFF-CHIP und Daten ON-CHIP
(c) Instruktion ON-CHIP und Daten OFF-CHIP
(d) Instruktion OFF-CHIP und Daten OFF-CHIP

Zusätzlich gilt, dass für Load- und Store-Instruktionen, die auf den externen Off-Chip-Speicher zugreifen, Halfword- und Byte-Zugriffe aus Leistungssicht teurer bewertet sind als Word-Zugriffe. Bei Zugriff auf den On-Chip-Speicher ist es genau entgegengesetzt. Ein Word-Zugriff ist hier die teuerste Operation. Entscheidend dabei ist die Anzahl der benötigten Taktzyklen. Eine höhere Taktzyklenzahl während der Befehlsverarbeitung reduziert den Prozessorstrom, da sich der nächste Instruction-Fetch verzögert. Am deutlichsten tritt dieser Effekt bei Multiple-Instruktionen auf. Für eine Energieoptimierung sollte jedoch immer die Instruktion mit geringster Anzahl an Taktzyklen ausgewählt werden, unabhängig davon, ob ihre Leistung am Prozessor höher ist als für vergleichbare Instruktionen.

4. **Inter-Instruction-Effekte**

   Der durchschnittliche Inter-Instruction-Effekt zwischen zwei Instruktionen, errechnet über alle Instruktionstupel, bei zusätzlicher Betrachtung der Hamming-Distanz, beträgt 2 mA oder umgerechnet 5 Prozent.

5. **Adressenwahl**

   Die Adressenwahl für Instruktion und Daten beeinflusst den Prozessorstrom um bis zu 6 mA. Ähnlich wie bei den Datenabhängigkeiten einer Instruktion ist die Anzahl der anliegenden Einsen auf dem Adressbus entscheidend. Bei einer Erhöhung der Einsen steigt der Prozessorstrom an.

6.3 Speicherstrom

Für die Betrachtung der Stromkosten des Speicherbausteins werden gemäß Energiemodell in einer ersten Messreihe die Kosten beim Laden einer Instruktion (Instruction-Fetch) betrachtet und in einer anschließenden zweiten Messreihe Kosten unterschiedlicher Datenzugriffe untersucht.

6.3.1 Kosten beim Laden einer Instruktion

Tabelle 6.15 zeigt durchschnittlich gemessene Stromwerte beim Instruction-Fetch von Registertransfer-, ALU- und Sprungbefehlen. Die Messergebnisse variieren dabei in einem Bereich von 58,5-60 mA.

Instruktion	OFF/OFF (mA)	CPU + Instr.Fetch-Zyklen
MOV HI TO LO	59,6	2
ADD WITH CARRY	59,1	2
SUB 3 BIT IMM	58,5	2
CMP LO AND LO	59,2	2
NEG LO AND LO	59,2	2
LOGICAL AND	59,5	2
BNE	60,0	4

Tabelle 6.15: Speicherstrom von ALU-, Branch-, Move-Befehlen beim Instruction-Fetch

Die geringfügigen Variationen im Stromverlauf einzelner Instruktionen werden am Beispiel einer Instruktion mit Immediate-Anteil deutlich. Die Tabelle 6.16 zeigt dazu gemessene Stromwerte einer Move-Anweisung bei Verwendung unterschiedlicher Immediate-Operanden. Die Spalte Opcode entspricht dabei der Umsetzung der Move-Instruktion in Binärdarstellung. Betrachtet man die Bitfolge beginnend von rechts, entsprechen die ersten 8 Bits dem Immediate-Anteil der Instruktion. Die übrigen Bits interpretieren den Befehl. Je mehr Einsen der Immediate-Operand aufweist, umso geringer ist der gemessene Strom des Speicherbausteins. Die maximale Differenz beträgt in diesem Fall 3,5 mA. Variationen im Stromverlauf beim Instruction-Fetch sind demnach von der Kodierung der Instruktion im Speicher abhängig. Eine Kodierung mit erhöhter Anzahl von Einsen im Befehlswort senkt den Speicherstrom ab. Dieses Verhalten ist entgegengesetzt zum Prozessorstrom.

Instr.	Opcode	Anzahl Einsen	OFF-CHIP (mA)
MOV R1, 0x0	00100001 00000000	2	60,5
MOV R1, 0xAA	00100001 01010101	6	58,5
MOV R1, 0XFF	00100001 11111111	10	57,0

Tabelle 6.16: Speicherstrom eines MOV-Immediate-Befehls beim Instruction-Fetch

Für alle mehrzyklischen Instruktionen mit Ausnahme von Branch-Befehlen reduziert sich der Speicherstrom für den Instruction-Fetch bei Erhöhung der Zyklenzahl. Tabelle 6.17 zeigt dazu gemessene Stromwerte. Für Speicherzugriffsbefehle sind die Daten zur Isolierung des Instruction-Fetch auf den On-Chip-Speicher gelegt worden. Der Grund für das Abfallen des Speicherstroms liegt bei mehrzyklischen Befehlen in der höheren Bearbeitungszeit des Prozessors, bevor der nächste Instruction-Fetch erfolgen kann. Das angeschlossene Messgerät am Speicherbaustein integriert den Speicherstrom in einem Zeitraum von 50 ms [COS00] zu einem Gesamtergebnis. Bei mehrzyklischen Befehlen dauert die Bearbeitung im Prozessor geringfügig länger. Die kurze Verzögerung

im Prozessor reicht aus, um einen Stromabfall am Speicherbaustein zu erzeugen. Diese Absenkung wird vom Messgerät mit integriert. Am Beispiel einer Push-Multiple-Instruktion, abgebildet in Tabelle 6.18, wird der Stromabfall bei steigender Zyklenzahl verdeutlicht.
Eine Ausnahme bei Betrachtung von mehrzyklischen Instruktionen tritt bei Branch-Befehlen auf. Der Strom am Speicherbaustein fällt beim Instruction-Fetch für diese Befehle nicht ab. Der Grund hierfür liegt in der Behandlungsweise von Branch-Befehlen innerhalb der Pipeline. Im ersten Zyklus erfolgt der eigentliche Instruction-Fetch, im zweiten Zyklus wird über eine Sprungvorhersageberechnung die nächste Instruktion des Sprungziels aus dem Speicher geladen. Im letzten Taktzyklus erfolgt das Refilling der Pipeline [ARM95b]. Der Speicher bleibt bei Bearbeitung dieser Instruktion bei jedem Taktzyklus im Zugriff.

Instruktion	OFF/ON (mA)	CPU + Instr.Fetch-Zyklen
LOGICAL SHIFT LEFT	48,9	4
MUL 32x8	49,3	4
LOAD WORD	37,8	4
STORE WORD	48,4	3
PUSH 1 REG	48,4	3
POP 1 REG	38,6	4

Tabelle 6.17: Speicherstrom von mehrzyklischen Befehlen beim Instruction-Fetch

Instruktion	OFF/ON (mA)	CPU + Instr.Fetch-Zyklen
PUSH 1 REG	49,9	3
PUSH 2 REG	38,4	4
PUSH 3 REG	30,9	5
PUSH 4 REG	23,1	6
PUSH 5 REG	17,8	7
PUSH 6 REG	14,3	8
PUSH 7 REG	11,5	9
PUSH 8 REG	9,5	10

Tabelle 6.18: Speicherstrom von Multiple-Befehlen beim Instruction-Fetch

6.3.2 Kosten für verschiedene Datenzugriffe

Zur Untersuchung unterschiedlicher Datenzugriffe werden alle Speicherzugriffsbefehle selbst auf den On-Chip-Speicher gelegt. Der Off-Chip-Speicher dient als Datenspeicher. Eine Isolierung des Datenzugriffs wird somit ermöglicht. In Tabelle 6.19 sind Messwerte von 32-Bit-, 16-Bit- und 8-Bit-Speicherzugriffen dargestellt. Es zeigt sich, dass Word-Zugriffe aufgrund ihrer höheren Zyklenzahl bezogen auf Energie- und Leistungskosten teurer sind als Halfword- oder Byte-Zugriffe. Das gleiche gilt für Multiple-Instruktionen, abgebildet in Tabelle 6.20. Für jedes zusätzliche Register, das abgelegt wird, steigt entgegengesetzt zum Instruction-Fetch der Strom an. Abbildung 6.11 visualisiert diesen Effekt um eine weitere Auffälligkeit festzustellen. Eine Steigerung der Registerzahl erwirkt keinen kontinuierlichen Anstieg des Speicherstroms. Bei Verwendung von drei Registern und mehr zeigt sich, dass die Stromkurve deutlich abflacht und ab fünf Registern fast gesättigt erscheint. Da der Anteil an Speicherzugriffen bezogen auf die Zyklenzahl ansteigt, nähert sich auch der Stromverbrauch des Speichers asymptotisch dem Maximimum.

Instruktion	ON/OFF (mA)	CPU + Daten-Zyklen
LOAD WORD	43,0	6
LOAD HALFWORD	34,3	4
LOAD BYTE	34,3	4
STORE WORD	42,1	5
STORE HALFWORD	34,9	3
STORE BYTE	34,9	3

Tabelle 6.19: Speicherstrom bei 32-Bit-, 16-Bit- und 8-Bit-Datenzugriffe

Instruktion	ON/OFF (mA)	CPU + Daten-Zyklen
PUSH 1 REG	43,8	5
PUSH 2 REG	47,0	9
PUSH 3 REG	48,2	13
PUSH 4 REG	48,9	17
PUSH 5 REG	49,5	21
PUSH 6 REG	49,8	25
LOAD 1 REG	46,1	6
LOAD 2 REG	52,4	10
LOAD 3 REG	55,1	14
LOAD 4 REG	56,4	18
LOAD 5 REG	57,7	22
LOAD 6 REG	58,5	26

Tabelle 6.20: Speicherstrom beim Datenzugriff von Multiple-Instruktionen

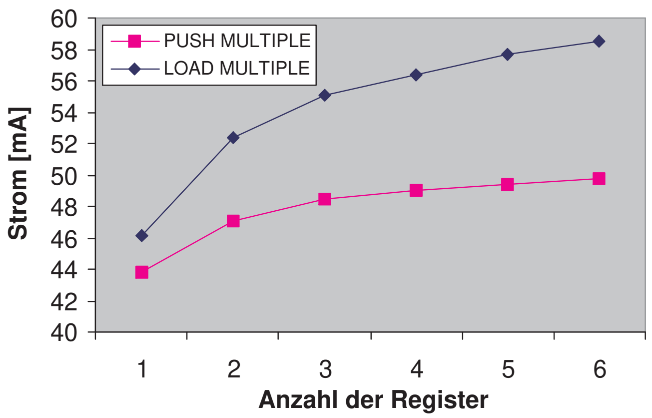

Abbildung 6.11: Stromverlauf beim Datenzugriff von Multiple-Instruktionen

6.3.3 Instruction-Fetch und Datenzugriff Off-Chip

Liegen verschiedene Speicherzugriffsbefehle und dazugehörige Daten auf dem externen Speicher, ist kein signifikanter Einfluss der Taktzyklenzahl auf den Speicherstrom festzustellen. Der Speicher steht bei dieser Speicherkombination im ständigen Zugriff des Prozessors, abwechselnd gefolgt von Instruction-Fetch und Datenzugriffen. Store-Instruktionen benötigen ca. 2 mA mehr an Strom, wie Tabelle 6.21 zeigt, jedoch aufgrund der unterschiedlichen Zyklenzahl unterschiedlich viel Energie.

Instruktion	OFF/OFF (mA)	CPU + Instr.Fetch + Daten-Zyklen
LOAD WORD	57,9	7
LOAD HALFWORD	57,2	5
LOAD BYTE	56,6	5
STORE WORD	59,6	6
STORE HALFWORD	59,5	4
STORE BYTE	59,5	4

Tabelle 6.21: Speicherstrom von 32-Bit-, 16-Bit-, 8-Bit-Datenzugriffe

6.3.4 Datenabhängigkeiten

Das Abfallen des Speicherstroms bei Erhöhung der Einsen im Instruktionswort zeigt sich nicht nur beim Instruction-Fetch, sondern auch bei Datenzugriffen auf Speicheradressen mit unterschiedlichen Dateninhalten. Abbildung 6.12 stellt dazu den Stromverlauf des Speichers bei Zugriff einer Load-Word-Instruktion auf unterschiedliche Dateninhalte dar. Das Abfallen des Speicherstroms bei Erhöhung der Einsen ist für alle Speicherzugriffsarten, insbesondere auch für Store- und Stack-Operationen, festgestellt worden.

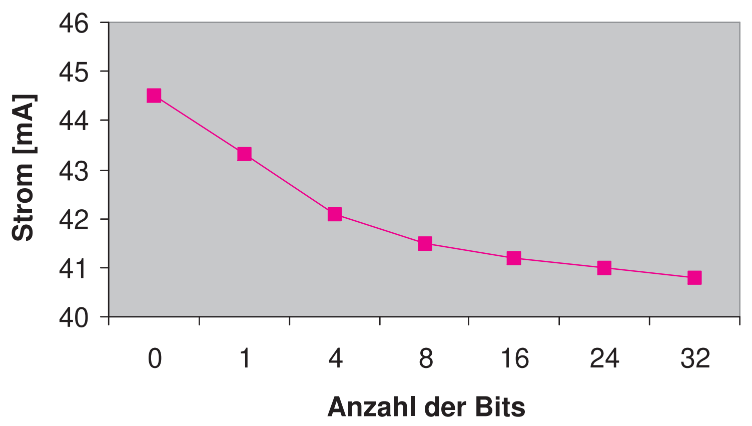

Abbildung 6.12: Stromverlauf einer LOAD-Word-Instruktion bei Variation der Speicherinhalte

6.3.5 Wahl des Adressbereichs

Variationen des Adressbereichs für Instruktionen, Daten und Stack zeigen keine signifikanten Stromschwankungen des Speichers beim Instruction-Fetch oder bei Datenzugriffen an.

6.3.6 Speicher im Idle-Zustand

Liegt eine Instruktion auf dem On-Chip-Speicher des Prozessors und erfolgt kein Datenzugriff auf den externen Speicher, wird dieser vom Prozessor nicht angesprochen. Der gemessene Strom des Speicherbausteins ohne Instruction-Fetch oder Datenzugriffe beträgt 0,58 mA.

6.3.7 Zusammenfassung und Bewertung der Ergebnisse

Folgende Ergebnisse können für den Speicher zusammengefasst werden:

1. **Kosten beim Instruction-Fetch**

   Für alle einzyklischen Instruktionen beträgt der gemessene Strom beim Instruction-Fetch 57-60,5 mA.

   Bei allen mehrzyklischen Instruktionen mit Ausnahme von Branch-Anweisungen reduziert sich der Speicherstrom für den Instruction-Fetch bei steigender Zyklenzahl. Beginnend bei durchschnittlich 48,5 mA für 2-Zyklenbefehle fällt der Strom bei 8-Zyklenbefehlen bis auf 9,7 mA ab. Eine erhöhte Bearbeitungszeit des Prozessors bei mehrzyklischen Instruktionen, die keine Datenzugriffe auf dem Speicher ausführen, senkt den Speicherstrom.

2. **Kosten bei Datenzugriffen**

   Bei Betrachtung unterschiedlicher Datenzugriffe zeigt sich, dass Word-Zugriffe aufgrund ihrer höheren Zyklenzahl mit 43 mA Stromkosten deutlich teurer sind als Halfword- oder Byte-Zugriffe, die durchschnittlich 34 mA an Stromkosten produzieren. Weiterhin sind Schreibzugriffe 2 mA teuerer als Lesezugriffe.

   Entgegengesetzt zum Instruction-Fetch steigt bei Datenzugriffen von Multiple-Instruktionen der Speicherstrom mit steigender Zyklenzahl an. Eine erhöhte Zugriffsfrequenz des Speichers ist der Grund des Anstiegs.

3. **Datenabhängigkeiten**

   Für den Instruction-Fetch und bei Datenzugriffen variiert der Strom für unterschiedlichen Speicherinhalte bis maximal 3,5 mA. Entgegengesetzt zum Prozessorstrom reduziert sich beim Speicher der Strom bei steigender Anzahl von Einsen.

Ein Optimierungsansatz zur Reduktion von Energiekosten ergibt sich bei Verwendung von Speicherzugriffsarten mit geringer Zyklenzahl. 32-Bit-Datenzugriffe sollten daher möglichst oft durch 16- oder 8-Bit-Zugriffe ersetzt werden, falls nur Daten mit einer Wortgröße kleiner als 16 Bit benötigt werden, da zwei Waitstate-Zyklen weniger anfallen. Zusätzlich senken 16- und 8-Bit-Zugriffe die Leistung des Speichers ab.

6.4 Genauigkeit der Ergebnisse

Die gemessenen Stromwerte werden zur Ergebnisvalidierung in zwei Programmsequenzen mit verschiedenen Thumb-Instruktionen ausgeführt. Die Sequenzen dürfen dabei nicht zu lang sein damit, damit das Messgerät gleichbleibende Werte anzeigt. Kleinere Befehlssequenzen mit maximal 20 Instruktionen erzeugen für diese Untersuchung gute Ergebnisse. Genau wie bei der Basiskostenbestimmung einer Instruktion wird die Sequenz in einer Schleife wiederholt. Die Validierung sieht vor, die angezeigten Messwerte des Amperemeters für Prozessor und Speicher mit den Daten der Energiekostentabelle im Anhang A zu vergleichen und anschließend eine Genauigkeitsberechnung durchzuführen. Diese Berechnung wird dabei nach demselben Verfahren durchgeführt wie für die Kostenberechnung von Inter-Instruction-Effekten. Jedoch sind in diesem Fall nicht zwei, sondern alle Instruktionen der Sequenz zu berücksichtigen. Die Differenz zwischen gemessenem und errechnetem Stromwert bestimmt in diesem Fall die Genauigkeit der Energiekostentabelle.
Für die Validierung werden folgende zwei Programmsequenzen verwendet:

1. Die erste Sequenz besteht aus verschiedenen Instruktionen, mit erhöhtem Anteil von ALU-Operationen.

2. Die zweite Sequenz besteht aus verschiedenen Instruktionen, die unterschiedliche Speicherzugriffe vornehmen. Dazu wurde eine typische Blockkopierroutine verwendet.

Die Tabellen 6.22 und 6.23 zeigen für beide Sequenzen abgelesene und berechnete Stromwerte für den Prozessor an. Für diese Berechung sind 2 mA Inter-Instruction-Kosten berücksichtigt worden. Die Angaben in den eckigen Klammern neben den Einzelstromwerten sind benötigte Taktzyklen der einzelnen Instruktionen. Die beiden Tabellen 6.24 und 6.25 zeigen die Ergebnisse der beiden Sequenzen für den Speicherstrom an.
Die durchschnittliche Abweichung von 1,7 Prozent für den Prozessorstrom und 2,8 Prozent für den Speicherstrom, ermittelt über alle Speicherkombinationen, zeigt die Genauigkeit des aufgestellten Energiemodells. Um eine noch höhere Präzision zu erreichen, ist es erforderlich, die Datenabhängigkeiten während des Programmlaufs zu berücksichtigen. Auf diese Berechnung ist jedoch aufgrund des hohen zeitlichen Aufwandes und der geringen Abweichung verzichtet worden.

Instr.-Sequenz	OFF/OFF (mA)	OFF/ON (mA)	ON/OFF (mA)	ON/ON (mA)
`Loop MOV r5, #255`	41,3 [2]	41,3 [2]	47,1 [1]	47,1 [1]
`LSL r5, r6,#2`	43,7 [2]	43,7 [2]	51,7 [1]	51,7 [1]
`MOV r9, r5`	42,4 [2]	42,4 [2]	49,1 [1]	49,1 [1]
`MOV r6, r9`	42,4 [2]	42,4 [2]	49,1 [1]	49,1 [1]
`ADD r7, r5, r6`	42,2 [2]	42,2 [2]	48,8 [1]	48,8 [1]
`STR r7, [r2,r3]`	49,1 [6]	51,3 [3]	45,5 [5]	56,0 [2]
`CMP r5, #8`	38,8 [2]	38,8 [2]	41,9 [1]	41,9 [1]
`BEQ stop`	41,7 [4]	41,7 [4]	46,7 [3]	46,7 [3]
`LDR r6, [r2,r3]`	44,2 [7]	47,8 [4]	42,1 [6]	50,2 [3]
`MUL r6, r5`	49,6 [3]	49,6 [3]	51,6 [2]	51,6 [2]
`ORR r7, r5`	42,4 [2]	42,4 [2]	49,3 [1]	49,3 [1]
`SUB r0, #1`	39,4 [2]	39,4 [2]	43,0 [1]	43,0 [1]
`CMP r0, #0`	38,8 [2]	38,8 [2]	41,9 [1]	41,9 [1]
`BNE Loop`	41,8 [4]	41,8 [4]	46,8 [3]	46,8 [3]
Summe Zyklen	42	36	28	22
Gemessen (Errechnet)	46,4 (45,5)	46,6 (45,7)	49,5 (47,9)	51,4 (50,5)
Abweichung	1,9 %	1,9 %	3,3 %	1,7 %

Tabelle 6.22: Prozessor: Sequenz 1

Instr.-Sequenz	OFF/OFF (mA)	OFF/ON (mA)	ON/OFF (mA)	ON/ON (mA)
`Loop MOV r2, #num`	41,3 [2]	41,3 [2]	47,1 [1]	47,1 [1]
`LDR r0, =src`	44,2 [7]	47,8 [4]	42,1 [6]	50,2 [3]
`LDR r1, =dst`	44,2 [7]	47,8 [4]	42,1 [6]	50,2 [3]
`LSR r3,r2,#2`	43,7 [2]	51,8 [2]	51,8 [1]	51,8 [1]
`PUSH {r4-r7}`	44,7 [18]	52,8 [6]	41,6 [17]	54,1 [5]
`Copy LDMIA r0!, {r4-r7}`	42,6 [19]	47,8 [7]	41,6 [18]	50,0 [6]
`STMIA r1!, {r4-r7}`	43,1 [18]	51,2 [6]	45,2 [17]	52,2 [5]
`SUB r3, #1`	39,4 [2]	39,4 [2]	43,0 [1]	43,0 [1]
`BNE Copy`	41,8 [4]	41,8 [4]	46,8 [3]	46,8 [3]
`POP, {r4-r7}`	46,0 [19]	50,5 [7]	43,3 [18]	50,6 [6]
`BEQ Loop`	41,7 [4]	41,7 [4]	46,7 [3]	46,7 [3]
Summe Zyklen	274	124	247	97
Gemessen (Errechnet)	44,5 (45,0)	48,4 (49,1)	45,1 (45,4)	50.9 (51,8)
Abweichung	1,1 %	1,4 %	0,6 %	1,7 %

Tabelle 6.23: Prozessor: Sequenz 2

Instr.-Sequenz	OFF/OFF (mA)	OFF/ON (mA)	ON/OFF (mA)	ON/ON (mA)
`Loop MOV r5, #255`	59,6	59,6	0,58	0,58
`LSL r5, r6,#2`	58,8	58,8	0,58	0,58
`MOV r9, r5`	59,6	59,6	0,58	0,58
`MOV r6, r9`	59,6	59,6	0,58	0,58
`ADD r7, r5, r6`	58,7	58,7	0,58	0,58
`STR r7, [r2,r3]`	56,6	48,5	42,0	0,58
`CMP r5, #8`	58,7	58,7	0,58	0,58
`BEQ stop`	60,0	60,0	0,58	0,58
`LDR r6, [r2,r3]`	57,9	37,5	42,1	0,58
`MUL r6, r5`	49,3	49,3	0,58	0,58
`ORR r7, r5`	59,5	59,5	0,58	0,58
`SUB r0, #1`	59,1	59,1	0,58	0,58
`CMP r0, #0`	58,7	58,7	0,58	0,58
`BNE Loop`	60,1	60,1	0,58	0,58
Gemessen (Errechnet)	59,0 (58,0)	56,6 (55,2)	17,5 (16,9)	0,58 (0,58)
Abweichung	1,7 %	2,5 %	3,5 %	0 %

Tabelle 6.24: Speicher: Sequenz 1

Instr.-Sequenz	OFF/OFF (mA)	OFF/ON (mA)	ON/OFF (mA)	ON/ON (mA)
`Loop MOV r2, #num`	59,6	59,6	0,58	0,58
`LDR r0, =src`	57,9	37,5	42,1	0,58
`LDR r1, =dst`	57,9	37,5	42,1	0,58
`LSR r3,r2,#2`	58,9	58,9	0,58	0,58
`PUSH {r4-r7}`	53,0	23,1	48,9	0,58
`Copy LDMIA r0!, {r4-r7}`	54,2	21,1	56,4	0,58
`STMIA r1!, {r4-r7}`	53,1	26,6	49,3	0,58
`SUB r3, #1`	59,1	59,1	0,58	0,58
`BNE Copy`	60,0	60,0	0,58	0,58
`POP, {r4-r7}`	56,5	17,4	48,5	0,58
`BEQ Loop`	60,0	60,0	0,58	0,58
Gemessen (Errechnet)	56,8 (54,9)	36,5 (35,2)	45,3 (46,5)	0,58 (0,58)
Abweichung	3,4 %	3,6 %	2,6 %	0 %

Tabelle 6.25: Speicher: Sequenz 2

6.5 Energiekosten von Instruktionen und Programmen

Für die Betrachtung des Gesamtenergieverbrauchs einer Instruktion wird die Leistung des Prozessors und die des Speichers berücksichtigt. Die Leistung des Speichers ist mit dem Faktor 2 in die Energiebewertung eingeflossen, da für einen Datenzugriff immer zwei Speicherbausteine verwendet werden. Die Erklärung hierfür ist in Kapitel 3.3.3 nachzulesen.

6.5.1 Leistung vs. Energie

Zur Motivation dieses Abschnittes wird ein Vergleich zwischen Leistungs- und Energiekosten anhand einer Multiplikationsoperation durchgeführt. Wie Tabelle 6.26 zeigt, hat die MUL-Instruktion bei Betrachtung des Prozessorstroms einen sehr konstanten Verlauf, unabhängig von ihrer Datenwahl. Jedoch benötigt diese Instruktion abhängig von der Wahl der Operanden zwischen zwei und fünf Taktzyklen zur Ergebnisberechnung. Der entsprechende Energieverlauf für diese Instruktion ist in Abbildung 6.13 dargestellt. Dominieren bei zwei Taktzyklen die Energiekosten des Speichers im Verhältnis 3:1 zur Prozessorenenergie, erhöhen sich, bei fast gleichbleibenden Energiekosten des Speichers, die Kosten des Prozessors bei steigender Zyklenzahl kontinuierlich. Bei fünf Taktzyklen benötigt der Prozessor annähernd soviel Energie wie der Speicher, um das Ergebnis zu berechnen.

MULTIPLY	Operand 1	Operand 2	CPU (mA)	RAM (mA)	Zyklen
32x8	FFFFFFFF	FF	49,6	49,3	3
32x16	FFFFFFFF	FFFF	49,9	38,1	4
32x24	FFFFFFFF	FFFFFF	50,3	31,0	5
32x32	FFFFFFFF	FFFFFFFF	50,8	26,2	6

Tabelle 6.26: Prozessor- und Speicherstrom einer MUL-Instruktion bei Variation der Operanden

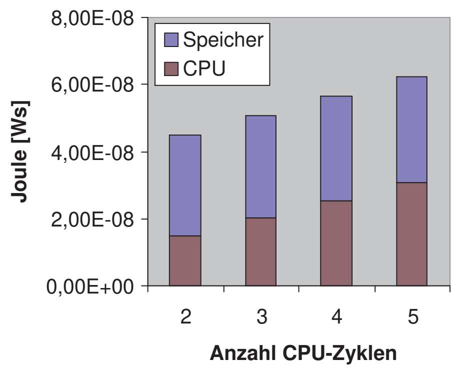

Abbildung 6.13: Energieverteilung einer MUL-Instruktion für Prozessor und Speicher

6.5.2 Prozessorenergie vs. Speicherenergie

Die Auswirkung einer Instruktion auf die Prozessor- und Speicherenergie ist in den beiden Abbildungen 6.15 und 6.16 dargestellt. Dabei liegen alle untersuchten Instruktionen und Daten auf dem externen Arbeitsspeicher. Bei Betrachtung der durchschnittlichen Energieverteilung im Kreisdiagramm 6.14 kann festgestellt werden, dass Kosten im Verhältnis 3:1 zu Lasten des Speichers anfallen. Innerhalb der einzelnen Instruktionen ist die Zyklenzahl entscheidend für den Kostenverlauf. Speicherzugriffsbefehle zeigen nicht nur die höchsten Leistungswerte für den Prozessor an, sondern erzeugen zusätzlich durch ihre hohe Anzahl an Taktzyklen die höchsten Energiekosten.

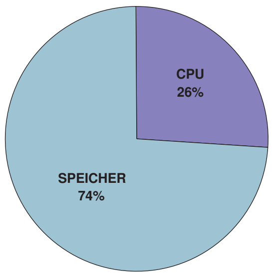

Abbildung 6.14: duchschnittliche Energieverteilung Prozessor vs. Speicher

Die hohen Energiekosten des externen Arbeitsspeichers resultieren aus dem höheren Stromverbrauch gegenüber dem Prozessor und der Beobachtung, dass zwei Speicherbausteine bei einem Datenzugriff angesprochen werden.

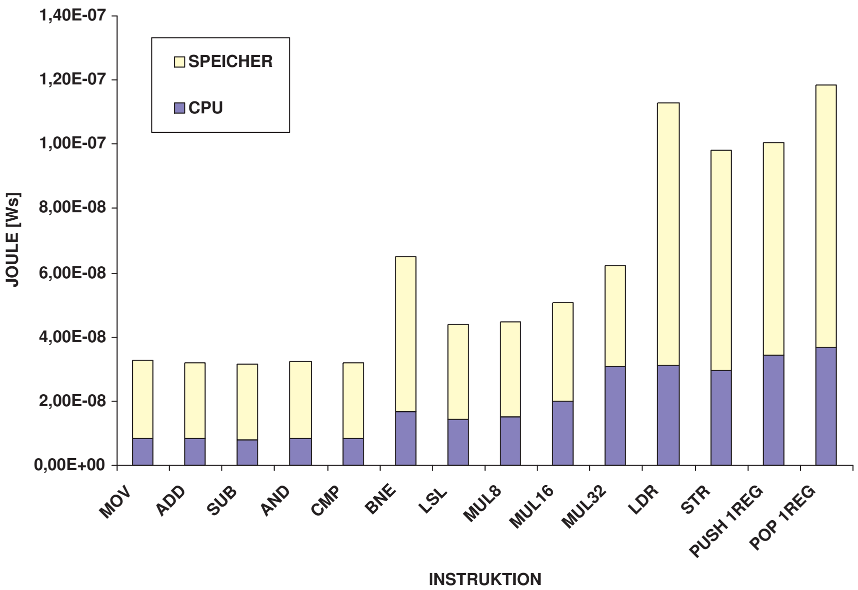

Abbildung 6.15: Gesamtenergie typischer Thumb-Instruktionen für Prozessor und Speicher

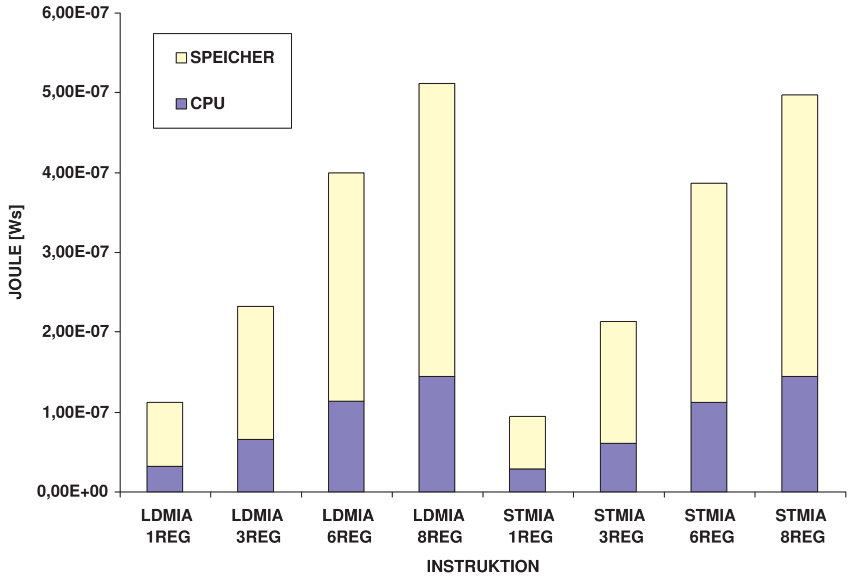

Abbildung 6.16: Gesamtenergie von Multiple-Instruktionen für Prozessor und Speicher

6.5.3 Off-Chip vs. On-Chip

Die Abbildungen 6.18 und 6.18 zeigen instruktionsabhängige Energiekosten bei Verwendung verschiedener Speicherbereiche für Instruktionen und Daten. Bei Betrachtung des Kreisdiagramms in Abbildung 6.17 zeigt sich, dass der Energiebedarf von ALU-, Registertransfer- und Sprungbefehlen um mehr als das sechsfache ansteigt, falls diese Instruktionen auf dem Off-Chip-Speicher anstatt auf dem On-Chip-Speicher liegen. Zwar ist die Prozessorleistung für Instruktionen auf dem On-Chip-Speicher im Durchschnitt um 10-15 Prozent höher als bei Verwendung des Off-Chip-Speichers, jedoch entfallen vollständig die hohen Energiekosten des externen Speichers, und zusätzlich reduziert sich die Zyklenzahl der einzelnen Instruktionen.

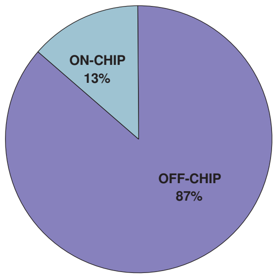

Abbildung 6.17: durchschnittliche Energieverteilung Off-Chip vs. On-Chip

Ein ähnliches Bild zeigt sich bei der Energiebetrachtung von Speicherzugriffsbefehlen. Auch hier dominieren die hohen Energiekosten bei Verwendung der Speicherkombination Off/Off für Instruktion und Daten. Der Worst-Case ist in diesem Fall eine Erhöhung um mehr als das siebenfache, im Gegensatz zur Plazierung von Instruktion und Daten auf dem On-Chip-Speicher. Da für größere Programme die Speicherkapazität des On-Chip-Speichers nicht ausreicht, sollte dieser Speicherbereich für die Datenhaltung oder zum Auslagern oft angesprochener Programmteile genutzt werden.

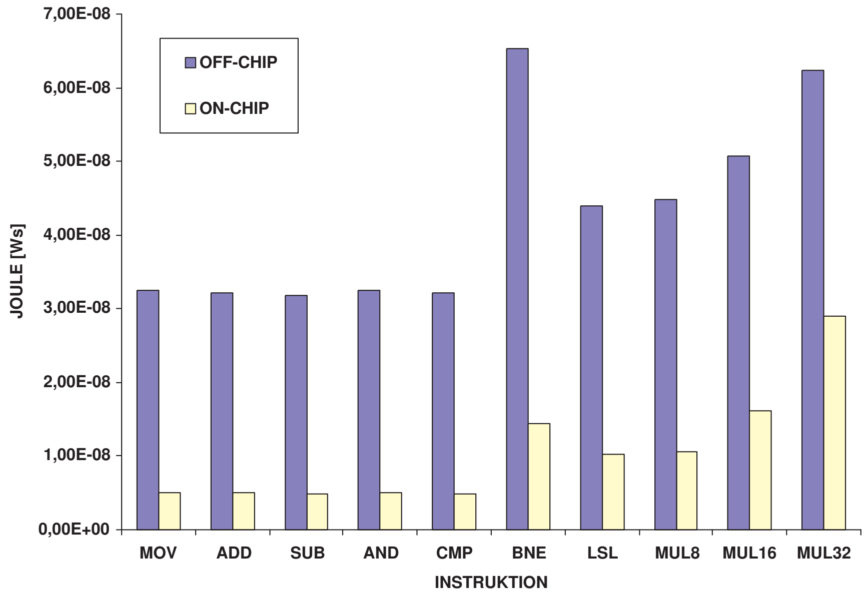

Abbildung 6.18: Gesamtenergie für ALU-, Registertransfer- und Sprungbefehle bei Verwendung des Off-Chip- oder On-Chip-Speichers

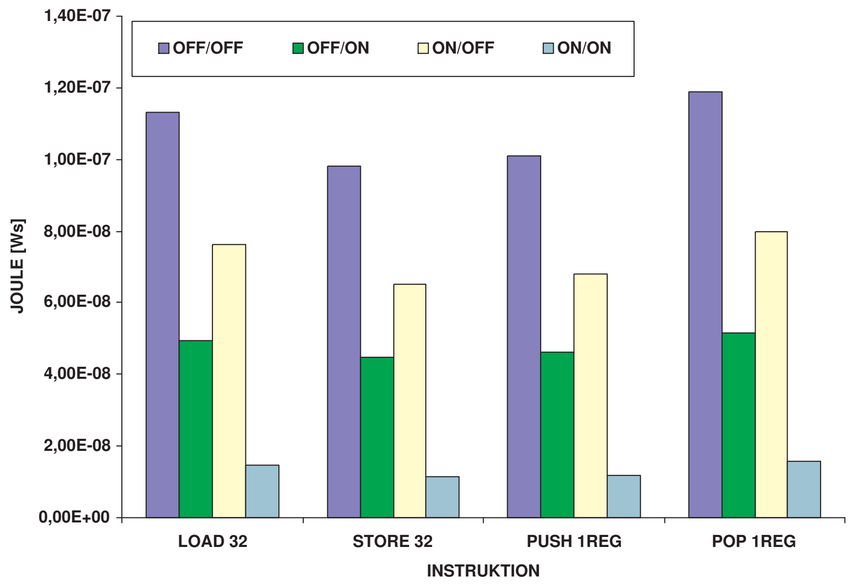

Abbildung 6.19: Gesamtenergie für Speicherzugriffsbefehle bei Verwendung des Off-Chip- oder On-Chip-Speichers

6.6 Energiekosten von Programmen

Mit Hilfe des Trace-Analyzers und den ermittelten Energiedaten für Prozessor und Speicher lässt sich für ein beliebig großes Programm der Energieverbrauch bestimmen. Zum Abschluss dieser Arbeit wird ein Energievergleich zwischen drei Sortierprogrammen durchgeführt, die jeweils 100, 200 und 400 Elemente sortieren. Die verwendeten Sortieralgorithmen sind dabei Bubble-Sort, Heap-Sort und Insertion-Sort. Abbildung 6.20 zeigt die benötigte Energie für Prozessor und Speicher dieser Sortierprogramme in Abhängigkeit der zu sortierenden Elemente. Programm-Code und Daten liegen dabei auf dem externen Speicher. Die benötigten Zyklen sind in Tabelle 6.27 dargestellt.

Sortierprogramm	Zyklen (100)	Zyklen (200)	Zyklen (400)
Bubble-Sort	246284	967092	3768056
Insertion-Sort	41344	481769	369709
Heap-Sort	24878	166699	124235

Tabelle 6.27: Zyklenzahlen verschiedener Sortieralgorithmen

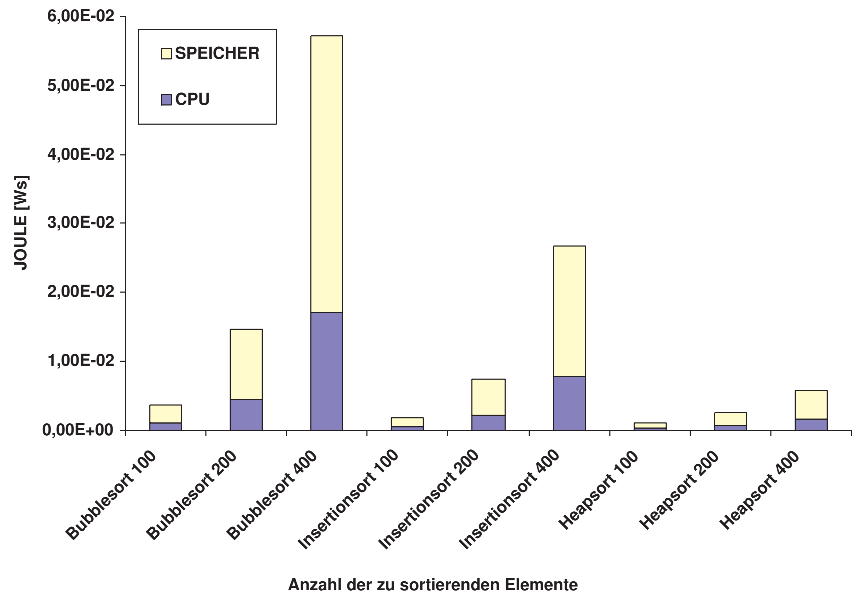

Abbildung 6.20: Energieverbrauch verschiedener Sortieralgorithmen

Die Ergebnisse des Trace-Analyzers zeigen, dass Bubble-Sort und Insertion-Sort durch ihre quadratischen Laufzeiteigenschaften eine hohe Zyklenzahl und somit auch einen hohen Energieverbrauch gegenüber Heap-Sort aufweisen. Die Energieverteilung zwischen Prozessor und Hauptspeicher liegt wie bei der Betrachtung einzelner Instruktionen im Durchschnitt bei 1:3 zu Lasten des externen Speichers. Auch dieses Beispiel macht deutlich, dass die Energiekosten des Speichers dominieren und Ansatzpunkt jeder Energieoptimierung sein sollten.

Kapitel 7

Zusammenfassung und Ausblick

In diesem letzten Kapitel erfolgt eine Zusammenfassung der erreichten Ergebnisse und anschließend ein Ausblick für zukünftige Forschungsmöglichkeiten auf dem Gebiet der Messung von Prozessor-Instruktionen.

7.1 Zusammenfassung

In dieser Diplomarbeit wurde der Energieverbrauch des Thumb-Instruktionssatzes für den ARM7TDMI-Prozessor untersucht. Dazu wurde im Vorfeld ein Energiemodell aufgestellt, das alle Kosten des Prozessors mit dazugehörigem Arbeitspeicher berücksichtigt. Unter Zuhilfenahme einer Versuchsanordnung und einer implementierten Messsoftware sind anschließend Strommessungen durchgeführt worden. Die Messergebnisse haben bestätigt, dass in Abhängigkeit von der gewählten Instruktion sowohl die Leistung als auch die Energie von Prozessor und Speicher unterschiedlich stark beeinflusst werden kann. Weiterhin wurde festgestellt, dass verschiedene Seiteneffekte wie Datenabhängigkeiten, Inter-Instruction-Effekte bei einem Instruktionswechsel sowie Wahl des Speicherbereichs für Programm-Code und Daten die Prozessor- und Speicherleistung und somit auch die Energie zusätzlich beeinflussen können.
Die Energiekosten des externen Arbeitspeichers dominieren dabei mit durchschnittlich 75 Prozent am betrachteten Gesamtsystem. Eine Vermeidung oder reduzierte Nutzung dieser Ressource sollte daher Gegenstand jeder Optimierung sein. Für eine detallierte Leistungs- und Energiebetrachtung einzelner Instruktionen werden die wichtigsten Ergebnisse dieser Arbeit zusammengetragen und daraus resultierende Optimierungsansätze dargelegt:

Leistungsbetrachtung

Für den Prozessor gilt, dass durch die Nutzung interner Prozessorressourcen wie Barrelshifter oder Multiplier die Grundleistung des Chips bei einem Programmlauf um bis zu 20 Prozent ansteigen kann. Für eine Leistungsoptimierung

sollten deswegen Instruktionen, die auf diese Ressourcen zugreifen, zu nennen sind dabei Shift-, Rotate- und Multiply-Operationen, möglichst oft vermieden werden. Gleiches gilt für alle Instruktionen, die Speicherzugriffe durchführen. Diese Anweisungen erwirken durch ihre Nutzung des Adress- und Datenbusses eine zusätzliche Erhöhung der Schaltkreisaktivität im Prozessor, die wiederum dafür verantwortlich ist, dass die Grundleistung des Prozessors um bis zu 50 Prozent ansteigt.
Für den externen Speicher gilt, dass nur durch eine Reduzierung der Datenzugriffe die Leistung sinnvoll abgesenkt werden kann. Weiterhin sollten notwendige Speicherzugriffe im stromgünstigeren 16- oder 8-Bit-Zugriffsmodus durchgeführt werden anstatt im teuren 32-Bit-Modus, falls nur Daten mit einer Wortgröße kleiner oder gleich 16 Bit während eines Programmlaufs benötigt werden.

Energiebetrachtung

Eine sinnvolle Optimierung der Energiekosten für Prozessor und Speicher wird bei Entscheidungsfreiheit durch die Verwendung von Instruktionen oder Instruktionssequenzen mit geringerer Anzahl an Taktzyklen erreicht. Lässt sich die Zyklenzahl nicht weiter absenken, sind Instruktionen auszuwählen, die eine geringere Leistung (siehe Leistungsbetrachtung) aufweisen. Bei der gewählten Ersetzungsstrategie darf jedoch nicht die Zyklenzahl ansteigen. Denn gerade die Zyklenzahl fließt als multiplikativer Wert in die Energieberechnung des Gesamtsystems mit ein. Wie in der Einleitung dieses Kapitels erwähnt worden ist, dominieren die Energiekosten des externen Speichers. Aus diesem Grund sollte der interne On-Chip-Speicher des Prozessors für Programm-Code und Daten häufig verwendet werden, um somit die Anzahl der Zugriffe auf den externen Speicher zu reduzieren. Gleichzeitig wird durch dieses Vorgehen die Zyklenzahl für die Vearbeitung der einzelnen Instruktionen durch das Vermeiden von Speicher-Waitstates gesenkt.

7.2 Ausblick

Im Rahmen dieser Diplomarbeit wurde für die Positionierung von Instruktionen und Daten der externe Hauptspeicher und der interne On-Chip-Speicher des Prozessors verwendet. Weitere Untersuchungsmöglichkeiten ergeben sich bei einer Kostenbetrachtung des Flash-Rom-Speichers. Dies erscheint sinnvoll, da die Energie- und Leistungskosten des Hauptspeichers im Gegensatz zum Prozessor hoch ausfallen. In diesem Zusammenhang ist auch die Implementierung verschiedener Algorithmen zu erwägen, um Programm-Code und Daten auf verschiedene Speicherbereiche energieoptimiert zu verteilen. Die Verteilungsstrategie sollte dabei die maximale Speicherkapazität, Waitstate-Zyklen und Energiekosten der einzelnen Speicherbereiche berücksichtigen.
Für Prozessor-Architekturen mit Cache-Speicher sind Untersuchungen der Energiekosten mit und ohne Cache-Miss-Effekte denkbar. Die verwendete Messmethodik in dieser Arbeit kann dazu als Anleitung dienen.

Anhang A

Gesamtergebnisse

Die Ergebnisse der durchgeführten Messreihen für die Basiskostenbestimmung einer Instruktion sind in diesem Anhang aufgelistet. Für jede untersuchte Speicherkombination von Instruktionen und Daten existieren zwei Tabellen. Es folgt eine Erklärung der verwendeten Spaltenkürzel:

INSTR: untersuchte Instruktion.

THUMB: entsprechender Thumb-Befehl.

LO: niedrigster Stromwert dieser Instruktion gemessen am Prozessor.

HI: höchster Stromwert dieser Instruktion gemessen am Prozessor.

AVG: durchschnittlicher Stromwert dieser Instruktion für den Prozessor.

CZ: Anzahl der benötigten CPU-Zyklen.

MEM: Stromwert dieser Instruktion gemessen am Speicher.

IF: Anzahl zusätzlicher Zyklen für den Instruktion-Fetch.

DF: Anzahl zusätzlicher Zyklen bei einem Datenzugriff.

SZ: Summe aller benötigten Zyklen.

POW (Ws): Energiekosten einer Instruktion für Prozessor und Speicher.

Folgende Speicherkombinationen entsprechen den abgebildeten Tabellen auf den nächsten Seiten:

OFF/OFF: Tabelle A.1 und Tabelle A.2

OFF/ON: Tabelle A.3 und Tabelle A.4

ON/OFF: Tabelle A.5 und Tabelle A.6

ON/ON: Tabelle A.7 und Tabelle A.8

INSTR	THUMB	LO	HI	AVG	CZ	MEM	IF	DF	SZ	POW(Ws)
MOVE										
Move Imm	MOV Rd, #8Bit	40,7	41,9	41,3	1	58,7	1	0	2	3,20E-08
Move Hi to Lo	MOV Rd, Hs	40,7	44	42,4	1	59,6	1	0	2	3,25E-08
Move Lo to Hi	MOV Hd, Rs	40,7	44,1	42,4	1	59,6	1	0	2	3,25E-08
Move Hi to Hi	MOV Hd, Hs	40,7	44,1	42,4	1	59,6	1	0	2	3,25E-08
ALU ARITHMETIC										
Add 3Bit	ADD Rd, Rs, #3Bit	40,2	43,6	41,9	1	58,2	1	0	2	3,19E-08
Add Lo and Lo	ADD Rd, Rs, Rn	40,3	44,1	42,2	1	58,7	1	0	2	3,21E-08
Add Hi to Low	ADD Rd, Hs	40,2	42,9	41,6	1	59,5	1	0	2	3,23E-08
Add Lo to Hi	ADD Hd, Rs	40,8	42,8	41,8	1	59,6	1	0	2	3,24E-08
Add Hi to Hi	ADD Hd, Hs	40,2	43	41,6	1	59,1	1	0	2	3,22E-08
Add 8Bit Imm	ADD Rd, #8Bit	40,2	42,2	41,2	1	59,5	1	0	2	3,23E-08
Add Value to Sp 7Bit	ADD SP, #7Bit	40,9	41,3	41,1	1	59,9	1	0	2	3,24E-08
Add Value to Sp 7Bit	ADD SP, #-7Bit	40,1	40,9	40,5	1	59,5	1	0	2	3,21E-08
Add with carry	ADC Rd, Rs	40,9	43,8	42,4	1	59,1	1	0	2	3,23E-08
Subtract	SUB Rd, Rs, Rn	37,8	44,4	41,1	1	58,5	1	0	2	3,18E-08
Subtract 3Bit Imm	SUB Rd, Rs, #3Bit	37,7	42,2	40	1	58,5	1	0	2	3,16E-08
Subtract 8Bit Imm	SUB Rd, #8Bit	37,7	41	39,4	1	58,5	1	0	2	3,15E-08
Subtract with carry	SBC Rd, Rs	37,7	42,9	40,3	1	59,2	1	0	2	3,20E-08
Negate	NEG Rd, Rs	40,9	44	42,5	1	59,2	1	0	2	3,24E-08
Multiply 32x8	MUL Rd, Rs	49,6	49,6	49,6	2	49,3	1	0	3	4,48E-08
Multiply 32x16	MUL Rd, Rs	49,9	49,9	49,9	3	38,1	1	0	4	5,08E-08
Multiply 32x24	MUL Rd, Rs	50,3	50,3	50,3	4	31	1	0	5	5,65E-08
Multiply 32x32	MUL Rd, Rs	50,8	50,8	50,8	5	26,2	1	0	6	6,24E-08
Compare Lo and Lo	CMP Rd, Rs	37,7	44,3	41	1	59,2	1	0	2	3,21E-08
Compare Lo and Hi	CMP Rd, Hs	37,7	44,3	41	1	59,4	1	0	2	3,22E-08
Compare Hi and Lo	CMP Hd, Rs	37,7	44,3	41	1	59,5	1	0	2	3,22E-08
Compare Hi and Hi	CMP Hd, Hs	37,7	44,3	41	1	59,2	1	0	2	3,21E-08
Comare Negative	CMN Rd, Rs	40,1	43,8	42	1	58,7	1	0	2	3,21E-08
Compare Imm	CMP Rd, #8Bit	37,7	39,9	38,8	1	58,7	1	0	2	3,15E-08
ALU LOGICAL										
Logical AND	AND Rd, Rs	39,5	44,2	41,9	1	59,5	1	0	2	3,24E-08
Logical EOR	EOR Rd, Rs	40,8	43,3	42,1	1	59,1	1	0	2	3,23E-08
Logical OR	ORR Rd, Rs	40,7	44,1	42,4	1	59,5	1	0	2	3,25E-08
Logical Bit clear	BIC Rd, Rs	37,7	42	39,9	1	59,2	1	0	2	3,19E-08
Logical MOVE NOT	MVN Rd, Rs	42	43	42,5	1	58,6	1	0	2	3,22E-08
Logical Test Bits	TST Rd, Rs	39,5	44,1	41,8	1	59,5	1	0	2	3,24E-08
ALU SHIFT AND ROTATE										
Logical shift left 5 Bit Imm	LSL Rd, Rs, 5Bit	43,1	44,2	43,7	1	58,8	1	0	2	3,25E-08
Logical shift left	LSL Rd, Rs	46,5	48,4	47,5	2	48,9	1	0	3	4,39E-08
Logical shift right 5 Bit Imm	LSR Rd, Rs, 5Bit	42,9	44,5	43,7	1	58,9	1	0	2	3,25E-08
Logical shift right	LSR Rd, Rs	46,3	48,2	47,3	2	48,6	1	0	3	4,36E-08
Arithmetic shift right 5Bit Imm	ASR Rd, Rs, 5Bit	42,9	44	43,5	1	58,9	1	0	2	3,25E-08
Arithmetic shift right	ASR Rd, Rs	45,9	48,7	47,3	2	49,2	1	0	3	4,40E-08
Rotate right	ROR Rd, Rs	46,1	49,2	47,7	2	48,7	1	0	3	4,38E-08
BRANCH										
Branch if Z set	BEQ label	40,9	42,4	41,7	3	60	1	0	4	6,51E-08
Branch if Z clear	BNE label	41,1	42,5	41,8	3	60,1	1	0	4	6,53E-08
Branch if C set	BCS label	41,1	42,5	41,8	3	60	1	0	4	6,52E-08
Branch if C clear	BCC label	41,1	42,6	41,9	3	60	1	0	4	6,52E-08
Branch if N set	BMI label	40,9	42,5	41,7	3	60	1	0	4	6,51E-08
Branch if N clear	BPL label	40,9	42,5	41,7	3	60	1	0	4	6,51E-08
Branch if V set	BVS label	40,7	42,3	41,5	3	60	1	0	4	6,51E-08
Branch if V clear	BVC label	41,1	42,5	41,8	3	60	1	0	4	6,52E-08
Branch if C set and Z clear	BHI label	40,8	42,4	41,6	3	60	1	0	4	6,51E-08
Branch if C clear and Z set	BLS label	40,9	42,5	41,7	3	60	1	0	4	6,51E-08
Branch if N set and V set or if N clear and V clear	BGE label	40,9	42,5	41,7	3	60	1	0	4	6,51E-08
Branch If N set and V clear, or if N clear and V set	BLT label	41,1	42,5	41,8	3	60	1	0	4	6,52E-08
Branch if Z clear and N or V set or if Z clear and N or V clear	BGT label	40,5	42,2	41,4	3	60	1	0	4	6,50E-08
Branch if Z set or N set and V clear or N clear and V set	BLE label	40,7	42,2	41,5	3	60	1	0	4	6,50E-08
Branch unconditional	B label	41,3	44,5	42,9	3	61,2	1	0	4	6,66E-08
Branch long with link	BL label	41,7	42,8	42,3	4	60,9	1	0	5	8,26E-08

Tabelle A.1: INSTRUKTION: OFF-CHIP DATEN: OFF-CHIP

INSTR	THUMB	LO	HI	AVG	CZ	MEM	IF	DF	SZ	POW(Ws)
LOAD										
Load word with 7Bit offset	LDR Rd, [Rb, 7Bit]	43,2	45	44,1	3	57,7	1	3	7	1,12E-07
Load halfword with 6Bit offset	LDRH Rd, [Rb, 6Bit]	46,7	48,3	47,5	3	57,9	1	1	5	8,22E-08
Load byte with 5Bit offset	LDRB Rd, [Rb, 5Bit]	47,3	48,4	47,9	3	56,6	1	1	5	8,11E-08
Load word with Reg offset	LDR Rd, [Rb,Ro]	43,6	44,7	44,2	3	57,9	1	3	7	1,13E-07
Load halfword with Reg offset	LDRH Rd, [Rb,Ro]	47,6	48,1	47,9	3	57,2	1	1	5	8,17E-08
Load signed halfword with Reg offset	LDRSH Rd, [Rb,Ro]	47,8	48,4	48,1	3	57,2	1	1	5	8,18E-08
Load byte with Reg offset	LDRB Rd, [Rb,Ro]	48	48,3	48,2	3	57,2	1	1	5	8,19E-08
Load signed byte with Reg offset	LDRSB Rd, [Rb,Ro]	47,9	48,8	48,4	3	57,3	1	1	5	8,21E-08
Load PC-relative	LDR Rd, [PC,10Bit]	39,6	43,3	41,5	3	55,2	1	3	7	1,07E-07
Load SP-relative	LDR Rd, [SP,10Bit]	42,6	43,7	43,2	3	57,9	1	3	7	1,12E-07
Load Adress using PC	ADD Rd, PC, 10Bit	41,4	41,5	41,5	3	60,3	1	3	7	1,14E-07
Load Adress using SP	ADD Rd, SP, 10Bit	40,9	41,2	41,1	3	60,3	1	3	7	1,14E-07
LOAD MULTIPLE										
Load Multiple 1 Reg	LDMIA Rb!, {1 Reg}	44,3	46,9	45,6	3	56,7	1	3	7	1,12E-07
Load Multiple 2 Reg	LDMIA Rb!, {2 Reg}	43,3	44,2	43,8	4	56,5	1	6	11	1,74E-07
Load Multiple 3 Reg	LDMIA Rb!, {3 Reg}	42	44,2	43,1	5	55,4	1	9	15	2,32E-07
Load Multiple 4 Reg	LDMIA Rb!, {4 Reg}	41,8	43,4	42,6	6	54,2	1	12	19	2,89E-07
Load Multiple 5 Reg	LDMIA Rb!, {5 Reg}	41,3	43,3	42,3	7	53,6	1	15	23	3,46E-07
Load Multiple 6 Reg	LDMIA Rb!, {6 Reg}	40,9	43	42	8	52,5	1	18	27	4,00E-07
Load Multiple 7 Reg	LDMIA Rb!, {7 Reg}	40,7	42,7	41,7	9	52,2	1	21	31	4,56E-07
Load Multiple 8 Reg	LDMIA Rb!, {8 Reg}	40,1	42,4	41,3	10	51,9	1	24	35	5,11E-07
STORE										
Store word with 7Bit offset	STR Rd, [Rb, 7Bit]	44,4	54,7	49,6	2	56,3	1	3	6	9,80E-08
Store halfword with 6Bit offset	STRH Rd, [Rb, 6Bit]	48,8	64,5	56,7	2	59,5	1	1	4	7,08E-08
Store byte with 5Bit offset	STRB Rd, [Rb, 5Bit]	49	61,6	55,3	2	58,8	1	1	4	6,96E-08
Store word with Reg offset	STR Rd, [Rb, Ro]	45,2	53	49,1	2	56,6	1	3	6	9,81E-08
Store halfword with reg. offset	STRH Rd, [Rb, Ro]	50,7	60	55,4	2	59,6	1	1	4	7,03E-08
Store byte with Reg offset	STRB Rd, [Rb, Ro]	50,2	59,4	54,8	2	59,5	1	1	4	7,00E-08
Store SP-relative	STR Rd, [SP, 10Bit]	44,1	46,1	45,1	2	54,9	1	3	6	9,36E-08
STORE MULTIPLE										
Store multiple 1 Reg	STMIA Rb!, {1 Reg}	45	49,7	47,4	2	55,3	1	3	6	9,54E-08
Store multiple 2 Reg	STMIA Rb!, {2 Reg}	42,4	46,6	44,5	3	54,6	1	6	10	1,55E-07
Store multiple 3 Reg	STMIA Rb!, {3 Reg}	41,3	45,3	43,3	4	53,8	1	9	14	2,13E-07
Store multiple 4 Reg	STMIA Rb!, {4 Reg}	41	45,2	43,1	5	53,1	1	12	18	2,71E-07
Store multiple 5 Reg	STMIA Rb!, {5 Reg}	40,9	44,9	42,9	6	52,7	1	15	22	3,29E-07
Store multiple 6 Reg	STMIA Rb!, {6 Reg}	40,6	44,6	42,6	7	52,3	1	18	26	3,85E-07
Store multiple 7 Reg	STMIA Rb!, {7 Reg}	40,4	44,5	42,5	8	51,9	1	21	30	4,42E-07
Store multiple 8 Reg	STMIA Rb!, {8 Reg}	40,3	44,3	42,3	9	51,4	1	24	34	4,97E-07
STACK										
Push 1 Reg	PUSH {1 Reg}	52,8	61,3	57,1	2	54,7	1	3	6	1,01E-07
Push 2 Reg	PUSH {2 Reg}	47,2	52,6	49,9	3	54	1	6	10	1,59E-07
Push 3 Reg	PUSH {3 Reg}	45,2	49,7	47,5	4	53,5	1	9	14	2,18E-07
Push 4 Reg	PUSH {4 Reg}	43,6	45,8	44,7	5	53	1	12	18	2,73E-07
Push 5 Reg	PUSH {5 Reg}	42,9	44,1	43,5	6	52,6	1	15	22	3,29E-07
Push 6 Reg	PUSH {6 Reg}	42,2	43,7	43	7	52,2	1	18	26	3,86E-07
Push 7 Reg	PUSH {7 Reg}	41,4	42,7	42,1	8	51,8	1	21	30	4,40E-07
Push 8 Reg	PUSH {8 Reg}	40,7	41,8	41,3	9	51,4	1	24	34	4,93E-07
Push 1 Reg and LR	PUSH {1 Reg, LR}	47,2	53,6	50,4	3	53,8	1	6	10	1,59E-07
Push 2 Reg and LR	PUSH {2 Reg, LR}	45,1	49,9	47,5	4	53,4	1	9	14	2,18E-07
Push 3 Reg and LR	PUSH {3 Reg, LR}	43,5	45,8	44,7	5	53,2	1	12	18	2,74E-07
Push 4 Reg and LR	PUSH {4 Reg, LR}	42,9	44,3	43,6	6	52,6	1	15	22	3,30E-07
Push 5 Reg and LR	PUSH {5 Reg, LR}	42,4	43,8	43,1	7	52,1	1	18	26	3,86E-07
Push 6 Reg and LR	PUSH {6 Reg, LR}	41,3	42,7	42	8	51,8	1	21	30	4,40E-07
Push 7 Reg and LR	PUSH {7 Reg, LR}	40,9	41,8	41,4	9	51,7	1	24	34	4,96E-07
Push 8 Reg and LR	PUSH {8 Reg, LR}	40,2	40,9	40,6	10	51,3	1	27	38	5,48E-07
Pop 1 Reg	POP {1 Reg}	50,4	53,4	51,9	3	58,1	1	3	7	1,19E-07
Pop 2 Reg	POP {2 Reg}	46,2	50,2	48,2	4	57,5	1	6	11	1,81E-07
Pop 3 Reg	POP {3 Reg}	44,6	48,5	46,6	5	57,1	1	9	15	2,43E-07
Pop 4 Reg	POP {4 Reg}	44,1	47,9	46	6	56,5	1	12	19	3,04E-07
Pop 5 Reg	POP {5 Reg}	43,4	47	45,2	7	56,2	1	15	23	3,65E-07
Pop 6 Reg	POP {6 Reg}	43	46,6	44,8	8	55,4	1	18	27	4,23E-07
Pop 7 Reg	POP {7 Reg}	42,7	46,4	44,6	9	55,2	1	21	31	4,84E-07
Pop 8 Reg	POP {8 Reg}	42,5	46,1	44,3	10	54,8	1	24	35	5,42E-07

Tabelle A.2: INSTRUKTION: OFF-CHIP DATEN: OFF-CHIP

INSTR	THUMB	LO	HI	AVG	CZ	MEM	IF	DF	SZ	POW(Ws)
MOVE										
Move Imm	MOV Rd, #8Bit	40,7	41,9	41,3	1	58,7	1	0	2	3,20E-08
Move Hi to Lo	MOV Rd, Hs	40,7	44	42,4	1	59,6	1	0	2	3,25E-08
Move Lo to Hi	MOV Hd, Rs	40,7	44,1	42,4	1	59,6	1	0	2	3,25E-08
Move Hi to Hi	MOV Hd, Hs	40,7	44,1	42,4	1	59,6	1	0	2	3,25E-08
ALU ARITHMETIC										
Add 3Bit	ADD Rd, Rs, #3Bit	40,2	43,6	41,9	1	58,2	1	0	2	3,19E-08
Add Lo and Lo	ADD Rd, Rs, Rn	40,3	44,1	42,2	1	58,7	1	0	2	3,21E-08
Add Hi to Low	ADD Rd, Hs	40,2	42,9	41,6	1	59,5	1	0	2	3,23E-08
Add Lo to Hi	ADD Hd, Rs	40,8	42,8	41,8	1	59,6	1	0	2	3,24E-08
Add Hi to Hi	ADD Hd, Hs	40,2	43	41,6	1	59,1	1	0	2	3,22E-08
Add 8Bit Imm	ADD Rd, #8Bit	40,2	42,2	41,2	1	59,5	1	0	2	3,23E-08
Add Value to Sp 7Bit	ADD SP, #7Bit	40,9	41,3	41,1	1	59,9	1	0	2	3,24E-08
Add Value to Sp 7Bit	ADD SP, #-7Bit	40,1	40,9	40,5	1	59,9	1	0	2	3,23E-08
Add with carry	ADC Rd, Rs	40,9	43,8	42,4	1	59,1	1	0	2	3,23E-08
Subtract	SUB Rd, Rs, Rn	37,8	44,4	41,1	1	58,5	1	0	2	3,18E-08
Subtract 3Bit Imm	SUB Rd, Rs, #3Bit	37,7	42,2	40	1	58,5	1	0	2	3,16E-08
Subtract 8Bit Imm	SUB Rd, #8Bit	37,7	41	39,4	1	58,5	1	0	2	3,15E-08
Subtract with carry	SBC Rd, Rs	37,7	42,9	40,3	1	59,2	1	0	2	3,20E-08
Negate	NEG Rd, Rs	40,9	44	42,5	1	59,2	1	0	2	3,24E-08
Multiply 32x8	MUL Rd, Rs	49,6	49,6	49,6	2	49,3	1	0	3	4,48E-08
Multiply 32x16	MUL Rd, Rs	49,9	49,9	49,9	3	38,1	1	0	4	5,08E-08
Multiply 32x24	MUL Rd, Rs	50,3	50,3	50,3	4	31	1	0	5	5,65E-08
Multiply 32x32	MUL Rd, Rs	50,8	50,8	50,8	5	26,2	1	0	6	6,24E-08
Compare Lo and Lo	CMP Rd, Rs	37,7	44,3	41	1	59,2	1	0	2	3,21E-08
Compare Lo and Hi	CMP Rd, Hs	37,7	44,3	41	1	59,4	1	0	2	3,22E-08
Compare Hi and Lo	CMP Hd, Rs	37,7	44,3	41	1	59,5	1	0	2	3,22E-08
Compare Hi and Hi	CMP Hd, Hs	37,7	44,3	41	1	59,2	1	0	2	3,21E-08
Comare Negative	CMN Rd, Rs	40,1	43,8	42	1	58,7	1	0	2	3,21E-08
Compare Imm	CMP Rd, #8Bit	37,7	39,9	38,8	1	58,7	1	0	2	3,15E-08
ALU LOGICAL										
Logical AND	AND Rd, Rs	39,5	44,2	41,9	1	59,5	1	0	2	3,24E-08
Logical EOR	EOR Rd, Rs	40,8	43,3	42,1	1	59,1	1	0	2	3,23E-08
Logical OR	ORR Rd, Rs	40,7	44,1	42,4	1	59,5	1	0	2	3,25E-08
Logical Bit clear	BIC Rd, Rs	37,7	42	39,9	1	59,2	1	0	2	3,19E-08
Logical MOVE NOT	MVN Rd, Rs	42	43	42,5	1	58,6	1	0	2	3,22E-08
Logical Test Bits	TST Rd, Rs	39,5	44,1	41,8	1	59,5	1	0	2	3,24E-08
ALU SHIFT AND ROTATE										
Logical shift left 5 Bit Imm	LSL Rd, Rs, #5Bit	43,1	44,2	43,7	1	58,8	1	0	2	3,25E-08
Logical shift left	LSL Rd, Rs	46,5	48,4	47,5	2	48,9	1	0	3	4,39E-08
Logical shift right 5 Bit Imm	LSR Rd, Rs, #5Bit	42,9	44,5	43,7	1	58,9	1	0	2	3,25E-08
Logical shift right	LSR Rd, Rs	46,3	48,2	47,3	2	48,6	1	0	3	4,36E-08
Arithmetic shift right 5Bit Imm	ASR Rd, Rs, #5Bit	42,9	44	43,5	1	58,9	1	0	2	3,25E-08
Arithmetic shift right	ASR Rd, Rs	45,9	48,7	47,3	2	49,2	1	0	3	4,40E-08
Rotate right	ROR Rd, Rs	46,1	49,2	47,7	2	48,7	1	0	3	4,38E-08
BRANCH										
Branch if Z set	BEQ label	40,9	42,4	41,7	3	60	1	0	4	6,51E-08
Branch if Z clear	BNE label	41,1	42,5	41,8	3	60,1	1	0	4	6,53E-08
Branch if C set	BCS label	41,1	42,5	41,8	3	60	1	0	4	6,52E-08
Branch if C clear	BCC label	41,1	42,6	41,9	3	60	1	0	4	6,52E-08
Branch if N set	BMI label	40,9	42,5	41,7	3	60	1	0	4	6,51E-08
Branch if N clear	BPL label	40,9	42,5	41,7	3	60	1	0	4	6,51E-08
Branch if V set	BVS label	40,7	42,3	41,5	3	60	1	0	4	6,51E-08
Branch if V clear	BVC label	41,1	42,5	41,8	3	60	1	0	4	6,52E-08
Branch if C set and Z clear	BHI label	40,8	42,4	41,6	3	60	1	0	4	6,51E-08
Branch if C clear and Z set	BLS label	40,9	42,5	41,7	3	60	1	0	4	6,51E-08
Branch if N set and V set or if N clear and V clear	BGE label	40,9	42,5	41,7	3	60	1	0	4	6,51E-08
If N set and V clear, or if N clear and V set	BLT label	41,1	42,5	41,8	3	60	1	0	4	6,52E-08
Branch if Z clear and N or V set or if Z clear and N or V clear	BGT label	40,5	42,2	41,4	3	60	1	0	4	6,50E-08
Branch if Z set or N set and V clear or N clear and V set	BLE label	40,7	42,2	41,5	3	60	1	0	4	6,50E-08
Branch unconditional	B label	41,3	43	42,2	3	61,2	1	0	4	6,63E-08
Branch long with link	BL label	41,7	42,8	42,3	4	60,9	1	0	5	8,26E-08

Tabelle A.3: INSTRUKTION: OFF-CHIP DATEN: ON-CHIP

INSTR	THUMB	LO	HI	AVG	CZ	MEM	IF	DF	SZ	POW(Ws)
LOAD										
Load word with 7Bit offset	LDR Rd, [Rb, 7Bit]	47,1	48,9	48	3	37,8	1	0	4	4,98E-08
Load halfword with 6Bit offset	LDRH Rd, [Rb, 6Bit]	45,7	46,9	46,3	3	37,8	1	0	4	4,91E-08
Load byte with 5Bit offset	LDRB Rd, [Rb, 5Bit]	45,8	46,7	46,3	3	38,1	1	0	4	4,93E-08
Load word with Reg offset	LDR Rd, [Rb,Ro]	47,3	48,3	47,8	3	37,5	1	0	4	4,95E-08
Load halfword with Reg offset	LDRH Rd, [Rb,Ro]	46	46,5	46,3	3	37,5	1	0	4	4,88E-08
Load signed halfword with Reg offset	LDRSH Rd, [Rb,Ro]	46,3	47,2	46,8	3	37,5	1	0	4	4,90E-08
Load byte with Reg offset	LDRB Rd, [Rb,Ro]	46,1	46,4	46,3	3	37,5	1	0	4	4,88E-08
Load signed byte with Reg offset	LDRSB Rd, [Rb,Ro]	45,9	46,9	46,4	3	37,5	1	0	4	4,89E-08
Load PC-relative	LDR Rd, [PC,10Bit]	39,6	43,3	41,5	3	55,2	1	0	4	6,12E-08
Load SP-relative	LDR Rd, [SP,10Bit]	47,1	48,1	47,6	3	37	1	0	4	4,90E-08
Load Adress using PC	ADD Rd, PC, 10Bit	41,4	41,5	41,5	3	60,3	1	0	4	6,53E-08
Load Adress using SP	ADD Rd, SP, 10Bit	40,9	41,2	41,1	3	58,1	1	0	4	6,33E-08
LOAD MULTIPLE										
Load Multiple 1Reg	LDMIA Rb!, {1 Reg}	48,1	49,2	48,7	3	38,62	1	0	4	5,07E-08
Load Multiple 2Reg	LDMIA Rb!, {2 Reg}	47,6	49	48,3	4	31,31	1	0	5	5,59E-08
Load Multiple 3Reg	LDMIA Rb!, {3 Reg}	47,4	48,8	48,1	5	26,25	1	0	6	6,08E-08
Load Multiple 4 Reg	LDMIA Rb!, {4 Reg}	47,1	48,4	47,8	6	21,1	1	0	7	6,34E-08
Load Multiple 5 Reg	LDMIA Rb!, {5 Reg}	46,5	48,1	47,3	7	16,3	1	0	8	6,44E-08
Load Multiple 6 Reg	LDMIA Rb!, {6 Reg}	46,2	47,5	46,9	8	12,5	1	0	9	6,51E-08
Load Multiple 7 Reg	LDMIA Rb!, {7 Reg}	45,9	47,2	46,6	9	11,4	1	0	10	6,98E-08
Load Multiple 8 Reg	LDMIA Rb!, {8 Reg}	45,3	46,9	46,1	10	7,9	1	0	11	6,86E-08
STORE										
Store word with 7Bit offset	STR Rd, [Rb, 7Bit]	50	53,2	51,6	2	48,4	1	0	3	4,48E-08
Store halfword with 6Bit offset	STRH Rd, [Rb, 6Bit]	47,9	51,6	49,8	2	48,8	1	0	3	4,45E-08
Store byte with 5Bit offset	STRB Rd, [Rb, 5Bit]	48,4	50,9	49,7	2	49,2	1	0	3	4,47E-08
Store word with Reg offset	STR Rd, [Rb, Ro]	50,2	52,3	51,3	2	48,5	1	0	3	4,48E-08
Store halfword with reg. offset	STRH Rd, [Rb, Ro]	48,9	50,6	49,8	2	48,5	1	0	3	4,43E-08
Store byte with Reg offset	STRB Rd, [Rb, Ro]	49	50,7	49,9	2	48,5	1	0	3	4,44E-08
Store SP-relative	STR Rd, [SP, 10Bit]	51,1	53,5	52,3	2	50	1	0	3	4,60E-08
STORE MULTIPLE										
Store multiple 1 Reg	STMIA Rb!, {1 Reg}	51,2	54,2	52,7	2	49,6	1	0	3	4,59E-08
Store multiple 2 Reg	STMIA Rb!, {2 Reg}	50,6	53,8	52,2	3	38,1	1	0	4	5,17E-08
Store multiple 3 Reg	STMIA Rb!, {3 Reg}	50	53,2	51,6	4	30,8	1	0	5	5,70E-08
Store multiple 4 Reg	STMIA Rb!, {4 Reg}	49,7	52,6	51,2	5	26,2	1	0	6	6,26E-08
Store multiple 5 Reg	STMIA Rb!, {5 Reg}	49,3	52,1	50,7	6	22,1	1	0	7	6,69E-08
Store multiple 6 Reg	STMIA Rb!, {6 Reg}	48,8	51,6	50,2	7	19,3	1	0	8	7,15E-08
Store multiple 7 Reg	STMIA Rb!, {7 Reg}	48,4	51,4	49,9	8	16,9	1	0	9	7,59E-08
Store multiple 8 Reg	STMIA Rb!, {8 Reg}	48,3	51,2	49,8	9	14,5	1	0	10	7,93E-08
STACK										
Push 1 Reg	PUSH {1 Reg}	51,6	54,5	53,1	2	49,9	1	0	3	4,62E-08
Push 2 Reg	PUSH {2 Reg}	51,5	54,5	53	3	38,39	1	0	4	5,23E-08
Push 3 Reg	PUSH {3 Reg}	51,3	54,3	52,8	4	30,9	1	0	5	5,77E-08
Push 4 Reg	PUSH {4 Reg}	51,3	54,3	52,8	5	23,1	1	0	6	5,98E-08
Push 5 Reg	PUSH {5 Reg}	51,1	54,2	52,7	6	17,8	1	0	7	6,22E-08
Push 6 Reg	PUSH {6 Reg}	51,1	54,2	52,7	7	14,3	1	0	8	6,55E-08
Push 7 Reg	PUSH {7 Reg}	51,4	53,7	52,6	8	11,5	1	0	9	6,85E-08
Push 8 Reg	PUSH {8 Reg}	51,3	53,7	52,5	9	9,7	1	0	10	7,24E-08
Push 1 Reg and LR	PUSH {1 Reg, LR}	51,7	55,6	53,7	3	38,39	1	0	4	5,25E-08
Push 2 Reg and LR	PUSH {2 Reg, LR}	51,5	55,5	53,5	4	30,6	1	0	5	5,78E-08
Push 3 Reg and LR	PUSH {3 Reg, LR}	51,4	55,5	53,5	5	23,1	1	0	6	6,02E-08
Push 4 Reg and LR	PUSH {4 Reg, LR}	51,3	55,2	53,3	6	17,7	1	0	7	6,25E-08
Push 5 Reg and LR	PUSH {5 Reg, LR}	51,1	54,7	52,9	7	14,4	1	0	8	6,58E-08
Push 6 Reg and LR	PUSH {6 Reg, LR}	50,6	54,5	52,6	8	11,7	1	0	9	6,88E-08
Push 7 Reg and LR	PUSH {7 Reg, LR}	50,5	54,5	52,5	9	8,9	1	0	10	7,08E-08
Push 8 Reg and LR	PUSH {8 Reg, LR}	50,3	54,2	52,3	10	7,5	1	0	11	7,45E-08
Pop 1 Reg	POP {1 Reg}	49,1	53	51,1	3	38,6	1	0	4	5,17E-08
Pop 2 Reg	POP {2 Reg}	48,8	52,9	50,9	4	31,2	1	0	5	5,70E-08
Pop 3 Reg	POP {3 Reg}	48,7	52,7	50,7	5	23,3	1	0	6	5,88E-08
Pop 4 Reg	POP {4 Reg}	48,4	52,6	50,5	6	17,4	1	0	7	6,01E-08
Pop 5 Reg	POP {5 Reg}	48,3	52,6	50,5	7	14,1	1	0	8	6,34E-08
Pop 6 Reg	POP {6 Reg}	48	52,3	50,2	8	11,6	1	0	9	6,65E-08
Pop 7 Reg	POP {7 Reg}	47,8	52,1	50	9	9,6	1	0	10	6,96E-08
Pop 8 Reg	POP {8 Reg}	47,7	52	49,9	10	7,2	1	0	11	7,12E-08

Tabelle A.4: INSTRUKTION: OFF-CHIP DATEN: ON-CHIP

INSTR	THUMB	LO	HI	AVG	CZ	MEM	IF	DF	SZ	POW(Ws)
MOVE										
Move Imm	MOV Rd, #8Bit	45,9	48,2	47,1	1	0,6	0	0	1	4,86E-09
Move Hi to Lo	MOV Rd, Hs	45,7	52,5	49,1	1	0,6	0	0	1	5,07E-09
Move Lo to Hi	MOV Hd, Rs	45,8	52,5	49,2	1	0,6	0	0	1	5,07E-09
Move Hi to Hi	MOV Hd, Hs	45,6	52,4	49	1	0,6	0	0	1	5,06E-09
ALU ARITHMETIC										
Add 3Bit	ADD Rd, Rs, #3Bit	45	51,8	48,4	1	0,6	0	0	1	5,00E-09
Add Lo and Lo	ADD Rd, Rs, Rn	45,1	52,5	48,8	1	0,6	0	0	1	5,04E-09
Add Hi to Low	ADD Rd, Hs	45,3	49,8	47,6	1	0,6	0	0	1	4,91E-09
Add Lo to Hi	ADD Hd, Rs	45,1	50,3	47,7	1	0,6	0	0	1	4,92E-09
Add Hi to Hi	ADD Hd, Hs	45	49,8	47,4	1	0,6	0	0	1	4,89E-09
Add 8Bit Imm	ADD Rd, #8Bit	44,8	48,9	46,9	1	0,6	0	0	1	4,84E-09
Add Value to Sp 7Bit	ADD SP, #7Bit	46,2	47,1	46,7	1	0,6	0	0	1	4,82E-09
Add Value to Sp 7Bit	ADD SP, #-7Bit	44,6	46,2	45,4	1	0,6	0	0	1	4,69E-09
Add with carry	ADC Rd, Rs	46,5	52,5	49,5	1	0,6	0	0	1	5,11E-09
Subtract	SUB Rd, Rs, Rn	40	53,3	46,7	1	0,6	0	0	1	4,82E-09
Subtract 3Bit Imm	SUB Rd, Rs, #3Bit	39,9	49,1	44,5	1	0,6	0	0	1	4,60E-09
Subtract 8Bit Imm	SUB Rd, #8Bit	39,7	46,3	43	1	0,6	0	0	1	4,45E-09
Subtract with carry	SBC Rd, Rs	39,7	50,6	45,2	1	0,6	0	0	1	4,67E-09
Negate	NEG Rd, Rs	46,2	52,5	49,4	1	0,6	0	0	1	5,09E-09
Multiply 32x8	MUL Rd, Rs	51,6	51,6	51,6	2	0,6	0	0	2	1,06E-08
Multiply 32x16	MUL Rd, Rs	52,5	52,5	52,5	3	0,6	0	0	3	1,62E-08
Multiply 32x24	MUL Rd, Rs	54,8	54,8	54,8	4	0,6	0	0	4	2,26E-08
Multiply 32x32	MUL Rd, Rs	56,4	56,4	56,4	5	0,6	0	0	5	2,90E-08
Compare Lo and Lo	CMP Rd, Rs	39,7	53	46,4	1	0,6	0	0	1	4,79E-09
Compare Lo and Hi	CMP Rd, Hs	39,7	53	46,4	1	0,6	0	0	1	4,79E-09
Compare Hi and Lo	CMP Hd, Rs	39,7	53	46,4	1	0,6	0	0	1	4,79E-09
Compare Hi and Hi	CMP Hd, Hs	39,7	53	46,4	1	0,6	0	0	1	4,79E-09
Comare Negative	CMN Rd, Rs	44,9	52,3	48,6	1	0,6	0	0	1	5,02E-09
Compare Imm	CMP Rd, #8Bit	39,6	44,1	41,9	1	0,6	0	0	1	4,34E-09
ALU LOGICAL										
Logical AND	AND Rd, Rs	43,3	52,8	48,1	1	0,6	0	0	1	4,96E-09
Logical EOR	EOR Rd, Rs	45,9	51	48,5	1	0,6	0	0	1	5,00E-09
Logical OR	ORR Rd, Rs	45,9	52,7	49,3	1	0,6	0	0	1	5,09E-09
Logical Bit clear	BIC Rd, Rs	39,8	47,2	43,5	1	0,6	0	0	1	4,50E-09
Logical MOVE NOT	MVN Rd, Rs	48,4	49,6	49	1	0,6	0	0	1	5,06E-09
Logical Test Bits	TST Rd, Rs	43,4	52,6	48	1	0,6	0	0	1	4,95E-09
ALU SHIFT AND ROTATE										
Logical shift left 5 Bit Imm	LSL Rd, Rs, #5Bit	50,4	52,9	51,7	1	0,6	0	0	1	5,32E-09
Logical shift left	LSL Rd, Rs	48,5	51,6	50,1	2	0,6	0	0	2	1,03E-08
Logical shift right 5 Bit Imm	LSR Rd, Rs, #5Bit	50,2	53,3	51,8	1	0,6	0	0	1	5,33E-09
Logical shift right	LSR Rd, Rs	48,8	51,8	50,3	2	0,6	0	0	2	1,04E-08
Arithmetic shift right 5Bit Imm	ASR Rd, Rs, #5Bit	50,1	52,5	51,3	1	0,6	0	0	1	5,29E-09
Arithmetic shift right	ASR Rd, Rs	48,5	52,8	50,7	2	0,6	0	0	2	1,04E-08
Rotate right	ROR Rd, Rs	49	53,8	51,4	2	0,6	0	0	2	1,06E-08
BRANCH										
Branch if Z set	BEQ label	45,9	47,4	46,7	3	0,6	0	0	3	1,45E-08
Branch if Z clear	BNE label	46	47,5	46,8	3	0,6	0	0	3	1,45E-08
Branch if C set	BCS label	45,9	47,5	46,7	3	0,6	0	0	3	1,45E-08
Branch if C clear	BCC label	46	47,5	46,8	3	0,6	0	0	3	1,45E-08
Branch if N set	BMI label	45,7	47,3	46,5	3	0,6	0	0	3	1,44E-08
Branch if N clear	BPL label	45,9	47,3	46,6	3	0,6	0	0	3	1,44E-08
Branch if V set	BVS label	45,7	47,5	46,6	3	0,6	0	0	3	1,44E-08
Branch if V clear	BVC label	46,2	47,5	46,9	3	0,6	0	0	3	1,45E-08
Branch if C set and Z clear	BHI label	45,7	47,3	46,5	3	0,6	0	0	3	1,44E-08
Branch if C clear and Z set	BLS label	45,9	47,5	46,7	3	0,6	0	0	3	1,45E-08
Branch if N set and V set or if N clear and V clear	BGE label	46	47,5	46,8	3	0,6	0	0	3	1,45E-08
If N set and V clear, or if N clear and V set	BLT label	46	47,5	46,8	3	0,6	0	0	3	1,45E-08
Branch if Z clear and N or V set or if Z clear and N or V clear	BGT label	46	47,5	46,8	3	0,6	0	0	3	1,45E-08
Branch if Z set or N set and V clear or N clear and V set	BLE label	46,1	47,5	46,8	3	0,6	0	0	3	1,45E-08
Branch unconditional	B label	47,3	49,1	48,2	3	0,6	0	0	3	1,49E-08
Branch long with link	BL label	49,4	50	49,7	4	0,6	0	0	4	2,05E-08

Tabelle A.5: INSTRUKTION: ON-CHIP DATEN: OFF-CHIP

INSTR	THUMB	LO	HI	AVG	CZ	MEM	IF	DF	SZ	POW(Ws)
LOAD										
Load word with 7Bit offset	LDR Rd, [Rb, 7Bit]	41	42,7	41,9	3	42,1	0	3	6	7,62E-08
Load halfword with 6Bit offset	LDRH Rd, [Rb, 6Bit]	44,3	45,7	45	3	34,6	0	1	4	4,60E-08
Load byte with 5Bit offset	LDRB Rd, [Rb, 5Bit]	44,8	45,5	45,2	3	34,6	0	1	4	4,61E-08
Load word with Reg offset	LDR Rd, [Rb,Ro]	41,5	42,7	42,1	3	42,1	0	3	6	7,63E-08
Load halfword with Reg offset	LDRH Rd, [Rb,Ro]	44,8	45,4	45,1	3	34,6	0	1	4	4,60E-08
Load signed halfword with Reg offset	LDRSH Rd, [Rb,Ro]	45	45,9	45,5	3	34,6	0	1	4	4,62E-08
Load byte with Reg offset	LDRB Rd, [Rb,Ro]	45,2	45,5	45,4	3	34,6	0	1	4	4,61E-08
Load signed byte with Reg offset	LDRSB Rd, [Rb,Ro]	45,1	46,1	45,6	3	34,6	0	1	4	4,62E-08
Load PC-relative	LDR Rd, [PC,10Bit]	47,3	49,9	48,6	3	0,6	0	3	6	3,01E-08
Load SP-relative	LDR Rd, [SP,10Bit]	41	41,8	41,4	3	42,1	0	3	6	7,59E-08
Load Adress using PC	ADD Rd, PC, 10Bit	47,3	47,6	47,5	3	0,6	0	3	6	2,94E-08
Load Adress using SP	ADD Rd, SP, 10Bit	46,2	46,4	46,3	3	0,6	0	3	6	2,87E-08
LOAD MULTIPLE										
Load Multiple 1Reg	LDMIA Rb!, {1 Reg}	42,6	46,5	44,6	3	46,1	0	3	6	8,26E-08
Load Multiple 2Reg	LDMIA Rb!, {2 Reg}	41,2	44,5	42,9	4	52,4	0	6	10	1,49E-07
Load Multiple 3Reg	LDMIA Rb!, {3 Reg}	40,7	43,3	42	5	55,1	0	9	14	2,15E-07
Load Multiple 4 Reg	LDMIA Rb!, {4 Reg}	40,7	42,5	41,6	6	56,4	0	12	18	2,80E-07
Load Multiple 5 Reg	LDMIA Rb!, {5 Reg}	40,6	42,2	41,4	7	57,7	0	15	22	3,47E-07
Load Multiple 6 Reg	LDMIA Rb!, {6 Reg}	40,4	42	41,2	8	58,5	0	18	26	4,14E-07
Load Multiple 7 Reg	LDMIA Rb!, {7 Reg}	40,3	41,9	41,1	9	59,1	0	21	30	4,81E-07
Load Multiple 8 Reg	LDMIA Rb!, {8 Reg}	40,2	41,6	40,9	10	59,4	0	24	34	5,47E-07
STORE										
Store word with 7Bit offset	STR Rd, [Rb, 7Bit]	43,8	46,9	45,4	2	42	0	3	5	6,51E-08
Store halfword with 6Bit offset	STRH Rd, [Rb, 6Bit]	48,6	53,6	51,1	2	34,9	0	1	3	3,65E-08
Store byte with 5Bit offset	STRB Rd, [Rb, 5Bit]	48,5	52	50,3	2	34,9	0	1	3	3,63E-08
Store word with Reg offset	STR Rd, [Rb, Ro]	44,3	46,6	45,5	2	42	0	3	5	6,52E-08
Store halfword with reg. offset	STRH Rd, [Rb, Ro]	50,2	52,3	51,3	2	34,9	0	1	3	3,66E-08
Store byte with Reg offset	STRB Rd, [Rb, Ro]	49,5	51,5	50,5	2	34,9	0	1	3	3,63E-08
Store SP-relative	STR Rd, [SP, 10Bit]	43,5	45,1	44,3	2	42	0	3	5	6,46E-08
STORE MULTIPLE										
Store multiple 1 Reg	STMIA Rb!, {1 Reg}	45,8	48,7	47,3	2	44	0	3	5	6,81E-08
Store multiple 2 Reg	STMIA Rb!, {2 Reg}	45,2	47,9	46,6	3	47,3	0	6	9	1,28E-07
Store multiple 3 Reg	STMIA Rb!, {3 Reg}	44,2	47,4	45,8	4	48,7	0	9	13	1,87E-07
Store multiple 4 Reg	STMIA Rb!, {4 Reg}	43,6	46,7	45,2	5	49,3	0	12	17	2,46E-07
Store multiple 5 Reg	STMIA Rb!, {5 Reg}	43,2	46,2	44,7	6	49,8	0	15	21	3,05E-07
Store multiple 6 Reg	STMIA Rb!, {6 Reg}	42,6	45,7	44,2	7	50,2	0	18	25	3,64E-07
Store multiple 7 Reg	STMIA Rb!, {7 Reg}	42,4	45,5	44	8	50,4	0	21	29	4,23E-07
Store multiple 8 Reg	STMIA Rb!, {8 Reg}	42,2	45,4	43,8	9	50,6	0	24	33	4,82E-07
STACK										
Push 1 Reg	PUSH {1 Reg}	46	48,8	47,4	2	43,8	0	3	5	6,80E-08
Push 2 Reg	PUSH {2 Reg}	43,1	45,1	44,1	3	47	0	6	9	1,25E-07
Push 3 Reg	PUSH {3 Reg}	41,7	43,6	42,7	4	48,2	0	9	13	1,82E-07
Push 4 Reg	PUSH {4 Reg}	40,8	42,3	41,6	5	48,9	0	12	17	2,39E-07
Push 5 Reg	PUSH {5 Reg}	40,6	42	41,3	6	49,5	0	15	21	2,97E-07
Push 6 Reg	PUSH {6 Reg}	40,4	41,9	41,2	7	49,8	0	18	25	3,54E-07
Push 7 Reg	PUSH {7 Reg}	40,3	41,7	41	8	50,2	0	21	29	4,13E-07
Push 8 Reg	PUSH {8 Reg}	40,1	41,6	40,9	9	50,4	0	24	33	4,71E-07
Push 1 Reg and LR	PUSH {1 Reg, LR}	44,2	46,8	45,5	3	47	0	6	9	1,26E-07
Push 2 Reg and LR	PUSH {2 Reg, LR}	42,1	44,1	43,1	4	47,6	0	9	13	1,81E-07
Push 3 Reg and LR	PUSH {3 Reg, LR}	41,8	43,7	42,8	5	48,3	0	12	17	2,39E-07
Push 4 Reg and LR	PUSH {4 Reg, LR}	41,5	43,6	42,6	6	48,8	0	15	21	2,96E-07
Push 5 Reg and LR	PUSH {5 Reg, LR}	41,2	43,2	42,2	7	49,6	0	18	25	3,56E-07
Push 6 Reg and LR	PUSH {6 Reg, LR}	40,9	43	42	8	50,1	0	21	29	4,15E-07
Push 7 Reg and LR	PUSH {7 Reg, LR}	40,9	42,9	41,9	9	50,3	0	24	33	4,74E-07
Push 8 Reg and LR	PUSH {8 Reg, LR}	40,6	42,8	41,7	10	50,5	0	27	37	5,32E-07
Pop 1 Reg	POP {1 Reg}	43,9	45,8	44,9	3	43,8	0	3	6	8,00E-08
Pop 2 Reg	POP {2 Reg}	42,8	44,8	43,8	4	47,1	0	6	10	1,39E-07
Pop 3 Reg	POP {3 Reg}	42,5	44,4	43,5	5	47,8	0	9	14	1,96E-07
Pop 4 Reg	POP {4 Reg}	42,3	44,2	43,3	6	48,5	0	12	18	2,54E-07
Pop 5 Reg	POP {5 Reg}	42	44,1	43,1	7	49,4	0	15	22	3,14E-07
Pop 6 Reg	POP {6 Reg}	41,7	43,9	42,8	8	49,8	0	18	26	3,73E-07
Pop 7 Reg	POP {7 Reg}	41,3	43,6	42,5	9	50	0	21	30	4,30E-07
Pop 8 Reg	POP {8 Reg}	41,2	43,4	42,3	10	50,1	0	24	34	4,88E-07

Tabelle A.6: INSTRUKTION: ON-CHIP DATEN: OFF-CHIP

INSTR	THUMB	LO	HI	AVG	CZ	MEM	IF	DF	SZ	POW(Ws)
MOVE										
Move Imm	MOV Rd, #8Bit	45,9	48,2	47,1	1	0,6	0	0	1	4,86E-09
Move Hi to Lo	MOV Rd, Hs	45,7	52,5	49,1	1	0,6	0	0	1	5,07E-09
Move Lo to Hi	MOV HD, Rs	45,8	52,5	49,2	1	0,6	0	0	1	5,07E-09
Move Hi to Hi	MOV HD, Hs	45,6	52,4	49	1	0,6	0	0	1	5,06E-09
ALU ARITHMETIC										
Add 3Bit	ADD Rd, Rs, #3Bit	45	51,8	48,4	1	0,6	0	0	1	5,00E-09
Add Lo and Lo	ADD Rd, Rs, Rn	45,1	52,5	48,8	1	0,6	0	0	1	5,04E-09
Add Hi to Low	ADD Rd, Hs	45,3	49,8	47,6	1	0,6	0	0	1	4,91E-09
Add Lo to Hi	ADD Hd, Rs	45,1	50,3	47,7	1	0,6	0	0	1	4,92E-09
Add Hi to Hi	ADD Hd, Hs	45	49,8	47,4	1	0,6	0	0	1	4,89E-09
Add 8Bit Imm	ADD Rd, #8Bit	44,8	48,9	46,9	1	0,6	0	0	1	4,84E-09
Add Value to Sp 7Bit	ADD SP, #7Bit	46,2	47,1	46,7	1	0,6	0	0	1	4,82E-09
Add Value to Sp 7Bit	ADD SP, #-7Bit	44,6	46,2	45,4	1	0,6	0	0	1	4,69E-09
Add with carry	ADC Rd, Rs	46,5	52,5	49,5	1	0,6	0	0	1	5,11E-09
Subtract	SUB Rd, Rs, Rn	40	53,3	46,7	1	0,6	0	0	1	4,82E-09
Subtract 3Bit Imm	SUB Rd, Rs, #3Bit	39,9	49,1	44,5	1	0,6	0	0	1	4,60E-09
Subtract 8Bit Imm	SUB Rd, #8Bit	39,7	46,3	43	1	0,6	0	0	1	4,45E-09
Subtract with carry	SBC Rd, Rs	39,7	50,6	45,2	1	0,6	0	0	1	4,67E-09
Negate	NEG Rd, Rs	46,2	52,5	49,4	1	0,6	0	0	1	5,09E-09
Multiply 32x8	MUL Rd, Rs	51,6	51,6	51,6	2	0,6	0	0	2	1,06E-08
Multiply 32x16	MUL Rd, Rs	52,5	52,5	52,5	3	0,6	0	0	3	1,62E-08
Multiply 32x24	MUL Rd, Rs	54,8	54,8	54,8	4	0,6	0	0	4	2,26E-08
Multiply 32x32	MUL Rd, Rs	56,4	56,4	56,4	5	0,6	0	0	5	2,90E-08
Compare Lo and Lo	CMP Rd, Rs	39,7	53	46,4	1	0,6	0	0	1	4,79E-09
Compare Lo and Hi	CMP Rd, Hs	39,7	53	46,4	1	0,6	0	0	1	4,79E-09
Compare Hi and Lo	CMP Hd, Rs	39,7	53	46,4	1	0,6	0	0	1	4,79E-09
Compare Hi and Hi	CMP Hd, Hs	39,7	53	46,4	1	0,6	0	0	1	4,79E-09
Comare Negative	CMN Rd, Rs	44,9	52,3	48,6	1	0,6	0	0	1	5,02E-09
Compare Imm	CMP Rd, #8Bit	39,6	44,1	41,9	1	0,6	0	0	1	4,34E-09
ALU LOGICAL										
Logical AND	AND Rd, Rs	43,3	52,8	48,1	1	0,6	0	0	1	4,96E-09
Logical EOR	EOR Rd, Rs	45,9	51	48,5	1	0,6	0	0	1	5,00E-09
Logical OR	ORR Rd, Rs	45,9	52,7	49,3	1	0,6	0	0	1	5,09E-09
Logical Bit clear	BIC Rd, Rs	39,8	47,2	43,5	1	0,6	0	0	1	4,50E-09
Logical MOVE NOT	MVN Rd, Rs	48,4	49,6	49	1	0,6	0	0	1	5,06E-09
Logical Test Bits	TST Rd, Rs	43,4	52,6	48	1	0,6	0	0	1	4,95E-09
ALU SHIFT AND ROTATE										
Logical shift left 5 Bit Imm	LSL Rd, Rs, #5Bit	50,4	52,9	51,7	1	0,6	0	0	1	5,32E-09
Logical shift left	LSL Rd, Rs	48,5	51,6	50,1	2	0,6	0	0	2	1,03E-08
Logical shift right 5 Bit Imm	LSR Rd, Rs, #5Bit	50,2	53,3	51,8	1	0,6	0	0	1	5,33E-09
Logical shift right	LSR Rd, Rs	48,8	51,8	50,3	2	0,6	0	0	2	1,04E-08
Arithmetic shift right 5Bit Imm	ASR Rd, Rs, #5Bit	50,1	52,5	51,3	1	0,6	0	0	1	5,29E-09
Arithmetic shift right	ASR Rd, Rs	48,5	52,8	50,7	2	0,6	0	0	2	1,04E-08
Rotate right	ROR Rd, Rs	49	53,8	51,4	2	0,6	0	0	2	1,06E-08
BRANCH										
Branch if Z set	BEQ label	45,9	47,4	46,7	3	0,6	0	0	3	1,45E-08
Branch if Z clear	BNE label	46	47,5	46,8	3	0,6	0	0	3	1,45E-08
Branch if C set	BCS label	45,9	47,5	46,7	3	0,6	0	0	3	1,45E-08
Branch if C clear	BCC label	46	47,5	46,8	3	0,6	0	0	3	1,45E-08
Branch if N set	BMI label	45,7	47,3	46,5	3	0,6	0	0	3	1,44E-08
Branch if N clear	BPL label	45,9	47,3	46,6	3	0,6	0	0	3	1,44E-08
Branch if V set	BVS label	45,7	47,5	46,6	3	0,6	0	0	3	1,44E-08
Branch if V clear	BVC label	46,2	47,5	46,9	3	0,6	0	0	3	1,45E-08
Branch if C set and Z clear	BHI label	45,7	47,3	46,5	3	0,6	0	0	3	1,44E-08
Branch if C clear and Z set	BLS label	45,9	47,5	46,7	3	0,6	0	0	3	1,45E-08
Branch if N set and V set or if N clear and V clear	BGE label	46	47,5	46,8	3	0,6	0	0	3	1,45E-08
If N set and V clear, or if N clear and V set	BLT label	46	47,5	46,8	3	0,6	0	0	3	1,45E-08
Branch if Z clear and N or V set or if Z clear and N or V clear	BGT label	46	47,5	46,8	3	0,6	0	0	3	1,45E-08
Branch if Z set or N set and V clear or N clear and V set	BLE label	46,1	47,5	46,8	3	0,6	0	0	3	1,45E-08
Branch unconditional	B label	47,3	49,1	48,2	3	0,6	0	0	3	1,49E-08
Branch long with link	BL label	49,4	50	49,7	4	0,6	0	0	4	2,05E-08

Tabelle A.7: INSTRUKTION: ON-CHIP DATEN: ON-CHIP

INSTR	THUMB	LO	HI	AVG	CZ	MEM	IF	DF	SZ	POW(Ws)
LOAD										
Load word with 7Bit offset	LDR Rd, [Rb, 7Bit]	47,7	52,1	49,9	3	0,6	0	0	3	1,54E-08
Load halfword with 6Bit offset	LDRH Rd, [Rb, 6Bit]	45,8	48,6	47,2	3	0,6	0	0	3	1,46E-08
Load byte with 5Bit offset	LDRB Rd, [Rb, 5Bit]	45,9	48,1	47	3	0,6	0	0	3	1,46E-08
Load word with Reg offset	LDR Rd, [Rb,Ro]	48,3	52	50,2	3	0,6	0	0	3	1,55E-08
Load halfword with Reg offset	LDRH Rd, [Rb,Ro]	46,4	48,3	47,4	3	0,6	0	0	3	1,47E-08
Load signed halfword with Reg offset	LDRSH Rd, [Rb,Ro]	46,6	49	47,8	3	0,6	0	0	3	1,48E-08
Load byte with Reg offset	LDRB Rd, [Rb,Ro]	46,5	47,5	47	3	0,6	0	0	3	1,46E-08
Load signed byte with Reg offset	LDRSB Rd, [Rb,Ro]	46,3	48,2	47,3	3	0,6	0	0	3	1,46E-08
Load PC-relative	LDR Rd, [PC,10Bit]	47,3	49,9	48,6	3	0,6	0	0	3	1,50E-08
Load SP-relative	LDR Rd, [SP,10Bit]	46,3	46,6	46,5	3	0,6	0	0	3	1,44E-08
Load Adress using PC	ADD Rd, PC, 10Bit	47,3	47,6	47,5	3	0,6	0	0	3	1,47E-08
Load Adress using SP	ADD Rd, SP, 10Bit	46,2	46,4	46,3	3	0,6	0	0	3	1,44E-08
LOAD MULTIPLE										
Load Multiple 1Reg	LDMIA Rb!, {1 Reg}	48,1	53,5	50,8	3	0,6	0	0	3	1,57E-08
Load Multiple 2Reg	LDMIA Rb!, {2 Reg}	47,7	53,3	50,5	4	0,6	0	0	4	2,08E-08
Load Multiple 3Reg	LDMIA Rb!, {3 Reg}	47,5	53,1	50,3	5	0,6	0	0	5	2,59E-08
Load Multiple 4 Reg	LDMIA Rb!, {4 Reg}	47,1	52,8	50	6	0,6	0	0	6	3,09E-08
Load Multiple 5 Reg	LDMIA Rb!, {5 Reg}	47	52,8	49,9	7	0,6	0	0	7	3,60E-08
Load Multiple 6 Reg	LDMIA Rb!, {6 Reg}	46,9	52,7	49,8	8	0,6	0	0	8	4,11E-08
Load Multiple 7 Reg	LDMIA Rb!, {7 Reg}	46,8	52,7	49,8	9	0,6	0	0	9	4,62E-08
Load Multiple 8 Reg	LDMIA Rb!, {8 Reg}	46,4	52,6	49,5	10	0,6	0	0	10	5,11E-08
STORE										
Store word with 7Bit offset	STR Rd, [Rb, 7Bit]	52,8	57,9	55,4	2	0,6	0	0	2	1,14E-08
Store halfword with 6Bit offset	STRH Rd, [Rb, 6Bit]	49,8	55,7	52,8	2	0,6	0	0	2	1,09E-08
Store byte with 5Bit offset	STRB Rd, [Rb, 5Bit]	50,9	55,3	53,1	2	0,6	0	0	2	1,09E-08
Store word with Reg offset	STR Rd, [Rb, Ro]	54,1	57,9	56	2	0,6	0	0	2	1,15E-08
Store halfword with reg. offset	STRH Rd, [Rb, Ro]	52	55,1	53,6	2	0,6	0	0	2	1,10E-08
Store byte with Reg offset	STRB Rd, [Rb, Ro]	52,3	55,3	53,8	2	0,6	0	0	2	1,11E-08
Store SP-relative	STR Rd, [SP, 10Bit]	54,1	56,9	55,5	2	0,6	0	0	2	1,14E-08
STORE MULTIPLE										
Store multiple 1 Reg	STMIA Rb!, {1 Reg}	53	56,4	54,7	2	0,6	0	0	2	1,13E-08
Store multiple 2 Reg	STMIA Rb!, {2 Reg}	50,4	55,7	53,1	3	0,6	0	0	3	1,64E-08
Store multiple 3 Reg	STMIA Rb!, {3 Reg}	49,8	55	52,4	4	0,6	0	0	4	2,16E-08
Store multiple 4 Reg	STMIA Rb!, {4 Reg}	49,5	54,8	52,2	5	0,6	0	0	5	2,69E-08
Store multiple 5 Reg	STMIA Rb!, {5 Reg}	49,4	54,8	52,1	6	0,6	0	0	6	3,22E-08
Store multiple 6 Reg	STMIA Rb!, {6 Reg}	49,1	54,5	51,8	7	0,6	0	0	7	3,74E-08
Store multiple 7 Reg	STMIA Rb!, {7 Reg}	48,5	54,4	51,5	8	0,6	0	0	8	4,24E-08
Store multiple 8 Reg	STMIA Rb!, {8 Reg}	48,5	54,2	51,4	9	0,6	0	0	9	4,76E-08
STACK										
Push 1 Reg	PUSH {1 Reg}	55,2	59,7	57,5	2	0,6	0	0	2	1,18E-08
Push 2 Reg	PUSH {2 Reg}	54,1	57,9	56	3	0,6	0	0	3	1,73E-08
Push 3 Reg	PUSH {3 Reg}	53,2	55,5	54,4	4	0,6	0	0	4	2,24E-08
Push 4 Reg	PUSH {4 Reg}	53	55,2	54,1	5	0,6	0	0	5	2,78E-08
Push 5 Reg	PUSH {5 Reg}	52,7	54,9	53,8	6	0,6	0	0	6	3,32E-08
Push 6 Reg	PUSH {6 Reg}	52,5	54,7	53,6	7	0,6	0	0	7	3,86E-08
Push 7 Reg	PUSH {7 Reg}	52,1	54,4	53,3	8	0,6	0	0	8	4,39E-08
Push 8 Reg	PUSH {8 Reg}	51,9	54,1	53	9	0,6	0	0	9	4,91E-08
Push 1 Reg and LR	PUSH {1 Reg, LR}	53,9	59,2	56,6	3	0,6	0	0	3	1,74E-08
Push 2 Reg and LR	PUSH {2 Reg, LR}	53,5	58,2	55,9	4	0,6	0	0	4	2,30E-08
Push 3 Reg and LR	PUSH {3 Reg, LR}	52,6	57,7	55,2	5	0,6	0	0	5	2,84E-08
Push 4 Reg and LR	PUSH {4 Reg, LR}	52	57,1	54,6	6	0,6	0	0	6	3,37E-08
Push 5 Reg and LR	PUSH {5 Reg, LR}	51,5	56,5	54	7	0,6	0	0	7	3,89E-08
Push 6 Reg and LR	PUSH {6 Reg, LR}	51,2	56,1	53,7	8	0,6	0	0	8	4,42E-08
Push 7 Reg and LR	PUSH {7 Reg, LR}	51,1	55,7	53,4	9	0,6	0	0	9	4,95E-08
Push 8 Reg and LR	PUSH {8 Reg, LR}	50,9	55,2	53,1	10	0,6	0	0	10	5,46E-08
Pop 1 Reg	POP {1 Reg}	49,6	53,3	51,5	3	0,6	0	0	3	1,59E-08
Pop 2 Reg	POP {2 Reg}	49,2	53,2	51,2	4	0,6	0	0	4	2,11E-08
Pop 3 Reg	POP {3 Reg}	48,9	52,7	50,8	5	0,6	0	0	5	2,62E-08
Pop 4 Reg	POP {4 Reg}	48,5	52,6	50,6	6	0,6	0	0	6	3,13E-08
Pop 5 Reg	POP {5 Reg}	48,2	52,3	50,3	7	0,6	0	0	7	3,63E-08
Pop 6 Reg	POP {6 Reg}	48	52,2	50,1	8	0,6	0	0	8	4,13E-08
Pop 7 Reg	POP {7 Reg}	47,9	52,1	50	9	0,6	0	0	9	4,64E-08
Pop 8 Reg	POP {8 Reg}	47,9	52	50	10	0,6	0	0	10	5,15E-08

Tabelle A.8: INSTRUKTION: ON-CHIP DATEN: ON-CHIP

Die durchgeführten Messreihen für Inter-Instruction-Effekte sind abschließend in Tabelle A.9 und Tabelle A.10 dokumentiert.

Instruktion 1	Instruktion 2	Inter-Instruction-Effekt (mA)
SUB Rd,Rs,Rn	ADD Rd,Rs,Rn	0,7
ADD Rd,Rs,Rn	ADC Rd,Rs	0,2
AND Rd,Rs	OR Rd,Rs	0,4
LDR Rd,Rs,Rn	STR Rd,Rs,Rn	0,4
BNE label	BCS label	0,2
LSL Rd,Rs	LSR Rd,Rs	0,6
LDR Rd,Rs,Rn	STR Rd,Rs,Rn	0,4
LDMIA Rb!, {2 Reg}	STMIA Rb!, {4 Reg}	0,8
PUSH {1 Reg}	POP {1 Reg}	0,3

Tabelle A.9: Inter-Instruction-Effekte zwischen Befehlen gleicher Klasse

Instruktion 1	Instruktion 2	Inter-Instruction-Effekt (mA)
MOV Rd,Hs	ADD Rd,Rs,Rn	3,8
MOV Rd,Hs	MUL Rd,Rs	2,1
CMP Rd,Rs	BNE label	3,9
ADD Rd,Rs,Rn	POP {1 Reg}	2,7
SUB Rd,#8Bit	LSR Rd,Rs,#5Bit	3,3
SUB Rd,Rs,Rn	LDR Rd,Rs,Rn	2,2
LDMIA Rb!, {4 Reg}	MUL Rd,Rs	3,6
STR Rd,Rs,Rn	ADD Rd,Rs,Rn	1,9
STR Rd,Rs,Rn	MUL Rd,Rs	2,5

Tabelle A.10: Inter-Instruction-Effekte zwischen Befehlen unterschiedlicher Klassen

Literaturverzeichnis

[ARM95a] Advanced RISC Machines Ltd (ARM): *An Introduction to Thumb*, Version 2.0, März 1995.

[ARM95b] Advanced RISC Machines Ltd (ARM): *ARM7TDMI Data Sheet*, Dokument-Nr: ARM DDI 0029E, 1995.

[ARM96] Advanced RISC Machines Ltd (ARM): *ARM Architecture Reference Manual*, Dokument-Nr: ARM DDI 0100B, 1996.

[ARM98a] Advanced RISC Machines Ltd (ARM): *ARM Software developmet toolkit: Reference Guide*, Dokument-Nr: ARM DUI 0041C, 1998.

[ARM98b] Advanced RISC Machines Ltd (ARM): *ARM Software developmet toolkit: User Guide*, Dokument-Nr: ARM DUI 0040D, 1998.

[ATM98a] Atmel Corporation: *AT91 ARM Thumb Microcontrollers: AT91M40400 Datasheet*, Dokument-Nr: 0768, 1998.

[ATM98b] Atmel Corporation: *AT91EB01 Evaluation Board: User Manual*, Dokument-Nr: 1144, 1998.

[BRA93] J. Bradley: *Calculation of TMS320C5x Power Dissipation Application Report*, Texas Instruments, 1993.

[COS00] COSINUS Computermesstechnik GmbH: *Handmultimeter ESCORT-95: Datenblatt*, http://www.cosinus.de/html/datenblaetter.html, 2000.

[GEB98] C. Gebotys und R. Gebotys: *An Empirical Comparison of Algorithmic, Instruction, and Architectural Power Prediction Models for High Performance Embedded DSP Processors*, Intern. Symposium on Low Power Electronics and Design, 1998.

[IDT99] Integrated Device Technology, Inc. (IDT): *IDT71V124SA, CMOS Static Ram: Datasheet*, November 1999.

[MAR00] P. Marwedel: *Skript zur Vorlesung Rechnerarchitektur*, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, April 2000.

[NKH00] N. Chang, K. Kim und H. Lee: *Cycle-Accurate Energy Consumption Measurement and Analysis: Case Study of ARM7TDMI*, in Proceedings of ACM/IEEE International Symposium on Low Power Electronics and Design (ISPLED00), July 2000.

[PRE85] R. Pregla: *Grundlagen der Elektrotechnik: Felder und Gleichstromnetzwerke*, Hüthig Verlag, Heidelberg, 1985.

[RUS98] J. Russel: *Assembly Code Power Analysis of a high performance processor family*, University of Texas, 1998.

[SIT99] G. Sinevriotis und T. Stouraitis: *Power Analysis of the ARM7 Embedded Microprocessor*, Proc. 9th Int. Workshop Power and Timing Modeling, Optimization and Simulation (PATMOS), Oct. 6-8, 1999.

[SS00] S. Steinke und R. Schwarz: *Low Power Code Generation for a RISC Processor by Register Pipelining*, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, August 2000.

[SW99] S. Steinke und L. Wehmeyer: *Low Power Code Generierung*, Universität Dortmund, Fakultät Informatik, Lehrstuhl 12, Technische Informatik, Vortragsfolien, 1999.

[SYN96] Synopsys, Inc.: *Power Products Reference Manual*, Version 3.5, Kapitel 2, Seite 2-9, 1996.

[TMW94a] V. TIWARI, S. MALIK und A. WOLFE: *Instruction Level Power Analysis and Optimization of Software*, Journal of VLSI Signal Processing Systems, August 1996.

[TMW94b] V. TIWARI, S. MALIK und A. WOLFE: *Power analysis of embeddded software: A first step towards software power minimization*, IEE Transactions on VLSI Systems, 2(4): Seite 437-445, Dezember 1994.

[TIW96] V. TIWARI: *Logic and System Design for Low Power Consumption*, University of Princeton, Dissertation: Seite 163-203, November 1996.

[TL98] V. TIWARI und T.C. Lee: *Power Analysis of a 32-bit Embedded Microcontroller*, VLSI Design Journal, 7(3), 1998.

[YAS00] H. Yasuura: *System Design Methods for Low Energy Consumption Embedded Systems*, Folienvortrag, Summer school, Universität Dortmund, 2000.

[ZIV94] V. Zivojnovic: *DSPstone: A DSP-oriented benchmarking methodology.* In International Conference on Signal Processing and Technology, 1994.