Energiecharakterisierung rekonfigurierbarer Rechensysteme

Diplomarbeit im Fach Informatik

vorgelegt von
Thomas Hirth
geboren am 30.08.1977 in Bayreuth

Institut für Informatik,
Lehrstuhl für Verteilte Systeme und Betriebssysteme,
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
	Dipl.-Inf. Andreas Weißel
	Dipl.-Inf. Christian Fiermann

Beginn der Arbeit:	01.11.2005
Abgabedatum:	28.04.2006

Kurzfassung

Heutzutage wird der Energiebedarf von Rechensystemen vor allem durch mobile Anwendungen zunehmend interessant. Zu einem Rechensystem gehört immer auch Arbeitsspeicher, der bislang bei Untersuchungen des Energiebedarfes des Gesamtsystems kaum beachtet wurde. Mittels der heutzutage zur Verfügung stehenden Technik ist es möglich, rekonfigurierbare Rechensysteme mit speziell angepasster Hardware aufzubauen.
In dieser Diplomarbeit werden zunächst die physikalischen Grundlagen behandelt, die überhaupt zu Energieverbrauch moderner digitaler Schaltungstechnik führen, wobei auch die Eigenschaften des heute weit verbreiteten SDRAM nicht zu kurz kommen.
Für das zu implementierende System wird ein rekonfigurierbares Rechensystem verwendet, um mit hinzugefügten Erweiterungen Speicherzugriffe zählen zu können, welche von einem leicht modifizierten Linux-Betriebssystem ausgewertet werden. Dabei werden Messungen des elektrischen Stromflusses während des Ablaufes von Programmen durchgeführt, die intensiv auf den Arbeitsspeicher zugreifen. Mit der durch die Zugriffszähler genau bekannten Zahl der Speicherreferenzen kann auf den Energiebedarf eines einzelnen Speicherzugriffs geschlossen werden.
Zusätzlich sind weitere Zähler hinzugefügt, die die Anzahl der Referenzierungen von Bereichen des Speichers zählen. Damit kann die Lokalität von Programmen festgestellt werden, mittels Benchmarkprogrammen wurde diese Funktion nachgewiesen. Mit dem Betriebssystem ist es nun möglich, nach Prozessen getrennt die Lese- und Schreibzugriffe zu erfassen. Damit kann festgestellt werden, ob eine gegebene Anwendung eher rechen- oder speicherintensiv ist, und wie lokal ihr Speicherzugriffsmuster aussieht.
Durch diese Ergebnisse ist es möglich, bislang nur theoretisch durchdachte Ansätze zur effizienteren Nutzung des Arbeitsspeichers und geschickter Ausnutzung der Vorteile verschiedener Speichersysteme auch erfolgsversprechend in die Praxis umzusetzen.

Abstract

Today the energy consumption of computer systems gains in interest especially because of mobile applications. But in every computer system a main memory is needed, which was up to now rarely considered in examinations of the energy consumption. With the nowadays available technology of reconfigurable computer systems it is possible to build systems with specialized hardware.
In this diploma thesis first of all the physical principles that lead to the energy consumption of modern digital circuitry are covered and the properties of commonly used SDRAM memory are not missed out.
For the implemented system a reconfigurable computer system is used to count memory accesses by added hardware. These counts are evaluated by a slightly modified Linux

operating system. While some programs that heavily access the main memory are run measurements of the electric current are conducted. With the known access count from the counters it is possible to calculate the energy usage of a single access.
Additionally counters were added to count the number of references of discriminated memory regions. With those the locality of programs can be observed, witch was verified by running benchmark programs. With the Operating system it is now possible to gather information of memory read and write operations discrete for each process. It can be stated which application is more computationally or memory intensive and even the locally of its reference pattern can be figured out.
With these results it is possible to put so far only theoretical proposals for the efficient usage of the main memory and even clever exploits of the advantages of different memory systems successfully into practice.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 27. April 2006 _____

Unterschrift

Inhaltsverzeichnis

Kurzfassung / Abstract ii

Versicherung iv

1. Motivation 1

2. Energie und Energieverbrauch 3

2.1. Definition des Energieverbrauches 4

2.2. Gründe für den Energiekonsum in CMOS 4

2.2.1. Strukturverkleinerung 6

2.2.2. Schaltquerströme 8

2.2.3. Leckströme 9

2.2.4. Kapazitive Lasten 11

2.3. Hardwareoptionen um den Energieverbrauch zu reduzieren 12

2.4. Eigenschaften von SDRAM 13

3. Thematisch verwandte Forschungsarbeiten 20

3.1. Ermittlung des Stromverbrauches 20

3.1.1. Messungen 21

3.1.2. Rechnerische Verfahren 23

3.2. Energieabschätzung mit Hilfe von Zählern 24

3.3. Anwendungen und Verfahrensweisen 26

3.3.1. Softwarebasierte Ansätze 26

3.3.2. Hardwareverfahren 26

4. Architekturüberlegungen und Implementierungen 29

4.1. Hardwarebeschreibungssprache VHDL 29

4.2. Field Programmable Gate Arrays 32

4.3. Grundlegende Überlegungen 34

4.4. Detaillierte Erläuterung der Implementierung der Zugriffszähler . . . 36

4.5. Details zu den Chunkzählern 42

4.6. Der Enabler 43

4.7. Auswertalgorithmen und Kerneltreiber 44

4.8. Verifikation der Hardware 46

4.9. Zusammenfassung der Hardware 47

5. Messungen und Ergebnisse **49**
5.1. Vorlaufuntersuchungen 49
5.2. Messungen mit dem Current Sense Amplifier 49
5.3. Vorabmessungen ohne Zählerstandbeachtung 54
5.4. Messungen mit Zählerstandsanalyse 55
5.4.1. Ermittlung des Energieverbrauches eines Zugriffes 56
5.4.2. Energiebedarf in Benchmark-Programmen 59

6. Fazit und Ausblick **66**

Abbildungsverzeichnis **68**

Tabellenverzeichnis **69**

Listings **70**

Literatur **71**

A. Bedienungsanleitung und Zusatzinformation **77**
A.1. Installation und Inbetriebnahme des Linux-Betriebssystems 77
A.2. Das JTAG-Interface 80

B. Listings **82**
B.1. VHDL-Quellen 82
B.2. Kernel-Quellen 90

1. Motivation

Energie geht nicht verloren.
Hermann Ludwig Ferdinand von Helmholtz, (1821 - 1894)

Mit diesem Ausspruch hat der Physiker Helmholtz sicherlich Recht. Energie bleibt erhalten, wird aber in andere Erscheinungsformen umgewandelt. Gerade elektrische Energie verwandelt sich dabei in thermische Energie, die, den Gesetzen der Thermodynamik folgend die höchstmögliche Entropie anstrebt, und somit für den Laien „verloren“ zu gehen scheint.

Der weltweite Energiebedarf wird in den nächsten Jahrzehnten weiter anwachsen. Die globalen Herausforderungen einer ausreichenden, umweltverträglichen Energieversorgung machen es erforderlich, dass wo immer möglich der Energieverbrauch minimiert wird. Durch die Turbulenzen auf den Energiemärkten steigen die Preise auch für elektrische Energie seit Jahrzehnten stark an. Darum wird der Energiebedarf von mobilen Geräten und auch von eingebetteten Systemen immer kritischer hinterfragt und inzwischen sogar für Kaufentscheidungen herangezogen. Denn eine hohe Energieaufnahme führt zusätzlich zu hohen Kosten zu verkürzter Laufzeit von batteriebetriebenen Geräten und auch zu erhöhten Anforderungen an die Kühlung bzw. Klimatisierung von stationären Geräten wie PCs oder Serverschränken, die dadurch oft unangenehm laut werden.

Zum Energiemanagement von Rechensystemen sind in den letzten Jahren viele Forschungen unternommen worden, die sich aber vor allem mit CPU, Festplatten und drahtlosen Netzwerkverbindungen beschäftigt haben. Dahingegen wurde der Energiebedarf des Hauptspeichers kaum betrachtet, obwohl mittlerweile in großen Servern der Speicher noch vor den CPUs im Leistungsbedarf liegt. Zu den einzelnen Speicherbausteinen liefern die Hersteller Datenblätter mit Daten zum Energieverbrauch und Zeitverhalten. Diese sagen aber praktisch nichts über das Verhalten im fertigen System unter realen Einsatzbedingungen aus. Mit Hilfe von rekonfigurierbarer Hardware können Rechensysteme aufgebaut werden, die – mit speziell angepasster Hardware – beliebige zusätzliche Aufgaben übernehmen können. So wurde die Idee geboren, mit Hilfe von Hardware in einem rekonfigurierbaren Rechensystem das Verhalten des Speichers in Hinblick auf den Energiekonsum genauer zu untersuchen, und gegebenenfalls durch Messungen zu neuen Erkenntnissen zu gelangen.

Als Grundlage dient ein mit einem FPGA (*Field Programmable Gate Array*) und SDRAM (*Synchronous Dynamic Random Access Memory*) ausgestattetes Developmentboard (Entwicklungsplatine). Der FPGA hat im Silizium einen PowerPC-Kern integriert, während große Teile der Peripherie, die in einem kommerziellen Prozessor ebenfalls integriert sind, im VHDL-Quelltext (*Very High Speed Integrated Circuits Hardware Description Language*) vorliegen. Diese können zusammen mit der hier in dieser Diplomarbeit in VHDL entwickelten Zusatzhardware eingesetzt werden. Sie liefern durch kleine zusätzliche Erweiterungen Daten welche die Zusatzhardware für ihre Aufgaben benutzt (mehr dazu in Kap. 4, Seite 29ff).

Hier wird zunächst mit den physikalischen Grundlagen von Energieverbrauch in elektronischen Schaltungen begonnen und auf thematisch verwandte Forschungsarbeiten hingewiesen. Im Weiteren wird dann die spezielle Implementierung von konfigurierbarer Hardware und deren Einbindung in ein Linux-Betriebssystem beschrieben, bevor mit Messungen begonnen wird, die schließlich zum Ziel haben, den Energieverbrauch eines einzelnen Speicherzugriffs zu charakterisieren. Am Schluss werden noch mögliche Ansätze für weiterführende Forschungsarbeiten oder Gebiete für die praktische Verwendung der hier gefundenen Zusammenhänge aufgezeigt.

2. Energie und Energieverbrauch

Aus [44], zum Suchwort „Energie“:

„Energie ist ein universeller Begriff, der in den unterschiedlichsten Zusammenhängen verwendet wird: [...]

- In der Philosophie ist *Energeia* (griechisch für Tätigkeit) seit Aristoteles der Inbegriff des Realen im Gegensatz zum bloß Möglichen (Dynamis).
- Im Alltag verwendet man das Wort Energie für den psychischen Antrieb und das körperliche Arbeitsvermögen eines Menschen.
- In Physik und Ingenieurwissenschaften hat der Begriff Energie [...] große Bedeutung erlangt. Energie ist dort die Fähigkeit, Arbeit zu verrichten.“

In dieser Arbeit geht es insbesondere um die Bedeutung der Energie und des Energieverbrauches im physikalischen bzw. ingenieurwissenschaftlichen Zusammenhang:

„Die als Energie gespeicherte Arbeit muss nicht in der Arbeitsform abgegeben werden, in der sie aufgenommen wurde. Diese Abgabe ist auch in anderen Arbeitsformen möglich. [...] Alle Naturerscheinungen gehorchen einem fundamentalen Gesetz, der *Erhaltung der Energie*:

> In einem abgeschlossenen System bleibt der Energieinhalt konstant. Energie kann weder vernichtet werden, noch aus dem Nichts entstehen; sie kann sich in verschiedene Formen umwandeln oder zwischen verschiedenen Teilen des Systems ausgetauscht werden.“ (Aus [18], S. 51f)

Dieser Satz ist einer der experimentell am besten abgesicherten Sätze der Physik.

Die Einheit der Energie (E) ist dieselbe wie die der Arbeit (W), das Joule [J]. Ein Joule ist eine Wattsekunde [Ws] oder ein Newtonmeter [Nm]. Die bekannteste Einheit für Energie ist jedoch die Kilowattstunde [kWh], nach der z.B. der Stromverbrauch eines Haushaltes abgerechnet wird. Eine kWh entspricht somit 3,6 Millionen Joule. Eng verknüpft mit Arbeit und Energie ist die Leistung (P), dabei handelt es sich physikalisch gesehen um Energie pro Zeitspanne ($P = \frac{\Delta W}{\Delta t}$). Die Einheit dafür ist [$\frac{J}{s}$] bzw. Watt [W].

Aufgrund dieser engen Verwandtschaft werden im Folgenden die Begriffe „Energie“ und „Leistung“ synonym verwendet, da $J = Ws = CV$ gilt.

2.1. Definition des Energieverbrauches

In dieser Arbeit wird der Begriff „Energieverbrauch“ wie folgt definiert:

> Energieverbrauch ist die Menge an elektrischer Energie, die aufgewendet wird, um eine Schaltung zu betreiben. Darin enthalten ist auch der Anteil an Energie, der beim Betrieb in Wärme umgewandelt wird, also nicht mehr für Schaltvorgänge zur Verfügung steht.

Der Energieverbrauch ist somit kein „Verbrauch“ im eigentlichen Sinne, sondern eine Energieumwandlung und zwar von elektrischer Energie in Wärme. Da diese thermische Energie jedoch praktisch nicht weiter genutzt wird, kann trotzdem von „Verbrauch“ oder „Verlust“ gesprochen werden.

Bei der Betrachtung des internen Energieverbrauchs einer elektrischen Schaltung muss grundsätzlich zwischen zwei Arten unterschieden werden:

1. Statischer Anteil: Er besteht vor allem aus dem sog. Leckstrom (siehe 2.2.3). Der statische Anteil ist nicht abhängig von der Schaltfrequenz (f), sondern steigt mit der Versorgungs- bzw. Betriebsspannung (V_{DD}) und der Temperatur (T).
2. Dynamischer Anteil: Dieser kommt nur zum Tragen, wenn die Schaltung betätigt wird, also Schaltvorgänge zustande kommen. Der dynamische Anteil ist praktisch nicht von T abhängig, sondern vor allem von der Schaltfrequenz (f), V_{DD} sowie parasitären Kapazitäten (siehe Abschnitt 2.2.4).

2.2. Gründe für den Energiekonsum in CMOS

Die derzeitig in der digitalen Elektronik dominierende Schaltungstechnologie nennt sich „CMOS“ (*Complementary Metal Oxide Semiconductor*). In Abbildung 1, Seite 5, ist ein N-Kanal-MOSFET (*Metal Oxide Semiconductor Field Effect Transistor*)

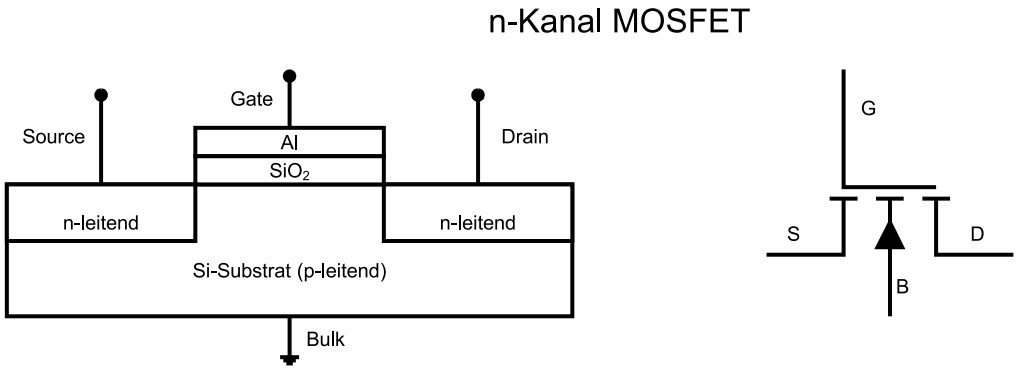

Abbildung 1: n-Kanal-MOSFET (links schematisch, rechts Schaltsymbol)

mit Skizze des Internen Aufbaus und seinem Schaltsymbol zu sehen. Die Anschlüsse G (*Gate*), S (*Source*) und D (*Drain*) werden immer eingezeichnet, während der B-Anschluss (*Bulk*) oft nicht mit Masse verbunden oder mit dem S verbunden dargestellt wird.

Das CMOS-Bauprinzip für logische Funktionen ist die Kombination von je einem P-Kanal-Feldeffekttransistor (FET) mit einem dazu komplementären N-Kanal-FET. Aufgrund der gleichen Steuerspannung und den komplementären Transistoren sperrt in jedem logischen Zustand genau einer der Transistoren, während der andere leitend ist. Dadurch können die Logikpegel den ganzen Spannungshub zwischen der Versorgungsspannung (V_{DD}) und der Masse (GND) ausnutzen, da der jeweils leitende Transistor einen *Pull-Up*- bzw. *Pull-Down*-Pfad für den Ausgang darstellt („*Rail-to-Rail*“). Das heißt, dass der Ausgang über den gerade leitenden Transistor entweder mit Masse oder Betriebsspannung verbunden ist. Abbildung 2, Seite 6, stellt den Schaltplan eines Inverters in CMOS dar. I ist der Eingang und O der Ausgang, an dem das invertierte Eingangssignal anliegt. Es ist deutlich der Aufbau aus komplementären Transistoren zu erkennen.

Als Nachteil erscheint, dass für jedwede logische Teilfunktion immer zwei Transistoren in eine Schaltung integriert werden müssen. Jedoch ist ein Transistor mit den verwendeten Lithographieprozessen wesentlich leichter und genauer herzustellen als ein Widerstand, der in anderen Schaltungstechniken eingesetzt werden müsste. Deswegen wird für VLSI (*Very Large Scale Integration*, d.h. der Integration von sehr vielen Gattern auf einem Chip) heutzutage praktisch ausschließlich CMOS verwendet.

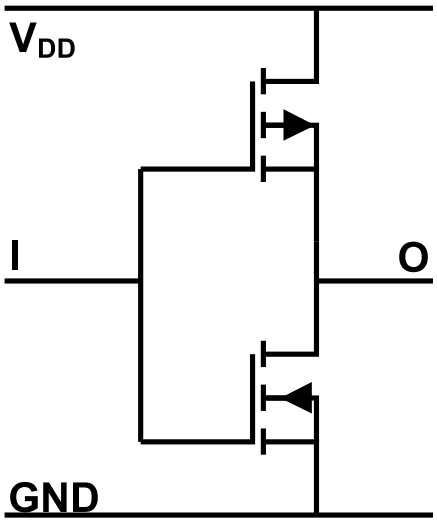

Abbildung 2: CMOS-Inverter (oben p-Kanal- unten n-Kanal-MOSFET)

Der Hauptvorteil von CMOS ist, dass ein Gatter nur beim Umschaltvorgang Energie benötigt. Allerdings gibt es vor allem durch die fortschreitende Miniaturisierung noch andere Situationen, in denen Energie in Verlustleistung übergeht, wie im Folgenden beschrieben.

2.2.1. Strukturverkleinerung

In der Mikroelektronikbranche wird oft von der Größeneinheit „f“ gesprochen, womit in diesem Abschnitt nicht die Frequenz, sondern die *Feature Size* („Merkmalsgröße“) gemeint ist. Damit wird laut [17] die kleinste Abmessung eines Transistors in Breite oder Länge gemessen. 1971 lag f noch bei 10μm, Anfang 2006 bei 65nm. Ein einzelner Transistor ist somit binnen 25 Jahren in der Fläche auf 1/23.500 geschrumpft. Dadurch entstehen natürlich auch massive Kosteneinsparungen, da auf einer Scheibe Silizium (*Wafer*) deutlich mehr Schaltungen produziert werden können und die Ausbeute steigt. Deswegen ist jeder Halbleiterhersteller bemüht, die Strukturen immer weiter zu verkleinern, um seinen Gewinn – trotz enormer Investitionen – zu maximieren. Aufgrund

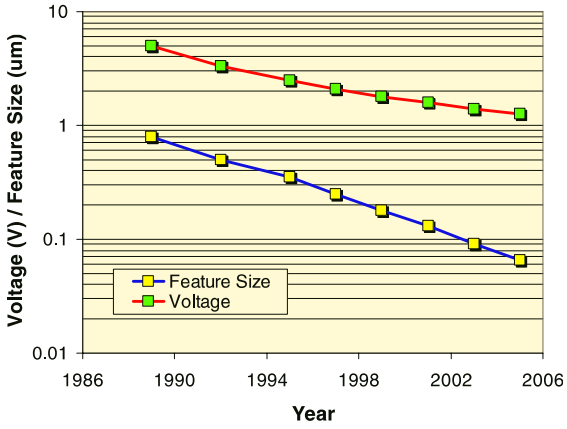

Abbildung 3: f und V_{DD} in den letzten Jahren (aus [24])

dieser ständigen Strukturverkleinerungen verdoppelt sich die Anzahl der Transistoren pro Chip etwa alle 18 Monate nach dem so genannten „Moore'schen Gesetz" (*Moore's Law*).

Wenn f schrumpft, verkleinern sich die Strukturen nicht nur in X- und Y-Richtung, sondern auch in der Z-Richtung, also der Tiefe im Substrat. Laut [17], Seite 13f, folgen daraus direkt drei Probleme, die in den nächsten Abschnitten immer wieder aufgegriffen werden:

1. Die Betriebsspannung der Transistoren muss gesenkt werden, um die korrekte und verlässliche Funktion der Schaltungen aufrecht zu halten. Dieser Zusammenhang ist sehr gut in Abbildung 3, Seite 7, zu erkennen.

2. Obwohl die Transistoren wegen der Strukturverkleinerungen immer besser werden, gilt dies keinesfalls auch für die Verbindungsleitungen in integrierten Schaltungen. Vor allem die Signalverzögerung, das so genannte *Propagation Delay* steigt, obwohl die Länge der Leitungen sinkt. Dies kommt durch die Vergrößerung des Widerstandes und der Kapazität zustande, mit dem sich die sog. *RC*-Zeitkonstante erhöht. Diese beiden Größen werden auf komplexe Weise durch den Fertigungsprozess, die Geometrie, die Last und sogar durch benachbarte Strukturen beeinflusst. Auf diese genauer einzugehen würde den Rahmen dieser Arbeit sprengen. Es sei aber beispielhaft erwähnt, dass laut [17] im Intel Pentium 4 von 2001 zwei von seinen über 20 Pipelinestufen alleine dazu dienen, die Signale über den Chip zu verbreiten.

3. Auch der Leistungsverbrauch wird zu einer Herausforderung durch das Verkleinern der Strukturen. In modernen Mikroprozessoren ist das Schalten der Transistoren die Hauptursache für den Energieverbrauch. Die Energie pro Transistor ist proportional zu dem Produkt aus Lastkapazitäten, der Schaltfrequenz und dem Quadrat der Betriebsspannung. Trotz der Abnahme der einzelnen Lastkapazitäten und der oben schon erwähnten Absenkung der Spannung führt der ständige Anstieg der Anzahl, der mit immer höherer Frequenz schaltenden Transistoren zu einer ständigen Zunahme des Leistungsverbrauches (siehe auch Abschnitt 2.2.2). Die ersten Mikroprozessoren benötigten Leistungen in der Größenordnung von einem zehntel Watt, während heutige Hochleistungsprozessoren leicht 150Watt verbrauchen. Deswegen wird laut [17] die Verteilung des Stromes, das Abführen der Wärme und die Vermeidung von *Hot Spots* (Punkte im Mikroprozessor, die durch viele Schaltvorgänge extrem heiß werden, da die Wärme nicht schnell genug verteilt wird) ein größeres Limit für das Fortschreiten der Technologie sein, als das stetige Vervielfachen der Transistoranzahl.

Diese Punkte führen alle zu einem Intradependenznexus zwischen der Leistungsfähigkeit der Schaltungen und f, wobei als Daumenregel gesagt werden kann, dass die Leistungsfähigkeit linear mit der Strukturverkleinerung steigt.

2.2.2. Schaltquerströme

Als Schaltquerströme bezeichnet man den Strom, der beim Umschalten technologiebedingt durch eine logische Schaltung fließt, aber nicht beabsichtigt wird. Er kommt folgendermaßen zustande:

Mit der in 2.2.1 angesprochene Miniaturisierung entstehen durch die sehr dünnen und relativ langen Verbindungsleitungen auf einem Chip erhebliche Leitungswiderstände. Diese bedingt, dass das Umladen der Gates an den an dieser Leitung angeschlossenen Transistoren mit einer gewissen *RC*-Zeitkonstante und einer damit einhergehenden Verzögerung geschieht. Erreicht nun einer der Transistoren seine Schaltspannung und schaltet vom sperrenden in den leitenden Zustand, hat der andere Transistor seine Schaltspannung noch nicht erreicht und befindet sich immer noch im leitenden Zustand. Zudem haben N- und P-Kanal-FETs herstellungstechnisch bedingt leicht unterschiedliche Schaltspannungen und Schaltgeschwindigkeiten.

Somit entsteht eine leitende Verbindung durch die beiden Transistoren von V_{DD} nach GND, also ein Kurzschluss in der Schaltung. Dieser Kurzschluss könnte für die Schaltung potentiell zerstörend wirken, wenn er nicht von folgenden Punkten begrenzt würde:

- Die FETs zeigen ein analoges Umschaltverhalten d.h., ihr Durchgangswiderstand liegt nicht sofort bei 0Ω, sondern hat einen exponentiellen Verlauf mit *RC*-Zeitkonstante in der Potenz von *e*.
- Der andere Transistor schaltet von leitend nach sperrend um, dies geschieht zwar nicht zum exakt gleichen Zeitpunkt wie bei dem oben genannten Transistor, aber doch nach einer nur extrem kurzen Zeitspanne. Tritt dieser Zustand ein, ist der Stromfluss durch den Querstrompfad nicht mehr möglich.

Durch vernünftiges Schaltungsdesign, welches lange, dünne Leitungen mit nur geringer Treiberstärke (*Fan-gut*) vermeidet, kann der Schaltquerstrom minimiert werden. Die Verlustleistung des Schaltquerstromes liegt üblicherweise um Größenordnungen unter den anderen Verlustleistungen, so dass sie in Datenblättern üblicherweise nicht erwähnt wird.

Eine sehr eingängige Erläuterung mit einer Animation dieses Zusammenhanges am Beispiel eines Inverters mittels eines Java-Applets ist unter [28] zu finden.

2.2.3. Leckströme

FETs haben einen Schwellwert der Gate-Source-Spannung (V_{th}; *Threshold Voltage*), unterhalb dessen der Stromfluss durch den Transistor (von Source nach Drain) exponentiell abfällt. Früher lag diese Schwelle bei etwa 700mV, während die Schaltung mit 5V arbeitete. Heutzutage liegt jedoch die Versorgungsspannung teilweise im Bereich von 1V, während V_{th} immer noch bei etwa 200mV liegt (vgl. die Spannungskurve in Abb. 3, Seite 7).

Wegen dieses geringen Abstandes ist der Transistor nicht optimal gesperrt und es fließt durch die geringe Potentialdifferenz von V_{DD} zu V_{th} ein Leckstrom durch den FET, der bei manchen Fertigungstechnologien die Größenordnung des Schaltstromes erreicht.

Dieser so genannte *Sub Threshold Current* ist momentan der dominierende Anteil des Leckstroms.

Ein weiterer Anteil des Leckstroms ist der Tunnelstrom von Elektronen, die aus dem Gate durch die dünne Oxidschicht (in aktuellen Technologien nur noch fünf Atomlagen dünn) in den Kanal tunneln, da einige Elektronen thermisch bedingt genügend Bewegungsenergie besitzen, um die Isolatorschicht zu überwinden. Dies führt dazu, dass die Transistoren in der vorhergehenden Logikstufe für einen größeren *Fan-Out* (Ausgangslast der Schaltung) ausgelegt sein müssen, weil sie noch zusätzlich den Gatestrom der nachfolgenden Stufe treiben müssen.

Somit hängt der Leckstrom von der Dimensionierung des Kanals, der Dotierung, den Abmaßen von Drain- und Source-Übergängen, der Dicke des Gateoxids, der Temperatur und vielen weiteren Faktoren ab. Eine vollständige Abhandlung aller Faktoren und Einflussgrößen würde jedoch dem Ziel dieser Arbeit nicht gerecht werden.

Als Beispiel sei hierzu aus [23], S.20, dem Datenblatt des Pentium M Mikroprozessors, zitiert:

> „*The Intel Pentium M processor supports the THERMTRIP# signal for catastrophic thermal protection. [...] Even with the activation of THERMTRIP#, that halts all processor internal clocks and activity, leakage current can be high enough such that the processor cannot be protected in all conditions without the removal of power to the processor. If the external thermal sensor detects a catastrophic processor temperature of 125°C (maximum), or if the THERMTRIP# signal is asserted, the VCC supply to the processor must be turned off within 500 ms to prevent permanent silicon damage due to thermal runaway.*“

Frei übersetzt bedeutet das unter anderem, dass die Summe der Leckströme so groß ist, dass alleine durch die dadurch in Wärme umgewandelte Verlustleistung das dotierte Silizium des Prozessors irreparabel geschädigt werden kann. Nebenbei sei erwähnt, dass der Pentium M als ein sehr energieeffizienter Prozessor gilt.

Zusammenfassend lässt sich sagen, dass durch die ständige Verringerung der Transistordimensionen der Leckstrom immer weiter ansteigt. Die Abbildung 4 zeigt, wie

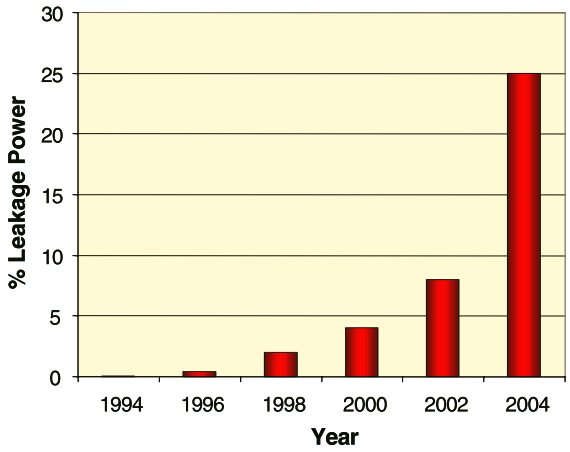

Abbildung 4: Leckstromanteil im letzen Jahrzehnt (aus [24])

die Leckströme im letzten Jahrzehnt einen immer höheren Anteil an der Gesamtleistung ausmachten. Wenn dieser Trend weiter anhalten sollte, werden die Leckströme binnen kurzer Zeit die Schaltströme übertreffen.

2.2.4. Kapazitive Lasten

Den dominierenden Anteil am Stromverbrauch haben die kapazitiven Lasten, die beim Schaltvorgang umgeladen werden müssen. Dazu gehören die parasitären Kapazitäten jedes einzelnen Transistors sowie der am Ausgang hängenden Leitungen und natürlich auch die Gates der in der Schaltung folgenden Transistoren. Eine ausführliche Aufstellung und Berechnung aller dieser Kapazitäten findet sich in [39] S. 208ff. Dieser für das Umladen benötigte Strom wird mit kleineren Strukturgrößen nicht geringer, da die Summe der Kapazitäten (C) in erster Näherung nicht nur von der Fläche (A), sondern auch vom Abstand (d) und den Dielektrika zwischen den Leitern (der Permittivität $\varepsilon_0\varepsilon_r$) nach folgender Gleichung abhängt:

$$C = \varepsilon_0 \varepsilon_r \frac{A}{d} \tag{1}$$

Die zum Schalten benötigte Energie ist somit die gespeicherte elektrische Energie der gesamten Kapazitäten und berechnet sich wie folgt:

$$W = \frac{1}{2} C U^2 \tag{2}$$

Die in Kapazitäten gespeicherte Energie ist für die Mikroelektronik von großer Bedeutung. Trotz der mikroskopischen Abmessungen der Schaltungen werden relativ hohe Kapazitäten erreicht, weil das üblicherweise als Isolator verwendete Siliziumdioxid ein relativ großes ε_r von 3,9 hat, was in Gleichung 1 als Faktor eingeht. Außerdem führen die immer kleineren Strukturen zu immer geringeren Abständen d, was die Kapazitäten zusätzlich vergrößert. Dies funktioniert so gut, dass beispielsweise die Zustandsspeicherkondensatoren in DRAM aus abwechselnden Schichten von dotiertem Silizium und SiO_2 hergestellt werden.

Die Arbeit aus Gleichung 2 muss bei jedem Zustandswechsel der Schaltung aufgebracht werden, so dass die aufgenommene Leistung auch noch von der Schaltfrequenz abhängt, die in modernen Schaltungen einige Megahertz bis hin zu mehreren Gigahertz beträgt. Somit errechnet sich die Verlustleistung näherungsweise nach Gleichung 3 (mit dem Schaltfaktor σ).

$$P = \sigma \frac{1}{2} C U^2 f \tag{3}$$

2.3. Hardwareoptionen um den Energieverbrauch zu reduzieren

Aus den in Abschnitt 2.1 genannten Einflussgrößen ergeben sich alle Größen, an denen Veränderungen vorgenommen werden können, um den Energieverbrauch zu senken.

- Temperatur: Wenn die Schaltung heruntergekühlt wird, sinken die Leckströme. Dies wird in der Praxis kaum angewendet, da eine wirkliche Kühlung zu energieaufwendig ist. Deswegen wird nur mit Ventilatoren versucht, die Temperatur nicht allzu stark über die Zimmertemperatur steigen zu lassen.

- Versorgungsspannung: Aus Gleichung 2 wird ersichtlich, dass der Energieverbrauch pro Schaltvorgang quadratisch mit der Versorgungsspannung steigt. Deswegen kann schon mit einem kleinen Absenken von V_{DD} Energie eingespart werden, da der Leckstrom nur linear von V_{DD} abhängt.

- Schaltfrequenz: Digitale Schaltungen arbeiten normalerweise getaktet, d.h. nur zu bestimmten, durch den Takt vorgegeben Zeitpunkten können die Transistoren geschaltet werden. Wie mit Gleichung 3 erkenntlich, kann durch eine Senkung der Taktrate die Energieaufnahme linear gesenkt werden. Dadurch wird die

Schaltung aber auch langsamer. Zudem spielen ungetaktete digitale Schaltungen für Rechensysteme spielen in der Praxis keine Rolle.

In ganz modernen Schaltungen findet man häufig Mischformen der letzten beiden Punkte, da sich Spannung und Schaltfrequenz gegenseitig beeinflussen: Für geringere Schaltfrequenzen genügt eine geringere Betriebsspannung. In modernen Schaltungen können diese Größen sogar im laufenden Betrieb verändert werden. Derartige Methoden sind meist mit dem Schlagwort DFVS (*Dynamic Frequency and Voltage Scaling*) gemeint.

Marktführer Intel schlägt in [24] vor, weniger auf Frequenzsteigerungen der Mikroprozessoren, sondern besser auf den Einsatz von Mehrfachkernen zu bauen, da dadurch auf langsamere Transistoren mit geringeren Leckströmen gesetzt werden kann.

Eine andere Möglichkeit zur Verringerung der Leckströme ist so genanntes *Strained Silicon* („gestrecktes Silizium“). Hierbei wird durch Änderungen im Herstellungsprozess der Chips die Ordnung des Kristallgitters manipuliert, was zu verringerten Leckströmen führt (siehe [22], S. 4).

2.4. Eigenschaften von SDRAM

SDRAM ist das Akronym für *Synchronous Dynamic Random Access Memory*. DRAM ist laut [45] eine besonders preiswerte Bauform für Halbleiter-Speicherbausteine, da die einzelnen Bitspeicher nur aus einem Transistor und einer Kapazität bestehen, der so genannten 1T1C-Anordnung. Daher ist der Aufbau in Lithographieprozessen prinzipiell einfach und auf einer kleinen Siliziumfläche möglich, was die Herstellung preiswert macht. Eine heutige Zelle benötigt nur etwa $0{,}025\mu m^2$. *Dynamic* deutet an, dass es sich dabei, wie bei den meisten RAM-Technologien, um einen dynamischen Speicher handelt. Das bedeutet, dass der Speicherinhalt beim Abschalten der Versorgungsspannung verloren geht. Synchrones DRAM (SDRAM) zeichnet sich gegenüber asynchronem DRAM dadurch aus, dass durch die taktsynchrone (*Synchronous*) Ansteuerung im Vergleich zu anderen DRAM-Typen wie *Fast-Page-Mode-DRAM* (FPM) oder *Enhanced-Data-Out-DRAM* (EDO) Wartezeiten kürzer ausfallen können. Deswegen wurde in der PC-Industrie ab 1996 der Einsatz von SDRAM forciert, so dass sich heute bis

auf einige Nischen kaum noch andersartige DRAM-Speicher in PCs finden lassen. Die einzigen Unterschiede zu damals sind die ermöglichte Erhöhung der Taktfrequenz durch die in Abschnitt 2.2.1, S. 6ff, angesprochene Strukturverkleinerung sowie die Erhöhung des Datendurchsatzes mittels dem so genanntem DDR-SDRAM (*Double Data Rate*), aktuell in den moderneren Varianten DDR2 und DDR3. DDR-SDRAM ist technisch prinzipiell mit SDRAM identisch, nur dass die Daten zu beiden Flanken des Taktsignals übertragen werden, also bei steigender und fallender Flanke je ein Bit.

Im Einzelnen ist eine Speicherzelle wie in 5, Seite 15 aufgebaut. Sie besteht, wie oben schon erwähnt, im Wesentlichen aus einer Speicherkapazität, einem Transistor, einer so genannten Wortleitung (WL) und einer Bitleitung, *Digit Line* oder auch Leseleitung (BL). Die einzelnen Speicherzellen sind im Speicherbaustein üblicherweise in einer Gitterstruktur angeordnet. Das Auslesen geschieht durch das Anlegen von Chip-Select-Signalen (CS), falls mehrere Chips in einem Modul zusammen eingebaut sind, dem Anlegen der Bankadresse und Reihenadresse und dem Eintreffen des RAS-Signals (*Row Address Select*). Die Leseverstärker lesen daraufhin die adressierte Zeile komplett aus, während mit dem mittlerweile eingetroffenen CAS-Signal (*Column Address Select*) das gewünschte Datenwort mit der Spaltenadresse aus dem Adressbus selektiert wird. Im Detail wird mit der dekodierten Zeilenadresse mittels der WL die entsprechende Zeile angesteuert, womit alle Transistoren in dieser Zeile leitfähig werden. Über die jeweilige BL wird dann der Ladungszustand des Kondensators an spezielle Leseverstärkerschaltungen (auch Schreib-Lese-Verstärker oder *Sense Amps* genannt) weitergereicht, welche aus den Kondensatorzuständen wieder korrekte digitale Signalpegel generieren. Laut [36] funktioniert das Lesen also über einen Ladungsausgleich, mittels dessen das in einen instabilen Zustand gebrachte Flipflop im Leseverstärker in einen stationären Zustand gebracht wird. Diesen Vorgang des instabilisierens nennt man *Precharging*. Durch das Lesen wird der Speicherinhalt wieder aufgefrischt, da quasi gleichzeitig mit dem Lesezugriff ein Schreibvorgang erfolgt. Zum Schreiben wird nämlich gleichermaßen eine Zeile selektiert und an die Bitleitung das gewünschte Potential angelegt, was über die Flip-Flop-Stellung des Leseverstärkers geschieht. Somit wird dann über den durchgeschalteten Transistor die Kapazität geladen oder entladen.

Einer der wichtigsten Gründe für oder gegen eine bestimmte Speichertechnologie ist die Geschwindigkeit. Der wichtigste Parameter ist nach [45] bei SDRAM die Zeitspanne t_{RC} (RC von RC-Zeitkonstante). Dies ist die Wartezeit, nach der eine Spei-

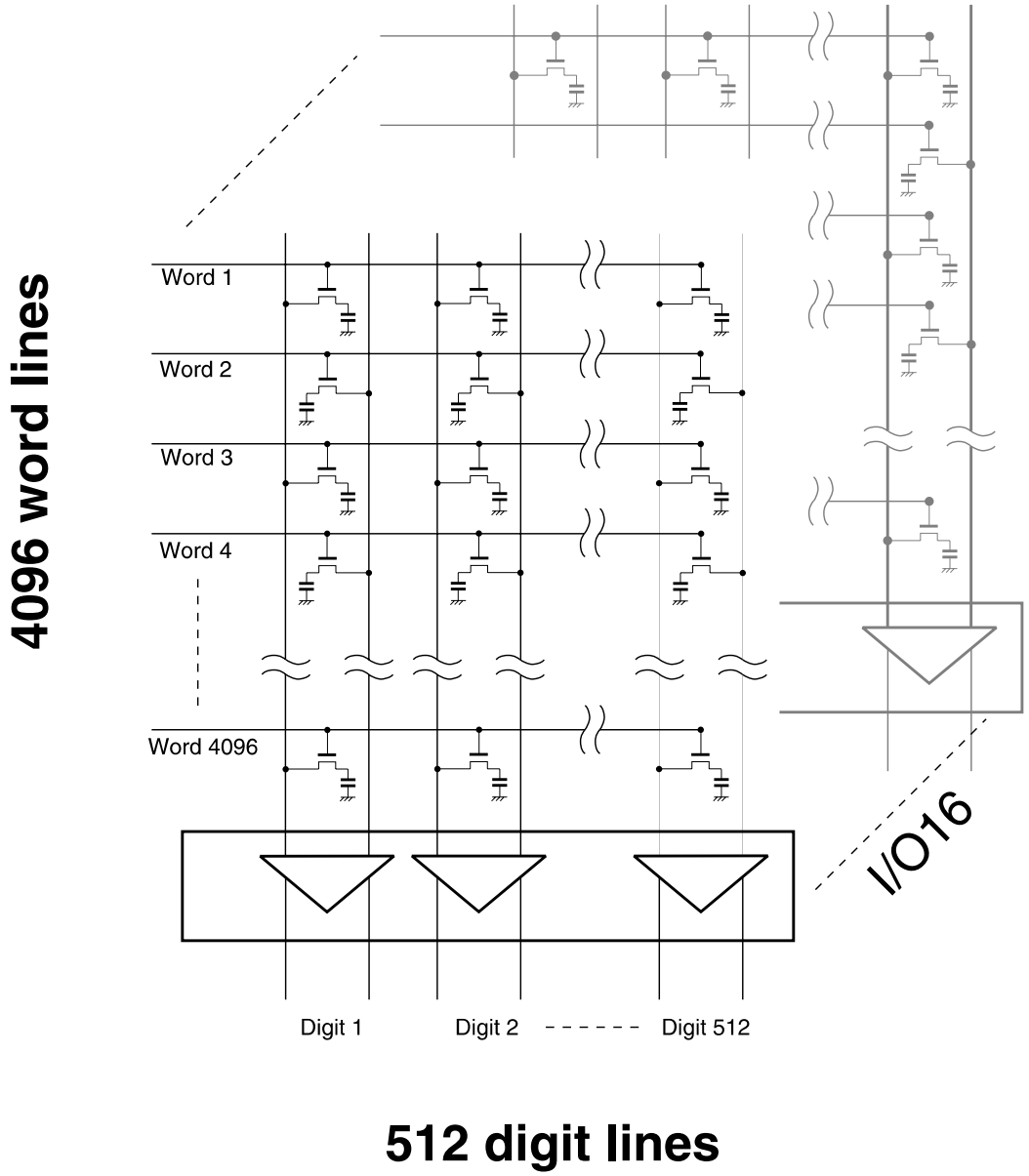

Abbildung 5: SDRAM-Speichermatrix aus einzelnen Zellen (aus [10])

cherzelle nach einem Zugriff wieder für den nächsten Zugriff bereit ist. Diese hängt davon ab, wie schnell die Leseverstärker den Zustand der Speicherzelle messen, also lesen, bzw. ändern, alias schreiben, können. Diese Geschwindigkeit hängt über die RC-Zeitkonstante eng mit der Zahl der an einer solchen Verstärkerschaltung hängenden Speicherzellen ab. Mit steigender Speicherkapazität wird der SDRAM-Speicher also erst einmal langsamer, da mehr Zellen an einem Leseverstärker angeschlossen sind. Um diesem Effekt entgegen zu wirken, unterteilt man die Speichermatrix in einzelne Bänke, die jeweils eigene Verstärkerschaltungen haben. Da durch die vielen Verstärker der Energieverbrauch massiv ansteigt, während die Kühl- und Stromversorgungsmöglichkeiten beschränkt sind, wird meistens zusätzlich die Betriebsspannung gesenkt, um eine Überhitzung zu vermeiden. Frühere SDRAM-Speicher liefen mit 3,3V, heute werden aktuelle Bausteine laut [45] über in den Speicher integrierte Spannungswandler mit 1,8V betrieben.

Mit der Unterteilung in einzelne Bänke kann zur Geschwindigkeitssteigerung das *Bank Interleaving* angewendet werden (vgl. [36]): Während die eine Bank gerade ihre Daten liefert wird schon die nächste Bank adressiert, so dass die verschiedenen Bänke sich mit dem Liefern der Daten abwechseln. Die auf dem Developmentboard verwendeten Speicher haben vier Bänke, da diese mit zwei Bits adressiert werden. Damit dies funktionieren kann, haben die Speicherbausteine selbst getrennte Anschlüsse für Daten, Adressen und Befehle. Pro Bank kommen jeweils eigene Adressdecoder und Leseverstärker zum Einsatz (siehe Abb. 6, Seite 17). Dahingegen werden die einzelnen Module, um Leitungen zu sparen, so angesteuert, dass die Spalten und die Zeilenadressen nacheinander angelegt werden, was mittels RAS- und CAS-Signalen unterschieden wird. Dahingegen sind zur Auswahl der Bänke zusätzliche Leitungen vorhanden. Die Adressierung mittels RAS, CAS und dem anschließenden Kommando (Lesen oder Schreiben) läuft in SDRAM-Bausteinen in einer dreistufigen Pipeline ab, so dass nach dem dritten Takt jeden weiteren Takt ein neues Wort geliefert werden kann.

Weil die Signale wegen der hohen Taktfrequenzen zu genau definierten Zeitpunkten ankommen müssen, um synchron zu bleiben, ist auch die Leiterführung auf der Platine wichtig. Auf normalem Platinenmaterial kommen elektrische Signale in einer Nanosekunde (entspricht 100MHz Frequenz) nur etwa 17cm weit. Parasitäre kapazitive und induktive Lasten zum Beispiel durch ungünstige Leitungsführung und Anschlussbeinchen verlangsamen die Signale zusätzlich. Deswegen kommen heute fast nur noch BGA-Gehäuse (*Ball Grid Array*) zum Einsatz, die anstelle von seitlichen An-

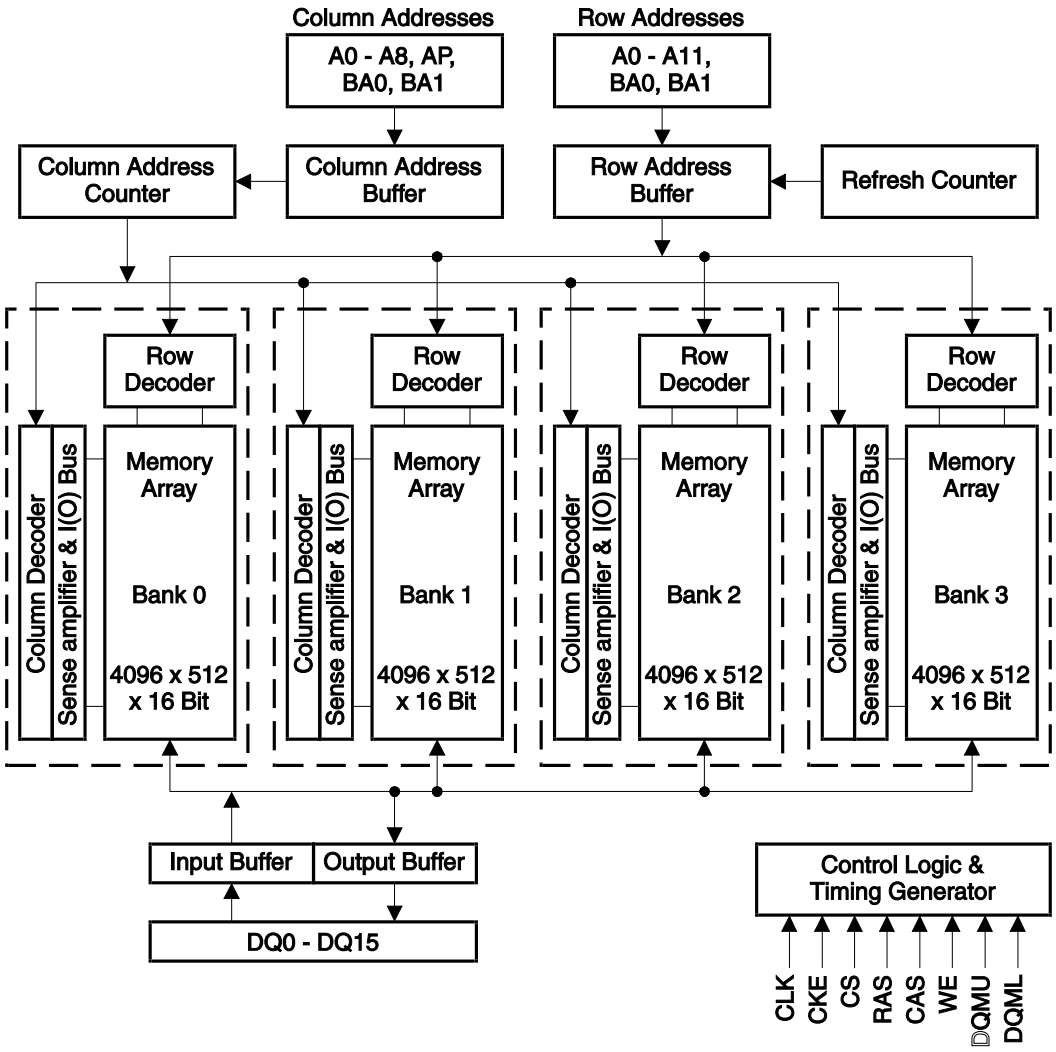

Abbildung 6: Blockdiagramm des SDRAMs auf der Experimentierplatine (aus [20])

schlussbeinen an der Unterseite der Chips kleine Lotkügelchen (*Balls*) tragen, die in maschinellen Verfahren direkt auf die Trägerplatinen gelötet werden.

Da die Kapazitäten über ähnliche Effekte, wie in den vorausgegangenen Kapiteln 2.2 (S. 4ff) erklärt, ihren Ladungszustand ändern, müssen die Speicherzellen, vor allem die, auf welche nicht zugegriffen wurde, in bestimmten Zeitabständen ausgelesen und wieder beschrieben werden, um ihren eindeutigen Ladezustand nicht zu verlieren. Diese sog. Refresh-Zyklen werden vom meist in den Speicherbaustein eingebauten *Refresh Controller* automatisch im Hintergrund durchgeführt (*automatic* oder *hidden Refresh*). Laut [36], Seite 52, beträgt diese Refreshperiode in aktuellen Bausteinen etwa 2ms, also etwa 500-mal pro Sekunde, was weniger als 1% der Arbeitsleistung verbraucht. Interessant ist, dass die Speicherkapazitäten einen Wert von etwa 15fF besitzen, während zum Vergleich die parasitären Kapazitäten an der Bitleitung gleichzeitig bei etwa 150fF liegen. Dadurch, dass die Speicherkapazitäten so klein sind, sind die „größten Leistungsfresser im SDRAM-Speicher die Leseverstärker“ ([36]).

An dieser Stelle kann der zweite Faktor zur Beschleunigung erklärt werden, das so genannte *Prefetching*. Dies bedeutet, laut [45], dass die ganze ausgelesene Zeile in einer Pufferstufe vorgehalten bleibt, solange keine andere Zeile per RAS selektiert wird. Somit stehen die Daten schon in Puffern zur Verfügung und müssen nicht nochmals erneut ausgelesen werden, wenn mittels CAS auf ein anderes Datenwort aus derselben Zeile zugegriffen wird. Dieses Prefetching kann aber nur seine Wirkung entfalten, wenn verschiedene Datenwörter derselben Spalte angefordert werden oder die Daten auf verschiedenen Bänken liegen. Wenn zwischenzeitlich auf eine andere Zeile zugegriffen wurde und dann wieder auf die Erste, kommt die oben angesprochene t_{RC} und die Zeit zum Precharging zur Wirkung. Diese liegt heute bei den schnellsten SDRAMS im Bereich oberhalb von 50ns. Bei den auf unserem Board verwendeten wird dafür ein Wert von 70ns angegeben, was einer maximal erreichbaren Zyklusfrequenz von etwas über 14MHz entspricht.

Wie [45], [36] und [10] feststellen, können SDRAM-Speicher ihre volle Geschwindigkeit nur bei optimalen Zugriffsmustern erreichen. Bei diesen liegt noch ein weiterer Ansatzpunkt für Geschwindigkeitssteigerung vor. Mittels des sog. *Burst*-Zugriffes werden die Chips so programmiert, dass sie ohne zusätzlichen Adressierungsvorgang, der nicht mit dem Prefetching verwechselt werden sollte, dazwischen gleich mehrere hintereinander liegende Datenworte liefern. Dieser Burst-Modus erlaubt es, größere Datenmengen in einem einzigen Zugriff zu übertragen. 7, Seite 19, zeigt einen sol-

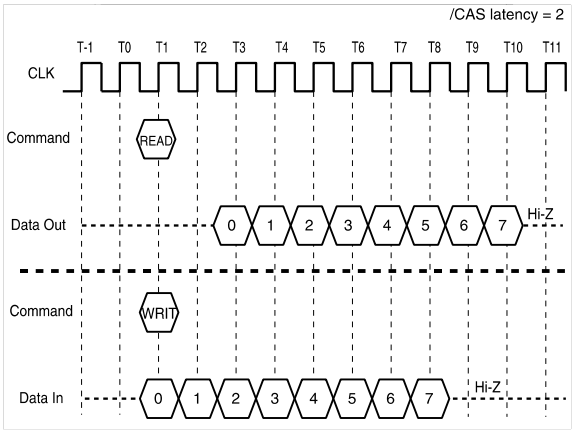

Abbildung 7: Ablauf eines Read- und eines Write-Bursts (aus [10])

chen Burst. Der große Vorteil besteht darin, dass nicht mehr jedes einzelne Datenwort extra adressiert und gelesen werden muss, sondern die jeweiligen Daten automatisch sequentiell vom Speicher geliefert werden. Üblicherweise ist die Burst-Länge so gewählt, dass sie genau eine Cachezeile (*Cache Line*) der Prozessorarchitektur füllt. Beim hier verwendeten PowerPC 4xx sind das acht Byte.

3. Thematisch verwandte Forschungsarbeiten

Über Energie, Energieverbrauch und Maßnahmen, beides zu beeinflussen, sind in den letzten Jahren eine Vielzahl von Artikeln erschienen. Vor allem seit das tägliche Leben immer mehr von mobiler Elektronik beeinflusst wird und die Energieversorgung der entscheidende Punkt für mobile Anwendungen ist (siehe z.B. die mobile Roboterplattform aus [19]), befassen sich immer mehr Forscher mit diesem Themenbereich. Aus diesem Grund sollen im Folgenden nur eine Auswahl der für die vorliegende Arbeit relevanten Quellen betrachtet und dem Ansatz dieser Diplomarbeit gegenübergestellt werden.

Es gibt viele Arbeiten, die sich mit DVFS (*Dynamic Voltage and Frequency Scaling*) beschäftigen, da Spannung und Frequenz in den Energieverbrauch quadratisch oder zumindest linear eingehen (siehe Gleichung 3, Seite 12). Da diese Verfahren inzwischen auch von kommerziell erhältlicher Hardware unterstützt werden, zielen die meisten Arbeiten darauf ab, die Energiespartaktiken (*Policies*) zu optimieren, um den Energieverbrauch weiter zu senken.

Zuerst wird auf Arbeiten eingegangen, die den Stromverbrauch, vor allem auch von Speichern, bestimmen. Danach folgen einige Arbeiten, die sich mit der Energieabschätzung befassen und sich dabei oft auf Zähler stützen. Abschließend wird am Ende dieses Kapitels auf verschiedene Beispiele zur Anwendung und Verfahrensweise eingegangen.

3.1. Ermittlung des Stromverbrauches

Um überhaupt eine Grundlage zu haben, warum es lohnenswert ist, sich speziell mit dem Energieverbrauch eines Speichers auseinander zu setzen, werden zunächst einige Arbeiten vorgestellt, die sich dieser Fragestellung von der praktischen Seite mittels Messungen gestellt haben. Danach soll auf Arbeiten eingegangen werden, die sich primär von der Theorie bzw. der Mathematik her angenähert haben, um dann ihre Ergebnisse mit Messwerten zu untermauern.

3.1.1. Messungen

Da Verbrauchsmessungen, vor allem des Speichers, ein Ziel dieser Diplomarbeit sind, möchte ich einige Arbeiten in diesem Gebiet vorstellen, die auf das Messen des Stromverbrauchs genauer eingehen. Der Messaufbau ist praktisch immer der gleiche, indem der Spannungsabfall an einem Shunt gemessen wird, kann bei bekannter Betriebsspannung der Stromfluss und damit der Verbrauch gemessen werden. Der Spannungsabfall wird meist noch verstärkt, digitalisiert und gespeichert, um für eingehendere Betrachtungen zur Verfügung zu stehen.

Barr et al. vermessen in [2] einen Handheld-Computer vom Stromverbrauch in Bezug auf Prozessor und Speicher. In diesem Experiment geht es um verlustlose Datenkompression, wobei festgestellt wird, dass ein Speicherzugriff das bis zu 200-mal mehr Energie verbraucht wie eine Berechnung. Hier wird nämlich die Anzahl der Speicherzugriffe und der arithmetischen Operationen mittels der Betaversion eines Simulationswerkzeuges („simple Scalar“) ermittelt. Außerdem ist bei den Tests auf der StrongARM Plattform insgesamt 64kB Cache aktiviert, so dass vor allem *Cache Misses* zu der gemessenen Energiebilanz des Speichers in dem System beitragen.

Das Paper von Weißel et al. ([41]) ist bemerkenswert, weil darin beschrieben wird, wie die aufgenommene Leistung eines Pentium4-Mainboards mittels eines *Shunt* und eines 8-Bit-AD-Wandlers mit 40ksps (*kilo Samples per Second*) gemessen wird. Mittels der so gewonnenen Messwerte werden dann über Differentialgleichungen aus der Physik die Wärmeentwicklung und damit der Energieverbrauch des Prozessors gesteuert. Diese Steuerung wird nicht nur bei einem Prozessor angewandt, sondern über das Netzwerk mit allen anderen Rechnern in einen Rechner-Cluster kommuniziert. Mit einem so geregelten Rechner-Cluster kann z.B. die Klimatisierung des Raumes effizienter dimensioniert werden, da die Kühlung nicht mehr für Lastspitzen ausgelegt sein muss.

In der Veröffentlichung von Farkas et al. ([12]) werden nicht nur qualitative Aussagen über Optimierungen in einem Taschencomputer und einer Java Virtual Machine (JVM) getroffen, sondern diese auch durch Messungen verifiziert. Diese erfolgen durch zwei Operationsverstärker, die den Spannungsabfall an einem Widerstand in der Hauptstromversorgung verstärken. Dieses Signal wird von einem 16-Bit-ADC (Analog-Digital-Wandler) digitalisiert und mit 5ksps abgespeichert. Dabei führen die Autoren Fehlerberechnungen und -messungen durch, um das unvermeidliche mit auf-

gezeichnete Rauschen und sonstige Messungenauigkeiten herauszurechnen. Darüber hinaus wird festgehalten, dass mit einem Cache die Ausführungsgeschwindigkeit proportional zur erhöhten Stromaufnahme ist, der Nutzen eines Caches aber nur bei darauf abgestimmten Zugriffsmustern der Anwendungen zum Tragen kommt. Eine weitere Feststellung ist, dass bei kleinen, vor allem tragbaren Computersystemen der Anteil des Speichers am gesamten Stromverbrauch deutlich größer ist, als bei größeren Rechensystemen.

Stitt und Vahid beschreiben im Aufsatz [38], wie mit rekonfigurierbaren Systemen je nach Anwendung 25% bis 71% Energie eingespart werden kann, indem geschwindigkeitskritische Teile der Software in rekonfigurierbare Hardware verlagert werden. Durch die größere Ausführungsgeschwindigkeit der Algorithmen in der Hardware, mittels Parallelisierung, Pipelining und anderen Beschleunigungsmethoden, kann Ausführungszeit gespart werden, was durch die kürzere Ablaufzeit auch Energieeinsparungen bewirkt. Der Artikel beschreibt zudem, wie neben einer Abschätzung wirkliche Messungen durchgeführt werden, indem Applikationen in einer lange laufenden Schleife ausgeführt werden, um stabile Messwerte mit einem Multimeter zu erhalten.

Die Quelle [16] beschreibt einen in Hardware implementierten Spannungscontroller, der aus einem *Hardware Performance Monitor*, einer Kontrolllogik und einem seriellen Interface besteht. Der Hardwarecontroller misst mit speziellen Testvektoren die Performance des SOC (*System on a Chip*) und regelt, je nach Leistungsanforderung der Applikation, die Versorgungsspannung in einer Reglerschleife, bis die gewünschte Ausführungsgeschwindigkeit erreicht wird. Zu dieser intelligenten Versorgungsspannungsregelung kommt noch eine Frequenzskalierung, so dass dieses System auch als DVFS-System eingestuft werden kann. Die meisten dieser Systeme können diese in Hardware implementierte Nachregelung jedoch nicht vorweisen können und schneiden dementsprechend schlechter beim Energiemanagement ab.

Auch Joseph und Martonsi ([27]) setzen prozessorinterne Performancezähler ein, um für den Stromverbrauch relevante Ereignisse zu zählen. Anstatt sich auf Simulationen zu verlassen, werden jedoch Messungen durchgeführt. Der Messaufbau ist praktisch identisch mit dem in dieser Diplomarbeit. Diese Arbeit schließt damit ab, dass es sich lohnen sollte, nicht nur die Register der Prozessoren zu überwachen, sondern auch den Speicher mit Zählern zu versehen, um die häufig benutzten Speicherbereiche herauszufinden. Denn nicht nur der Prozessor in einem System verbraucht Energie, sondern

auch die übrigen Komponenten, was die Autoren in ihrer Arbeit nicht weiter untersucht haben.

3.1.2. Rechnerische Verfahren

Der Verbrauch von Systemen lässt sich nur wirklich bestimmen, wenn nicht nur Messwerte, sondern auch Vergleichswerte bekannt sind, mit deren Hilfe bestimmt werden kann, ob Messungen in der richtigen Größenordnung liegen und ob alle an der Versorgung angeschlossenen Peripheriegeräte berücksichtig wurden. Auch zu solchen Abschätzungen gibt es spezielle Literatur, von der hier zwei beispielhaft aufgeführt werden sollen.

Der Artikel [26] von Joe ist eine Praxisanleitung um ein System mit einem Blackfin-Mikroprozessor nach dem zu erwartenden Stromverbrauch zu entwickeln. Es ist interessant, wie erfahrene Entwickler mit einfachen Abschätzungen doch sehr genaue Werte erreichen, deren Fehler zu den nachfolgenden Messungen unter 3% liegt, während manche Arbeiten mit deutlich größeren Abweichungen zu den Schätzwerten aufwarten. Es wird mit dieser Arbeit auch nahe gelegt, die Datenblätter nicht in jedem Fall als absolut genau zu erachten, da die wirkliche Stromaufnahme außer den Fertigungsstreuungen des „Siliziumstücks“ des Kunden auch noch von der Umgebungstemperatur, dem Kern und den Systemfrequenzen, Spannungsversorgungen, Anschlusskapazitäten, der Anwendungssoftware und der Nutzung von Peripherie abhängt. Dies bedeutet für diese Diplomarbeit, dass die ermittelten Messwerte nur für genau dieses eine Developmentboard gelten und nur prinzipiell übertragbar sind. Demzufolge könnten sich mit einer anderen Hardware andere Werte ergeben, deren Abweichungen nicht von Messfehlern verursacht werden.

Der Magazinartikel von Jansen ([25]) beschäftigt sich mit rechnerischen Verfahren, um den Stromverbrauch von Speichern zu ermitteln. Jansen stellt fest, dass sich die von den Herstellern von SDRAM angegebenen Werte in den Datenblättern unterscheiden, nicht aber die Verfahren und Konzepte, die zur Ermittlung dieser Werte führen. Diese werden in diesem Artikel ausführlich dargestellt, mit Hintergründen erläutert und anhand eines Beispielspeicherbausteines durchexerziert. Zu diesem Artikel gehört auch noch eine Tabellenkalkulationsvorlage, mit der anhand von Datenblattangaben und Schätzungen des Betriebszustand Betriebswerte von beliebigen DRAM-

Bausteinen bestimmt werden können, was sich vor allem bei der Abschätzung des statischen Verbrauches als praktisch erweist.

3.2. Energieabschätzung mit Hilfe von Zählern

Üblicherweise werden für die Abschätzung des Energieverbrauchs Zähler verwendet, da es zwischen den Zählerständen und dem Energieverbrauch einen Zusammenhang gibt. Im Folgenden werden deswegen einige Quellen genauer Beschrieben.

Mittels der Präsentation [8] von Choi, Pedram et al. wird eine Forschungsarbeit über DVFS vorgestellt, die um optimale Ergebnisse zu erzielen, in die XScale-CPU eingebaute „Performance Monitor“ Einheiten nutzt. Unter diesen Einheiten findet sich auch ein Zähler, der die Anzahl der externen Speicherzugriffe misst. Jedoch lässt sich dieser Wert nur indirekt aus anderen Werten berechnen (vgl. [21]) und unterscheidet nicht zwischen Lese- und Schreibzugriffen. Der aktuelle Stromverbrauch wird auch über einen Spannungsabfall an Widerständen bestimmt und mit 40ksps (*kilo Samples per Second*) abgetastet und aufgezeichnet. Die Präsentation von Pedram über das Apollo-Testprojekt ([37]) ist die Weiterführung der oben genannten Idee, wie die prozessorinternen Performancezähler eingesetzt werden, um DVFS-Verfahren zu implementieren. Jedoch werden in der letztgenannten Präsentation zusätzlich die Speichercontroller eines Systems durch rekonfigurierbare Hardware ersetzt und modifiziert, um den Speicher in das Powermanagement mit einbinden zu können.

In [4] wird von Bellosa festgestellt, dass die Auflösung von Leistungsmessgeräten zumeist nicht ausreicht, um den genauen Ort sowie den auslösenden Prozess des Energieverbrauchs festzustellen. Deswegen wird zur Verwendung von prozessorinternen Ereigniszählern ermuntert, um die Auflösungsgenauigkeit und die korrekte Feststellung des Initiators zu verbessern. In weiteren Ausführungen wird postuliert, dass SDRAM vor allem zum statischen Energieverbrauch beiträgt, der somit vor allem von der Größe der Speicher abhängt. Des Weiteren wird auch der Zusammenhang zwischen dem Energieverbrauch von SDRAM und den Zugriffscharakteristika (siehe Abschnitt 2.4, Seiten 13ff) erklärt und festgestellt, dass der optimale Betriebspunkt einer DFVS-CPU nur festgelegt werden kann, wenn die Speicherzugriffscharakteristik der Anwendungen genau bekannt ist. Außerdem kann es sein, dass durch das Umschalten von Komponenten in Stromsparmodi wegen der stark erhöhten Latenz beim „Aufwecken“ die

Energiesparbemühungen ad absurdum geführt werden. Es wird weiterhin behauptet, dass durch Trägheiten der Spannungsregulatoren und von AD-Messwertumsetzern ein Energieverbrauch nicht akkurat nachzuweisen ist. Daher wird das Anbringen von Zählern an Geräte empfohlen, die noch nicht von den prozessorinternen Ereigniszählern erfasst werden. Speziell zum SDRAM wird in Übereinstimmung mit anderen Quellen festgestellt, dass die Energie pro Speicherzugriff mit der Betriebsfrequenz steigt, aber mit der Zahl der Speicherreferenzen sinkt, was leicht mit den Erläuterungen aus Abschnitt 2.4 verständlich ist: Der Stromverbrauch bei CMOS ist proportional zur Umschaltfrequenz. Optimierungen im Speicher ermöglichen bei einem speziellen Referenzverhalten Einsparungen der Zugriffe und damit eine Absenkung der Umschaltfrequenz.

Frank Bellosa schlägt in [3] als neuen Ansatz für das Ressourcenmanagement Zähler im Speichersystem vor, vor allem um den Effekt der sog. *Memory Pre-Emption* zu entgegnen. Dieser Effekt bedeutet, dass sich parallel auf gemeinsamen Speicher zugreifende Subsysteme gegenseitig behindern. Anstelle von realen Speicherzugriffszählern werden jedoch prozessorinterne *Performance Counter* eingesetzt, um mittels *Cache Misses* auf Speicherzugriffe zu schließen. Jedoch werden fünf theoretische Stellen aufgezeigt, welche für das Platzieren von Speicherzugriffszählern geeignet erscheinen:

1. Im Prozessor selbst, und zwar für die einzelnen *Load*- und *Store*-Operationen.
2. Im *Cache Controller* für *Cache Misses*, *Cache Invalidations* und *Stall Cycles* (zur Erklärung dieser Begriffe siehe [17]).
3. Im Interface zwischen Prozessorcache und dem Bussystem (*Northbridge*).
4. Im Anschlusssystem der einzelnen Speicherbänke.
5. Am Interface von DMA-fähigen (*Direct Memory Access*) Ein-/Ausgabegeräten.

In dieser Diplomarbeit wird eine Mischung aus den Punkten 3 und 4 verwendet. Die Zähler sind an den Speicherkontroller angebracht, welcher aber an dem Bussystem (vom Prozessor aus gesehen) vor den Speichermodulen sitzt.
Am Ende gibt dieses Paper noch den Anstoß, sich mit dem Speichersystem genauer auseinander zu setzen, „da Rechnen mehr als das Ändern von Daten bedeutet“. Es bedeutet laut Bellosa nämlich sowohl Datenveränderung als auch das Bewegen von Daten.

3.3. Anwendungen und Verfahrensweisen

Selbstverständlich bleiben die oben vorgestellten Verfahren nicht reine Theorie, sondern werden inzwischen auf vielfältige Weise in der Praxis eingesetzt. Besonderes Augenmerk liegt dabei auf den Policies, die das Verhalten der Systeme maßgeblich bestimmen.

3.3.1. Softwarebasierte Ansätze

Außer den bis jetzt vorgestellten Arbeiten, die sich mit der Ermittlung des Stromverbrauches und Abschätzungen beschäftigt haben, wird das Thema des Energieverbrauches auch von der Softwareseite her angegangen, wofür stellvertretend die folgenden Papers stehen:

Einen stark softwareorientierten Ansatz wird von Neugebauer und McAuley in [34] beschrieben, indem sie mit einer optimierten Betriebssystemstruktur Energie genauso verwalten, als wäre sie eine ganz normale Hardwareressource. So kann das vorgestellte „Nemesis“-Betriebssystem die Energieverbraucher genau benennen, weil es von mehreren Verbrauchern genutzte Einheiten wie bestimmte Serverprozesse oder den Kernel eliminiert. Im Idealfall wäre dann noch zusätzlich Hardwareunterstützung vorhanden, um für jeden Tatbestand die Energie zu messen. Da es solch ein System nicht gibt, werden in dieser Arbeit Abschätzungen getroffen und mit Informationen aus einem SBS (Smart Battery System, siehe z.B. [29]) abgeglichen.

Im Paper [30] gehen Mamagkakis et al. davon aus, dass das dynamische Speichersystem heutzutage einer der hauptsächlichen Verbraucher an Energie ist und dadurch das Gesamtsystem stark beeinflusst. Deswegen werden Untersuchungen angestellt, um eine gegebene Aufgabe, in diesem Fall eine drahtlose Datenübertragung, mit möglichst energiesparenden Datenstrukturen zu lösen. Indem die Daten effizienter verwaltet werden, können Speicherzugriffe und damit Energie eingespart werden. Leider stellen die Autoren dieser Arbeit Ergebnisse vor, aber nicht, wie diese genau gewonnen werden.

3.3.2. Hardwareverfahren

Außer den Softwareansätzen gibt es auch Arbeiten, die sich dem Thema Energieverbrauch und Energiesparmaßnahmen von der Hardwareseite her nähern:

Fan et al. stellt in [11] Methoden vor, um den einen immer größeren Anteil am Stromverbrauch verursachenden SDRAM-Speicher effizienter zu verwenden. Dabei werden die von den Speicherbausteinen unterstützten Stromsparmodi so weit als möglich ausgenutzt. Es werden die Eigenheiten der Betriebsmodi *Active*, *Standby*, *Nap* („Schlummern“) und *Power Down* vorgestellt und auch die Nachteile der verlängerten Latenzzeiten beim „aufwecken“ der Speicherbausteine. Der Speichercontroller auf dem in dieser Diplomarbeit verwendeten Developmentboard unterstützt aber diese Stromsparmodi nicht, so dass die Ergebnisse aus diesem Paper hier keine Anwendung finden. Fan zieht als Fazit, dass in allen Systemen mit Caches die SDRAM-Bausteine nach jedem Zugriff sofort in einen Energiesparzustand versetzt werden sollten, da diese nicht von ausgefeilten Powermanagementstrategien profitieren.

Mit [33] stellt Bellosa et al. eine Arbeit vor, in der es vor allem um die Prozessmigration in SMP-Servern (*Symmetric Multi Processing*) geht. Zur Abschätzung des Energieverbrauches werden prozessorinterne Ereignisse mit den integrierten Performancezählern erfasst. Es wird vorgeschlagen, diese gezählten Ereignisse mittels Gewichtung in Leistung umzurechnen. Laut Bellosa können die Gewichte durch Messen des tatsächlichen Energieverbrauches mit einem Multimeter kalibriert werden. Im Ausblick wird darüber hinaus die Idee vorgebracht, die einzelnen funktionalen Einheiten eines Prozessors genauer zu untersuchen, um zum Beispiel der Entstehung von *Hot Spots* gezielter entgegensteuern zu können.

Die Arbeit [7] von Brock und Rajamani beschreibt die Arbeiten an einem DVFS-System, welches dem hier auf dem Developmentboard verwendeten System sehr ähnlich ist. Es geht um einen PowerPC 405LP mit SDRAM-Speicher. Das Powermanagement wird nicht nur von der Hardware ausgeführt, sondern darüber hinaus von einem angepassten Linux-Betriebssystem unterstützt. In dieser Arbeit wird daneben festgehalten, dass eine optimale Powermanagementstrategie nur mit einer aktiven Teilnahme von energiegewahren Tasks und einem Betriebssystem erreicht werden kann, welches die Fähigkeit zu Task- und ereignisspezifischem Power- und Performancemanagement besitzt. Um ein solches Management zu ermöglichen, sind Zähler wie in dieser Diplomarbeit geradezu prädestiniert, da die Implementierung im Linux-Betriebssystem auch hier prozessspezifisch abläuft. Das heißt, dass jedem Prozess die von ihm verursachten Zugriffe angerechnet werden können. Im zweiten Paper [35] von Brock et al. wird ebenfalls der PowerPC 405LP Prozessor in einem SOC (System on a Chip) verwendet, das ein DVFS-System implementiert. Dabei werden Messungen beim De-

kodieren eines MPEG-Datenstromes (Motion Pictures Experts Group), also beim Abspielen eines komprimierten Videofilmes durchgeführt, um die Leistungsfähigkeit der Stromsparalgorithmen bei einer typischen Anwendung zu untersuchen.

Von energiegewahren Caches für SOCs handelt die letzte Quelle von Weißel et al. [43]. Dieses Paper beschreibt, dass in den LEON-Prozessor (eine VHDL-Implementierung eines SparcV8-Prozessors) in das Cache- und in das Speicher-Subsystem Zähler integriert werden sollen. Die damit gewonnenen Ergebnisse sollen in zukünftige Designentscheidungen einfließen. Genau dies wird in dieser Diplomarbeit getan, nur dass aus Platzgründen im FPGA als Prozessor der integrierte PowerPC-Kern verwendet wird.

4. Architekturüberlegungen und Implementierungen

In diesem Kapitel wird zuerst auf die Grundlagen für die Bearbeitung der Aufgabenstellung in Form von VHDL und FPGAs eingegangen. Im Weiteren werden dann grundsätzliche Überlegungen vorgestellt, bevor mit der konkreten Beschreibung der Implementierung begonnen wird.

4.1. Hardwarebeschreibungssprache VHDL

In dieser Diplomarbeit wird mittels VHDL spezielle Hardware entwickelt, um Speicherzugriffe und Speicherbereichsreferenzen in einem rekonfigurierbaren Rechensystem energetisch zu betrachten. VHDL ist das Akronym für *Very High Speed Integrated Circuits Hardware Description Language*. Es gibt verschiedene Hardwarebeschreibungssprachen kurz HDLs, die sich mehr oder weniger gut für gewisse Aufgaben eignen. Die wichtigsten, um Hardware zu beschreiben, sind VHDL und Verilog, wobei VHDL in Europa die weiter verbreitete Sprache ist.

Laut [44] wurde VHDL in den frühen Achtzigern des vergangenen Jahrhunderts entwickelt und das US-Verteidigungsministerium hat die Einhaltung der Syntax zur Voraussetzung für die Erteilung von Aufträgen gemacht. Das Ziel war, die Dokumentation zu vereinheitlichen und den Datenaustausch von komplexen digitalen Systemen zu erleichtern. Deswegen wird auch oft davon gesprochen, dass VHDL eine „selbstdokumentierende“ Sprache ist. Dies macht aber sinnvolle Kommentare keineswegs überflüssig. VHDL wurde vom IEEE im Standard 1076 1987 genormt, der 1993, 2000 und 2002 überarbeitet wurde. Daneben wurde die Sprache in dem Standard IEEE 1076.1 (VHDL-AMS) um Möglichkeiten zur Beschreibung von Analogen- und Mixed-Signal-Systemen erweitert. Für praktische Tätigkeiten relevant sind jedoch momentan nur die Standards von 1987 und 1993, da die Erweiterungen der Sprache in den neueren Standards noch nicht von allen verbreiteten Synthesewerkzeugen vollständig unterstützt werden.

Nach [1] ist eine Hardwarebeschreibungssprache von einer Programmiersprache vor allem dadurch zu unterscheiden, dass eine HDL nicht zwingend einen sequentiellen Ablauf von Anweisungen festlegt, sondern ein Modell beschreibt. Sämtliche Anweisungen laufen zunächst einmal parallel voneinander ab. Dadurch kann VHDL die

Möglichkeiten von realer Hardware überhaupt erst nachbilden. Wird eine Schritt-für-Schritt-Abarbeitung von bestimmten Zuweisungen oder Algorithmen gewünscht, muss dies explizit beschrieben werden (`function`, `process`). Darum gibt es auch zwei grundlegend verschiedene Arten der Datenübertragung und Speicherung. Zum einen die Signale (`signal`), die immer erst am Ende eines Zuweisungsblockes den letzten ihnen zugewiesenen Wert annehmen. Zum anderen die Variablen (`variable`), welche einen zugewiesenen Wert sogleich annehmen und somit eher den aus konventionellen Programmiersprachen bekannten Variablen entsprechen.

Der Aufbau einer VHDL-Datei ist im Wesentlichen immer identisch. Zuerst werden die verwendeten Bibliotheken und Packages eingebunden, was bei dem in dieser Arbeit realisierten Chunkzähler folgendermaßen aussieht:

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  library proc_common_v1_00_b;
7  use proc_common_v1_00_b.proc_common_pkg.all;
```

An dieser Stelle wird darauf hingewiesen, dass VHDL keine Unterscheidung zwischen Groß- und Kleinschreibung vornimmt. Trotzdem ist es Konvention die Schlüsselworte, Variablen und Bezeichner durch die Groß- und Kleinschreibung zu unterscheiden. Es bleibt jedoch dem Programmierer überlassen, welchen Stil er bevorzugt.

Danach wird eine `entity` (Funktionseinheit) angelegt, wobei es üblich ist, dass die Quelltextdatei genauso wie die Entity heißt. In dieser werden Konstanten (`generic`) mit Zahlenwert und Datentyp definiert, während in der `port`-Deklaration die Eingänge und Ausgänge mit ihrem Namen, der entsprechenden Signalart und Breite festgelegt werden, was im Beispiel so aussieht:

```
8  entity SEITE is
9            generic(
10                C_CHUNKZAEHLER : integer := 32;
11                C_CHUNKLOG     : integer := 5; --=ld 32
12                C_CHUNKZB      : integer := 32;
13                C_ZAEHLBREITE  : integer := 32
14           );
15           port (
16                CLK       : in std_logic;
```

```
17                RST       : in std_logic;
18                ENABLE    : in std_logic;
19                CNT       : out std_logic_vector(0 to C_CHUNKZB-1)
20          );
21 end entity SEITE;
```

In diesem Beispiel wird in Zeile 11 ein Kommentar verwendet, der an einer beliebigen Stelle mit „- -“ eingeleitet wird und sich bis zum Ende der Zeile erstreckt. Außerdem gibt es noch „inout“-Ports, die einen bidirektionalen Datenfluss erlauben. Auf die Entity folgt anschließend die Architektur (`architecture`), in der die Entity „mit Leben gefüllt“ wird. Die Architektur kann sehr unterschiedlich aufgebaut sein, je nachdem, was die Entity beschreiben soll. In der Architektur können Hilfssignale und Variablen definiert werden, Untermodule, die wiederum aus VHDL-Dateien bestehen, definiert und instantiiert werden und/oder parallel oder sequentiell ablaufende Zuweisungen stehen. Durch diese große Vielfalt ist die Sprache sehr mächtig, so dass der Entwickler sehr diszipliniert arbeiten muss, um nicht den Überblick zu verlieren.

Der Vollständigkeit des Beispiels halber sei hier noch die Architektur eines einzelnen Chunkzählers angegeben, die aus einem sequentiell ablaufenden Prozess und einer einzigen parallelen Zuweisung besteht. Da dieser Abschnitt kein Kurs in VHDL sein kann und soll, sei für eine deutlich vielfältigere Architektur auf das Listing der Datei „user_logic.vhd“ im Anhang B.1 verwiesen.

```
22 architecture SEITE_ARCH of SEITE is
23      constant EINSEN     : std_logic_vector(0 to C_CHUNKZB-1) := (
           others => '1');
24      signal STAND        : std_logic_vector(0 to C_CHUNKZB-1);
25      signal MERKER       : std_logic;
26
27 begin
28
29      PG : process (ENABLE, CLK) is
30      begin
31           if (CLK'event and CLK = '1') then
32                if (RST = RESET_ACTIVE) then
33                     STAND     <= (others => '0');
34                     MERKER    <= '0';
35                elsif (ENABLE = '1') then
36                     if (MERKER = '0') then
37                          MERKER    <= '1';
38                          STAND     <= STAND + 1;
39                     end if; -- MERKER
40                elsif (ENABLE = '0') then
41                     MERKER    <='0';
```

```
42                end if; --RST
43                if (STAND = EINSEN) then
44                     STAND     <= (others => '0');
45                end if; -- STAND
46           end if; -- CLK
47      end process PG;
48
49      CNT <= STAND;
50
51 end architecture SEITE_ARCH;
```

Die große Anzahl der Möglichkeiten erlaubt es in VHDL, je nach Abstraktionsebene des Modells auf Gatterebene, mit einzelnen elektronischen Bauelementen (RTL „Register-Transfer-Logik“) bis hin zu Verhaltensbeschreibungen zu arbeiten. Durch die Black-Box-Abstraktionen wird das Arbeiten auf verschiedenen Abstraktionsebenen in einem Projekt und ein *Top-Down Design Flow* ermöglicht. Dadurch ist mit VHDL das relativ schnelle Entwickeln großer und komplexer (digitaler) Schaltungen möglich, die vom Zeitbudget als auch von der ökonomischen Seite her hohe Effizienz erfordern. Zum Vergleich: Das hier zum Teil entwickelte Gesamtsystem hat eine „Total equivalent gate count“ von 2.603.121 Gattern. Ohne Softwareunterstützung hätte dieses Projekt nicht binnen eines halben Jahres soweit vorangetrieben werden können.

Nach der Beschreibung kann ein System simuliert, verifiziert und schließlich eine Netzliste erstellt werden. Das Erstellen einer Netzliste besteht aus der Synthese sowie dem Place-and-Route und dem Technology-Mapping. In VHDL kann man sowohl synthetisierbaren Code schreiben als auch nicht synthetisierbaren. Der zweite Fall kann gewollt sein, indem eine sog. *Test Bench* (Prüfstand) für die Simulation und Verifikation einer VHDL-Quelle geschrieben wurde, oder auftreten, weil das Synthesewerkzeug das gewählte Sprachkonstrukt nicht unterstützt oder weil der Entwickler einen Fehler gemacht hat. Aus einer Netzliste können die Masken für die Produktion von ICs gewonnen werden, oder nach einer Konvertierung die Programmierdateien für einen FPGA (Field Programmable Gate Array) oder ein PLD (Programmable Logic Device).

4.2. Field Programmable Gate Arrays

Ein FPGA ist laut [44] ein frei programmierbarer Logikschaltkreis. Ein FPGA besteht prinzipiell aus konfigurierbaren Logikblöcken (CLBs, *Configurable Logic Block*,

deswegen „Field“), Leiterbahnen zur Verbindung dieser Blöcke und Ein- und Ausgabetreibern der I/O-Pins des Gehäuses. Außer den Logikblöcken existieren je nach Hersteller und Modell auch SRAM-Blöcke, so genannte BRAM-Blöcke (*Block RAM*), PLLs (*Phase Locked Loop*), DLLs (*Delay Locked Loop*), Taktaufbereitungen (DCM, *Digital Clock Manager*) sowie einfache ALUs (*Arithmetic Logic Unit*) bis hin zu Prozessorkernen, wie in dem hier verwendeten Xilinx Virtex2 Pro. Da die CLBs der FPGAs einen synchronen Takt erhalten müssen, um unkalkulierbare Laufzeitunterschiede zu vermeiden (synchrones Design), sind Taktverteilerbäume notwendig. Es gibt sowohl wiederprogrammierbare flüchtige (SRAM-basierte), nicht flüchtige wiederprogrammierbare (Flash basierend)) als auch nur einmal programmierbare (One Time Programmable, Antifuse-Technologie) FPGAs.

Ein FPGA wird in [39], S. 746ff als ein vom Anwender selbst programmierbares Schaltwerk beschrieben, das von sich aus Blöcke aus Gattern, Flip-Flops und teilweise PLDs enthält, mit denen sehr komplexe Schaltungen entworfen werden können. FPGAs sind besonders dann vorteilhaft, wenn sich die Aufgabe schlecht auf einen normalen PLD, die oft nur Schaltnetze zulassen, abbilden lässt. Im Gegensatz zu vielen anderen programmierbaren Schaltungen werden bei FPGAs die Verbindungen zwischen den einzelnen Blöcken mit der so genannten „Programmable-Link-Technik“ auf dem Chip zu der gewünschten Schaltung verbunden.

Mehrere CLBs werden zu so genannten Slices zusammengefasst. Ein CLB besteht aus LUTs (*Look Up Table*), Registern, Multiplexern, einer IO-Matrix. Eine kombinatorische Logik mit Operationen wie AND, OR, NOT, XOR wird in den LUTs dargestellt. Diesen sind Register nachgeschaltet. Alle Ein- und Ausgänge innerhalb des CLB führen zur IO-Matrix, welche frei programmierbare Verbindungen bereitstellt. Die kombinatorische Logik wird bei den hier verwendeten FPGAs des Markführers Xilinx durch LUTs mit vier Eingängen und einem Ausgang gebildet. Eine LUT enthält einen 16x1 Bit-RAM-Speicher, dessen Programmierung alle binären Ausgangswerte eines Ausgangs bei vier Eingangswertekombinationen zulässt. In den CLBs sind Flipflop-Register, in Form von D-Latches. Die der kombinatorischen Logik nachgeschalteten Latches können mit Hilfe der Multiplexer genutzt oder umgangen werden.

FPGAs unterscheiden sich von den einfacheren CPLDs (*Complex PLD*) vor allem durch den Aufbau aus Blöcken und dem frei programmierbaren Signalfluss. Somit sind exakte Gatterdurchlaufzeiten nur mittels aufwendiger Simulation zu ermitteln.

Von ASICs (*Application Specific Integrated Circuit*) unterscheiden sie sich laut [44] in folgenden Punkten:

- geringe Entwicklungskosten
- sehr kurze Implementierungszeiten
- einfach korrigier- und erweiterbar (rekonfigurierbar)
- geprüftes Silizium
- geringeres Designrisiko, da es nicht 6 Monate vor der Hardwareauslieferung fertig sein muss
- ab mittleren Stückzahlen höherer Stückpreis
- geringere Taktraten (max. ca. 500 MHz)
- geringere Logikdichte (ca. 10facher Flächenbedarf gegenüber ASIC gleicher Technologie)
- höherer Energiebedarf

4.3. Grundlegende Überlegungen

Die Aufgabenstellung war zunächst, Zugriffszähler zu implementieren, die die Schreib- und Lesezugriffe des Prozessors in den SDRAM-Speicher mitzählen. Wenn nun geeignete Programme ablaufen, die z.B. nur lesend auf den Speicher zugreifen, kann mit den in Abschnitt 5 gezeigten Verfahren die Mehraufnahme von Strom im Vergleich zum Ruhezustand gemessen werden. Mit der von den Zugriffszählern gemessenen Anzahl der Zugriffe kann dann eine einfache Rechnung durchgeführt werden, um den Leistungsbedarf eines einzelnen Lesezugriffs zu ermitteln. Das gleiche Verfahren kann bei Schreibzugriffen angewendet werden. Dabei wäre es ideal, wenn das Programm selbst in einem anderen Speicher, wie den Block-RAMs, liegen würde, da ja jeder Schritt im Programm auch einen Lesezugriff auf eine Instruktion darstellt. Dies beeinflusst die Stromverbrauchsmessung der Schreibzugriffe, da nicht ausschließlich schreibend zugegriffen wird.

Nach der Implementierung der Zugriffszähler, die hier als zwei gleichartige 32-Bit-Zähler mit jeweils 16-Bit-Überlaufzählern, also effektiven 48Bit implementiert wurden, blieben noch genügend freie Ressourcen auf dem FPGA übrig, um weitere Untersuchungen mit Hilfe von rekonfigurierbarer Hardware durchzuführen.

Im Hinblick auf einschlägige Quellen (siehe Abschnitt 3, Seite 20f) wurde vorgeschlagen, „Seitenzähler“ einzurichten, welche die Lokalität der Programme und des Linux-Betriebssystems erfassen sollen. Dies erwies sich als aussichtslos, da diesem Vorhaben zwei Probleme im Weg standen:

- Bei einer typischen Seitengröße von 4kB wären bei 32MB RAM 8192 Seiten zu zählen gewesen, was zu ebenso vielen Zählern geführt hätte. So viele Ressourcen stehen im FPGA nicht zur Verfügung.
- Zum Zweiten sind mit den Assistenten von Xilinx nur maximal 32 je 64Bit breite verschiedene Ausgangsregister möglich, was an der Implementierung des Multiplexers für den Zugriff auf die 32 einzelnen Registerinhalte liegt.

Deshalb wurde angedacht, aus den Seitenzählern *Chunk Counter* („Blockzähler“) zu machen. Dies bedeutet, dass nicht einzelne Seitenzugriffe gezählt werden, sondern Zugriffe auf zusammenhängende Bereiche des Speichers.

Zuerst wurde eine Chunkgröße von 512kB ausgewählt, was zu 64 Zählern führte. Diese Menge an Zählern nutzt die Hardware des FPGA fast vollständig, wie im nächsten Absatz beschrieben. Da für diese Zähler neben den Zugriffszählern in den Ausgangsregistern nicht genügend Platz besteht, um sie ebenfalls 32Bit breit zu machen, und die Zähler dann auch von der Softwareseite aus einigermaßen effizient auslesbar sein sollen, wurden jeweils drei Zählerstände in einem 64-Bit-Register zusammengefasst. Jeder Zählerstand war somit 21 Bit breit, somit blieb nur jeweils ein Bit je Register ungenutzt. Insgesamt wurden dabei von den 32 Registern mit den Seitenzählern, den Overflow-Zählern und dem reservierten Wort $\frac{4+64}{3} = 26$ der 32 Register genutzt.

Die komplexe Verkabelung und die vielen 21-Bit Zähler führten nach Durchlauf der Synthese zu 7% ungenutzten Slices des FPGA. Um länger zählen zu können, konnten die Zähler wegen des praktisch vollen FPGA nicht breiter gebaut werden. Es lässt sich mit einfacher Rechnung zeigen, dass ein 21-Bit-Zähler im schlimmsten Fall bei 100MHz Taktfrequenz binnen 21ms überläuft. Ist die Zahl der Überläufe jedoch größer als eins, kann dies nicht erkannt und vor allem dem korrekten Zähler und damit

Speicherbereich zugeordnet werden. Diese Lösung erwies sich angesichts der erwarteten Lokalität der Speicherzugriffe als ungeeignet.

Aus den oben genannten Gründen wurden die Zähler auf je 32Bit verbreitert und damit auf 32 Stück beschränkt, die dann die Zugriffe auf einen Chunk von je einem MB Größe erfassen. Damit können je zwei Zähler in ein Ausgaberegister gesteckt werden, was zu einer Nutzung von insgesamt 20 Registern führt. Nach Ablauf der Synthese kann festgestellt werden, dass das Synthesewerkzeug offensichtlich Kenntnis vom optimalen Synthetisieren von 32-Bit-Zählern hat, da nach der Synthese noch 12% der FPGA-Slices unbenutzt sind. Obwohl diese Lösung die zur Verfügung stehenden Ressourcen nicht komplett ausnutzt, wird dennoch an ihr festgehalten, da die Softwareauswertung relativ unkompliziert ist und ein 32-Bit-Zähler im *Worst Case* immerhin circa 43 Sekunden bis zum Überlauf braucht. Der zugehörige Kerneltreiber macht bei der Auswertung aus den 32 bzw. 48Bit der Zähler je 64Bit (Datentyp: „unsigned long long int“). Deshalb stellt ein möglicher Überlauf auch kein ernsthaftes Problem mehr da: Der Software bleibt ja fast eine dreiviertel Minute Zeit, um zwischenzeitlich die internen 64-Bit Zähler zu aktualisieren.

4.4. Detaillierte Erläuterung der Implementierung der Zugriffszähler

Um hardwareseitige Speicherzugriffe, zu zählen, werden folgende Einzelteile und Signale benötigt, die jeweils gesondert erläutert werden:

- Takt: Der Power PC, der Speicher und die ganze Logik im FPGA ist getaktete digitale Hardware. Deswegen werden auch die Zähler an das Taktnetz angeschlossen und synchron zum Takt betrieben. Nach der Synthese konnte für die Zugriffszählereinheit eine geschätzte Ausführungsgeschwindigkeit von 184MHz gemessen werden, was noch 84% über der Taktfrequenz des Gesamtsystems liegt. Diese Zahl ist nur eine Schätzung, da nach der Synthese noch das *Place & Route* kommt, was die real erreichbaren Taktfrequenzen noch ändern kann. Im Normalfall kann allerdings davon ausgegangen werden, dass in der Realität eher höhere Betriebsfrequenzen erreicht werden. Nur, wenn die Syntheseschätzung schon unterhalb der gewünschten Betriebsgeschwindigkeit liegt, sollte nachgebessert werden. So kommt nach der Synthese für das Gesamtsystem eine max.

Betriebsgeschwindigkeit von 99,246MHz heraus. Der „kritische Pfad“, also der Pfad mit durch die Schaltung mit der längsten Verzögerung und einer Verbindung zum Datenpin 13 des Speichers war nicht durch die in dieser Diplomarbeit vorgenommenen Änderungen der Hardware beeinflusst. Trotzdem läuft das Gesamtsystem mit 100MHz stabil.

- Reset: In der gesamten PowerPC-Architektur auf dem FPGA ist der Reset ein sogenannte synchroner Reset, das heißt er wird nur an der steigenden Flanke des Taktsignals erkannt. Außerdem ist er ein „Active High Reset“. Normalerweise ist ein Reset immer asynchron und „active low“, damit zu völlig beliebigen Zeiten ein Reset erfolgen kann. Vor allem nach dem Anlegen von Betriebsspannung startet das System so in einem definierten Zustand. Es lag überall ein Low-Pegel an. Der Kaltstart für das System ist also dasselbe wie ein Reset, der über die Defaultwerte alles in einem definierten Zustand bringt. In diesem Fall werden bei einem Reset alle Zähler auf „0“ zurückgesetzt sowie die Signale für die vergangenen Lese- oder Schreibindikatoren gelöscht, d.h. auf „0“ gesetzt.
- Zähler: Ein Zähler ist in VHDL relativ einfach zu beschreiben, da die Umwandlung von std_logic nach integer und zurück implizit eingebaut ist. Das heißt, es wird ein Signal als Zählerstand definiert. In dem Zählprozess wird diesem Signal dann der eigene Wert „+1“ bzw. „-1“ zugewiesen. Das inkrementierte bzw. dekrementierte Signal ist der neue Zählerstand, der dann dem Ausgangszählerstandsvektor übergeben wird. Das Ausgangssignal kann dann an anderer Stelle weiterverarbeitet werden.

```
1 ...
2       RD_CNT              : out std_logic_vector(0 to 31);
3 ...
4       signal RD_ZAEHL     : std_logic_vector(0 to 31);
5 ...
6             RD_ZAEHL      <= RD_ZAEHL + 1;
7 ...
8       RD_CNT    <= RD_ZAEHL;
```

Natürlich muss bei einem Zähler darauf geachtet werden, dass er im Reset-Fall ordentlich zurückgesetzt wird und dass er auch bei einem Overflow d.h. der Zählvektor ist binär voller Einsen) sicherheitshalber zurückgesetzt wird. Bei VHDL ist es immer wichtig, alle Fälle zu betrachten, damit die synthetisierte Hardware einen nicht definierten Zustand erreichen könnte. Falls ein Zustand nicht erreicht wird, optimiert diesen das Synthesewerkzeug mit Sicherheit weg.

In der Implementierung (siehe Listing 1, Seite 82) wird bei einem Überlauf der Overflowzähler um eins inkrementiert und der Zähler zurückgesetzt. Zu guter Letzt muss noch beachtet werden, dass der Zähler nur dann inkrementiert wird, wenn es auch wirklich einen Grund dafür gibt. Das führt zur Betrachtung der Signale für den Lese- und Schreibzugriff weiter unten.

Üblicherweise werden für die meisten Zwecke Rückwärtszähler verwendet, da diese im Normalfall in schnellere Hardware übersetzt werden. Da Zähler allerdings zu den Standardmakros gehören, die von den Synthesewerkzeugen erkannt und somit üblicherweise durch vorgefertigte, optimierte Hardware ersetzt werden, wurde keine besondere Aufmerksamkeit auf eine möglichst schnelle Implementierung gelegt. Deswegen wird ein Aufwärtszähler eingesetzt, was bei dem späteren Auslesen und Auswerten Umrechnungen spart.

- Zugriffssignale: Damit wirklich Lese- und Schreibzugriffe auf den SDRAM-Speicher gezählt werden können, müssen diese Ereignisse dort erfasst werden, wo sie sicher von Lesezugriffen auf Peripheriegeräte oder eventuell aktivierte Caches oder Block RAMs unterschieden werden können. Daher bleibt nur der SDRAM-Speicherkontroller übrig, der als Verbindungsstück von PLB (Processor Local Bus) und den außerhalb des FPGA liegenden SDRAM-Speicherbausteinen fungiert. Abb. 8, Seite 39, zeigt den logischen Aufbau des Systems. Markiert ist der PowerPC-Kern, der Speichercontroller und das hier in dieser Diplomarbeit implementierte Modul „PLBADDON“, in dem sich die einzelnen Zähler verbergen. Die Abbildung 9, Seite 40, zeigt den schematischen Aufbau des mitgelieferten Speicherkontrollers, der im Quelltext vorliegt. In diesem befindet sich auch die FSM (*Finite State Machine*), also der endliche Automat für die *Data State Machine*. Am Automatengraph der Data State Machine (vgl. Abb. 10, Seite 41), kann erkannt werden, dass dort die Signale „pend_read“ und „pend_write“ verwendet werden, um in den Read- bzw. Write-Zustand zu wechseln. Diese beiden Signale werden als Indikatoren für einen Lese- oder Schreibvorgang verwendet, indem beide über die Signale read_indicator und write_indicator in die oberen Hierarchien zurückgeschleift werden und im Toplevel mit den entsprechenden Eingängen des PLBADDON verbunden werden. Somit stehen in den Zugriffszählern die pend_*-Signale über eine reale physikalische Leitungsverbindung zur Verfügung.

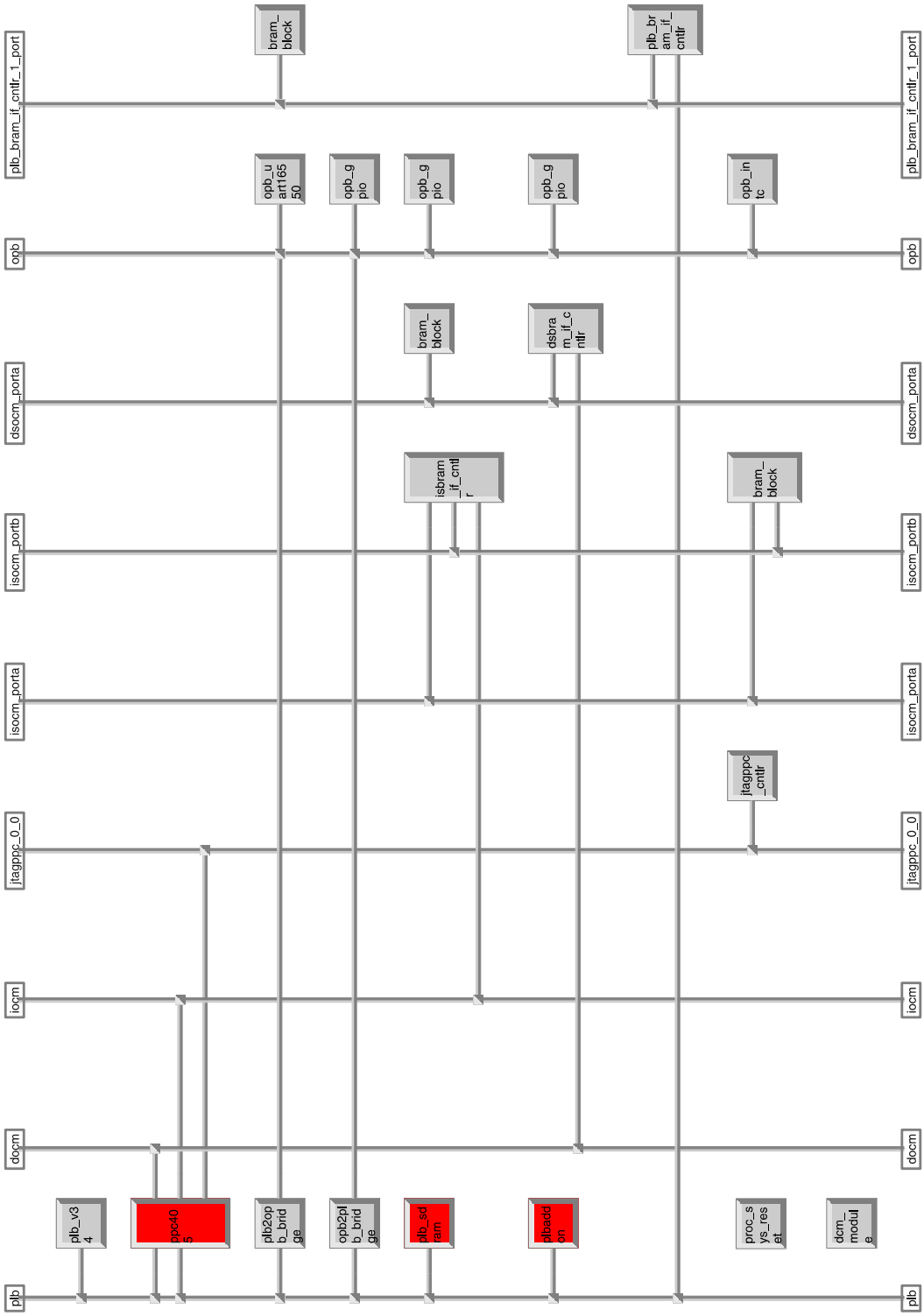

Abbildung 8: Module und Busverbindungen des synthetisierten Systems

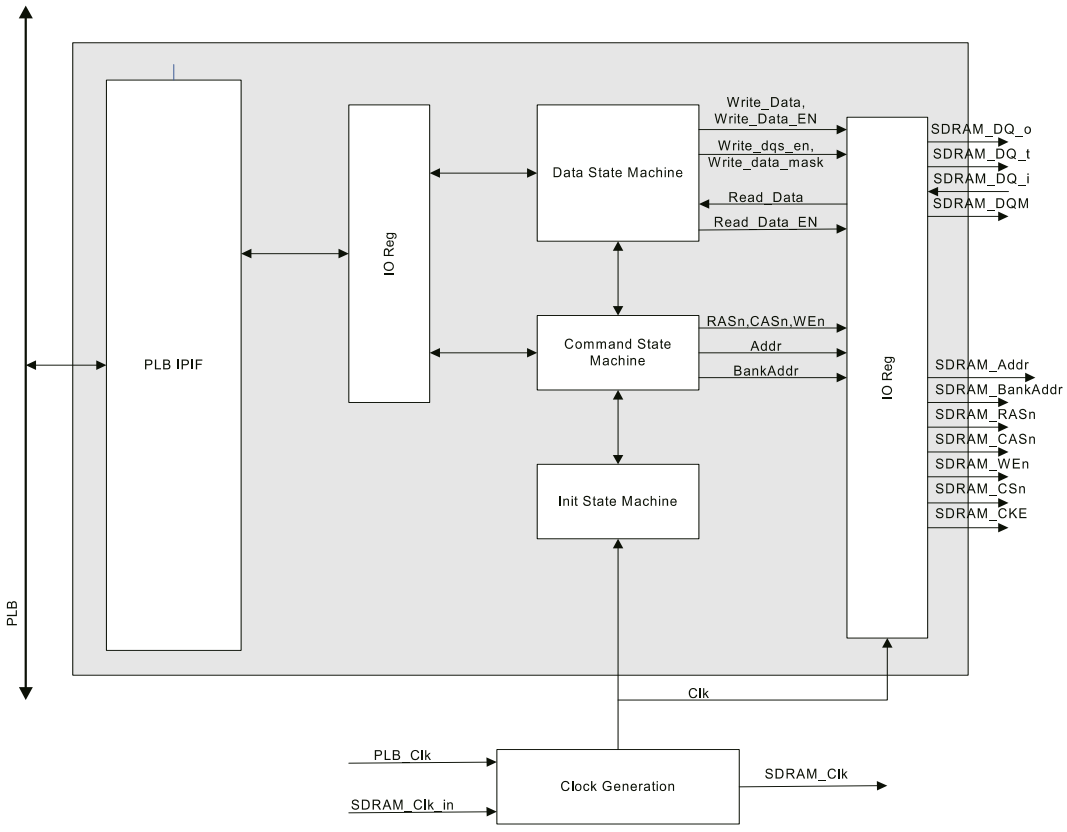

Abbildung 9: Blockskizze des SDRAM-Controllers (aus [47])

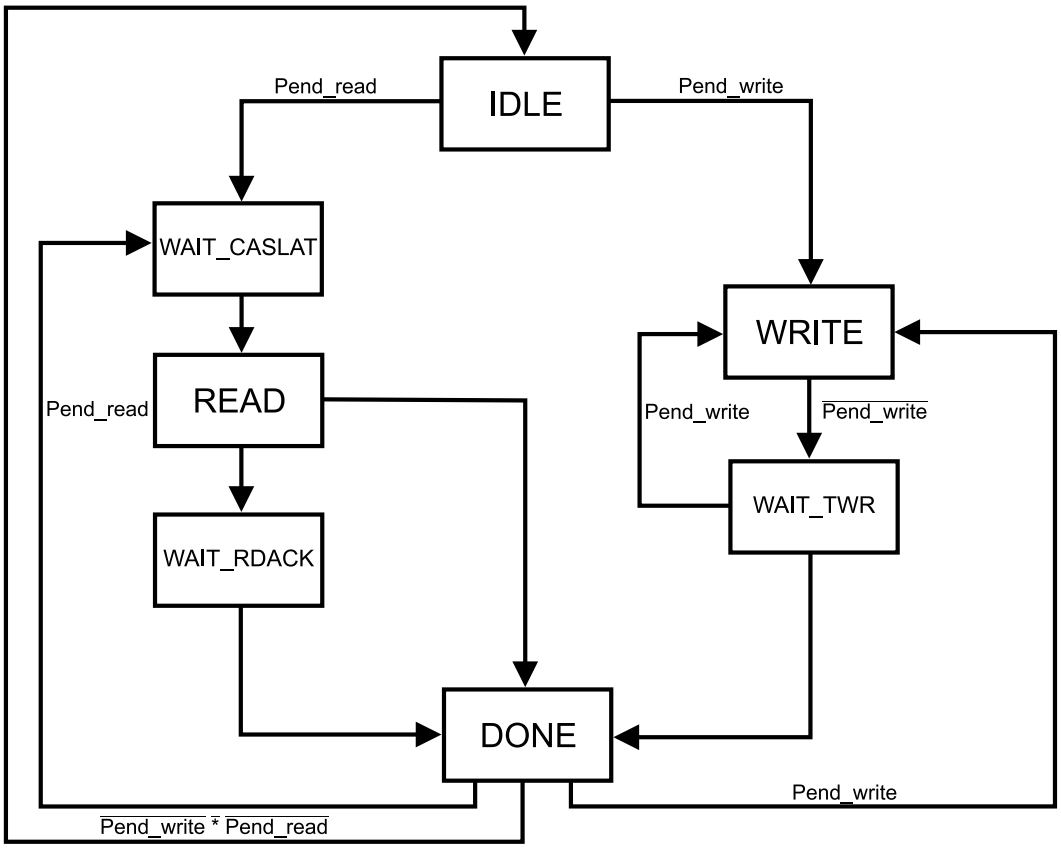

Abbildung 10: Automatengraph der Data-Statemachine (aus [47], vereinfacht)

- Ausgaberegister: Das Ansteuern der Memory-Mapped Register, in denen die Zählerstände abgefragt werden können, ist wohl das schwierigste Teilproblem. Es gibt jedoch im EDK einen „Create and Import Peripheral Wizard“, der Vorlagen für die Anbindung von eigenen IP-Cores, so heißen bei Xilinx selbst erstellte Zusatzmodule, an den PLB erstellt. Optional erstellt dieser Wizard auch ein funktionsfähiges Beispiel für die Anbindung von Registern, die gelesen und beschrieben werden können. Diese Vorlagen mussten nur noch etwas abgeändert werden, um für die Implementierung der Zähler geeignet zu sein.

4.5. Details zu den Chunkzählern

Die Chunkzähler wurden geschaffen, um die Lokalität des Betriebssystems und der Programme untersuchen zu können. Lokalität bedeutet in diesem Fall, dass immer wieder nur auf bestimmte Speicherbereiche zugegriffen wird und auf andere fast nie.

Zu den 32-Bit Zählern muss nicht viel erklärt werden, da sie mit den im Abschnitt 4.4, Seite 36 erklärten im Prinzip identisch sind. Der Unterschied besteht darin, dass es keine Overflow-Behandlung gibt, sondern die Zähler einfach wieder auf „0“ zurückgesetzt werden. Außerdem sperren sie sich nach einer Inkrementierung selbst, bis das Enable-Signal wieder auf „0“ gefallen ist, weil sie sonst falsche Zählerstände generieren würden.

```
1       PC : for i in 0 to C_CHUNKZAEHLER-1 generate
2            INSTANZ : SEITE port map(
3                 CLK            => Bus2IP_Clk,
4                 RST            => Bus2IP_Reset,
5                 CNT            => INT_CNT(i),
6                 ENABLE         => ENABLE_VECTOR(i)
7                 );
8       end generate PC;
```

Die Besonderheit an den Chunkzählern ist die Art ihrer Instantiierung, wie im oben angegebenen Codebeispiel erkennbar. Sie werden mittels `generate` eingebunden, was dafür sorgt, dass der eine Zähler in 32 Kopien bei der Synthese eingegliedert wird. Der Einzige Unterschied zwischen den verschiedenen Zählern ist ihre Verkabelung mit den Zählerstandausgängen und dem Enable-Vektor, welche über Arrayindices dem jeweiligen Zähler bei der Instantiierung zugewiesen werden. Somit hat jeder Zähler in den Vektoren seinen eigenen, ihm zugewiesenen Platz. Das ist sehr vorteilhaft, weil dann

im weiteren Verlauf einfach mit Schleifen über die Vektoren bzw. Arrays iteriert werden kann, um weitere Zuweisungen zu erledigen. Besondere Beachtung sollte dem Enable-Signal gewidmet werden, denn nur wenn dieses aus dem ENABLE_VECTOR übernommene Signal aktiv ist, darf der entsprechende Zähler einen Zugriff auf „seinem“ ein Megabyte großen Speicherbereich zählen, ansonsten erhöht sich der Zählerstand nicht. Die Erzeugung des geeigneten Einschaltsignals wird im nächsten Abschnitt erläutert.

4.6. Der Enabler

Wie zuvor schon angesprochen ist die Aufgabe des Enablers, bei einem Zugriff auf einen bestimmten Speicherbereich den korrespondierenden Zähler zu aktivieren, indem letztendlich das richtige Bit im ENABLE_VECTOR gesetzt wird.

Um diese Aufgabe erledigen zu können, braucht die Komponente aber weitere Informationen als Eingabe. Und zwar einmal die Adresse im Speicher, auf die zugegriffen wird und zum zweiten das Signal dafür, dass die Adresse gültig ist, d.h. ein Zugriff erfolgt.

In der Dokumentation zum Speichercontroller, [47] S. 20ff, sind Timingdiagramme für den Speicherzugriff abgebildet, in denen sich die Quellen für diese Information leicht ausmachen lassen. Diese beiden Signale werden in dem PLB2IPIF-Modul (*Processor Local Bus to Intellectual Property Interface*) abgegriffen, das die Verbindung zwischen dem PLB und dem Speichercontroller herstellt.

- Das PLB_PAvalid-Signal deutet an, dass die auf dem PLB angelegte Adresse gültig ist. Ein physikalischer Zugriff auf den Speicher findet dann wenige Taktzyklen später statt. Zu beachten ist, dass dieses Signal für drei Taktzyklen anliegt und durch die feste Verbindung die Seitenzähler selbstsperrend nach einer Inkrementierung sein müssen.

- Die mit PLB_ABus(0 to 31) angelegte 32-Bit-Adresse wird ausgewertet und nach der Rechenvorschrift [47], S. 10f, in Bankadresse (Bits 7 und 8), Reihenadresse (Bits 9 bis 20) und Spaltenadresse (Bits 21 bis 28) zerlegt. Für die 32 Zähler werden die Bits 7 bis 11 aus der angelegten Adresse verwendet, da die so gewonnene 5-Bit-Adresse mit einem der 32 jeweils ein Megabyte großen

zusammenhängenden Chunks im physikalischen Speicher korrespondiert. Dieser Teil der Adresse wird in einen Integer umgewandelt und in einer Schleife verglichen, ob der Integerwert mit dem Schleifenindex übereinstimmt. Ist dies der Fall, wird das Bit im ENABLE_VECTOR mit dem Schleifenindex gesetzt, ansonsten gelöscht. Um keine Adressen auf dem PLB fälschlicherweise zu erfassen, die nicht nur den physikalischen Speicher adressieren, wird beim Abgreifen aus dem PLB2IPIF geprüft, ob die Bits 0 bis 6 und 29 bis 31 auch wirklich „0“ sind (vergleiche Listing 4, Seite 89).

Im Listing 3, Seite 88, ist der komplette Quelltext des Enablers zu finden.

4.7. Auswertalgorithmen und Kerneltreiber

Nachdem nun schon ausführlich berichtet wurde, wie die Hardware zusammengestellt wurde, soll hier nun beschrieben werden, wie die Hardwarezähler für Software unter dem Linux-Betriebssystem bereitgestellt wurden.

Zuerst wurde ein Kernelmodul (`treiber-driver.c`, siehe Listing B.2, Seite 94ff) geschrieben, das aber durch einfache Veränderungen im *Makefile* fest in den Kernel eingebunden werden kann, was später für die Zusammenarbeit mit dem *Scheduler* notwendig war. Das Kernelmodul ist dafür zuständig, den Adressbereich für die *Memory-Mapped-Register* zu reservieren und einen Eintrag im /proc-Pseudodateisystem bereit zu stellen.

In der Datei /proc/PLBADDON/PLBADDON finden sich die absoluten Zählerstände der Zugriffs- und der Chunkzähler seit dem letzten Reset. Diese Auflistung alleine genügt aber wegen dem Multitasking unter Linux nicht, um einzelnen Prozessen die von ihnen verursachten Zugriffe zuzuordnen, da ständig andere Prozesse im System aktiv sind, und ebenfalls auf den Speicher zugreifen.

Das Kernelmodul und alle anderen Dateien, die sich mit den Zählerständen befassen, binden die Datei `funktionen.h` ein (siehe Listing B.2, Seite 95), die alle Funktionen vom Auslesen des Speichers bis zum Abspeichern des aktuellen und vorangegangenen Zählerstandes in einer globalen Struktur in sich vereint. Zusätzlich wird dort auch eine Struktur deklariert, die den logischen und physikalischen Adressbereich des PLBADDONS speichert und ein Flag, welches anzeigt, ob das PLBADDON betriebsbereit ist oder nicht, mitführt.

Das Auslesen der Messwerte beginnt damit, dass über eine von Xilinx bereitgestellte Kernelfunktion der Adressbereich des PLBADDONS in 32-Bit Stücken ausgelesen und in ein Array kopiert wird. Aus diesem Array werden dann die einzelnen Zählerstände extrahiert, die Overflow-Zähler zu den Zugriffszählern hinzugefügt, und alle Zähler in je 64-Bit breite Variablen des aktuellen Zählerstandsarrays übertragen. Dabei werden die beim letzten Aktualisieren ermittelten Zählerstände gesichert, indem die vorletzten Stände überschrieben werden. Zugleich wird hier noch die fest in die Hardware einkodierte Prüfzahl („RESERVED“-Zahl) auf Korrektheit überprüft und nicht mit in das Zählerarray übertragen. Falls die Prüfzahl nicht korrekt sein sollte, wird dies im Kernellog vermerkt, da entweder der falsche Speicherbereich ausgelesen wurde, oder ein anderer Fehler auftrat.

Um prozessspezifisch Zugriffe zu zählen, mussten in den Dateien `sched.c` und `sched.h` (siele ausschnittsweises Listing B.2 und B.2, Seite 97ff) sowie in der Datei `exit.c` (siehe Ausschnitt in Listing B.2, Seite 98) Änderungen vorgenommen werden. Es wurde in die Struktur eines Tasks ein zusätzliches Array hinzugefügt, um dessen Zählerstände aufzunehmen. Diese Struktur wird beim erstmaligen Einplanen eines Tasks initialisiert. Bei jedem Taskwechsel werden dann durch Aufruf der bereitgestellten Funktion `aktualisiere_zaehler_global(device.baseaddress)` aus der in dieser Diplomarbeit erstellten Datei `funktionen.c` (siehe Listing B.2, Seite 96) die Zählerstände in der globalen Struktur aktualisiert, indem die Hardwarezähler ausgelesen werden. Per Differenzbildung von dem zuvor ermittelten Zählerstand mit dem aktuellen kann die von dem gerade aktiven Task verursachte Zählerstandsänderung erfasst werden. Diese Differenzen werden über die Laufzeit des Tasks einfach aufsummiert. Zusätzlich werden die Taskzähler auch direkt vor der Terminierung eines Tasks aktualisiert, um auch die letzten von dem Task während seiner Arbeit verursachten Zugriffe mitzählen zu können. Bei der Terminierung des Tasks wird dann in das Kernellog des Systems die PID und der Name des gerade beendeten Tasks mit allen seinen Zählerständen geschrieben.

Da keine Möglichkeit zur Verfügung stand, irgendwelche Daten von dem Developmentboard zur Auswertung zum PC zurück zu übertragen, wurden die Zählerstände manuell zur Auswertung in Tabellen übernommen. Mehr zur Auswertung und den gewonnenen Ergebnissen findet sich im Kapitel 5, Seite 49ff.

4.8. Verifikation der Hardware

Bei der Erstellung von Hardware ist es essentiell, diese auch auf die gewünschte Funktionalität hin zu überprüfen. Meist wird dabei nach einem Black-Box-Modell vorgegangen: Es werden bekannte Signale in die Eingänge eingespeist, die Ausgangssignale aufgezeichnet und diese mit den erwarteten Werten verglichen. Dieser Vergleich wird zuerst mit Hilfe von Simulationsprogrammen durchgeführt. Entweder wird für die zu testende Komponente eine so genannte *Testbench* geschrieben oder es werden direkt im Simulationswerkzeug Stimuli angelegt und die Reaktion des Systems beobachtet. Offenbart die Simulation keine Fehlfunktion mehr, wird in der industriellen Praxis zumindest nach dem *Place and Route* nochmals simuliert, um auch das Zeitverhalten des Systems mit den Gatter und Laufzeitverzögerungen überprüfen zu können. In den hier erstellten Komponenten konnte aufgrund der hohen Performancereserven auf diesen oft sehr zeitaufwendigen Schritt verzichtet werden. An programmierten FPGAs und vor allem an den ersten produzierten IC-Prototypen werden noch umfangreichere Messungen der Hardware vorgenommen, entweder mit Logikanalysatoren, die die Ausgangssignale aufzeichnen und auswerten können oder mit professionellen Testmaschinen, in denen einzelnen ICs komplett angeschlossen und ausgemessen werden können.

Letztendlich muss trotz aller Simulation das Gesamtsystem getestet werden, da das Testen per Simulation des gesamten Rechensystems mit all seinen Bussen und verschiedenartigen Komponenten nicht in der Zeit, die für diese Diplomarbeit zur Verfügung stand, geleistet werden kann. Die meisten Komponenten des Systems lagen sowieso im kostenpflichtigen Quelltext vor, bei dem man davon ausgehen kann, dass diese Komponenten schon ausgiebig getestet und eingesetzt wurden.

Für die ersten Praxistests wurde ein vereinfachtes System erzeugt, in dem die im Rahmen dieser Diplomarbeit erstellte Hardware mit eingebaut war. Das Xilinx EDK bietet die Möglichkeit einen FPGA ganz ohne zugrunde liegendes Betriebssystem mit einem einzigen C-Programm zu programmieren, was dann im FPGA ausgeführt wird. Von dieser Möglichkeit wurde ausgiebig Gebrauch gemacht, um nachzuweisen, dass die Zähler und die Seitenzähler korrekte Werte ermitteln. Somit konnte auch belegt werden, dass bei der Adressierung die richtigen Bits für den *Enabler* herangezogen wurden, um den jeweils richtigen Zähler für das physikalische Stück SDRAM laufen zu lassen.

Erst nach erfolgreicher Überprüfung aller von den Komponenten zur Verfügung gestellten Funktionen konnte mit sinnvollen Tests und den Messungen am unter Linux laufenden Rechensystem begonnen werden. Auch bei diesem konnte die Kernelunterstützung für die Hardware nicht einfach auf fehlerfreie Funktion überprüft werden, da nur eine einfache serielle Schnittstelle die Verbindung zur Außenwelt darstellt. Deswegen war der Treiber für die Hardware anfangs als ladbares Modul implementiert worden, damit nicht immer ein komplettes Kernelimage hochgeladen werden musste, sondern das kleine Kernelmodul über die serielle Schnittstelle durch ein verbessertes Ersetzt werden konnte. Näheres zu dieser Vorgehensweise findet sich im Anhang A.1, Seite 77f.

4.9. Zusammenfassung der Hardware

An dieser Stelle soll kurz auf die erzeugte Hardware eingegangen werden. Einige wichtige Kenngrößen sind in Tabelle 1, Seite 48 zusammengefasst. Die komplett vom Autor dieser Diplomarbeit geschriebenen Quellen sind in den Zeilen für ENABLER, SEITE und ZAEHLER zu finden. Diese werden anschließend in der vom Autor abgeänderten Datei USER_LOGIC instantiiert, welche nur den Rahmen vorgab. Die Zeile PLBADDON enthält die Daten für das komplette Modul, inklusive der vom *Wizard* erstellten Busanbindung an den PLB. Zu beachten ist in der Tabelle, dass in der Zeile SEITE das Syntheseresultat für einen einzigen Chunkzähler zu finden ist. Von diesem werden aber 32 Stück instantiiert (`generate`), was zu einem enormen Plus an Logik führt, die in der Komponente USER_LOGIC im Vergleich zur Summe ihrer einzelnen Teilmodule enthalten ist.

Die Abbildung 11, Seite 48 veranschaulicht noch mal das Gesamtsystem mit seinen wichtigsten Komponenten. Grau sind das FPGA und die SDRAM-Module zu erkennen. Der im Silizium des FPGA integrierte Power-PC-Kern ist gelb Eingezeichnet, während die in VHDL vorliegenden Module weiß sind. Die hier erstellte Erweiterung ist rot eingezeichnet, und bekommt ihre Zusatzinformationen über die Speicherzugriffe nicht über den PLB, über den der Prozessor die Zählerstände auslesen kann, sondern über zusätzliche Signale und Busse direkt aus dem Speicherkontroller. Als die einzigen Verbindungen nach außen zum PC sind die serielle Schnittstelle und der JTAG-Anschluss zu erkennen.

Komponente	Slices	Slice FFs	4-Input-LUTS	max. Taktfrequenz
ENABLER	21	23	36	274,8 MHz
SEITE	24	33	47	244,3 MHz
ZAEHLER	94	99	179	184,6 MHz
USER_LOGIC	1475	1187	2731	184,6 MHz
PLBADDON	1530	1425	2835	140,7 MHz
System	4378	4237	4539	99,2 MHz
XC2VP7-6FG456	4928	9856	9856	166,7 MHz

Tabelle 1: Syntheseergebnisse der erstellten Hardware

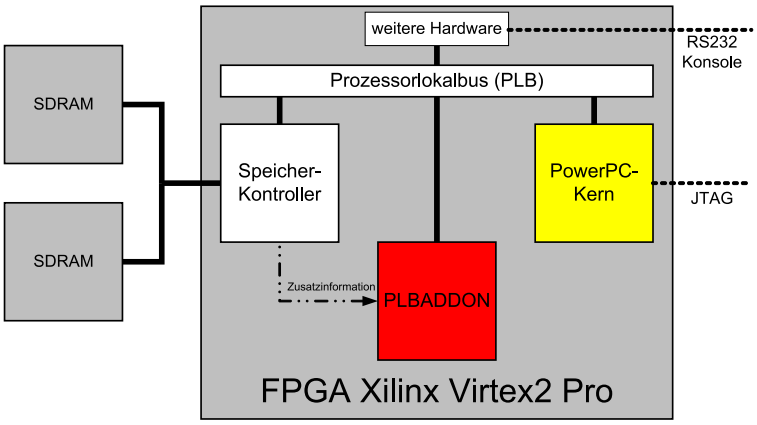

Abbildung 11: Blockdiagramm des Gesamtsystems

In der Zusammenfassung sind das Gesamtsystem und die maximalen Ressourcen des FPGA aufgelistet. An der Sliceanzahl gemessen macht das hier erstellte Modul allein 35% aus. Wie man leicht erkennt, sind alle selbst erstellten Komponenten schnell genug für das Gesamtsystem, da deren maximale Taktfrequenzen für den hier verwendeten FPGA zwischen 184,6MHz und 274,8MHz liegen. Dabei ist jedoch festzustellen, dass für die Komponente USER_LOGIC der limitierende Faktor offensichtlich die Komponente ZAEHLER ist. Es ist kein Widerspruch, dass die maximal mögliche Arbeitsfrequenz einzelner Komponenten höher liegt als die des gesamten FPGA, sondern darin begründet, dass die physikalischen Eingänge und Ausgänge nicht so schnell wie die interne Logik sind. Dieses Phänomen wird in der Praxis auch in modernen Mikroprozessoren genutzt, deren interne Taktfrequenz aktuell 4GHz überschreitet, während die Anschlüsse mit maximal 200MHz arbeiten.

5. Messungen und Ergebnisse

5.1. Vorlaufuntersuchungen

Um eine Vorstellung von den zu messenden Größen zu bekommen, wurden Datenblätter herangezogen sowie die Evaluations-Platinen an Labornetzgeräten betrieben, welche zusätzlich zu einer stabilen Spannungsversorgung auch die Stromaufnahme der angeschlossenen Geräte anzeigen können. Dabei stellte sich schnell heraus, dass der naive Ansatz, den Spannungsabfall über einen kleinen, präzisen Widerstand, einen (*Shunt*), im Strompfad mit einem Oszilloskop (ein normales Multimeter ist nicht schnell genug) aufzuzeichnen, nicht geeignet ist, aussagekräftige Messwerte zu erhalten.

Darum wurde zu Beginn der Arbeit eigens eine Experimentierschaltung aufgebaut, die im folgenden Abschnitt beschrieben wird. Mit dieser konnten dann weitere Messungen unternommen werden, die in den darauf folgenden Abschnitten vorgestellt werden.

5.2. Messungen mit dem Current Sense Amplifier

Da die Stromaufnahmedifferenzen sehr klein waren und auch sehr schnell wechselten, wurde eine Experimentierschaltung mit einem *Current Sense Amplifier* (Stromfühler mit Verstärkung, kurz CSA), dem MAX4070 aufgebaut. Abbildung 12, Seite 50, zeigt den Schaltplan. Für einen Verstärkungsfaktor von 50 wird Pin 5 (oben rechts) mit Masse anstelle von V_{CC} verbunden.

Um innerhalb eines sinnvollen Bereichs zu messen und gleichzeitig sofort verfügbare Widerstände verwenden zu können, wurde der Messwiderstand R_{sense} auf $\frac{1}{8}\Omega$ dimensioniert. Da kein Shunt mit diesem speziellen Wert verfügbar war, wurden 8 parallel geschaltete 1Ω 0603 SMD-Widerstände verwendet. Als CSA wurde der MAX4070 ausgewählt (siehe [31]).

Aufgrund des relativ weiten Strombereichs in der Gesamtstromaufnahme des „Virtex2 Pro Developmentboard“ von etwa 140-270mA wurde zunächst ein Verstärkungsfaktor von 50 ausgewählt. Nach den ersten Messungen wurde dieser jedoch auf 100 angehoben, um den tatsächlichen Wertebereich besser auflösen zu können. Es stellte sich nämlich heraus, dass sich der Strom fast zu gleichen Teilen auf die drei Spannungsschienen (3,3V, 2,5V und 1,5V) aufteilt und nicht, wie zuvor vermutet, vor allem die

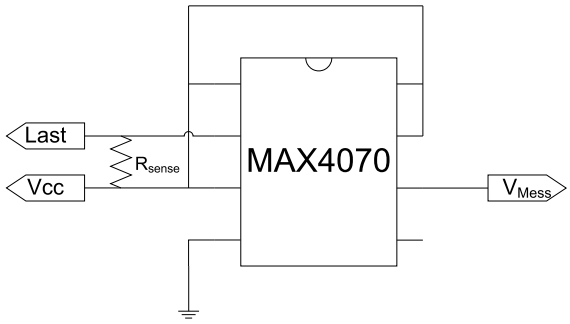

Abbildung 12: Schaltplan des CSA mit Verstärkungsfaktor 100

2,5V-Schiene belastet wird. An dieser sind fast alle ICs auf dem Testbord angeschlossen, genauso wie die Speicher und Teile des FPGA [32].

Der Current Sense Amplifier hat eine $-3dB$-Bandbreite von 20kHz, was dazu führt, dass kurzfristige Spitzenwerte oder Nulldurchgänge in der Stromaufnahme durch das tiefpassartige Verhalten herausgefiltert werden. Allzu extreme Spitzen sind sowieso nicht messbar, da jeder Stromversorgungsanschluss am FPGA und auch an den Spannungsreglern über mehrere Kondensatoren stabilisiert wird, so dass Grund zu der Annahme besteht, dass ordentliche Messwerte gewonnen werden können.

Mit dem CSA wurden zuerst einige Messreihen gefahren, um seine Funktionalität und seine Genauigkeit zu verifizieren. Dazu wurde der V_{CC}-Eingang mit einem Labornetzgerät von Toellner (TOE 8752) verbunden, der Ausgang für die Last über entsprechend gewechselte Kohleschichtwiderstände mittels des Multimeters PM2519 (von Philips) zur Strommessung mit der Masse des Netzteils verbunden, genauso wie die Masse der Experimentierschaltung. Am Ausgang V_{MESS} wurde mit einem Oszilloskop TDS220 von Tektronix unter Zuhilfenahme der Mittelwertfunktion aus 128 Messwerten die vom CSA generierte Spannung abgelesen. Abbildung 14, Seite 52, visualisiert das Ganze.

Die gewonnenen Messwerte von Strom am Multimeter (I_{multi}) und Spannung am Oszi (V_{oszi}) wurden mittels einer Tabellenkalkulation zu den Diagrammen in Abbildung 13, Seite 51, zusammengefasst. Wie man sieht, ist der Zusammenhang zwischen Stromfluss durch die Last und der Spannung am Messausgang sehr linear. Rein theoretisch ist der Zusammenhang:

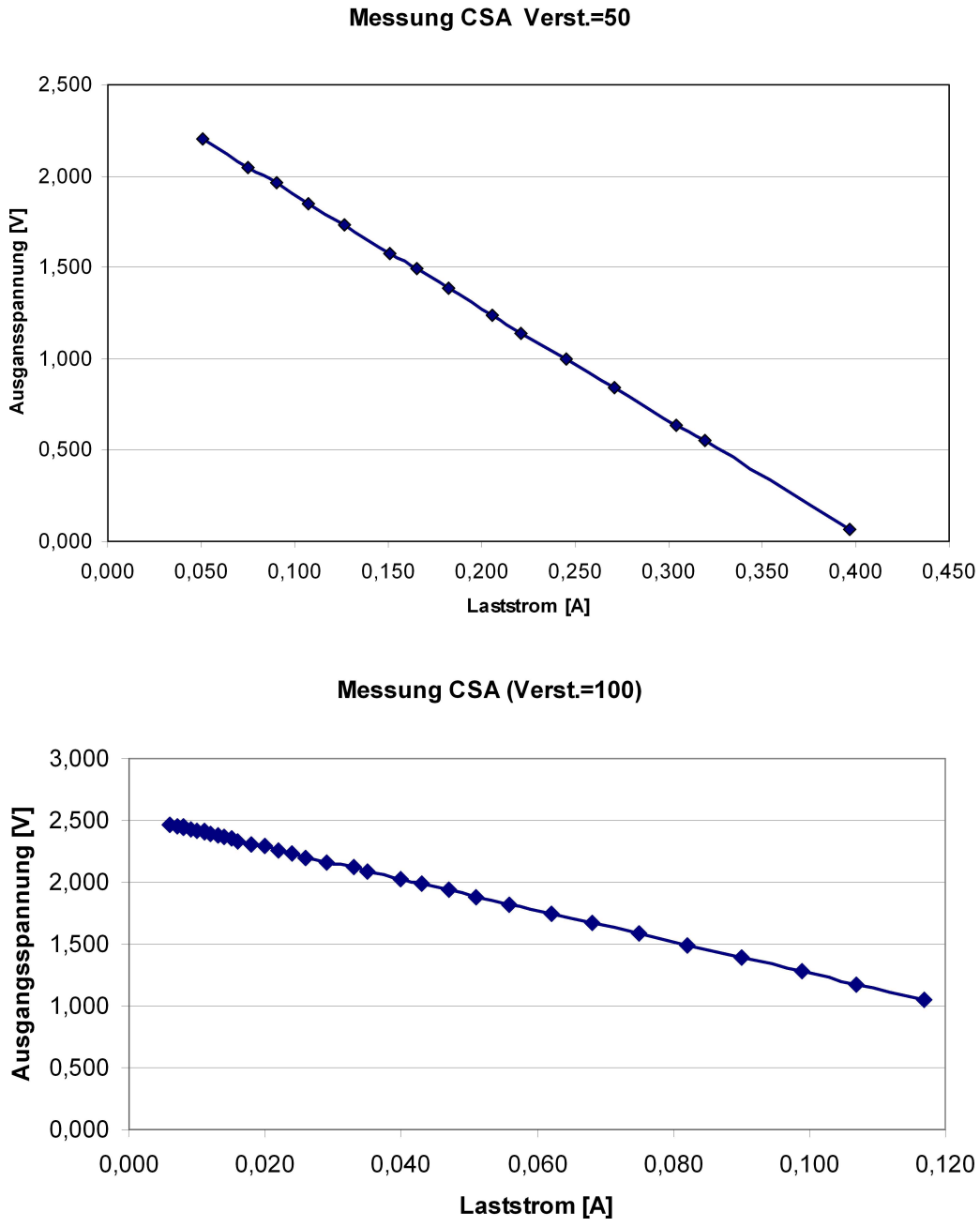

Abbildung 13: Vermessung des CSA

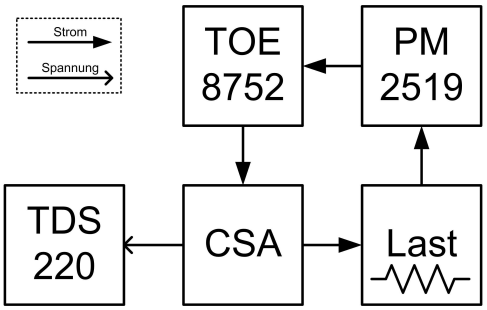

Abbildung 14: Skizze des Messaufbaus

$$I_{Last} = \frac{\text{Offset} - V_{oszi}}{R_{sense} * \text{Verstärkung}} \tag{4}$$

Mit dem Einsetzen der entsprechenden Werte, also für die Verstärkung$= 50\frac{V}{V}$ bzw. $100\frac{V}{V}$ für den Offset$= 2,5V$ und für den Widerstand $R_{sense} = \frac{1}{8}\Omega$) stimmt dies gut mit den experimentell ermittelten Formeln überein:

$$I_{Last} = \frac{2,5101 - V_{oszi}}{6,1576} \text{ für den Verstärkungsfaktor 50 und} \tag{5}$$

$$I_{Last} = \frac{2,5411 - V_{oszi}}{12,797} \text{ für den Verstärkungsfaktor 100} \tag{6}$$

Mit den so gewonnenen Formeln wurde das Developmentboard mit dem CSA (siehe Abb. 15, Seite 53) und dem Multimeter vermessen. Dabei konnten reproduzierbar dieselben Werte gewonnen werden, die auch mit den vom Multimeter gemessenen Werten im Rahmen der Messgenauigkeit übereinstimmten. Die Summe der gemessenen Ströme in den drei Spannungsschienen stimmt nicht mit dem gesamten, aus dem Netzteil entnommenen Strom überein. Dies liegt daran, dass einige Bauteile direkt an 5V aus dem Netzteil angeschlossen sind und sich deren Stromverbrauch damit einer genaueren Messung entzieht. Mit aktivierten PROMs und laufendem Demoprogramm werden so 42,3mA und ohne PROMs und laufendes Programm 34,8mA direkt aus der 5V Versorgungsspannung verbraucht.

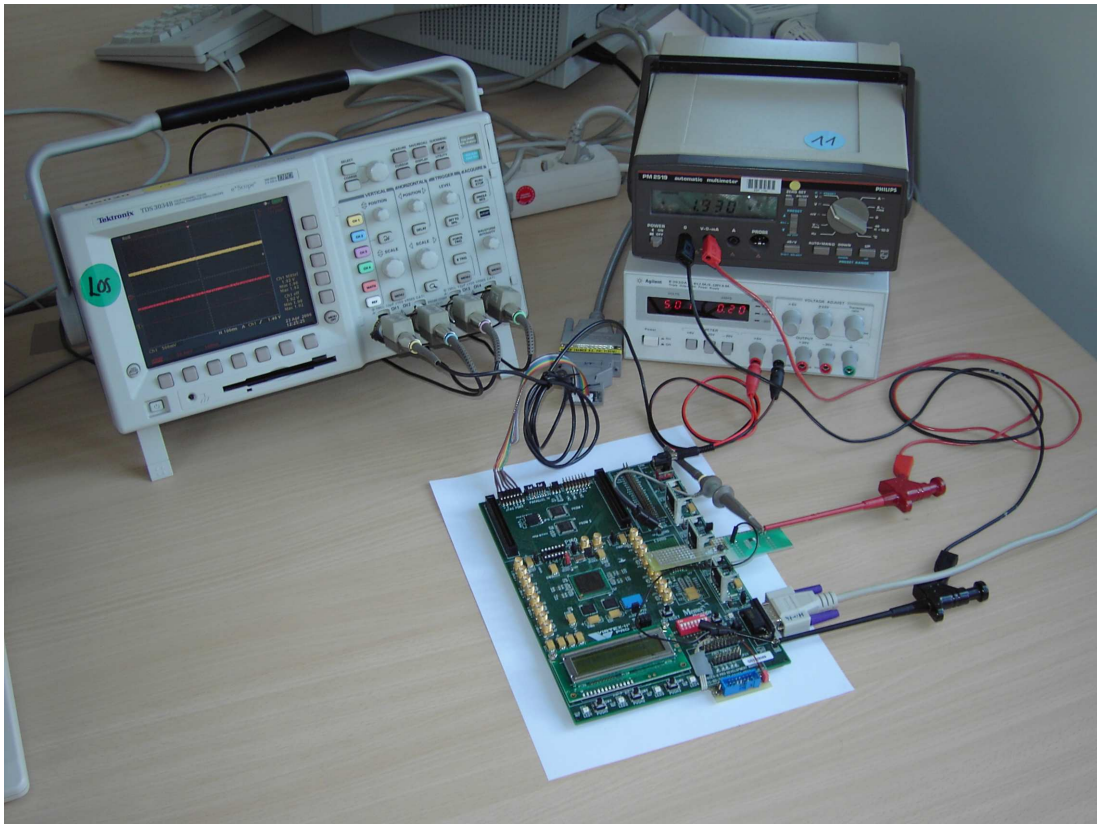

Abbildung 15: Foto des Messaufbaus mit dem CSA am 2,5V Messpunkt

Nr.	Schiene:	3,3V	in mA	2,5V	in mA	1,5V	in mA
1	PWR ON	1,69	66,35	1,92	48,77	2,11	33,69
2	Konfiguriert	1,72	64,32	1,62	72,13	1,10	112,85
3	STOP	1,72	64,40	1,63	71,43	1,45	85,34
4	CON (U-Boot)	1,71	64,79	1,62	71,98	1,32	95,11
5	JTAG-Download	1,71	64,79	wechselnd	-	1,45	85,34
6	IMI (U-Boot)	1,71	64,79	1,31	96,20	1,25	100,66
7	Linux (shell)	1,71	64,71	1,62	71,90	1,25	101,28
8	Linux (stop)	1,71	64,71	1,63	71,51	1,45	85,42
9	U-Boot	1,71	64,71	1,65	69,95	1,33	94,87
10	IMI	1,71	64,79	1,34	94,01	1,25	100,58
11	MTEST	1,71	64,63	1,44	85,73	wechselnd	-

Tabelle 2: Stromaufnahme des Developmentboards in verschiedenen Zuständen

5.3. Vorabmessungen ohne Zählerstandbeachtung

Um eine erste Übersicht zu bekommen, wurden an allen drei Stromversorgungsschienen (3,3V, 2,5V und 1,5V) unter Zuhilfenahme des CSA mit dem Multimeter PM 2519 die Stromaufnahme gemessen und die Messwerte auf 2 Dezimalstellen gerundet dargestellt. Tabelle 2, Seite 54 enthält die Messwerte, wobei in der zweiten Spalte der momentane Zustand zur Zeit der Messung benannt ist und die Werte in den Spalten der Schienen in Volt angegeben sind und rechts daneben in Milliampere. Alle Messwerte entstanden mit derselben Konfiguration und mit deaktivierten Caches.

Wie sich zeigt, ändert sich die Stromaufnahme in der 3,3V-Schiene nur unwesentlich, was darauf zurückzuführen ist, dass fast alle mit 3,3V versorgten Bauteile auf dem Developmentboard von dem momentanen Zustand des FPGA unabhängig sind.

Ein anderes Bild ergibt sich bei der Betrachtung der 2,5V und der 1,5V Spannungsversorgung. Bei beiden fällt auf, dass zwischen dem unkonfigurierten Zustand 1 und dem konfigurierten Zustand 2, bei dem der FPGA seine „Verdrahtung“ erhalten hat, ein sehr deutlicher Sprung in der Stromaufnahme zu verzeichnen ist. In der 2,5V Schiene zeichnet sich ab, dass bei hohem Rechenaufwand (IMI: Prüfsummenbestimmung des hochgeladenen Kernel- und Initrd-Images) der Stromverbrauch noch höher ist, als bei der reinen Speicherbelastung (MTEST: Ein eingebauter Speichertest im U-Boot-Bootloader). Auffallend ist auch, dass sich die Verbindung mit dem XMD (Xilinx Microprocessor Debugger) mit etwa 2mA Mehrverbrauch bei den gleichen Zuständen

auswirkt (4 und 6 gegenüber 9 und 10). Wenn das Linux-Betriebssystem läuft oder steht, wirkt sich das an der 2,5V-Schiene praktisch nicht auf die Stromaufnahme aus. Die Stromaufnahme während des Downloads des Images über JTAG war so schwankend, dass das Multimeter nicht sinnvoll abzulesen war, so dass der Messwert 5 nicht angegeben werden kann.

Bei der 1,5V-Schiene war der Messwert 5 dagegen derselbe wie im Stopp-Zustand 2. Dahingegen ist der Stromverbrauch beim Ablauf eines Programms deutlich höher (Werte 4, 6, 7, 9, 10), wobei die Stromaufnahme bei der Prüfsummenberechnung von der am Eingabeprompt der Linux-Shell noch übertroffen wurde. Währenddessen war die Stromaufnahme bei angehaltenem Mikroprozessor im Zustand 8 offensichtlich nicht von dem geladenen Image und Betriebssystem abhängig. Hier war im Zustand 11 kein Messwert sinnvoll ablesbar, was wohl daran liegt, dass während des Speichertests immer abwechselnd ein Bitmuster geschrieben und dann gelesen wird und die dafür benötigte Zeit keinen gemeinsamen Faktor mit der Messwertumsetzungsperiode des Multimeters hat.

Aus den Messergebnissen lässt sich somit folgern, dass der Mikroprozessor und die Interna des FPGA mit 1,5V betrieben werden, während der Speicher und die IO-Gates des FPGA mit 2,5V arbeiten. Erhöhte Werte in der Speicherstromaufnahme während der Prüfsummenberechnung deuten darauf hin, dass der weniger sequentielle Speicherzugriff zu erhöhter Stromaufnahme führt, da die Zwischensummen in einem anderen Bereich geschrieben werden müssen als der, aus dem die geprüften Werte stammen.

5.4. Messungen mit Zählerstandsanalyse

Nachdem nun gezeigt wurde, dass es genügt, bei der Analyse der Speicherstromaufnahme sich auf die Messung der 2,5V-Schiene zu beschränken, können nun Messungen unter gleichzeitiger Betrachtung der hier in dieser Arbeit implementierten Zähler stattfinden. Dazu wird zuerst mit einfachen Programmen der Energiebedarf von Speicherzugriffen analysiert, um später den Energiebedarf mittels Benchmarks bei realistischeren Aufgaben zu analysieren.

5.4.1. Ermittlung des Energieverbrauches eines Zugriffes

Für die Ermittlung des Verbrauches von Speicherzugriffen wurden mehrere Versionen eines kurzen Programms geschrieben, welches in einer Schleife ständig Speicherzugriffe durchführt. Wenn ein solches Programm hinreichend lange läuft, lässt sich der Stromfluss durch die ständigen Speicherzugriffe mit dem CSA und Gleichung 6, Seite 52, messen. Nach Beendigung des Programms liefern die Zählern aus dem /proc-Dateisystem, wie viele Speicherzugriffe tatsächlich stattgefunden haben, da nicht nur das Programm, sondern auch das zugrunde liegende Betriebsystem in parallel ablaufenden Tasks Speicherzugriffe verursacht haben. Deswegen muss das Programm hinreichend lange laufen, damit der Messfehler durch den Aufruf von `cat /proc/PLBADDON/PLBADDON` nicht ins Gewicht fällt. Dasselbe Verfahren wird nachher mit einem Programm angewendet, welches nur Lesezugriffe erzeugt.

Fall	Ort	Muster	Strom [mA]	Differenz [mW]
Linux-Shell	-	-	70,02	0
Schreiben	Variable	wechselnd	70,18	0,78
Schreiben	Variable	0x00000000	70,26	1,17
Schreiben	Variable	0x55555555	70,41	1,95
Schreiben	Variable	0xAAAAAAAA	70,41	1,95
Schreiben	Variable	0xFFFFFFFF	70,26	1,17
Schr. & Rechnen	Array	wechselnd	87,06	85,18
Schreiben	Array	wechselnd	120,72	253,46
Schreiben	Array	0x00000000	119,33	246,54
Schreiben	Array	0x55555555	120,11	250,45
Schreiben	Array	0xAAAAAAAA	120,04	250,06
Schreiben	Array	0xFFFFFFFF	119,65	248,11
Lesen	Variable	ALLE	70,26	1,17
Lesen	Array	wechselnd	84,87	74,24
Lesen	Array	0x00000000	84,79	73,85
Lesen	Array	0x55555555	84,01	69,94
Lesen	Array	0xAAAAAAAA	84,01	69,94
Lesen	Array	0xFFFFFFFF	83,23	66,03
Stop des PPC	-	-	72,29	11,33

Tabelle 3: Energiemehrverbrauch kleiner Testprogramme

In Tabelle 3, Seite 56, sind die gemessenen Werte dargestellt. In der Spalte Fall steht, was das Programm hauptsächlich gemacht hat. Zum Vergleich ist in der ersten Zeile der Strom dargestellt, wenn Linux läuft, und an der Shell auf eine Eingabe wartet. Dieser Wert dient in dieser Tabelle als Referenzwert, da jeder andere Zustand, sogar der letzte Fall, in dem der PPC-Kern mit dem Debugger angehalten wird, mehr Strom verbraucht. Wie man sieht, ist sowohl beim Schreiben, als auch beim Lesen an einer einzigen Speicheradresse der Leistungsunterschied minimal (<2mW). Dies liegt wahrscheinlich daran, dass ständig dieselbe Bank und Reihe adressiert wird. Dies hat zur Folge, dass die Datenwerte in den Pufferregistern der SDRAM-Module stehen (vgl. Abschnitt 2.4, Seite 13ff) und der leicht erhöhte Leistungsverbrauch von der Aktivierung der Treiberstufen herrührt.

Ganz anders stellt sich der Fall dar, wenn auf ein Array (1MB groß) zugegriffen wird. In diesem Fall steigt die Leistungsaufnahme deutlich an, und es lässt sich erkennen, dass die Leistungsaufnahme im Fall des Schreibens deutlich über der des Lesens liegt. Dies liegt aber nicht nur daran, dass Schreibzugriffe in den Speicher mehr Energie benötigen, sonder auch daran, dass während des Ablaufes eines Schreibprogramms die ganze Zeit über auch Leseinstruktionen stattfinden, was auch die Kompilierung mit `-03` nicht verhindern kann. So lässt sich anhand Zählerstände nachweisen, dass auf 4 Schreibzugriffe statistisch etwa 5 Lesezugriffe entstehen. Dahingegen liegt die Quote bei den Fällen für das Lesen bei mindestens 22 Lesezugriffen zu eins für den Fall des Zugriffs auf eine Variable, während das Verhältnis beim Lesen aus einem Array bei etwa 150 zu eins liegt.

Als weitere Erkenntnis stellte sich heraus, dass das Erstellen eines nicht konstanten Bitmusters einen deutlichen Einfluss auf die Leistungsaufnahme hatte (siehe Fall „Schreiben und Rechnen“). Im Zusammenhang mit den Zugriffszählern und der Laufzeit des Programms stellte sich heraus, dass die Integer-Operation „Modulo“ soviel Rechenzeit verbraucht, dass der Speicher nicht mehr mit der selben Rate angesprochen werden kann, wie bei konstanten Mustern. Eine hohe Speicherzugriffsrate ließ sich aber wieder erreichen, indem das wechselnde Muster durch Additionen und Verschiebeoperationen, die sehr schnell ausgeführt werden können, gebildet wird.

Um den Energiebedarf eines einzigen Zugriffes auf den SDRAM-Speicher abzuschätzen, bin ich wie folgt vorgegangen:

1. Die Zählerzustände wurden mit dem Programm, welches wechselnde Bitmuster aus einem Array liest, festgehalten wobei 506,6 Millionen Lesezugriffe und 3,3 Millionen Schreibzugriffe gezählt wurden.

2. Die Zahl der Schreibzugriffe wird in dieser Rechnung vernachlässigt, so dass die im Vergleich zum Ruhezustand des Betriebsystems mehr aufgenommene Leistung sich durch die Zahl der Lesezugriffe dividieren lässt, um die Leistung pro Zugriff zu erhalten.
$$\frac{74,23\text{mW}}{506599014} = 146,5\text{pW} \tag{7}$$

3. Die Ablaufzeit des Programms waren 62,62 Sekunden, woraus sich mit der Anzahl der Zugriffe die Zeitdauer für einen einzigen Zugriff errechnet.
$$\frac{62,62\text{s}}{506599014} = 123,61\text{ns} \tag{8}$$

4. Wenn man nun die Gleichungen 7 und 8 miteinander multipliziert erhält man den Energiemehrverbrauch pro Zugriff:
$$146,5\text{pW} * 123,61\text{ns} = 1,811 * 10^{-17}\text{J} \tag{9}$$

5. Zieht man nun die ganzen Messungenauigkeiten und Rundungsfehler, sowie den Energieverbrauch des als Referenz festgelegten Verbrauchs der wartenden Shell in Betracht, kommt man zu dem Schluss, dass ein Lesezugriff auf den Speicher in der Größenordnung von $2 * 10^{-17}$J liegt, was zu dem Verbrauch der wartenden Shell hinzuaddiert werden muss.

Mit einer genauso verlaufenden Rechnung habe ich auch den Energiemehrverbrauch für einen Schreibzugriff abgeschätzt, indem die 168 Millionen Lese und 134 Millionen Schreibzugriffe eines 13,99s lang laufenden Programms in die Formeln eingesetzt. Dieses Programm hatte einen Mehrverbrauch von 253,45mW. Die Resultierenden $3,866 * 10^{-17}$ Joule müssen noch mit dem Lese-Schreib-Verhältnis von 4:5 multipliziert werden, womit sich der Speicherverbrauch eines Schreibzugriffes auf etwa $3 * 10^{-17}$J abschätzen lässt. Dies ist der Aufschlag auf den Verbrauch einer wartenden Linux-Shell.

Da eine wartende Linux-Shell in einer Testzeit von 120s immerhin 2895883 Lese- und 95532 Schreibzugriffe auf den Speicher durchführt, bin ich davon ausgegangen, dass die Summe der Zugriffe den Stromverbrauch von 0,47mA (2,43mW) zwischen den Zuständen 7 und 3 aus der Tabelle 2 verursacht. Setzt man diese Werte in obige Formeln ein, beträgt der Grundverbrauch eines Speicherzugriffes etwa $3,26 * 10^{-14}$J, was über 1000-mal mehr ist, als der zusätzliche Bedarf für einen Schreib oder Lesezugriff. Dies deckt sich mit der Erklärung aus Abschnitt 2.2, S. 4ff, wonach ein Schreibzugriff prinzipiell genauso wie das Lesen funktioniert. Die leicht erhöhte Energieaufnahme beim Schreibzugriff lässt sich damit erklären, dass die Kondensatoren im SDRAM beim Schreiben vollständig aufgeladen werden müssen, während sie beim Lesen nur um den Teil wieder aufgeladen werden müssen, um den sie sich in der Zwischenzeit selbst entladen haben.

Wird das bekannte Zugriffsmuster der wartenden Linux-Shell in die Tabellenkalkulationsvorlage von Jensen (siehe [25]) eingesetzt, und die sonstigen Daten aus dem Datenblatt des Speichers von Infineon (siehe [20]) eingetragen, ergibt sich dafür eine abgeschätzte Grundleistungsaufnahme für die SDRAM-Bausteine von etwa 35mW. Dies bedeutet, dass etwa der gleiche Betrag auf der 2,5V Schiene an anderen Stellen der Platine verbraucht wird. Dies erscheint beim Studium des Schaltplans in etwa korrekt zu sein, da z.B. sieben LEDs mit 2,5V betrieben werden, genauso wie der Pegelwandler für die serielle Schnittstelle.

5.4.2. Energiebedarf in Benchmark-Programmen

Um nicht nur mit synthetischen Programmen unrealistische Anwendungsgebiete abzudecken und daraufhin eventuell spekulative Aussagen zu treffen, wurden einige „realistischere“ Anwendungsfälle aus der Benchmarksuite „MiBench“ (siehe [14]) ausgewählt. Die Auswahl beschränkte sich auf Testfälle, die in die Ramdisk des Systems passten, und sich mittels des Crosscompilers übersetzen ließen. Diese Benchmarks wurden für das Developmentboard kompiliert, in `.tar.gz`-Archive verpackt, über die serielle Schnittstelle wie in Anhang A.1 beschrieben auf das Board geladen, dort entpackt, ausgeführt und die Zählerstände aus dem Kernellog ermittelt. Diese Benchmarks sind hier kurz aufgeführt, die Beschreibung des Verhaltens stammt aus dem Paper zu MiBench ([14]).

1. Basicmath: Es werden einfache mathematische Operationen ausgeführt, wie das Lösen von quadratischen Gleichungen, das Ziehen der Quadratwurzel und das Umrechnen von Rad nach Winkelgrad.

2. Qsort: Mit dem bekannten Quicksort-Algorithmus wird eine Liste von Wörtern sortiert. Die Wörterliste stellte das kleine Testfeld dar, weil das große nicht in den Speicher passte. Interessant war das Energieverhalten am Ende des Algorithmus, an dem die Daten in sortierter Reihenfolge wieder in die Ramdisk zurückgeschrieben werden, was zu starken Messwertausschlägen führte.

3. Jpeg: Dieser Benchmark besteht aus 2 Programmen, wobei das erste eine Bilddatei vom PGM in das JPEG-Format konvertiert, und die andere die Rückkonvertierung durchführt. Es wurde das große Testbild verwendet. Da dies zwei getrennte Prozesse sind, wurden die Zugriffe der beiden Prozesse in der Auswertung zusammengezählt. Da die Programme relativ kurz laufen, kann der gesamte Ablauf in Abb. 16, Seite 61 betrachtet werden. Die schwarze Kurve in allen folgenden Abbildungen stellt den gleitenden Mittelwert über 32 Messwerte dar um Rauschen von signifikanten Stromaufnahmeänderungen zu unterscheiden. Auffällig ist, wie der Benchmarkteil des Konvertierens nach JPEG direkt nach dem Starten heftige Speicheraktivitäten entwickelt, während das Dekomprimierungsprogramm Erst zum Ende des Benchmarks hin verstärkte Speicheraktivitäten zeigt.

4. Typeset: Mit diesem Programm wird das darstellen einer HTML Seite nachgebildet, indem nur das Verarbeiten der Daten ohne eine grafische Darstellung oder Bildschirmausgabe durchgeführt wird. Es wurde hierbei der große Testdatensatz verwendet.

5. Dijkstra: In diesem Benchmark wird der bekannte Dijkstra-Algorithmus angewendet, um die kürzeste Verbindung zwischen zwei Knoten eines Graphen in Adjazenzmatrixdarstellung zu finden. Dieser Algorithmus hat eine Komplexität von $O(n^2)$.

6. SHA: Der *Secure Hash Algorithm* ist ein Algorithmus, um aus einer gegebenen Eingabe einen 160-Bit Hashwert zu bilden. Er wird oft eingesetzt, wenn es um Kryptographie oder Peer-to-Peer-Netze geht. Die bekannten MD4 und MD5

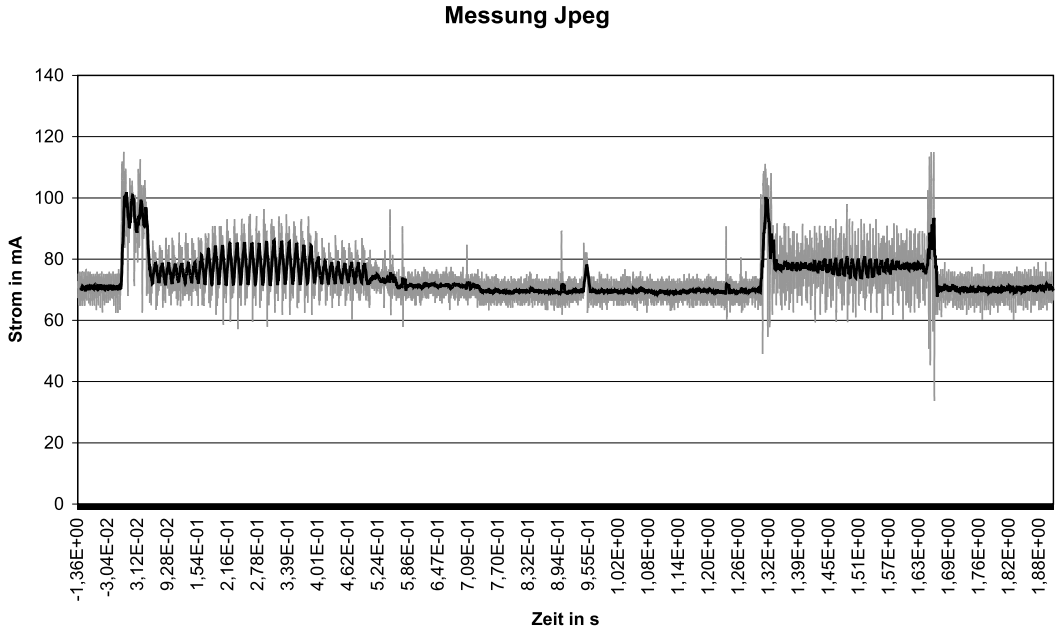

Abbildung 16: Stromverbrauch des JPEG Benchmarks

Hashfunktionen basieren auf diesem Algorithmus. Es wurde der große Datensatz zum Hashen verwendet. Das Energieverbrauchsmuster dieses Benchmarks ist in Abbildung 17, Seite 62 zu sehen. Sehr auffällig ist bei diesem Benchmark der gleichmäßige Energieverbrauch, der nur zum Start und zum Ende des Programms nennenswerte Ausschläge aufweist.

7. ADPCM: Dieses Kürzel steht für adaptive Puls-Code-Modulation, und wird zur Übertragung von Sprache in mobilen Kommunikationssystemen eingesetzt. In dieser Implementierung werden 16-Bit breite lineare PCM-Wörter aus einer Sprachaufzeichnung in 4-Bit-Wörter konvertiert, und damit eine konstante Datenkompressionsrate von 4:1 erreicht. Es wurde hier der kleine Testdatensatz angewandt, da auch dieser Benchmark aus der Kompression und dem Entpacken besteht. In der Auswertung wurden die Zähler beider Prozesse aufaddiert.

8. CRC32: Mit diesem Benchmark wird der populäre 32-Bit *Cyclic Redundancy Check* (CRC) ausgeführt. Dieser CRC wird oft bei der Datenübertragung oder Kompression verwendet, um Fehler aufzuspüren. Die Eingabedatei war dieselbe wie bei dem Test 7, nur aufgrund der hohen Geschwindigkeit des Algorithmus

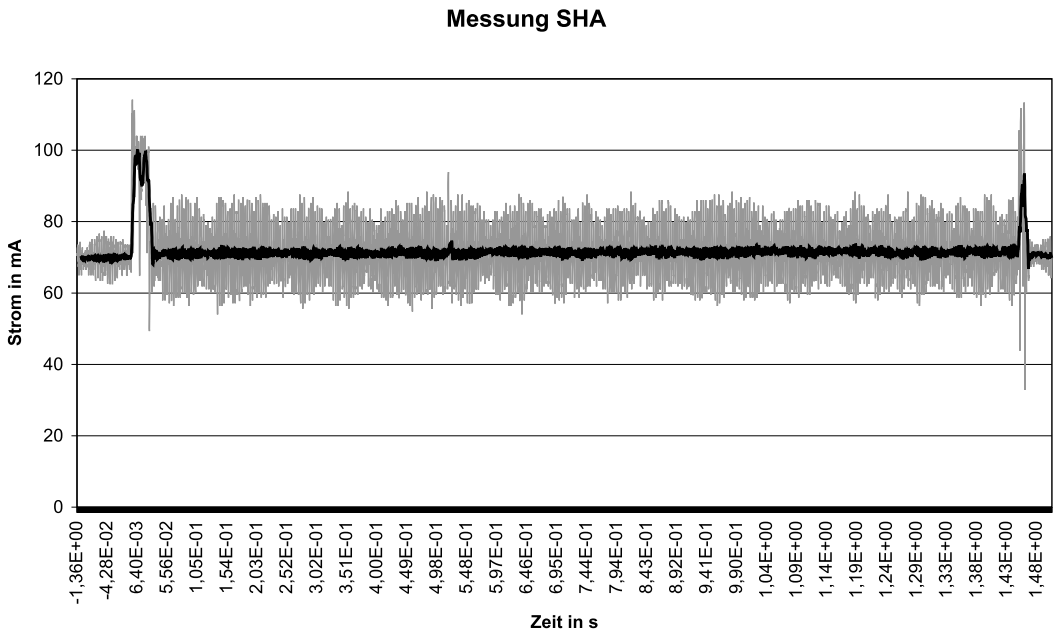

Abbildung 17: Oszilloskopmesswerte des Stromverbrauches des SHA-Benchmarks

acht Mal per `cat` aneinanderkopiert. Das Energieverbrauchsmuster ist trotz der vielen Speicherzugriffe auf die große Datei den ganzen Test über konstant.

Die Benchmarks liefen alle jeweils 5-mal ab, wobei allerdings ein nicht gewerteter Lauf zuvor durchgeführt wurde, um eventuelles Cachen von Daten anzustoßen. Das System war immer gleich konfiguriert, ohne aktivierte Caches im Prozessor und einer dynamischen Ramdisk als Massendatenspeicher. Dies kann die Messwerte etwas verfälschen, weil die Programme ihre Eingabe und Ausgabe von der Ramdisk beziehen bzw. darauf schreiben, so dass diese E/A-Operationen auch als Speicherzugriffe mitgezählt werden. Leider gibt es aber auf dem Developmentboard keine Möglichkeit die Programme ohne eine Ramdisk laufen zu lassen.

In der Tabelle 4, Seite 63, sind nun die Ergebnisse der Benchmarks getrennt nach Lese- und Schreibzugriffen angegeben. Die Obere Zeile eines jeden Benchmarks steht für die Lesezugriffe, die untere für die Schreibzugriffe. Tabelle 5, Seite 63, stellt die die einzelnen Benchmarks nochmals für die Summe aus Schreib- und Lesezugriffen dar.

Name	Zugr./sec.	Verbr. [J]	E(Zugr.)	Stdabw.	in [%]
Basicmath	1691048	8,77802E-07	26921482	26431,5	0,098
	10006	5,19560E-09	159296	4665,6	2,929
Qsort	1024120	5,26933E-07	16160618	66673,3	0,413
	229810	1,18279E-07	3626409	13878,8	0,383
Jpeg	2096429	1,12104E-07	3438143	186582,8	5,427
	613927	3,28391E-08	1006841	26394,6	2,622
Typeset	7856036	2,90222E-06	89008885	69009,5	0,078
	2199428	8,12775E-07	24919523	15297,7	0,061
Dijkstra	2356443	2,98885E-07	9166563	1063481,8	11,602
	240947	3,05704E-08	937283	147120,9	15,697
SHA	817270	3,83730E-08	1176869	34598,6	2,94
	182764	8,58387E-09	263180	95442,9	36,265
ADPCM	1383309	4,14958E-08	1272645	27942,9	2,196
	477503	1,43283E-08	439303	13967,7	3,18
CRC32	356573	1,32890E-07	4075630	103292,4	2,534
	47862	1,78430E-08	547064	50166,4	9,17

Tabelle 4: Ergebnisse der MiBench-Benchmarks, nach Lesen und Schreiben aufgeschlüsselt

Name	UC	Zeit [s]	Zugr./s	Zugr. insges.	Verbr. insges. [J]	Rd:Wr
Basicmath	22	15,92	1701054	27080778	8,82997E-07	169,00
Qsort	18	15,78	1253931	19787027	6,45212E-07	4,46
Jpeg	16	1,64	2710356	4444984	1,44943E-07	3,41
Typeset	8	11,33	10055464	113928408	3,71500E-06	3,57
Dijkstra	11	3,89	2597390	10103846	3,29455E-07	9,78
SHA	12	1,44	1000034	1440049	4,69569E-08	4,47
ADPCM	12	0,92	1860813	1711948	5,58242E-08	2,90
CRC32	13	11,43	404435	4622694	1,50733E-07	7,45

Tabelle 5: Ergebnisse der MiBench-Benchmarks in der Summe aller Zugriffe

In der ersten Spalte beider Tabellen befindet sich der Name des Benchmarks. In der zweiten Spalte von Tabelle 4 steht Zahl der Zugriffe pro Sekunde der Algorithmen. Umso größer der Quotient, umso speicherintensiver ist die Anwendung, während ein kleinerer Quotient eher auf eine rechenintensive Anwendung hinweist. Die dritte Spalte beinhaltet den resultierenden Verbrauch aufgrund der Anzahl der Speicherzugriffe. Dieser Wert wird mit den Werten aus dem vorhergehenden Unterabschnitt errechnet, wobei angenommen wurde, dass jedes Zugriff $3,15 * 10^{-14}$J als Grundleistung benötigt, wozu noch der Aufschlag für Lese- bzw. Schreibzugriffe addiert ist. Dieser Wert ist also der Wert, um den die Stromaufnahme im Vergleich zum stehenden System erhöht ist. Um den totalen Stromverbrauch des Speichersubsystems zu ermitteln, muss der Strom, den die SDRAMs ständig verbrauchen zu diesem Wert hinzuaddiert werden. Es steht in der vierten Spalte jeweils der arithmetische Mittelwert (Erwartungswert E) der absoluten Anzahl der Zugriffe. In der fünften Spalte die Standartabweichung während der Messungen. Um die Zahlen zu veranschaulichen steht in der letzten Spalte die Prozentzahl der Standartabweichung vom Erwartungswert.

In der Tabelle 5 steht nach dem Namen als zweiter Kennwert die Anzahl der UC (*Unused Chunks*), also wie viele der 32 Speicherbereiche nicht berührt wurden. Umso mehr nicht berührt wurden, umso höher ist die Lokalität des Prozesses. In den weiteren Spalten folgen dann die Anzahl der Zugriffe pro Sekunde, die absolute Anzahl der Zugriffe, sowie der Verbrauch, der auf den Grundverbrauch addiert werden muss. Die Werte in Tabelle 5 beziehen sich immer auf die Summe von Schreib und Lesezugriffen, was bei den Chunkzählern nicht unterschieden werden kann. In der sechsten Spalte sieht man das Verhältnis der Lese- zu Schreibzugriffen, was aussagt, wie viele Lesezugriffe auf einen Schreibzugriff in den Speicher kommen.

Zur Auswertung ist zu sagen, dass der Benchmark Typeset der mit Abstand speicherintensivste Benchmark ist, wobei auch auffällig ist, dass Lese- und Schreibzugriffe in der gleichen Zehnerpotenz liegen, und nur etwa 3,7-mal mehr Lese- als Schreibzugriffe stattgefunden haben. Damit ist der Energieverbrauch beim Aufbereiten einer HTML-Seite auch der größte unter diesen Programmen. Der rechenintensivste Benchmark ist CRC32, wobei dieser noch weniger Speicherzugriffe gehabt hätte, wenn seine Eingabedatei nicht in der Ramdisk gelegen hätte. Die Standartabweichungen vom Erwartungswert sind relativ gering. Nur beim Dijkstra-Algorithmus liegen diese über 10%, was wohl daran liegt, dass der eingeschlagene Pfad durch den Graphen nicht deterministisch ist. Der einzige unerklärliche Wert in der Tabelle ist die sehr große Stan-

dartabweichung bei den Schreibzugriffen des SHA-Benchmarks. Zur Anzahl der Lesezugriffe im Verhältnis zu den Schreibzugriffen fällt auf, dass die ADPCM-Kodierung ein sehr niedriges Verhältnis von 2,9 Lesezugriffen pro einem Schreibzugriff aufweist. Wahrscheinlich kommt diese Komprimierungsart mit sehr wenigen Schleifen aus. Die andern Benchmarks liegen alle im Mittelfeld mit 3,41 bis 9,78 Lesezugriffen pro Schreibzugriff, während der Benchmark Basicmath durch ein ziemlich extremes Verhältnis von 169 zu 1 hervorsticht. Offensichtlich durchläuft dieser Benchmark ziemlich viele Schleifen über die Zahlen, was das extreme Verhältnis erklärt. Schleifen erzeugen erfahrungsgemäß viele Lesezugriffe, da immer die Schleifenvariable mitgeführt werden muss, mit der arithmetische Operationen und Vergleiche durchgeführt werden, die jedes Mal lesend auf den Speicher wirken, da bei der hier verwendeten Standartkonfiguration die Caches, die solche Referenzen auffangen würden, inaktiv sind. Mit aktivierten Caches würden zwar die Messergebnisse noch realitätsnäher, aber da viele der MiBench-Benchmarks laut [14] komplett im Cache ablaufen, würde man nicht mehr gut zwischen rechenlastigen und speicherintensiven Anwendungen unterscheiden können.

6. Fazit und Ausblick

In dieser Diplomarbeit wurde mit der Erläuterung der physikalischen Grundlagen von Energie, Energieverbrauch, CMOS-Schaltungstechnik und SDRAM die Vorraussetzung geschaffen, warum es lohnenswert erschien, diese Größen an einem rekonfigurierbaren Rechensystem genauer zu untersuchen. Nachdem mittels der Vorstellung von thematisch verwandten Forschungsarbeiten die schon bekannten Ansätze vorgestellt wurden, wurde in einem weiteren Kapitel ausführlich auf die hier implementierte Hardware und die dazugehörige Betriebssystemsoftware eingegangen. Mit diesen Voraussetzungen konnten dann mittels geeigneten Zubehörs Messungen durchgeführt werden. Diese Ergebnisse können als Grundlage für weitere Forschungsarbeiten dienen, oder aber vielleicht schon bald auch kommerziell eingesetzt werden.

Frank Bellosa gibt in seinem Paper [5] ziemlich detailliert einen Teil der Aufgabenstellung dieser Arbeit vor, indem er Seitenzähler fordert, um im Endeffekt wenig referenzierte Speicherbereiche in einer Art „Auslagerung“ in energieverbrauchsärmere Speicherarchitekturen zu transferieren. Dazu schlägt er einen so genannte MMC (*Memory Management Controller*) vor. Dieser Kontroller könnte transparent für jedwedes Betriebssystem und jede Rechnerarchitektur die energetisch optimale Seitenverteilung im Speicher realisieren.

Diese vorliegende Diplomarbeit kann als eine Vorstufe dazu dienen, denn der MMC muss die Anzahl der Referenzen pro Seite kennen, um über ihre Auslagerung zu entscheiden. Zwar kann die hier implementierte Hardware nicht bis auf Seitenniveau auflösen, was aber eher an den verfügbaren Ressourcen in der Hardware, als an der technischen Machbarkeit scheiterte. Es wäre für die erhöhte Anzahl an Chunkzählern ein größeres FPGA benötigt worden.

Die hier in dieser Arbeit angestellten Messungen weisen eine hohe Lokalität der Speicherzugriffe nach, wobei anzumerken ist, dass anstelle einer Festplatte oder eines anderen Hintergrundspeichers eine Ramdisk eingesetzt wurde. Mit einem Speicher aus SDRAM und Flash als Zusatz könnte auf die Ramdisk verzichtet werden, zumal Flash-Speicher mit dem Ziel entwickelt wurde, zukünftig Festplatten abzulösen. Deswegen sind Flash-Speicher für Block-Zugriffe optimiert und damit geeignet, als Auslagerungsort wenig benutzter Speicherseiten herangezogen zu werden.

Eine andere Idee ist, Speicherbausteine durch eine völlig andere Technologie, z.B. MRAMs (*Magnetic Random Access Memory*) zu ersetzen, die im Ruhezustand keine Spannungsversorgung benötigen aber trotzdem Zugriffszeiten wie SDRAMs haben (vgl. [36]). Dadurch würde die ständig vorhandene Grundlast deutlich sinken und es müssten keine Speicherbänke periodisch immer wieder für einen Refresh aktiviert werden. Wenn z.B. ständig nur lesend referenzierte Speicherbereiche (z.B. die `glibc`) in ein PROM (*Programmable Read Only Memory*) ausgelagert werden könnten, müssten diese nicht (gecacht) vorgehalten werden und wären mit ähnlicher Geschwindigkeit wie im SDRAM auslesbar. Der einzige Nachteil wäre, das Schreibzugriffe energetisch teuerer werden würden, da die Magnetisierung umgepolt werden muss.

Eine andere Anwendungsmöglichkeit ergibt sich, weil ohne einen radikalen Speicherbaustein- bzw. Hardwareaustausch die Laufzeit von mobilen Geräten verlängert werden kann. Dies kann durch geschickten Einsatz der hier gewonnenen Ergebnisse erreicht werden: Ein mobiles Navigationsgerät aktualisiert mit Satellitensignalen einmal pro Sekunde seine Position, was viele Speicherzugriffe erfordert. Zum Beispiel könnte dann – gegebenenfalls automatisch – bei sinkendem Batteriestand die Updatefrequenz herabgesetzt werden, so dass der Nutzer den Komfort einer Positionsbestimmung hat und das Gerät trotzdem länger läuft. Möglich ist aber auch, dass per Benutzerabfrage die Frequenz der Positionsaktualisierung gesenkt werden kann. Gleichzeitig kann somit über den dann vorauszusehenden Energieverbrauch eine Laufzeitschätzung durchgeführt werden, was bei bekannten sonstigen Eckwerten des Systems eventuell sogar spezielle Batteriemonitoringhardware wie in [19] einsparen könnte.

Da mit Linux ein verbreitetes Betriebssystem zum Einsatz kam, kann man davon ausgehen, dass sich das Referenzverhalten auch auf anderen Rechnerarchitekturen ähnlich verhält. Dies gilt selbstverständlich auch, wenn dort keine Spezialhardware zur Messung zur Verfügung steht. Somit können die hier gewonnenen Ergebnisse auch auf anderen Rechnern eingesetzt werden. Als weitere Untersuchung könnte die hier implementierte Hardware in einem anderen System eingesetzt werden, welches über Hintergrundspeicher in Form einer Festplatte oder ähnlichem verfügt, so dass die Speicherzugriffe auf die Ramdisk zum Einlesen der Dateien oder Ausgeben der Ergebnisse nicht mehr als Speicherzugriffe mitgezählt werden.

Abbildungsverzeichnis

1. n-Kanal-MOSFET (links schematisch, rechts Schaltsymbol) 5
2. CMOS-Inverter (oben p-Kanal- unten n-Kanal-MOSFET) 6
3. f und V_{DD} in den letzten Jahren (aus [24]) 7
4. Leckstromanteil im letzen Jahrzehnt (aus [24]) 11
5. SDRAM-Speichermatrix aus einzelnen Zellen (aus [10]) 15
6. Blockdiagramm des SDRAMs auf der Experimentierplatine (aus [20]) 17
7. Ablauf eines Read- und eines Write-Bursts (aus [10]) 19
8. Module und Busverbindungen des synthetisierten Systems 39
9. Blockskizze des SDRAM-Controllers (aus [47]) 40
10. Automatengraph der Data-Statemachine (aus [47], vereinfacht) 41
11. Blockdiagramm des Gesamtsystems 48
12. Schaltplan des CSA mit Verstärkungsfaktor 100 50
13. Vermessung des CSA 51
14. Skizze des Messaufbaus 52
15. Foto des Messaufbaus mit dem CSA am 2,5V Messpunkt 53
16. Stromverbrauch des JPEG Benchmarks 61
17. Oszilloskopmesswerte des Stromverbrauches des SHA-Benchmarks . 62
18. Oszilloskopanzeige beim booten von Linux 78
19. Oszilloskopanzeige beim Hochladen per JTAG-Kabel 81

Tabellenverzeichnis

1. Syntheseergebnisse der erstellten Hardware 48
2. Stromaufnahme des Developmentboards in verschiedenen Zuständen . 54
3. Energiemehrverbrauch kleiner Testprogramme 56
4. Ergebnisse der MiBench-Benchmarks, nach Lesen und Schreiben aufgeschlüsselt 63
5. Ergebnisse der MiBench-Benchmarks in der Summe aller Zugriffe . . 63

Listings

seite.vhd 30
1. zaehler.vhd 82
2. user_logic.vhd 84
3. enabler.vhd 88
4. plb_slave_attachment_indet.vhd 89
5. treiber-driver.c 90
6. funktionen.h 94
7. funktionen.c 95
8. sched.c 96
9. sched.h 97
10. exit.c 97

Literatur

[1] J. Bäsig:
Entwicklung digitaler Systeme mit VHDL,
6. Überarbeitung, Eigenverlag, Nürnberg, 2000

[2] K. Barr, K. Asanovic:
Energy aware Lossless Data Compression,
In Usenix Association: Proceedings of MobiSys 2003, San Francisco, Mai 2003, S. 231-244

[3] F. Bellosa:
Processor Cruise Control: Throttling Memory Access in a Soft Real-Time Environment,
Technical Report TR-I4-97-02, Universität Erlangen, July 1997

[4] F. Bellosa:
The Case for Event-Driven Energy Accounting,
Technical Report TR-I4-01-07, Universität Erlangen, 29. Juni 2001

[5] F. Bellosa:
When Physical Is Not Real Enough,
In Proceedings of the 11th ACM SIGOPS European Workshop, Leuven, 20.-22. September 2004

[6] F. Bellosa, A. Weißel:
Power Management,
Jahresbericht 2003 der Informatik, Band 36 Nummer 8, Universität Erlangen, Mai 2004, S.67ff

[7] B. Brock, K. Rajamani:
Dynamic Power Management for Embedded Systems,
In Proceedings of the IEEE International SOC Conference, Portland, September 2003

[8] K. Choi, R. Soma, M. Pedram:
Off-chip Latency-Driven Dynamic Voltage and Frequency Scaling for an MPEG Decoding,
Dep. of Electrical Engineering, University of Southern California, Los Angeles

[9] C. Clark, A. Ley et al:
IEEE Standard Test Access Port and Boundary-Scan Architecture
Institute of Electrical and Electronics Engineers, New York, 23. Juli 2001

[10] Elpida Memory:
Users's Manual - How to use SDRAM
http://www.elpida.com/pdfs/E0123N60.pdf , Februar 2005

[11] X. Fan, C. Ellis, A. Lebeck:
Memory Controller Policies for DRAM Power Management,
International Symposium on Low Power Electronics and Design, Huntington Beach, 2001, S. 129-134

[12] K. Farkas, J. Flinn, G. Back, D. Grundwald, J. Anderson:
Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine,
In Proceedings of ACM SIGMETRICS, Santa Clara, Juni 2000

[13] Fraunhofer Institut für Integrierte Schaltungen:
Power- und Batterie- Management
http://www.iis.fraunhofer.de/ec/power/index_d.html Stand: 07. Juli 2005

[14] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown:
MiBench: A free, commercially representative embedded benchmark suite,
University of Michigan, Electrical Engineering and Computer Science,
http://www.eecs.umich.edu/mibench/

[15] J. Haid, G. Käfer, C. Steger, R. Weiss, W. Schögler, M. Manninger:
Run-Time Energy Estimation in System-On-a-Chip Designs,
Asia South Pacific - Design Automation Conference, 2003

[16] M. Hartman, S. Dhar:
On-Chip Power Management Utilizing an Embedded Hardware Controller and a Low-Power Serial Interface,
Embedded World Conference, Nürnberg, 17. Februar 2004

[17] J. Hennessy, D. Patterson:
Computer Architecture - A Quantitative Approach,
3. Ausgabe, Morgan Kaufmann Publishers, San Francisco, 2003

[18] E. Hering, R. Martin, M. Stohrer:
Physik für Ingenieure,
6. Auflage, Springer, Berlin, 1996

[19] T. Hirth:
Batteriemanagement und Batteriemonitoring in mobilen Robotersystemen,
Studienarbeit, Fraunhofer IIS & FAU Erlangen, 2005

[20] Infineon:
128-MBit Synchronous Low-Power DRAM in Chipsize Packages Datasheet (Rev.

12/01)
http://quicklinks.infineon.com/s/go/index.html&rid=1880500&noWait=1 Stand: 20. August 2005

[21] Intel:
Intel 80200 Processor based on Intel XScale Microarchitecture Developer's Manual,
www.intel.com/design/iio/manuals/27341103.pdf , März 2003

[22] Intel:
Designing for Power,
ftp://download.intel.com/technology/silicon/power/download/design4power05.pdf
Stand: 15.03.2006

[23] Intel:
Intel®Pentium®M Processor Datasheet
http://download.intel.com/design/mobile/datashts/25261203.pdf Stand: 23. November 2005

[24] Intel:
Power Delivery for High-Performance Microprocessors
download.intel.com/technology/itj/2005/volume09issue04/art02_powerdelivery/vol09_art02.pdf
Stand: 23. November 2005

[25] J. Jansen:
Calculating Memory System Power for DDR SDRAM,
Micron designline, Vol. 10, issue 2, Boise, 2. Quartal 2001

[26] B. Joe:
Estimating Power for ADSP-BF533 Blackfin Processors,
Analog Devices Engineer-to-Engineer Note EE-229, Rev. 1, 20. Februar 2004

[27] R. Joseph, M. Martonsi:
Run-Time Power Estimation in High-Performance Microprocessors,
In Proceeding of the International Symposium on Low Power Electronic Device, August 2001, S.135-140

[28] W. Lamersdorf (Verantwortlich im Sinne des Pressegesetzes):
CMOS Gates demonstration
http://tech-www.informatik.uni-hamburg.de/applets/cmos/cmosdemo.html
Stand: 25. November 2005

[29] N. Lucas, C. Codrea, T. Hirth, J. Gutierrez, F. Dressler:
RoBM^2: Measurement of Battery Capacity in Mobile Robot Systems,

GI/ITG KuVS Fachgespräch „Energiebewußte Systeme und Methoden“, Erlangen, Oktober 2005

[30] S. Mamagkakis, A. Mpartzas, G. Pouiklis, D. Atienza, F. Catthoor, D. Soudris, J. Mendias, A. Thanailakis:
Design of Energy Efficient Wireless Networks Using Dynamic Data Type Refinement Methodology,
Proceedings of 2nd International Conf. on Wired/Wireless Internet Communications, (WWIC 2004), Frankfurt/Oder, February 2004

[31] Maxim:
Bidirectional, High-Side, Current Sense Amplifiers with Reference MAX4069-MAX4072,
www.maxim-ic.com/MAX4070 , Stand: Januar 2003

[32] Memec Design:
2VPx-FG456 REV4 (Schematics),
2VPxFG456_REV.pdf , in der Dokumentation zum Development Board

[33] A. Merkel, F. Bellosa:
Verteilung der Leistungsaufnahme in Mehrprozessorsystemen,
GI/ITG KuVS Fachgespräch „Energiebewußte Systeme und Methoden“, Erlangen, Oktober 2005

[34] R. Neugebauer, D. McAuley:
Energy ist just another ressource: Energy accounting an pricing in the Nemesis OS,
In Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII), Schloß Elmau, Mai 2001

[35] K. Nowka, G. Carpenter, B. Brock:
The Design and application of the PowerPC 405LP energy-efficient system-on-a-chip,
IBM J. Res. & Dev., Vol 47, No. 5/6, September/November 2003

[36] F. Oehme:
Skript zur Vorlesung Speichertechnologie,
FAU Erlangen-Nürnberg, WS 2004/05

[37] M. Pedram:
Review of the Apollo Testbed Project,
University of Southern California, Los Angeles, 4. Dezember 2003

[38] G. Stitt. F. Vahid:
Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic,
IEEE Design and Test of Computers, November/December 2002, S. 36-43

[39] U. Tietze, Ch. Schenk:
Halbleiter-Schaltungstechnik,
12. Auflage, Springer, Berlin, 2002

[40] U. Tietze:
Unterlagen zum Praktikum Programmable Logic Devices 2005"(PLD2005),
FAU Erlangen-Nürnberg, Lehrstuhl für Technische Elektronik, WS 2004/05

[41] A. Weißel, F. Bellosa:
Dynamic Thermal Management for Distributed Systems,
First Workshop on Temperature-Aware Computer Systems (TACS-1), München, 20.06.2004, S. 1-11

[42] A. Weißel, F. Bellosa:
Quantifying Application Performance for Adaptive Power Management,
GI/ITG KuVS Fachgespräch „Energiebewußte Systeme und Methoden“, Erlangen, Oktober 2005

[43] A. Weißel, F. Popelta, A. Fröhlich, F. Bellosa:
Power-Aware Caches for SoCs,
Fachgespräch der GI/ITG-Fachgruppe Kommunikation und Verteilte Systeme, Universität Stuttgart, 14.-15. Oktober 2004

[44] Wikipedia die freie Enzyklopädie:
www.wikipedia.de, Stand: November und Dezember 2005, März 2006

[45] C. Windeck:
Merk-Zellen,
c’t Magazin, Heise Verlag, Heft 6/2006, Seite 278-285

[46] K. Yaghmour:
Building Embedded Linux Systems,
O’Reilly &Associates, Sebastopol, 2003

[47] Xilinx Logicore:
PLB Synchronous DRAM (SDRAM) Controller (v1.00e)
plb_sdram.pdf, in der Dokumentation zum Xilinx EDK 6.3i, 16. November 2004

[48] Xilinx:
PowerPC Processor Reference Guide
http://www.xilinx.com/bvdocs/userguides/ppc_ref_guide.pdf , 2. September 2003

[49] Xilinx:
Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet
http://www.xilinx.com/bvdocs/publications/ds083.pdf , 10. Oktober 2005

[50] Xilinx:
Embedded System Tools Reference Manual (Embedded Development Kit EDK 6.3i)
http://www.xilinx.com/ise/embedded/edk6_3docs/est_rm.pdf , 20. August 2004

A. Bedienungsanleitung und Zusatzinformation

A.1. Installation und Inbetriebnahme des Linux-Betriebssystems

Zuallererst muss die Hardware auf dem Board so konfiguriert werden, damit ein Linux-Betriebssystem überhaupt geeignete Bedingungen zum Starten vorfindet. Dies wird mit dem XPS-Tool (Xilinx Platform Studio) erledigt, mit welchem man letztendlich ein geeignetes Bitfile generiert. Dabei ist darauf zu achten, dass bei den Softwareprojekten eingestellt wird, dass die `Default: ppc405_0_bootloop` mit `Mark to initialize BRAMS` eingebunden wird. Dies stellt, laut dem Kommentar im Assembler-Sourcecode der Datei, sicher, dass sich der Prozessor nach dem Einschalten oder einem Reset in einem definierten Zustand befindet. Nachdem die Datei `download.bit` erzeugt wurde benutzt man die generierte Header-Datei `xparameters_ml300.h`, um einen Bootloader zu kompilieren (in unserem Fall U-Boot), und einen Kernel für die Hardware anzupassen. Dies geschieht genauso, wie mit einem Kernel für den PC, nur dass als Compiler der GNU-Compiler für den PowerPC benutzt wird. Hat man nun den Bootloader und den Kernel erzeugt, kopiert man Beide am besten in das Projektverzeichnis, um an der Debuggerkonsole nicht fehlerträchtig lange Pfade eintippen zu müssen.

1. Mit Xilinx Impact die HW-Konfiguration `implementation/download.bit` schreiben.

2. XMD (Xilinx Microprocessor Debugger) mit `xmd` starten, mit dem PowerPC verbinden (`ppc`), und das Kommando `dow u-boot.elf` ausführen. Nach dem Hochladen des Bootloaders stürzt XMD mit einem *Segmentation Fault* ab.

3. Dann muss die CPU-Reset-Taste am Board gedrückt werden, um den Prozessor wieder in einen definierten Zustand zu bringen.

4. XMD ist wieder starten, mit dem PPC zu verbinden (Schritt 2), `stop` eingeben , mit `mrd 0x1c00000 20 b` nachschauen, ob sich in der Speicherausgabe die Zeichenkette „`U-Boot`“ findet, um sicherzustellen, dass sich der Bootloader noch im Speicher befindet.

5. Mit dem Befehl `rwr pc 0x1c00100` wird der Program-Counter an die Programmstartadresse des Bootloaders gesetzt. Dies ist unbedingt mit dem Befehl `srrd` nachzuprüfen, da das setzten manchmal fehlschlägt.

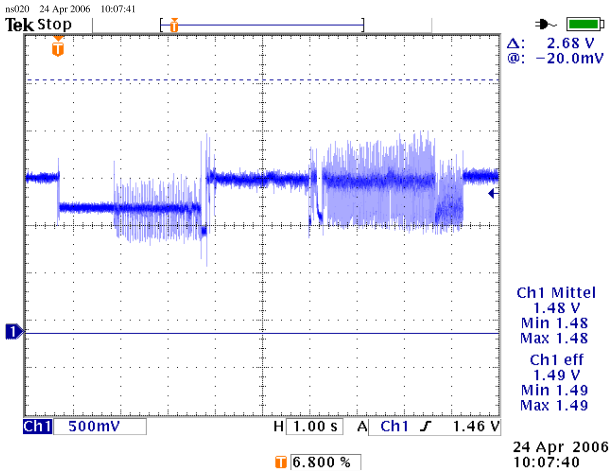

Abbildung 18: Oszilloskopanzeige beim booten von Linux

6. Mit `con` wird das Programm ab der Stelle, die im Program Counter steht fortgesetzt. Dann muss ganz schnell auf die Konsole gewechselt werden, an der ein Terminalprogramm (z.B. minicom) an der seriellen Schnittstelle lauscht, denn nach 5 Sekunden wird automatisch gebootet, wenn nicht vorher eine Taste gedrückt wird. Nach dem Tastendruck wird wieder in den XMD gewechselt, um mit `stop` den PPC anzuhalten, während an der serielle Konsole vom Bootloader auf eine Eingabe gewartet wird.

7. Dann wird mit `dow -data initrd.uimage 0x400000` das Kernelimage in eine üblicherweise freie Stelle im Speicher geladen, was aufgrund der Größe etwa 10 Minuten dauert. Nun erfolgt die Eingabe von `con`, um den PPC weiterlaufen zu lassen

8. An der jetzt wieder auf Eingaben reagierenden Minicom-Konsole sollte `imi 0x400000` eingegeben werden, um den erfolgreichen Schreibvorgang zu verifizieren (Meldung „Checksum OK“).

9. Als letztes wird `run boot_initrd` eingegeben, um die Startmeldungen des Linux-Betriebssystems an der Minicom-Konsole zu verfolgen. Abbildung 18, Seite 78, zeigt die Stromaufnahmeänderungen an der 2,5V-Schiene beim booten des Betriebssystems.

10. Nachdem nun Linux beim Starten beobachtet wurde, geht es daran, die serielle Schnittstelle so umzukonfigurieren, dass das eigene Kernelmodul per RS232-

Verbindung hochgeladen werden kann, da es zu lange dauert, jedes Mal bei einer kleinen Änderung das ganze Kernelimage wieder hochzuladen. Deswegen lautet der erste Befehl an der Shell: `/sbin/getty -L ttyS0 115200 ansi`.

11. Danach folgt das einloggen als `root`.

12. An der Konsole wechselt man dann in ein Verzeichnis, z.B. `cd /tmp`.

13. Dort wird `rx plb.ko` eingegeben, um danach die Tastenkombination Strg+A, S einzugeben, um in die Minicom-Uploadumgebung zu wechseln. Dort wird „xmodem“ ausgewählt, und mit 2x [Space] bis in das Verzeichnis gewechselt, in dem das hochzuladende Kernelmodul liegt. Dieses wird mit der [Leertaste] selektiert, und der Upload mit [Enter] gestartet. Es kann ein paar Sekunden dauern, bis dieser beginnt, aber das Kernelmodul ist binnen weniger Sekunden komplett hochgeladen.

14. Danach kann man mit `insmod plb.ko` das Kernelmodul laden, und z.B. mit `dmesg` nachsehen, ob alles geklappt hat. Die Ausgabe der Zählerstände erfolgt dann mit `cat /proc/PLBADDON/PLBADDON`.

15. Falls irgendwas nicht so wie vorgesehen geklappt hat, kann man mir `rmmod plb.ko` das Kernelmodul wieder entfernen, und mit Schritt 13 wieder ein neues Kernelmodul hochladen.

16. Ansonsten kann das Linux wie ein ganz gewöhnliches Linux benutzt werden.

Falls anstatt eines Kernelmodules einen Benchmark oder ein anderes Programm hochgeladen werden soll, sind die Schritte 13 bis 15 entsprechend zu ändern bzw. zu überspringen.

Falls während dieses Prozesses XMD Fehlermeldungen über die „cable connection“ „mutex“ bringen sollte, sollte überprüft werden, ob Impact geschlossen ist. Wenn das der Fall ist, wird mit [Strg]+[C] XMD abgebrochen. Nach der Eingabe von `ipcs` muss mit `ipcrm -s [nummer]` dasjenige Semaphorenflag, welches nicht die Nummer 0 trägt, gelöscht werden. Ab dann kann mit `xmd`, `ppc` und ggf. `con` dort weitergemacht werden, wo XMD aufgegeben hat.

Genaueres über die Installation und Konfiguration und Inbetriebnahme von Linux auf eingebetteten Systemen kann man in [46] nachlesen.

A.2. Das JTAG-Interface

JTAG ist die Abkürzung für *Joint Test Action Group*, ein Zusammenschluss verschiedener Halbleiterhersteller, und ist inzwischen eine Synonym für den IEEE-Standard 1149.1 (*Institute of Electrical and Electronics Engineers*). Der Standard wurde in den Jahren 1985/86 festgelegt, und hat sich seitdem weit verbreitet. Der Hauptzweck des auch als *Boundary Scan* bekannten Verfahrens, ist das Testen von fertig in einer Schaltung eingelöteten (*In Circuit*) ICs. Der Anschluss besteht aus fünf Steuerleitungen: Test Data Input (TDI), Test Data Output (TDO), Test Clock (TCK), Test Mode Select Input (TMS) und Test Reset (TRST), der oft weggelassen wird. Masse und V_{CC} werden für saubere Signalpegel meist auch mitgeführt, obwohl sie der Standard nicht vorsieht (vgl. [9]). Inzwischen wird der JTAG-Anschluss auch für das Programmieren von Mikrocontrollern und FPGAs eingesetzt, so auch auf dem hier verwendeten Developmentboard.

JTAG ist ein Beispiel für eine serielle Datenübertragung, da die Daten nur über den TDI eingespeist werden, was mit der Frequenz von TCK geschieht. Obwohl TCK durchaus bis zu 25MHz betragen kann, ist TCK bei dem in dieser Arbeit verwendeten Programmierkabel auf 200kHz beschränkt. Deswegen ist die Übertragung vergleichsweise langsam. Nichtsdestotrotz wird eine JTAG-Verbindung hier in dieser Arbeit genutzt, um die Konfiguration auf den FPGA aufzuspielen, den Bootlader, das Kernel- und Initrd-Image hochzuladen und den PowerPC mit dem Debugger anzusprechen. Die Abbildung 19, Seite 81 zeigt in einem Ausschnitt die Stromaufnahme an der 2,5V-Schiene beim Hochladen des Kernelimages.

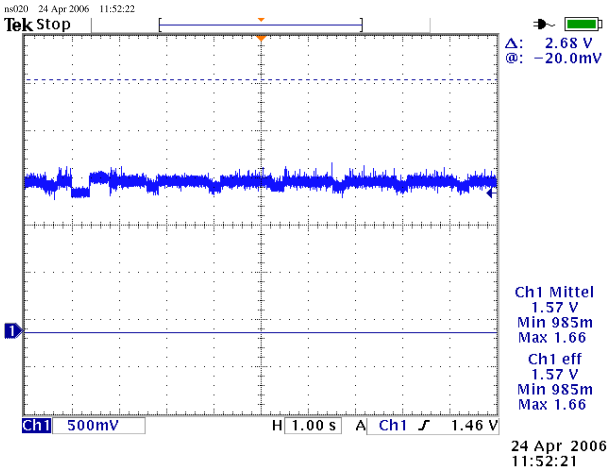

Abbildung 19: Oszilloskopanzeige beim Hochladen per JTAG-Kabel

B. Listings

Hier im Anhang werden aus Platzgründen die Quellcodes nur in verkürzter Form abgedruckt, weil die ausgelassenen Zeilen, markiert mit „...“, für das Verständinis nicht notwendig sind. Ausserdem wurde in den ausgelassenen Regionen des Codes nichts geändert. Vollständig abgedruckt sind nur die hier im Rahmen dieser Diplomarbeit erstellten Quellen.

B.1. VHDL-Quellen

Hier die Quellen, welche die erstellte Hardware beschreiben:

Listing 1: zaehler.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  -- Fuer RESET_ACTIVE
7  library proc_common_v1_00_b;
8  use proc_common_v1_00_b.proc_common_pkg.all;
9
10 entity ZAEHLER is
11     generic (
12             C_CHUNKZAEHLER : integer := 32;
13             C_CHUNKLOG     : integer := 5; --=ld 32
14             C_CHUNKZB      : integer := 32;
15             C_ZAEHLBREITE  : integer := 32
16             );
17     port ( CLK                    : in std_logic;
18             RST                    : in std_logic;
19             RD_INDICATOR           : in std_logic;
20             WR_INDICATOR           : in std_logic;
21             RD_CNT                 : out std_logic_vector(0 to
                   C_ZAEHLBREITE-1);
22             WR_CNT                 : out std_logic_vector(0 to
                   C_ZAEHLBREITE-1);
23             OVERFLOW               : out std_logic_vector(0 to
                   C_ZAEHLBREITE-1);
24             RESERVED               : out std_logic_vector(0 to
                   C_ZAEHLBREITE-1)
25             );
26 end entity ZAEHLER;
27
28 architecture ZAEHLER_ARCH of ZAEHLER is
29      -- Wenn proc_common nicht drin ist:
30 --       constant RESET_ACTIVE  : std_logic    :='1';
31
```

```
32      constant EINSEN          : std_logic_vector(0 to C_ZAEHLBREITE
            -1)
33                                   := (others => '1');
34      constant NULLEN          : std_logic_vector(0 to C_ZAEHLBREITE
            -1)
35                                   := (others => '0');
36      constant EINSLEIN        : std_logic_vector(0 to (C_ZAEHLBREITE
            /2)-1)
37                                   := (others => '1');
38      constant NULLCHEN        : std_logic_vector(0 to (C_ZAEHLBREITE
            /2)-1)
39                                   := (others => '0');
40      constant DUMMYWERT       : std_logic_vector(0 to C_ZAEHLBREITE
            -1)
41                                   := x"42412322";
42      signal RD_ZAEHL          : std_logic_vector(0 to C_ZAEHLBREITE
            -1);
43      signal WR_ZAEHL          : std_logic_vector(0 to C_ZAEHLBREITE
            -1);
44      signal OV_ZAEHL          : std_logic_vector(0 to C_ZAEHLBREITE
            -1);
45      signal RES_ZAEHL         : std_logic_vector(0 to C_ZAEHLBREITE
            -1);
46      signal OVERFLOW_RD       : std_logic_vector(0 to (C_ZAEHLBREITE
            /2)-1);
47      signal OVERFLOW_WR       : std_logic_vector(0 to (C_ZAEHLBREITE
            /2)-1);
48      signal LAST_RD_IND       : std_logic;
49      signal LAST_WR_IND       : std_logic;
50
51  begin
52
53  -- RST nicht in Sensivity, da Synchroner Reset.
54  ZAEHLEN : process (CLK, RD_INDICATOR, WR_INDICATOR)
55      begin
56           if (CLK'EVENT and CLK = '1') then
57                -- RESET-FALL (Synchron, wie X sonst auch macht)
58                if (RST = RESET_ACTIVE) then
59                     RD_ZAEHL             <= NULLEN;
60                     WR_ZAEHL             <= NULLEN;
61                     LAST_RD_IND          <= '0';
62                     LAST_WR_IND          <= '0';
63                     OVERFLOW_RD          <= NULLCHEN;
64                     OVERFLOW_WR          <= NULLCHEN;
65                 -- LESEN
66                elsif (RD_INDICATOR = '1' and LAST_RD_IND = '0' and
                       not(OVERFLOW_RD = EINSLEIN)) then
67                     LAST_RD_IND    <= '0'; -- Falls Burst
68                     if (RD_ZAEHL = EINSEN) then
69                          OVERFLOW_RD     <= OVERFLOW_RD + 1;
70                          RD_ZAEHL        <= NULLEN;
```

```
71                    else
72                         RD_ZAEHL        <= RD_ZAEHL + 1;
73                    end if;
74                -- SCHREIBEN
75               elsif (WR_INDICATOR = '1' and LAST_WR_IND = '0' and
                    not(OVERFLOW_RD = EINSLEIN)) then
76                    LAST_WR_IND    <= '0'; -- Falls Burst
77                    if (WR_ZAEHL = EINSEN) then
78                         OVERFLOW_WR     <= OVERFLOW_WR + 1;
79                         WR_ZAEHL        <= NULLEN;
80                    else
81                         WR_ZAEHL        <= WR_ZAEHL + 1;
82                    end if;
83                -- ZAEHLER VOLL (READ)
84               elsif (OVERFLOW_RD = EINSLEIN) then
85                     OVERFLOW_RD         <= NULLCHEN;
86                -- ZAEHLER VOLL (WRITE)
87               elsif (OVERFLOW_WR = EINSLEIN) then
88                     OVERFLOW_WR         <= NULLCHEN;
89                -- DEFAULT
90               else
91                    LAST_RD_IND         <= RD_INDICATOR;
92                    LAST_WR_IND         <= WR_INDICATOR;
93               end if; -- RST
94          end if; -- CLK
95     end process ZAEHLEN;
96
97      -- Overflow macht Effektiv 48-Bit zaehler
98      OV_ZAEHL(0 to (C_ZAEHLBREITE/2)-1)               <=
           OVERFLOW_RD(0 to (C_ZAEHLBREITE/2)-1);
99      OV_ZAEHL((C_ZAEHLBREITE/2) to C_ZAEHLBREITE-1)    <=
           OVERFLOW_WR(0 to (C_ZAEHLBREITE/2)-1);
100     -- Signalzuweisungen
101     RD_CNT    <= RD_ZAEHL;
102     WR_CNT    <= WR_ZAEHL;
103     OVERFLOW  <= OV_ZAEHL;
104     -- Reserved bleibt erstmal fuer Testzwecke konstant!
105     RES_ZAEHL <= DUMMYWERT;
106     RESERVED  <= RES_ZAEHL;
107
108 end architecture ZAEHLER_ARCH;
```

Listing 2: user_logic.vhd

```
1  ...
2  entity user_logic is
3       generic
4       (
5            -- ADD USER GENERICS BELOW THIS LINE ---------------
6            --USER generics added here
7            C_CHUNKZAEHLER : integer := 32;
8            C_CHUNKLOG     : integer := 5; --=ld 32
```

```
9            C_CHUNKZB      : integer := 32;
10           C_ZAEHLBREITE  : integer := 32;
11                 C_ADDRBREITE    : integer := 22;
12           -- ADD USER GENERICS ABOVE THIS LINE ---------------
13 ...
14      );
15      port
16      (
17           -- ADD USER PORTS BELOW THIS LINE ------------------
18           --USER ports added here
19           SDRAM_CLOCK             : in std_logic;
20           RD_INDICATOR            : in std_logic;
21           WR_INDICATOR            : in std_logic;
22           ADRESSE                 : in std_logic_vector(0 to 21);
23           ADR_VALID               : in std_logic;
24           -- ADD USER PORTS ABOVE THIS LINE ------------------
25
26           -- DO NOT EDIT BELOW THIS LINE ---------------------
27           -- Bus protocol ports, do not add to or delete
28           Bus2IP_Clk     : in std_logic;
29 ...
30           -- DO NOT EDIT ABOVE THIS LINE ---------------------
31      );
32 end entity user_logic;
33
34 architecture IMP of user_logic is
35
36      --USER signal declarations added here, as needed for user logic
37
38      -- Komponentendeklaration des Zugriffszaehlers
39      component ZAEHLER port (
40            CLK                     : in std_logic;
41            RST                     : in std_logic;
42            RD_INDICATOR            : in std_logic;
43            WR_INDICATOR            : in std_logic;
44            RD_CNT                  : out std_logic_vector
45                                          (0 to C_ZAEHLBREITE-1);
46            WR_CNT                  : out std_logic_vector
47                                          (0 to C_ZAEHLBREITE-1);
48            OVERFLOW                : out std_logic_vector
49                                          (0 to C_ZAEHLBREITE-1);
50            RESERVED                : out std_logic_vector
51                                          (0 to C_ZAEHLBREITE-1)
52            );
53      end component ZAEHLER;
54
55      -- Komponentendeklaration der Chunkzaehler
56      component SEITE port (
57                CLK                 : in std_logic;
58                RST                 : in std_logic;
59                CNT                 : out std_logic_vector
```

```
60                                              (0 to C_CHUNKZB-1);
61              ENABLE              : in std_logic
62              );
63      end component SEITE;
64
65      -- Komponentendeklaration des Enablers
66      component ENABLER port (
67           CLK            : in std_logic;
68           RST            : in std_logic;
69           ADRESSE        : in std_logic_vector
70                               (0 to C_ADDRBREITE-1);
71           ADR_VALID      : in std_logic;
72           ENABLE_VEC     : out std_logic_vector
73                               (0 to C_CHUNKZAEHLER-1)
74           );
75      end component ENABLER;
76
77      function EINPACKEN (C1,C2 : std_logic_vector(0 to C_CHUNKZB-1)
            )
78      return std_logic_vector is
79              variable TMP : std_logic_vector(0 to C_DWIDTH-1);
80           begin
81              TMP(0 to C_CHUNKZB-1)         := C1(0 to C_CHUNKZB-1)
                    ;
82              TMP(C_CHUNKZB to C_DWIDTH-1)    := C2(0 to C_CHUNKZB
                    -1);
83              return(TMP);
84      end function EINPACKEN;
85
86      -- Seitenzahler[i] fuellt INT_CNT[i].
87      type COUNTARRAY is array (0 to C_CHUNKZAEHLER-1) of
           std_logic_vector(0 to C_CHUNKZB-1);
88      signal INT_CNT    :    COUNTARRAY;
89
90      -- Vektor fuer das Enable der Seitenzaehler
91      signal ENABLE_VECTOR : std_logic_vector(0 to C_CHUNKZAEHLER-1);
92
93      -- Die von der Xilinx-Vorlage generiertern slv_regX durch ein
           Array ersetzen
94      -- um ueber Schleifenfunktionen auf die Indices zugreifen zu
           koennen
95      type SLAVEARRAY is array (0 to 31) of std_logic_vector(0 to
           C_DWIDTH-1);
96      signal slv_reg    : SLAVEARRAY;
97
98      signal slv_reg_write_select   : std_logic_vector(0 to 31);
99      signal slv_reg_read_select    : std_logic_vector(0 to 31);
100     signal slv_ip2bus_data   : std_logic_vector(0 to C_DWIDTH-1);
101     signal slv_read_ack : std_logic;
102     signal slv_write_ack     : std_logic;
103
```

```
104 begin
105
106     --USER logic implementation added here
107
108     -- Den Zugriffszaehler instantieren
109     Z : ZAEHLER port map (
110          CLK                 => Bus2IP_Clk,
111          RST                 => Bus2IP_Reset,
112          RD_INDICATOR        => RD_INDICATOR,
113          WR_INDICATOR        => WR_INDICATOR,
114          RD_CNT              => slv_reg(0)(0 to C_ZAEHLBREITE-1),
115          WR_CNT              => slv_reg(0)(C_ZAEHLBREITE to
                C_DWIDTH-1),
116          OVERFLOW            => slv_reg(1)(0 to C_ZAEHLBREITE-1),
117          RESERVED            => slv_reg(1)(C_ZAEHLBREITE to
                C_DWIDTH-1)
118     );
119
120     -- Den Enabler instantieren
121     E : ENABLER port map (
122          CLK            => Bus2IP_Clk,
123          RST            => Bus2IP_Reset,
124          ADRESSE        => ADRESSE,
125          ADR_VALID      => ADR_VALID,
126          ENABLE_VEC     => ENABLE_VECTOR
127          );
128
129     -- Per generate die Chunkzaehler a 32Bit erzeugen und
           Instantieren
130     PC : for i in 0 to C_CHUNKZAEHLER-1 generate
131          INSTANZ : SEITE port map(
132               CLK            => Bus2IP_Clk,
133               RST            => Bus2IP_Reset,
134               CNT            => INT_CNT(i),
135               ENABLE         => ENABLE_VECTOR(i)
136               );
137     end generate PC;
138
139
140     -- Die Chunkzaehler zugaenglich machen per EINPACKEN(), und den
           letzten nicht vergessen:
141     ACCESSIBLE : for i in 0 to (C_CHUNKZAEHLER/2)-1 generate
142          slv_reg(i+2) <= EINPACKEN(INT_CNT(i*2), INT_CNT((i*2)+1));
143     end  generate ACCESSIBLE;
144
145
146     ----------------------------------------
147     -- Example code to read/write user logic slave model s/w
           accessible registers
148     slv_reg_write_select <= Bus2IP_WrCE(0 to 31);
149     slv_reg_read_select  <= Bus2IP_RdCE(0 to 31);
```

```
150       slv_write_ack        <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1)
151       ... Bus2IP_WrCE(29) or Bus2IP_WrCE(30) or Bus2IP_WrCE(31);
152       slv_read_ack         <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1)
153       ... Bus2IP_RdCE(29) or Bus2IP_RdCE(30) or Bus2IP_RdCE(31);
154
155
156       SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg(0),
157       ... slv_reg(28), slv_reg(29), slv_reg(30), slv_reg(31) ) is
158       begin
159            case slv_reg_read_select is
160                 when "10000000000000000000000000000000" =>
                          slv_ip2bus_data <= slv_reg(0);
161  ...
162                 when "00000000000000000000000000000001" =>
                          slv_ip2bus_data <= slv_reg(31);
163                 when others => slv_ip2bus_data <= (others => '0');
164            end case;
165       end process SLAVE_REG_READ_PROC;
166
167       IP2Bus_Data        <= slv_ip2bus_data;
168       IP2Bus_WrAck       <= slv_write_ack;
169       IP2Bus_RdAck       <= slv_read_ack;
170       IP2Bus_Busy        <= '0';
171       IP2Bus_Error       <= '0';
172       IP2Bus_Retry       <= '0';
173       IP2Bus_ToutSup     <= '0';
174  end IMP;
```

Listing 3: enabler.vhd

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.STD_LOGIC_ARITH.ALL;
4   use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6   library proc_common_v1_00_b;
7   use proc_common_v1_00_b.proc_common_pkg.all;
8
9   entity ENABLER is generic(
10                C_CHUNKZAEHLER : integer := 32;
11                C_CHUNKLOG     : integer := 5; --=ld 32
12                C_CHUNKZB      : integer := 32;
13                C_ZAEHLBREITE  : integer := 32;
14                C_ADDRBREITE   : integer := 22
15          );
16
17          port (
18          CLK            : in std_logic;
19          RST            : in std_logic;
20          ADRESSE        : in std_logic_vector(0 to C_ADDRBREITE-1);
21          ADR_VALID      : in std_logic;
```

```
22          ENABLE_VEC     : out std_logic_vector(0 to C_CHUNKZAEHLER
                -1)
23          );
24 end entity ENABLER;
25
26 architecture ENABLER_ARCH of ENABLER is
27
28 begin
29      -- Die Adresse auf die ld(Chunkzaehlerbreite) ausschneiden, und
            den 'Multiplexerwert'
30      -- fuer die Enable-Signale der Chunkzaehler aktivieren
31      EN : process (ADR_VALID, ADRESSE, CLK)
32              variable ADDR           : std_logic_vector(0 to
                    C_CHUNKLOG-1);
33          begin
34              ADDR(0 to C_CHUNKLOG-1) := ADRESSE(0 to 4);
35              if (CLK'event and CLK = '1') then
36                  if (RST = RESET_ACTIVE) then
37                      ENABLE_VEC(0 to C_CHUNKZAEHLER-1) <= (
                            others => '0');
38                  elsif (ADR_VALID = '1') then
39                      for i in 0 to C_CHUNKZAEHLER-1 loop
40                          if ( conv_integer(unsigned(ADDR)) = i
                                ) then
41                              ENABLE_VEC(i) <= '1';
42                          else
43                              ENABLE_VEC(i) <= '0';
44                          end if;
45                      end loop;
46                  else -- ADR_VALID='0'
47                      ENABLE_VEC(0 to C_CHUNKZAEHLER-1) <= (
                            others => '0');
48                  end if; --RST ADR_VALID
49              end if; --CLK
50      end process EN;
51
52 end architecture ENABLER_ARCH;
```

Listing 4: plb_slave_attachment_indet.vhd

```
1  ...
2  -- PLB Slave attachment entity and architecture
3  ...
4  entity plb_slave_attachment_indet is
5  ...
6      port(
7  ...
8          -- MEINE AENDERUNGEN EINGEPFLEGT:
9          ADRESSE         : out std_logic_vector(0 to 21);
10         ADR_VALID       : out std_logic
11         );
12 end entity plb_slave_attachment_indet;
```

```
13
14 architecture implementation of plb_slave_attachment_indet is
15 ...
16     begin
17 ...
18 -- MEINE AENDERUNGEN EINGEPFLEGT:
19 FANGEN : process (plb_abus_reg, plb_pavalid_reg, Bus_Clk)
20           constant NULLEN_1 : std_logic_vector(0 to 6)   := (others
                 => '0');
21           constant NuLLEN_2 : std_logic_vector(29 to 31) := (others
                 => '0');
22      begin
23           if (Bus_Clk'event and Bus_Clk = '1') then
24                if ((plb_abus_reg(0 to 6) = NULLEN_1) and (
                       plb_abus_reg(29 to 31) = NULLEN_2)) then
25                  ADRESSE(0 to 21)    <= plb_abus_reg(7 to 28);
26                  ADR_VALID           <= plb_pavalid_reg;
27                end if; -- AUF sonstige Zeichen Checken
28           end if; --CLK
29      end process FANGEN;
30 --
31 ...
32 end implementation;
```

B.2. Kernel-Quellen

Listing 5: treiber-driver.c

```
1  #include <linux/kernel.h>
2  #include <linux/module.h>
3  #include <linux/fs.h>
4  #include <linux/version.h>
5  #include <linux/module.h>
6  #include <linux/init.h>
7  #include <linux/ioport.h>
8  #include <linux/errno.h>
9  #include <asm/io.h>
10 #include <linux/proc_fs.h>
11 #include <linux/types.h>
12 #include <linux/errno.h>
13 #include <linux/time.h>
14 #include <linux/kernel.h>
15 #include <linux/kernel_stat.h>
16 #include <linux/tty.h>
17 #include <linux/string.h>
18 #include <linux/mman.h>
19 #include <linux/proc_fs.h>
20 #include <linux/ioport.h>
21 #include <linux/config.h>
22 #include <linux/mm.h>
```

```
23 #include <linux/mmzone.h>
24 #include <linux/pagemap.h>
25 #include <linux/swap.h>
26 #include <linux/slab.h>
27 #include <linux/smp.h>
28 #include <linux/signal.h>
29 #include <linux/module.h>
30 #include <linux/init.h>
31 #include <linux/smp_lock.h>
32 #include <linux/seq_file.h>
33 #include <linux/times.h>
34 #include <linux/profile.h>
35 #include <linux/blkdev.h>
36 #include <linux/hugetlb.h>
37 #include <linux/jiffies.h>
38 #include <linux/sysrq.h>
39 #include <linux/vmalloc.h>
40 #include <linux/crash_dump.h>
41 #include <asm/uaccess.h>
42 #include <asm/pgtable.h>
43 #include <asm/io.h>
44 #include <asm/tlb.h>
45 #include <asm/div64.h>
46 #include "xparameters_ml300.h"
47 #include "funktionen.h"
48
49 #ifndef KERN_INFO
50      #define KERN_INFO
51 #endif
52 #ifndef KERN_ERR
53      #define KERN_ERR
54 #endif
55
56 #ifndef XPAR_PLBADDON_0_BASEADDR
57      #define XPAR_PLBADDON_0_BASEADDR 0xFFFE0000
58      #define XPAR_PLBADDON_0_HIGHADDR 0xFFFEFFFF
59 #endif
60 #define MAX_MELDUNG PAGE_SIZE
61
62 MODULE_DESCRIPTION("PLBADDON-Treiber");
63 MODULE_AUTHOR("Thomas Hirth (hirthts@iis.fraunhofer.de)");
64 MODULE_LICENSE("GPL");
65
66 struct plbaddon __initdata zaehl;
67 struct plbaddon zaehl;
68 struct dev device = {0,0,0,0, "keiner" ,0};
69
70 static struct proc_dir_entry *plbaddon_dir;
71 static struct proc_dir_entry *plbaddon_file;
72
73
```

```
74 static int proc_read_plbaddon (char *page, char **start, off_t off,
       int count, int *eof, void *data)
75 {
76      int len = 0;
77      int i = 0;
78      char tmp[4095];
79
80      if (device.initialisiert == 1) {
81           aktualisiere_global_zaehler(device.baseaddress);
82
83           len  = sprintf(tmp, "ALLE SPEICHERWERTE von 0x%x an:\n",
                 device.baseaddress);
84           len  = len + sprintf(tmp+len, "Lesezugriffe    = %llu\n",
                 zaehl.zaehler[0]);
85           len  = len + sprintf(tmp+len, "Schreibzugriffe = %llu\n",
                 zaehl.zaehler[1]);
86           for(i = 0; i < 32; i++){
87                len  = len + sprintf(tmp+len, "Chunkzaehler[%2d] = 0x
                     %16llx\n", i, zaehl.zaehler[i+2]);
88           }
89      } else {
90           len  = sprintf(tmp, "PLBADDON nicht intitialisiert. Keine
                 Werte da.\n");
91      }
92
93      if(len != sprintf(page, "%s", tmp)) printk("PLBADDON:
           Kopierfehler mit sprintf()\n");
94
95      if (len > MAX_MELDUNG) {
96           printk( KERN_ERR "PLBADDON: proc_fs Eintrag zu gross (%
                 dByte)\n", len );
97      }
98
99      return len;
100 }; /* proc_read_plbaddon() */
101
102
103 static int __init treiber_init_module(void)
104 {
105      int rv = 0;
106
107      device.initialisiert = 0;
108      device.baseaddr_real = XPAR_PLBADDON_0_BASEADDR;
109      device.highaddr_real = XPAR_PLBADDON_0_HIGHADDR;
110      sprintf(device.name, "PLBADDON");
111
112      printk( KERN_INFO "PLBADDON: wird intitialisiert\n" );
113
114      if (check_mem_region(device.baseaddr_real,
115                (device.highaddr_real-device.baseaddr_real)+1) < 0) {
116           printk( KERN_ERR "PLBADDON: check_mem Fehlgeschlagen\n");
```

```
117           rv = -EBUSY;
118           goto ende;
119           }
120       request_mem_region(device.baseaddr_real,
121                 (device.highaddr_real-device.baseaddr_real)+1, device
                      .name);
122
123       if ((device.baseaddress = (unsigned int) ioremap(device.
              baseaddr_real,
124                 (device.highaddr_real-device.baseaddr_real)+1)) == 0)
125                 printk( KERN_ERR "PLBADDON: IoReMap Fehlgeschladen\n
                      ");
126       device.highaddress = device.baseaddress +
127                 (device.highaddr_real-device.baseaddr_real);
128
129       plbaddon_dir =(struct proc_dir_entry*) proc_mkdir(device.name,
              NULL);
130           if( plbaddon_dir == NULL) {
131                 rv = -ENOMEM;
132                 printk( KERN_ERR "PLBADDON: mkdir Fehlgeschladen\n");
133                 remove_proc_entry(device.name, NULL);
134                 goto ende;
135           }
136       plbaddon_dir->owner = THIS_MODULE;
137       plbaddon_file =(struct proc_dir_entry*) create_proc_read_entry
138                       (device.name, 0, plbaddon_dir,
                            proc_read_plbaddon, NULL);
139       if( plbaddon_file == NULL) {
140           rv  = -ENOMEM;
141           printk( KERN_ERR "PLBADDON: Datei Fehlgeschladen\n");
142           remove_proc_entry(device.name, plbaddon_dir);
143           goto ende;
144       }
145       plbaddon_file->owner = THIS_MODULE;
146
147       device.initialisiert = 1;
148
149       printk( KERN_INFO "PLBADDON: Adresse von 0x%08X nach 0x%08X
              gemappt\n"
150                           , device.baseaddr_real, device.baseaddress)
                                ;
151       printk( KERN_INFO "PLBADDON: initialisiert\n" );
152       return rv;
153
154   ende:
155       printk( KERN_ERR "PLBADDON: Fehlgeschladen\n");
156       return rv;
157   }; /* treiber_init_module() */
158
159
160   static void __exit treiber_exit_module(void)
```

```
161 {
162      printk( KERN_INFO "PLBADDON: wird entladen\n" );
163 
164      device.initialisiert = 0;
165      remove_proc_entry(device.name, plbaddon_dir);
166      remove_proc_entry(device.name, NULL);
167      iounmap((void*) device.baseaddress);
168      release_mem_region(device.baseaddr_real,
169           (device.highaddr_real-device.baseaddr_real)+1);
170 
171      printk( KERN_INFO "PLBADDON: entladen.\n" );
172 }; /* treiber_exit_module() */
173 
174 
175 module_init(treiber_init_module);
176 module_exit(treiber_exit_module);
```

Listing 6: funktionen.h

```
1  #ifndef FUNKTIONEN_H
2  #define FUNKTIONEN_H
3  
4  #include <linux/kernel.h>
5  #include "xdma_channel.h"
6  #include "xbasic_types.h"
7  #include "xio.h"
8  
9  #ifndef KERN_INFO
10      #define KERN_INFO
11 #endif
12 #ifndef KERN_ERR
13      #define KERN_ERR
14 #endif
15 
16 struct plbaddon
17 {
18      unsigned long long int hwget[32];
19      unsigned int hwraw[4+32];
20      /* Soll das Auswertungsarray mit einzelnen Zaehlerstaenden
           enthalten: */
21      unsigned long long int zaehler[2+32];
22       /* äEnthlt den letzten Stand, um die Differenz ermitteln zu
            koennen: */
23      unsigned long long int letzter[2+32];
24 }; /* sruct plbaddon */
25 extern struct plbaddon zaehl; /* Globale Variable */
26 
27 struct dev
28 {
29      unsigned int baseaddr_real;
30      unsigned int highaddr_real;
31      unsigned int baseaddress;
```

```
32      unsigned int highaddress;
33      char name[10];
34      int initialisiert;
35 }; /* struct dev */
36 extern struct dev device;
37
38 void aktualisiere_global_zaehler(unsigned int);
39
40 #endif
```

Listing 7: funktionen.c

```
1  #include "funktionen.h"
2
3  static void dataaquisition(unsigned int addr)
4  {
5       int i = 0;
6       unsigned int thisaddr = 0;
7
8       for(i = 0; i < 32; i++){      /* lese 64 Register aus */
9            thisaddr = addr + (2*i*sizeof(int));
10           zaehl.hwget[i] = (((unsigned long long int)XIo_In32(
                 thisaddr)) << 32);
11           thisaddr += sizeof(int);
12           zaehl.hwget[i] += XIo_In32(thisaddr);
13      };
14 }; /* ENDE dataaqusition() */
15
16
17 static void entpacken(unsigned long long int *in)
18 {
19      int i = 0;
20
21      for (i = 0; i < ((32+2)/2) ; i++){
22           zaehl.hwraw[i*2]   = in[i] >> (64-32);
23           zaehl.hwraw[i*2+1] = in[i] & 0x00000000FFFFFFFFull;
24      }
25      if (zaehl.hwraw[3] != 0x42412322)
26                printk(KERN_INFO "\nPLBADDON: Auslesefehler.
                      Reserved=0x%08X ist nicht 0x42412322.\n", zaehl.
                      hwraw[3]);
27 }; /* ENDE entpacken() */
28
29
30 static void update (void) {
31      unsigned long long int tmp[2+32];
32      int i;
33
34      /* reads: */
35      tmp[0] = zaehl.hwraw[0] + (zaehl.hwraw[2] >> 16);
36
37      /* writes: */
```

```
38      tmp[1] = zaehl.hwraw[1] + (zaehl.hwraw[2] & 0x0000FFFF);
39
40      /* zaehl.hwraw[3] ist der Reserverd ---> brauchen wir nicht
           mehr */
41
42      /* Fuer .hwraw von [4] bis [32+4] */
43      for ( i = 4; i < 4+32; i++){
44                tmp[i-2] = zaehl.hwraw[i];
45      }
46
47      /* Externe Arrays updaten */
48      /* es sind in .letzter und .zaehler immer nur 2+32 Stueck drin:
            */
49      for ( i = 0; i < 32+2; i++){
50           zaehl.letzter[i] = zaehl.zaehler[i];
51           zaehl.zaehler[i] = tmp[i];
52      }
53  }; /* ENDE update()*/
54
55
56  void aktualisiere_global_zaehler(unsigned int addr){
57       dataaquisition(addr);
58       entpacken(zaehl.hwget);
59       update();
60  }; /* ENDE aktualisiere_zahler() */
```

Listing 8: sched.c

```
1   /*
2    *  kernel/sched.c
3    *
4    *  Kernel scheduler and related syscalls
5    *
6    *  Copyright (C) 1991-2002  Linus Torvalds
7    ...
8    */
9   ...
10
11  #include "funktionen.h" /* Fuer meine Erweiterung (hirthts)*/
12  ...
13  /*
14   * wake_up_new_task - wake up a newly created task for the first
       time.
15   *
16   * This function will do some initial scheduler statistics
       housekeeping
17   * that must be done for every newly created context, then puts the
       task
18   * on the runqueue and wakes it.
19   */
20  void fastcall wake_up_new_task(task_t *p, unsigned long clone_flags)
21  {
```

```
...
/** Meine Ergaenzung, damit die Tasks Initialisierte Zaehler haben //hirthts */
      for(i =0; i < 32+2; i++){p->zaehler_lokal[i] = 0;}

...
}
...
/*
 * context_switch - switch to the new MM and the new
 * thread's register state.
 */
static inline
task_t * context_switch(runqueue_t *rq, task_t *prev, task_t *next)
{
     int i;
     extern struct plbaddon zaehl;
     extern struct dev device;

     struct mm_struct *mm = next->mm;
     struct mm_struct *oldmm = prev->active_mm;

     /** Hier die Zaehler aktualisieren //hirthts */
     if (device.initialisiert == 1) {
          aktualisiere_global_zaehler(device.baseaddress);
          for(i=0; i < (2+32); i++){
               prev->zaehler_lokal[i] += zaehl.zaehler[i] - zaehl.letzter[i];
          }
     } else printk(KERN_ERR "SCHED: PLBADDON nicht intitialisiert\n");
...
}
...
```

Listing 9: sched.h

```
#ifndef _LINUX_SCHED_H
#define _LINUX_SCHED_H
...
struct task_struct {
     volatile long state;/* -1 unrunnable, 0 runnable, >0 stopped */
...
     atomic_t fs_excl;   /* holding fs exclusive resources */
/** Hier muessen meine Erweiterungen rein: in task_struct zum finden //hirthts */
     unsigned long long int zaehler_lokal[2+32];
};
...
```

Listing 10: exit.c

```
/*
```

```
2   *  linux/kernel/exit.c
3   *
4   *  Copyright (C) 1991, 1992  Linus Torvalds
5   */
6   ...
7   #include "funktionen.h" /* Fuer meine Erweiterung (hirthts)*/
8   ...
9   void release_task(struct task_struct * p)
10  {
11       int zap_leader;
12       task_t *leader;
13       struct dentry *proc_dentry;
14
15  /** Meine Aenderung: Ich glaube, hier kommen die Tasks weg (hirthts)
         */
16       int i;
17       extern struct plbaddon zaehl;
18       extern struct dev device;
19       if (device.initialisiert == 1) {
20            aktualisiere_global_zaehler(device.baseaddress);
21            for(i=0; i < (2+32); i++){
22                 p->zaehler_lokal[i] += zaehl.zaehler[i] - zaehl.
                       letzter[i];
23            }
24       }
25       printk(KERN_DEBUG "PLBADDON: Task #%d (%s) beended.\n", p->pid,
              p->comm);
26       for(i=0;i < 32+2;i++){
27            printk(KERN_DEBUG "Zaehlerstand[%2d]: %llu\n", i, p->
                  zaehler_lokal[i]);
28       }
29  ...
```

Danksagung

Diese Arbeit wurde vom Lehrstuhl 4 des Instituts für Informatik (INF4) an der Technischen Fakultät (TF) der Friedrich-Alexander Universität Erlangen-Nürnberg (FAU) betreut, aber extern im Fraunhofer Institut für Integrierte Schaltungen (IIS), Abteilung Leistungsoptimierte Systeme (LOS) durchgeführt.

Zuallererst möchte ich mich bei meinen Betreuern Christian Fiermann, Andreas Weißel und Prof. Dr.-Ing. Wolfgang Schröder-Preikschat für die Überlassung dieses Themas, für Erlaubnis zur externen Durchführung und die Betreuung bedanken.

Dankend möchte ich an dieser Stelle auch andere erwähnen, die mir bei der Anfertigung dieser Arbeit stets mit Rat und Tat zu Seite standen. Dies betrifft in der Abteilung Leistungsoptimierte Systeme (LOS) des Fraunhofer IIS neben den übrigen Mitarbeitern vor allem Marco Zebisch, Javier Gutierrez, Cosmin Codrea und Stefan Kerpes.

Mein ganz besonderer Dank für das Ausmerzen unzähliger Typografie- und Stilfehler gilt Susanne Müller und Monika Zehnder.

Vielen Dank auch an meine Partnerin für die Geduld und Verständnis sowie an meine Eltern, die mich in meiner langen schulischen und universitären Ausbildung jederzeit unterstützt haben.