Methodik für die Verlustleistungsabschätzung von Prozessoren mit Pipeline-Strukturen

Robert Fischer

Universität der Bundeswehr München

der Bundeswehr
Universität München

Methodik für die Verlustleistungsabschätzung von Prozessoren mit Pipeline-Strukturen

vorgelegt von

Robert Fischer

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

an der
Fakultät für Informatik
der Universität der Bundeswehr München

Promotionsausschuss:

Vorsitzender:	**Prof. Dr. Peter Hertling**
1. Gutachter:	**Prof. Klaus Buchenrieder, Ph.D.(OSU)**
2. Gutachter:	**Prof. Dr. Gabi Dreo Rodosek**

Tag der Prüfung: 17.05.2010

Neubiberg, Mai 2010

Kurzfassung

Die derzeit im Hardware- und Software-Design zur Abschätzung der Verlustleistung von Prozessoren eingesetzten Methoden auf Register-Transfer-Ebene oder Gatterebene sind sehr zeit- und arbeitsaufwendig. Gute Alternativen bieten daher Abschätzungsverfahren auf einer höheren Abstraktionsebene, so auch die instruktionsbasierte Abschätzung. Doch ist bei diesen Verfahren der noch zu optimierende Faktor die Genauigkeit. Die bisherigen Ansätze bilden die Verlustleistungsänderungen durch Pipeline-Hazards nicht im vollen Umfang ab oder vernachlässigen sie ganz. Hierdurch ist einerseits eine taktgenaue Abschätzung, die vor allem zur Hardware-Optimierung benötigt wird, nicht möglich und andererseits wird auch bei der Abschätzung der Verlustleistung für Programmabschnitte, wie sie von Software-Entwicklern benötigt wird, ein vermeidbarer Fehler in Kauf genommen.

In dieser Arbeit wird eine neue Methodik vorgestellt, um alle gängigen von Pipelining verursachten Veränderungen der Verlustleistung zu erfassen und diese in eine Berechnung für eine abgewandelte instruktionsbasierte Verlustleistungsabschätzung einzubeziehen.

Es werden die Pipeline-Strukturen von Prozessoren und ihr Einfluss auf die Verlustleistung untersucht, mit besonderer Berücksichtigung der verschiedenen Pipeline-Hazards. Daraus ergibt sich ein allgemein gültiges Modell sowohl für die Betrachtung der Verlustleistung von unterschiedlichen Pipeline-Strukturen als auch für Verlustleistungsoptimierungen von Pipelines in Prozessoren.

Die dafür notwendige Charakterisierung des Prozessors erfolgt durch Methoden, die im vom Bundesministerium für Bildung und Forschung geförderten Projekt LEMOS (**L**ow-Power - **E**ntwurfsmethoden für **mo**bile **S**ysteme) zusammen mit der Infineon Technologies AG entwickelt wurden.

In dieser Arbeit werden Strategien vorgeschlagen, um ein an die entsprechende Aufgabe angepasstes Optimum zwischen Genauigkeit und Geschwindigkeit bei der Abschätzung zu erreichen. Hierzu werden in drei Fallbeispielen mit Soft-Core Prozessoren verschiedene Aspekte dieser neuen Methode den Standardverfahren gegenübergestellt und verglichen. Die in dieser Arbeit vorgestellte Methode zeigt dabei deutlich genauere Ergebnisse – bei nur minimal höherem Rechen- und Modellierungsaufwand.

Abstract

The methods currently used in hardware and software design for estimating the power consumption of processors on the register-transfer-level or logic-level are very time consuming and complex. A good alternative are power estimation methods that work on a higher abstraction level, like the instruction based power estimation. Accuracy, however, could still be optimized with this method. The approaches so far do not fully take into account the changes in power consumption caused by pipeline hazards or even neglect them completely. Therefore a cycle accurate estimation, particularly necessary for optimizing the hardware, is not possible. In addition there is an unnecessary error introduced for the power estimation of whole program parts, which is needed by software engineers.
In this thesis a new method is being introduced, which is able to cover all common alterations of power consumption caused by pipelining and, at the same time, is taking them into account for a modified instruction based calculation of power consumption.
Various pipeline structures of processors and their influences on power consumption are being examined, with special emphasis on power consumption caused by pipeline hazards. From the results of these tests follows a model of universal validity as well for the examination of power consumption of various pipeline structures as also for the optimizing of power consumption of pipelines in processors.
The characterization of the processor needed for this, is being achieved by methods which have been developed in cooperation with Infineon Technologies AG in the German Bundesministerium für Bildung und Forschung supported LEMOS (Low-Power Design Methods for Mobile Systems) project.
Suggestions for strategies are being given in order to obtain an optimum between accuracy and speed as the application may require. Various aspects of this new method are being set against and compared with the standard proceedings by means of three case examples with soft-core processors. The method introduced in this thesis gives hereby clearly more accurate results – with only a minimal higher work cost.

Danksagung

Ich möchte mich ganz besonders bei meinem Doktorvater Prof. Buchenrieder bedanken, einerseits dafür, dass er mir bei der Verwirklichung dieser Arbeit viel freie Hand gelassen hat, aber mir andererseits auch immer mit fachlichem Rat zur Seite stand. Weiterhin danke ich allen meinen Kollegen am Institut für Technische Informatik der Universität der Bundeswehr für das ideale Arbeitsklima und die freundschaftliche Atmosphäre, insbesondere Heike Rolfs, Rainer Scholz und Dr. Herbert Kleebauer auch für die fachliche Unterstützung.
Besonderer Dank gilt Prof. Huazhong Yang von der Tsinghua Universität in Peking, der mir einen dreimonatigen Forschungsaufenthalt an seinem Institut ermöglicht und durch Diskussionen und Ratschläge somit wesentlich zu Kapitel 7.1 beigetragen hat.
Außerdem möchte ich mich noch bei meinen Eltern und besonders bei meiner Frau Jing für ihre Geduld und Fürsorge bedanken.
Schließlich danke ich allen weiteren Personen, die mich direkt oder indirekt beim Zustandekommen dieser Arbeit unterstützt haben.

Teile dieser Arbeit wurden unterstützt durch das Institut für Technik Intelligenter Systeme (ITIS e.V.) an der Universität der Bundeswehr München, durch die Infineon Technologies AG und den Deutschen Akademischen Austausch Dienst (DAAD).

Der Druck wurde gefördert aus Haushaltsmitteln der Universität der Bundeswehr München.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Geringer Leistungsverbrauch ist vor allem bei eingebetteten Systemen ein Wettbewerbsvorteil, der zusehends an Bedeutung gewinnt. Produkte länger ohne externe Stromversorgung betreiben zu können ist bei mobilen Geräten neben neuen Funktionen mittlerweile das Hauptziel der Investitionen in neue Forschung. Dass der Trend in letzter Zeit mehr zu energieeffizienten Systemen geht, zeigt auch der extrem energiesparende Atom Prozessor der Firma Intel, der schon nach kurzer Zeit, nicht zuletzt durch die neue Geräteklasse der Netbooks, zu einem weltweiten Erfolg wurde.
Aber auch bei Produkten, die mit einer netzgebundenen Stromversorgung betrieben werden, hat eine geringere Verlustleistung und der damit verbundene niedrigere Energieverbrauch eine hohe Priorität. Große Rechenzentren wie das Leibnitz Rechenzentrum in München verbrauchen mit fast fünf Megawatt genauso viel Strom wie 5000 Haushalte. Das Unternehmen Google, das eine der größten Server-Farmen der Welt besitzt, hat dieses Problem erkannt und versucht unter anderem durch effizientere Computer seinen Energieverbrauch zu senken [2].
Auch bei einfachen Desktop PCs ist das Einsparungspotential gewaltig, da mittlerweile in fast jedem Haushalt mindestens ein PC steht, und kaum ein Arbeitsplatz mehr ohne ihn auskommt. Wenn man durch energieeffizientere Prozessoren nur wenige Prozent der Verlustleistung sparen könnte, würde sich das allein durch die enorme Anzahl der Computer zu einem gewaltigen Energiesparpotenzial zusammenrechnen. Und als größter Energieverbraucher im PC [22] ist das Einsparungspotential beim Prozessor auch durch geringe Optimierungen noch am größten.
Des weiteren wird durch einen niedrigeren Energieverbrauch auch die Wärmeabgabe von integrierten Schaltungen reduziert. Diese Hitzeentwicklung erfordert nicht nur zusätzliche Kühlungsmaßnahmen, sondern lässt auch andere nahe gelegene Bauteile schneller altern oder zerstört sie sogar schlimmstenfalls. Die hohe Wärmeentwicklung ist bei Standard PC-

Prozessoren sogar ein so großes Problem geworden, dass man von dem traditionellen Single-Core-Ansatz abweichen musste, um die Prozessorleistung noch weiter zu steigern und den Aufwand für die Kühlung beherrschbar zu halten. Gerade in Rechenzentren braucht diese Kühlung sehr viel Energie: Im bereits genannten Münchner Leibnitz Rechenzentrum werden zwei Megawatt, also fast die Hälfte der gesamten verbrauchten Energie, für die Kühlung der Rechner verwendet [27].

1.2 Problemstellung

Um die Verlustleistung von Prozessoren zu senken, kann man nicht mehr nur die Hardware betrachten, sondern muss auch die Software optimieren. Vor allem das Zusammenspiel von Hardware und Software spielt eine immer wichtigere Rolle.
Dabei ist es notwendig, verschiedene Programm- und Prozessorkonfigurationen schnell in Bezug auf den Energieverbrauch untersuchen und vergleichen zu können. Gerade der Aufbau und die Struktur der Prozessor-Pipeline kann hier eine bedeutende Rolle spielen. Bei der Entwicklung des Intel Atom Prozessor konnte auch durch Änderungen an der Pipeline-Struktur eine enorme Reduzierung des Energieverbrauchs erreicht werden [9].
Um diese Untersuchungen durchführen zu können, waren bisher aufwendige Messungen an einem realen Prozessor notwendig. Dies ist oft schwierig, da die Anschlüsse des Prozessors für die Messungen nicht direkt erreichbar sind, und das Bauteil meist fest auf einer Platine eingebettet ist.
Eine Alternative ist die simulative Abschätzung der Verlustleistung auf einem Modell des Prozessors. Mit einer Simulation auf einer niedrigen Abstraktionsebene (Low-Level Simulation) kann die Verlustleistung eines Systems recht genau bestimmt werden ohne es physikalisch aufbauen zu müssen. Diese Methode ist aber wegen des relativ hohen Zeitaufwands für einen schnellen Vergleich zweier Alternativen nicht gut geeignet. Bei großen Designs und längerer Beobachtungsdauer über mehrere Millionen von Takten kann die Simulation mit heutiger Technologie sogar mehrere Tage dauern. Auch muss bei jeder Änderung am ursprünglichen Hardware-Design das komplette System neu synthetisiert und simuliert werden.
Es ist daher wünschenswert, die Verlustleistungsanalyse auf eine höhere Abstraktionsebene zu verlagern und so bereits möglichst in einer frühen Design-Phase Schätzwerte über den späteren Stromverbrauch zu erhalten.

Die bisherigen Methoden, welche eine schnelle Abschätzung gestatten, sind jedoch nicht vorrangig für Prozessoren mit Pipelining ausgelegt, bei denen die Verlustleistung vom ausgeführten Code und von den zu bearbeitenden Daten abhängt, aber zusätzlich die Zustände in der Pipeline beachtetet werden müssen.

1.3 Ziel der Arbeit

In dieser Arbeit wird eine schnelle und doch sehr genaue Methodik zur Abschätzung der Verlustleistung von Pipeline-Strukturen in Prozessoren vorgestellt. Die Pipeline wird dabei in Pipeline-Stufen zerlegt, und die Verlustleistung der einzelnen Stufen abgeschätzt. Mittels einer genauen Nachbildung des Befehlsflusses durch die Pipeline können Besonderheiten bei der Abarbeitung von Befehlen in einer Prozessor-Pipeline für die Abschätzung der Verlustleistung besser berücksichtigt werden.
Auch wird dadurch ein flexiblerer Umgang mit unterschiedlichen Pipeline-Strukturen und ein leichtes Anpassen des Verlustleistungsmodells an verschiedene Prozessor-Pipeline-Typen ermöglicht.
Daraus wird ein allgemein gültiges Modell für die Betrachtung der Verlustleistung von unterschiedlichen Pipeline-Strukturen entwickelt, und es werden allgemeine Empfehlungen für Verlustleistungsoptimierungen von Pipelines in Prozessoren gegeben.
Im Vergleich zu den bisherigen Methoden zur Abschätzung der Verlustleistung hat das in dieser Arbeit vorgestellte Verfahren zwei Vorteile: Einerseits ist es flexibler einsetzbar, da verschiedene Strukturen von Prozessor-Pipelines leicht bezüglich des Verlustleistungsverbrauchs gegeneinander abgewogen werden können. Andererseits erhöht es die Genauigkeit der bisherigen instruktionsbasierten Methode ohne mehr Rechenzeit für die Abschätzung zu benötigen.
Die Genauigkeit und die Geschwindigkeit dieser neuen Methodik werden durch drei repräsentative Fallbeispiele nachgewiesen und es werden Hinweise zur Durchführung von Abschätzungen an Prozessoren mit Pipelines gegeben.

1.4 Begriffserklärungen

FPGAs (Field Programmable Gate Arrays) sind integrierte Halbleiterbausteine, deren Logik und Verdrahtung noch im Endgerät vom Anwender verändert werden können.

Soft-Core Prozessoren sind Prozessoren, die in einer Hardware-Beschreibungssprache beim Anwender vorliegen, von diesem noch verändert und so an eine spezifische Aufgabe angepasst werden können. Sie kommen meist auf FPGAs zum Einsatz.

IP-Cores sind kommerzielle Soft-Cores, deren Hardware-Beschreibung nicht im Quellcode beim Anwender vorliegt. Sie können aber meist durch eine vom Hersteller mitgelieferte Konfigurationssoftware in gewissem Maße angepasst werden.

1.5 Überblick über diese Arbeit

Ein Überblick über Vorgehen und Ergebnisse der vorliegenden Arbeit ist in Abbildung 1.1 schematisch dargestellt. Die wichtigsten Ergebnisse sind dick umrandet und Informationsflüsse werden dabei durch gestrichelte Pfeile gekennzeichnet.

Zuerst werden in Kapitel 2 die Grundlagen zur Verlustleistungsabschätzung von Prozessoren erklärt. Hierzu wird der Energieverbrauch in integrierten Schaltungen beschrieben und in einen statischen und einen dynamischen Verbrauch aufgeteilt. Außerdem werden verschiedene Methoden der Verlustleistungsabschätzung von Prozessoren vorgestellt, und relevante Beispiele aus der aktuellen Forschung zu diesen gegeben.
Kapitel 3 gibt einen Überblick über Pipelining von Prozessoren. Es werden Pipeline-Hazards und deren technische Lösungen erklärt und verschiedene von der einfachen Pipeline abweichende Strukturen vorgestellt.
Das Prinzip der in dieser Arbeit vorgestellten Methode wird in Kapitel 4 vorgestellt. Die Verlustleistung des Prozessors wird dabei in die Verlustleistungen der einzelnen Pipeline-Stufen unterteilt, wodurch die Abläufe in der Prozessor-Pipeline besser nachvollzogen und der Energieverbrauch des Prozessors taktgenau abgeschätzt werden können.

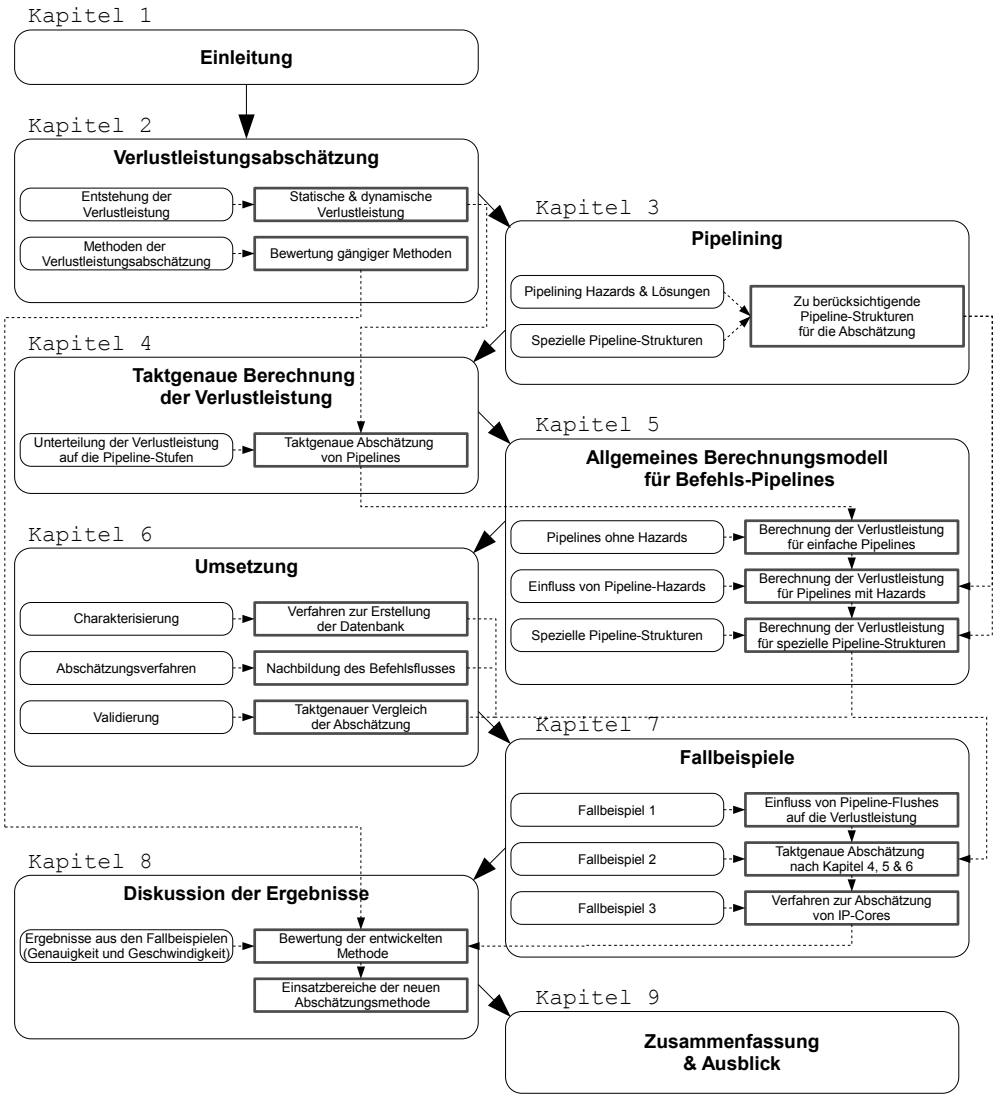

Abbildung 1.1: Vorgehensmodell und Ergebnisse der Arbeit

In Kapitel 5 werden allgemein gültige Formeln und theoretische Methoden erarbeitet, welche für die Berechnung des Energieverbrauchs der in Kapitel 3 beschriebenen Pipeline-Strukturen angewandt werden können.
Erläuterungen zur Umsetzung dieser Methoden werden in Kapitel 6 gegeben. Dazu werden die Charakterisierung der Energiewerte für die Pipeline-Stufen erläutert und der Aufbau der benötigten Datenbank beschrieben. Weiterhin wird die Erstellung eines um Pipeline-Stalls und -Flushes erweiterten Befehls-Traces gezeigt, welcher für die Abschätzung des Energieverbrauchs benötigt wird. Außerdem wird das Verfahren zur Validierung der Ergebnisse vorgestellt.
Anhand von drei Fallbeispielen in Kapitel 7 wird die neue Methode praktisch nachgewiesen. Fallbeispiel 1 zeigt an dem AVR-Core Prozessor den Einfluss von Branch Instruktionen und speziell den Einfluss von Pipeline-Flushes auf den Energieverbrauch des Prozessors. Fallbeispiel 2 zeigt die taktgenaue Abschätzung nach der in den Kapiteln 4, 5 und 6 entwickelten Methode anhand des Jam CPU Prozessors. Für die Abschätzung eines IP-Cores, dessen innerer Aufbau nicht bekannt ist, wird in Fallbeispiel 3 anhand des MicroBlaze Prozessors eine Methode vorgestellt.
Die Ergebnisse dieser Fallbeispiele werden in Bezug auf Zeitaufwand und Abschätzgenauigkeit in Kapitel 8 diskutiert. Es werden generelle Ergebnisse aus den Untersuchungen diskutiert und mögliche Einsatzbereiche der in dieser Arbeit vorgestellten Methode aufgezeigt.
Kapitel 9 fasst die gesamte Arbeit noch einmal zusammen und gibt einen Ausblick für künftige Forschungsarbeiten.

2 Methoden der Verlustleistungsabschätzung

Als Verlustleistung eines Prozessors bezeichnet man die Differenz zwischen aufgenommener Leistung und der für die Befehlsausführung notwendigen Leistung. Die Verlustleistung ist nahezu identisch mit der abgegebenen Wärmemenge, da ein Prozessor die Energie weder speichern noch in eine andere Form umwandeln kann.
In diesem Kapitel wird zunächst auf die Verlustleistung in integrierten Schaltungen im Allgemeinen eingegangen. In Kapitel 2.2 werden gängige Methoden der Verlustleistungsabschätzung für Prozessoren vorgestellt und nach Genauigkeit und Aufwand bewertet.

2.1 Verlustleistung in integrierten Schaltungen

Die Leistung P (von englisch *Power*) ist definiert als der Quotient aus aufgewendeter Energie ΔE und benötigter Zeit Δt und entspricht damit der Rate, mit der Energie aufgewendet oder „verbraucht“ wird:

$$P = \frac{\Delta E}{\Delta t} \tag{2.1}$$

Unterschieden wird dabei die Momentanleistung $P(t)$ zu einem bestimmten Zeitpunkt t, für den Δt gegen Null geht, und die mittlere verrichtete Leistung $\overline{P}$ über ein Zeitintervall $T = [t_1, t_2]$, welche durch Mittelung berechnet wird:

$$\overline{P} = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} P(t)dt; \tag{2.2}$$

Die momentane Leistungsaufnahme $P_{in}(t)$ eines elektrischen Systems ist das Produkt aus anliegender Spannung $U(t)$ und dem durch das System fließenden Strom $I(t)$:

$$P_{in}(t) = U(t) \cdot I(t) \tag{2.3}$$

Wenn es sich bei Strom und Spannung um konstante, bzw. durch Integration gemittelte Größen handelt, kann die mittlere Leistungsaufnahme P_{in} nach Formel (2.4) berechnet werden.

$$\overline{P}_{in} = U \cdot I \tag{2.4}$$

Im folgenden wird, wenn nicht anders angegeben, P gleich $\overline{P}$ angenommen.
Die *Verlust*leistung wird generell definiert als die Differenz zwischen der aufgenommenen Leistung P_{in} und der in der gewünschten Form abgegebenen Leistung (Nutzleistung). In der Literatur hat sich jedoch für den Leistungsverbrauch von Halbleiterschaltungen durchgesetzt, die durch architekturbedingte Verluste in der Schaltung verbrauchte Leistung plus die zur Verarbeitung von Nutzdaten benötigte Leistung als Verlustleistung zu bezeichnen [35]. Diese wird in Halbleiterbauelementen, neben einer gewissen hochfrequenten Abstrahlung und der zum Treiben von Ausgangssignalen benötigten Energie, hauptsächlich in Wärme umgewandelt.
Damit man aus der Verlustleistung die Energie berechnen kann, die über einen Zeitraum T verbraucht wird, muss man die mittlere Verlustleistung über diesen Zeitraum mit der Zeit multiplizieren:

$$E = P \cdot T \tag{2.5}$$

In der Halbleiterelektronik teilt sich die gesamte Verlustleistung eines Bauelements in zwei Teile: Die statische Verlustleistung, die auch ohne Schaltaktivität im Bauelement auftritt, und die dynamische Verlustleistung, die durch Verluste beim Umschalten von Signalen entsteht:

$$P_{ges} = P_{stat} + P_{dyn} \tag{2.6}$$

2.1.1 Statische Verlustleistung

Die statische Verlustleistung, die beim Anlegen einer Betriebsspannung ohne jegliche Schaltaktivität in einer integrierten Halbleiterschaltung auftritt, ist, bei gleich bleibender Temperatur und Versorgungsspannung, eine feste Bauelement-spezifische Größe und damit ein fixer Anteil der gesamten Verlustleistung. Sie entsteht durch Leckströme, Subschwellströme und Tunnelströme in integrierten Schaltungen.
Den größten Anteil an der statischen Verlustleistung haben die Leckströme von in Sperrrichtung betriebenen parasitären Dioden an den pn-Übergängen in den Transistoren, welche die Grundelemente einer integrierten Schaltung in CMOS Halbleitern bilden (Siehe Abb. 2.1).

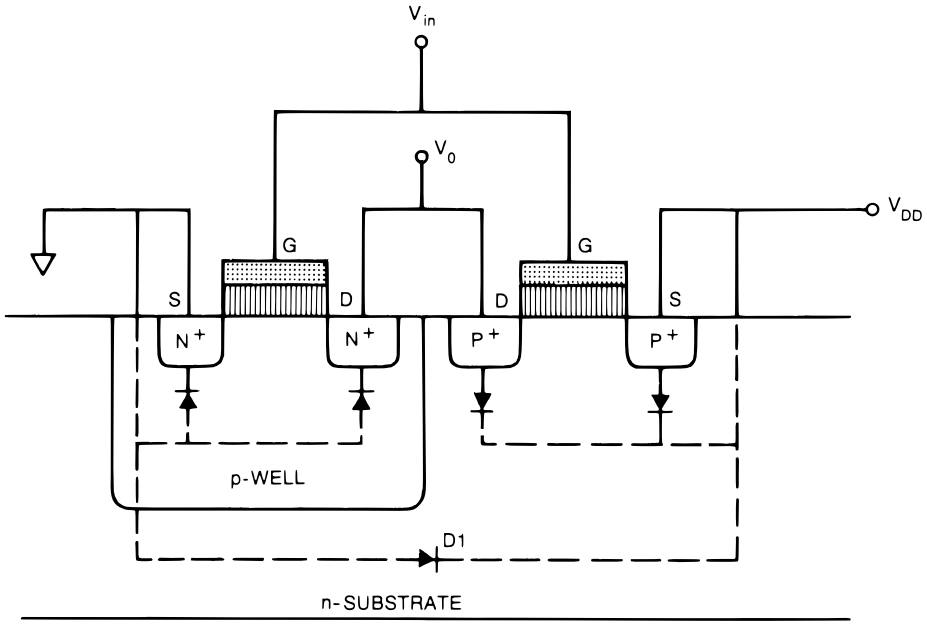

Abbildung 2.1: Modell eines Inverters in CMOS Technologie, welcher durch zwei komplementäre MOS-FET Transistoren gebildet wird, mit Leckströmen (gestrichelt) und parasitären Dioden an den pn-Übergängen im Silizium [34]

Der Leckstrom I_D für jeden pn-Übergang entspricht dabei dem Sperrstrom einer Diode [34]:

$$I_D = I_s(e^{\frac{qU}{kT}} - 1) \tag{2.7}$$

I_s steht dabei für den Sättigungsstrom der Diode in Sperrrichtung, U für die Diodenspannung, q für die Elementarladung, also die elektrische Ladung eines Elektrons, k für die Boltzmann-Konstante und T für die Temperatur der Halbleiterschaltung.
Eine weitere Ursache für die statische Verlustleistung stellen die Subschwellströme dar, welche durch Ladungsträgerdiffusion in dem Kanal eines selbstsperrenden Halbleitertransistors entstehen – selbst wenn die Gate Spannung U_{gs} unterhalb der Schwellspannung U_{th} liegt und sich der Transistor deshalb im Sperrzustand befindet [35][32].
Bei Technologien mit einer Gate-Länge von kleiner $0,18\mu m$ erhöht sich die statische Verlustleistung noch weiter über Tunnelströme durch das Gate-Oxid (In Abb. 2.1 vertikal gestreift) [35][20].
Diese drei Ursachen hängen hauptsächlich vom Fertigungsprozess der integrierten Schaltung und der Strukturgröße ab. Je kleiner die Transistoren sind, desto kürzer sind die Kanallängen, wodurch die Ladungsträger geringere Widerstände überwinden müssen und sich höhere statische Ströme ergeben.
Weitere Faktoren, welche die statische Verlustleistung beeinflussen, sind die Versorgungsspannung und die Temperatur des Siliziums (Siehe auch Formel (2.7)). Je höher die Temperatur ist, desto mehr Ladungsträger diffundieren im Silizium, wodurch sich die statische Verlustleistung weiter erhöht. Auch die Versorgungsspannung wirkt sich direkt auf die Leck- und Tunnelströme und damit auf die statische Verlustleistung aus.
Für die Abschätzung der Verlustleistung werden diese Faktoren als konstante Größen angenommen. Die statische Verlustleistung wird meist im Datenblatt des Prozessors als fester Wert angegeben, kann aber auch durch eine Messung bei einer Taktfrequenz von 0 Hertz statisch ermittelt werden. Einzig die Veränderung der Temperatur muss für eine Abschätzung berücksichtigt werden. Durch eine höhere dynamische Verlustleistung steigt die Temperatur der integrierten Schaltung und damit auch die statische Verlustleistung. Diese Schwankungen sind jedoch gegenüber den Veränderungen der dynamischen Verlustleistung gering. Durch eine entsprechende Kühlung kann ferner auch die Temperatur annähernd konstant gehalten werden. Somit wird im Folgenden von einer konstanten statischen Verlustleistung ausgegangen.
Bei einem Prozessor hat das ausgeführte Programm keinen Einfluss auf die statische Verlustleistung. Einzig Befehle die bestimmte Bereiche eines Prozessors in einen Schlafmodus umschalten können, der die Stromzufuhr zu diesen Bereichen wirklich trennt und nicht nur ein Clock-Gating durchführt, können diese Verlustleistung beeinflussen.

2.1.2 Dynamische Verlustleistung

Trotz der immer geringeren Strukturgrößen und dem damit verbundenen Anstieg der statischen Verlustleistung, überwiegt bei den meisten Anwendungen noch immer der dynamische Leistungsverbrauch. Dieser beruht auf zwei Phänomenen: Einerseits entstehen beim Umschalten, z.B. an einem CMOS Inverter, durch die nicht idealen Schaltkurven der Transistoren hohe Ströme, während der p-Kanal Transistor und der n-Kanal Transistor sich gleichzeitig im linearen Bereich befinden (Siehe Abb. 2.2).

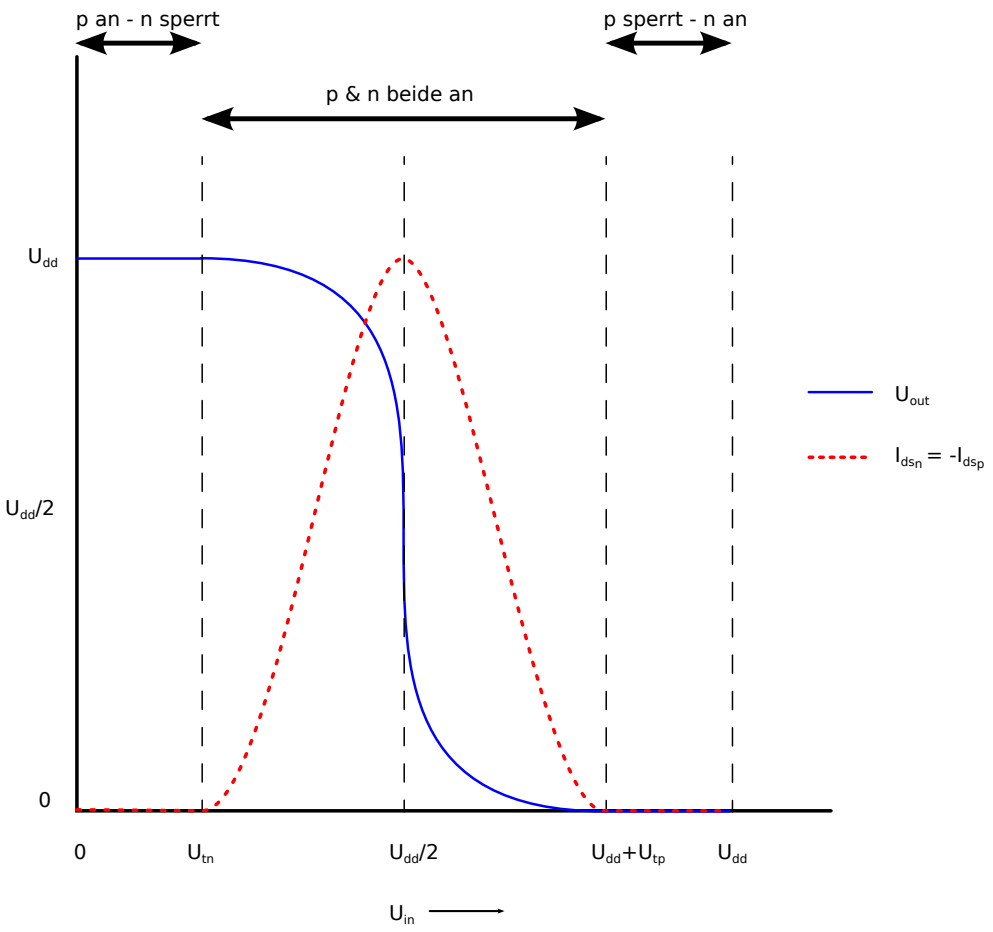

Abbildung 2.2: Transferkurve eines CMOS Inverters

In der Praxis bedeutet das, dass der schließende Transistor noch nicht ganz abgeschaltet ist, der öffnende Transistor schon frühzeitig einen Stromfluss

zulässt, und s
nung (U_{dd}) zu

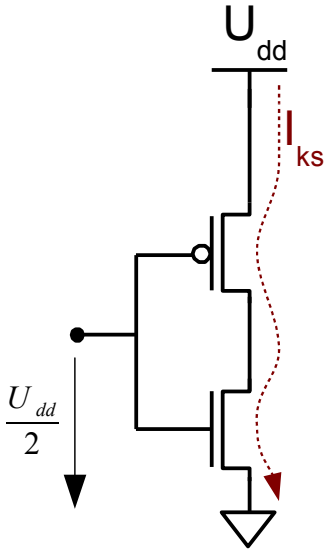

Abbildung 2.3: Schaltbild eines einfachen Inverters in CMOS Technologie, bei dem bei einer Umschaltung ein Quasi-Kurzschlussstrom I_{ks} von U_{dd} nach Masse fließt

Andererseits müssen bei Schaltvorgängen Auf- und Entladungen von Knoten- und Verbindungskapazitäten berücksichtigt werden. Diese Kapazitäten liegen bei integrierten Schaltungen hauptsächlich in den Verbindungsleitungen, die gegenüber Masse einen Kondensator bilden. Die Kapazität einer Leitung kann anhand eines parallelen Plattenmodells approximiert werden:

$$C = \frac{\varepsilon}{t} A \tag{2.8}$$

Hierbei ist A die Fläche der parallelen Platten, t die Dicke der Isolationsschicht und ε die Dielektrizitätskonstante des Materials in der Isolationsschicht.
Durch Randeffekte erhöht sich jedoch dieser approximierte Wert der Kapazität je nach Fertigungsverfahren noch einmal um den Faktor $1,5$ bis 3. Jeder Knoten und jede Verbindungsleitung bilden ein RC-Glied aus einem Widerstand und einem Kondensator. Somit entsteht aus der auf eine

Logikeinheit folgenden Verbindungsstruktur eine Kette aus RC-Gliedern, die sich jedoch mathematisch auf eine einzige Ersatzschaltung mit einer

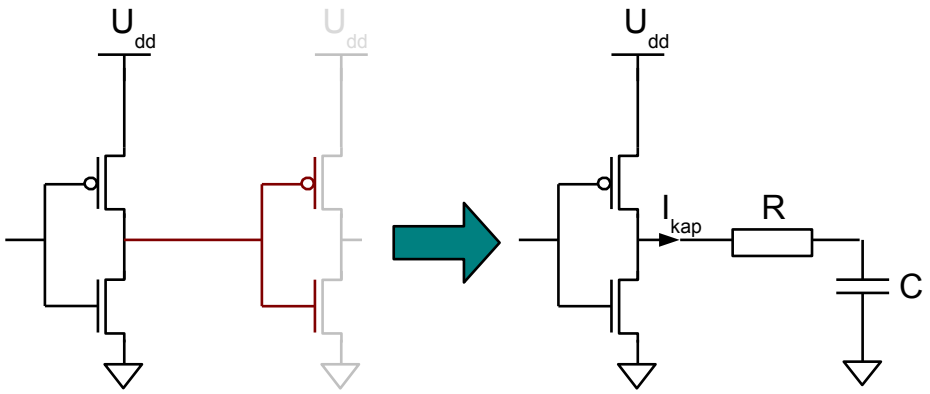

Abbildung 2.4: Die von dem Inverter abgehende Verbindungsleitung und folgende Logikeinheiten können in einem Ersatzschaltbild (re.) als RC-Glied dargestellt werden, durch das ein Strom I_{kap} zum Umladen der Kapazität C fließt.

Die Formel zur Berechnung dieser kapazitiven Verlustleistung ist dann:

$$P_{kap} = U_{dd}^2 \cdot C \cdot f_{clk} \cdot \alpha \tag{2.9}$$

U_{dd} ist dabei die Versorgungsspannung der Schaltung, f_{clk} die Frequenz, mit der die Schaltung getaktet wird, und α die Aktivitätsrate. Diese Aktivitätsrate ist der Faktor, mit der sich ein Signal im Bezug zur Frequenz f_{clk} ändert. Somit liegt dieser Faktor meist zwischen 0 und 1, kann aber durch Taktverdoppler, oder auch durch das Auftreten von Glitchen[1] größer als 1 sein [5]. Meist liegt der Faktor jedoch bei $0,5$ oder darunter, da sich die Signale normalerweise entweder zum Zeitpunkt der steigenden oder der fallenden Flanke des Taktsignals ändern.
Um diese beiden Teile der dynamischen Verlustleistung in einer Formel

[1] Glitche sind ungewollte kurzzeitige Umschaltungen eines Signals, verursacht durch unterschiedlich lange Signallaufzeiten, die in asynchronen Schaltungen häufig auftreten. Obwohl sie bei Synchronisierung an Registern korrigiert werden, verursachen sie trotzdem in der Logik vor dem Register eine höhere Schalthäufigkeit und führen damit zu einem erhöhten dynamischen Energieverbrauch.

auszudrücken, wird aus dem Kurzschlussstrom I_{ks} eine virtuelle Ersatz-Kapazität C_{ks} nach Formel (2.10) berechnet:

$$C_{ks} = \frac{I_{ks}}{U_{dd} \cdot f_{clk} \cdot \alpha} \tag{2.10}$$

Diese bildet nun zusammen mit der „echten“ zu ladenden Kapazität C des Knotens eine neue Ersatz-Kapazität C_{knoten}:

$$C_{knoten} = C + C_{ks} \tag{2.11}$$

Mit (2.9) ergibt sich die gesamte dynamische Verlustleistung für einen Knoten:

$$P_{dyn\_knoten} = U_{dd}^2 \cdot C_{knoten} \cdot f_{clk} \cdot \alpha \tag{2.12}$$

Für eine komplette Schaltung aus mehreren Knoten werden die Ersatz-Kapazitäten von jedem Knoten der Schaltung zu einer Kapazität C_{ges} aufsummiert und eine durchschnittliche Aktivitätsrate α_{ds}aus den Aktivitätsraten der einzelnen Knoten gebildet (Siehe auch [7] S. 23ff). Die Formel zur Berechnung der gesamten dynamischen Verlustleistung einer integrierten Schaltung ergibt sich somit zu:

$$P_{dyn} = U_{dd}^2 \cdot C_{ges} \cdot f_{clk} \cdot \alpha_{ds} \tag{2.13}$$

2.1.3 Verlustleistung von FPGAs

Obwohl die in dieser Arbeit vorgestellten Methoden und Ansätze auch auf normale Prozessoren angewendet werden können, werden hier in den Fallbeispielen sogenannte Soft-Core Prozessoren verwendet, da diese genauere Analysemöglichkeiten bieten und die inneren Abläufe im Prozessor besser nachvollzogen werden können.
Diese Soft-Core Prozessoren werden auf FPGAs konfiguriert, weshalb hier die Besonderheiten des Verlustleistungsverbrauchs von FPGAs beschrieben werden.
Schaltungen auf FPGAs haben im Vergleich zu fest verdrahteten integrierten Schaltungen eine höhere statische Verlustleistung. Diese reduziert

sich prozentual zur dynamischen Verlustleistung, je besser die verfügbare Logik auf einem FPGA genutzt wird. Deshalb ist es für einen geringen Energieverbrauch einer Anwendung auf einem FPGA wichtig, den kleinstmöglichen FPGA zu verwenden, auf dem das zu implementierende Design gerade noch Platz findet.
Bei der dynamischen Verlustleistung von FPGAs ist zu beachten, dass die Interconnect-Verlustleistung einen höheren Anteil an der Gesamtverlustleistung hat als bei Standard-Designs, da die Verdrahtung über fest vorgegebene, meist sehr lange, Interconnect-Leitungen erfolgt und so nur schlecht optimiert werden kann. Ansonsten gilt die Formel (2.13) zur Berechnung der dynamischen Verlustleistung in gleicher Weise für FPGAs[2].

2.2 Methoden der Verlustleistungsabschätzung von Prozessoren

Die Abschätzung der Verlustleistung von Prozessoren hat gegenüber der Abschätzung von generellen integrierten Schaltungen den Vorteil, dass die Zustände in der sich ein Prozessor befinden kann, durch die Zahl der Assemblerbefehle des Prozessors (und deren Kombination mit Eingangsdaten) beschränkt ist.
Folgende Methoden werden zur Abschätzung der Verlustleistung von Prozessoren vorrangig eingesetzt:

- Direkte Messungen an einem Prototyp
- Low-Level Simulation
- Modulbasierte Abschätzung
- Instruktionsbasierte Abschätzung
- Funktionsbasierte Abschätzung

Die Genauigkeit der Methoden nimmt von der Messung an einem Prototyp, welche die genauesten Ergebnisse liefert, bis zur funktionsbasierten Abschätzung ab. Aber gleichzeitig reduziert sich auch der Aufwand um

[2]Eine genauere Untersuchung zur Verlustleistung der Interconnect-Leitungen von FPGAs fand im Projekt LEMOS durch eine Diplomarbeit [16] von Xiangfeng Jiang bei Infineon statt. Die Messungen zur Verifikation wurden dabei mit von uns entwickelten Methoden in unseren Labors durchgeführt.

mehrere Größenordnungen[3].
Die in dieser Arbeit vorgestellte ACCPWR (von englisch „accurate power“) Methode basiert dabei auf der instruktionsbasierten Abschätzung, verbessert aber deren Genauigkeit deutlich, ohne dass große Einbußen bei der Geschwindigkeit in Kauf genommen werden müssen.
Eine grobe Übersicht über die Genauigkeit und die zur Abschätzung benötigte Zeit der verschiedenen Methoden gibt die Grafik in Abb. 2.5.

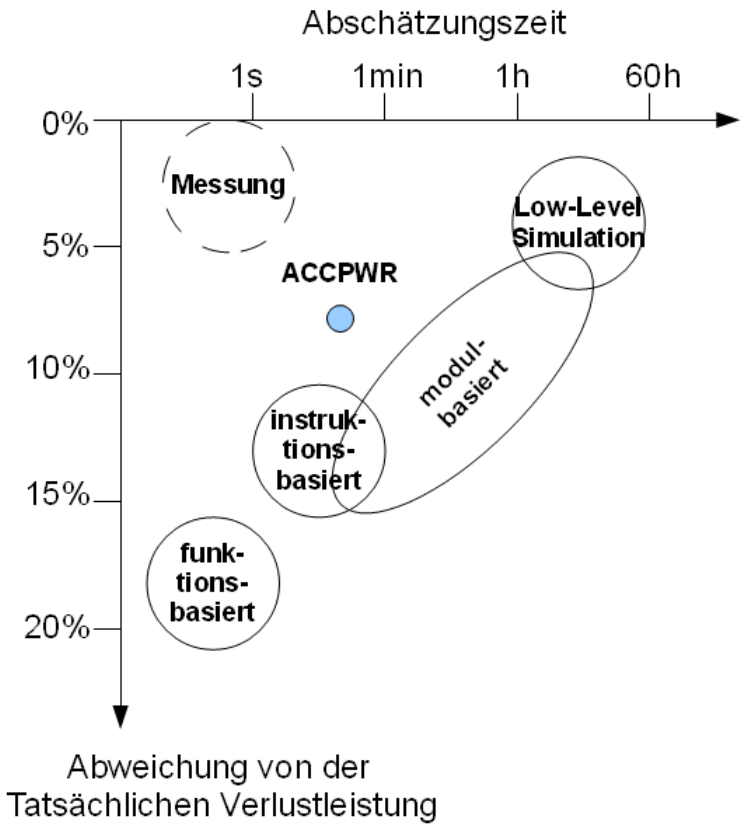

Abbildung 2.5: Qualitativer Vergleich der verschiedenen Methoden zur Abschätzung der Verlustleistung in Bezug auf die Abweichung gegenüber der Zeit für die Abschätzung.

[3]Der Aufwand für die Messung ist hierbei nicht direkt mit dem Aufwand für die verschiedenen Arten der Modellierung vergleichbar, da hierfür vor allem der Hardware-Aufwand (Labor, Messgeräte, Messaufbau) erheblich höher ist.

2.2.1 Direkte Messung

Die genaueste Methode, die Verlustleistung eines Prozessors zu ermitteln, ist es, eine Messung an einem baugleichen Prototypen unter Laborbedingungen durchzuführen. Hierbei wird bei konstanter Spannung U_{dd} der Strom $I(t)$, der an den Stromversorgungsleitungen des Prozessors fließt, gemessen und die Verlustleistung anhand Formel (2.4) berechnet.
Für diese Messungen muss der Prototyp mit dem zu testenden Programm geladen werden, und die Eingänge des Prozessors müssen mit den gleichen Signalen, die der Prozessor auch im realen Umfeld erhalten würde, versorgt werden. Dies geschieht mit Hilfe eines Patterngenerators, der vorher entsprechend programmiert werden muss. Gleichzeitig werden die Ausgangssignale des Prozessors mit einem „Digital Analyzer“ erfasst und ausgewertet. Dabei können die Strommessungen mit den Ausgangssignalen korreliert und eine genaue zeitliche Verbrauchsanalyse durchgeführt werden.
Messungen im realen Umfeld sind meist nicht möglich, da man nur selten an die Stromversorgungsleitungen des Prozessors herankommt, und somit nur die Verlustleistung des gesamten Systems, in das der Prozessor eingebettet ist, messen kann. Hierbei stören nicht nur der Leistungsverbrauch anderer Bauteile wie z.B. jener des externen Speichers oder diverser Leitungstreiber, sondern der Eingangsstrom wird meist noch über Spannungswandler nichtlinear verändert.
Durch die Messung an einem „Ersatzprozessor“ können abweichende Verlustleistungswerte gemessen werden, da zwei Prozessoren, auch wenn Sie im gleichen Prozess hergestellt wurden, durch Fertigungstoleranzen nie einen identischen Stromverbrauch aufweisen. Um die Abweichung möglichst gering zu halten, muss der Ersatzprozessor durch Kühlung auf einer konstanten Temperatur gehalten werden, die der Temperatur im Einsatzgebiet des Prozessors möglichst gut entspricht.
Ein weiterer Nachteil gegenüber der Simulation und anderen Abschätzverfahren ist, neben dem hohen Aufwand, dass mit diesem Verfahren nicht einzelne Teile des Prozessors auf ihren Leistungsverbrauch hin untersucht und optimiert werden können, sondern nur der Prozessor als Ganzes betrachtet werden kann.
Im vom Bundesministerium für Bildung und Forschung geförderten Projekt LEMOS (**L**ow-Power - **E**ntwurfsmethoden für **mo**bile **S**ysteme) [8] wurden von uns in Zusammenarbeit mit Infineon Technologies AG Methoden entwickelt, die Verlustleistung von eingebetteten Systemen genau

zu messen. Hierzu wurde eine Versorgungsleitung nach dem Spannungswandler auf der Platine aufgetrennt und die Strommessungen an diesem Knoten durchgeführt. Das entwickelte Verfahren entspricht industriellen Standards und wird qualifiziert eingesetzt. Die Ergebnisse wurden zur Charakterisierung und Validierung für das kommerzielle Tool Orinoco [26] vom Projektpartner ChipVision verwendet.

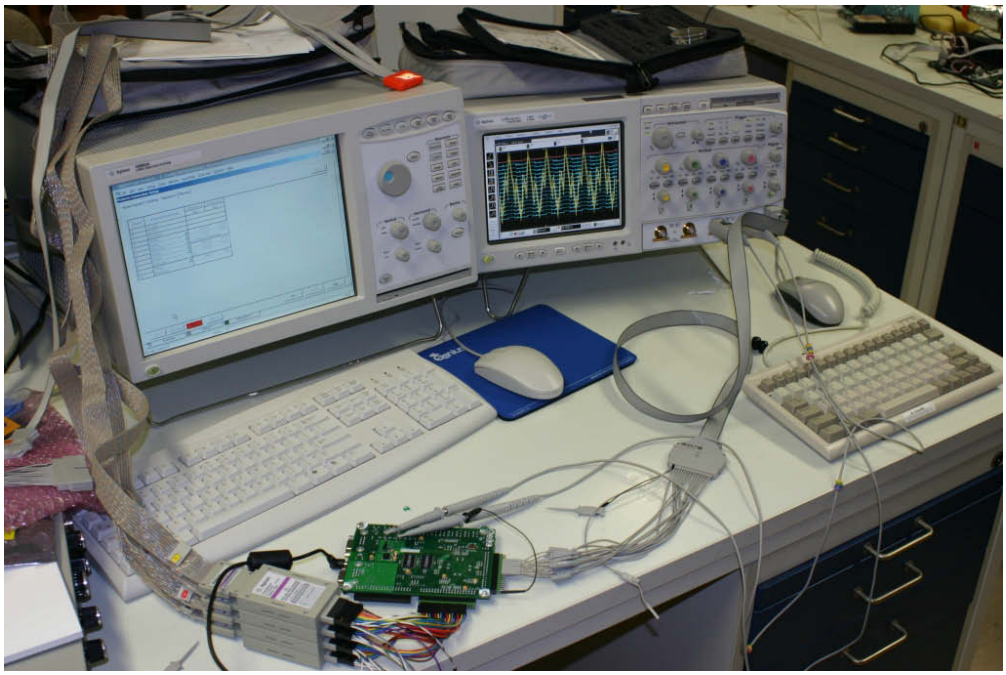

Abbildung 2.6: Der Patterngenerator Agilent 16902A (li.) liefert die Eingangssignale an das Test-Board. Die Ausgangssignale werden zusammen mit den Stromwerten auf einem digitalen Oszilloskop (Agilent Infiniium 54832D) mit integriertem digitalen Analyzer (re.) dargestellt

Genaue Messungen zur Charakterisierung eines FPGAs wurden auch in [19] durchgeführt. Hierbei wird eine zusätzliche Lastkapazität periodisch in die Versorgungsleitungen des FPGAs parallel hinzugeschaltet und über den resultierenden Spannungsabfall Rückschlüsse auf das im FPGA geladene Design gemacht.

2.2.2 Low-Level Simulation

Eine Low-Level Simulation ist eine Simulation auf einer niedrigen Abstraktionsebene, wie der Register-Transfer-Ebene, Gatterebene oder Schaltkreisebene. Für diese Art der Simulation wird das Design einer integrierten Schaltung in einer Hardware-Beschreibungssprache, wie z.B. VHDL oder Verilog, synthetisiert, auf eine niedrigere Ebene der Abstraktion übersetzt und auf dieser für gegebene Stimulationswerte an den Eingängen simuliert. Dabei werden alle Signaländerungen an allen Knoten und Verbindungsleitungen in eine Value-Change-Dump-Datei (VCD-Datei) aufgezeichnet. Anhand dieser aufgezeichneten Werte kann man die Aktivitätsraten α aller Knoten bestimmen. Ferner kann durch Informationen über die Implementierung des Designs (Länge der Verbindungsleitungen, Strukturgröße, etc.) die zu ladende Kapazität C bestimmt werden. Nach Formel (2.13) kann nun die dynamische Verlustleistung P_{dyn} berechnet werden.
Für FPGAs wird das Design in VHDL oder Verilog zu einer Netzliste synthetisiert. Diese wird beim „Mapping“ auf die Ressourcen des FPGA angepasst und anschließend für den gegebenen Platz auf dem FPGA platziert und verdrahtet. Mit einem Simulationsprogramm, wie z.B. ModelSim, wird dieses fertig verdrahtete Design (NCD-Datei) nun in einer Testbench simuliert und die Signaländerungen in eine VCD-Datei geschrieben. Diese kann mit einem vom FPGA-Hersteller meist mitgelieferten Programm zur Berechnung der Verlustleistung ausgewertet werden. Bei der Entwicklungsumgebung von Xilinx geschieht dies über das Programm XPower, in das man die VCD-Datei zusammen mit der NCD-Datei importiert. Dieses Programm liefert dann eine hinreichend genaue Abschätzungen der statischen und dynamischen Verlustleistung. Auch eine Berechnung der Verlustleistung einzelner Module ist möglich.

Diese Methode liefert, was die Präzision der Abschätzung betrifft, die optimal erzielbaren Ergebnisse, die man ohne Messung erhalten kann. Der Vorteil ist dabei, dass man die Verlustleistung sehr gut lokal und zeitlich untersuchen kann. Jedoch ist vor allem die Simulation sehr zeitaufwendig und für längere Applikationsprogramme nicht durchführbar, da die Zeit für die Simulation mit der Länge des Applikationsprogramms proportional und mit der Größe des Designs sogar überproportional zunimmt. Zusätzlich dauert das Synthetisieren, Mappen, Platzieren und Verdrahten des Designs, je nach Größe des Designs und der jeweiligen Bedingungen (Timing- oder Flächenbeschränkungen, etc.), von wenigen Minuten bis zu mehreren Stunden.

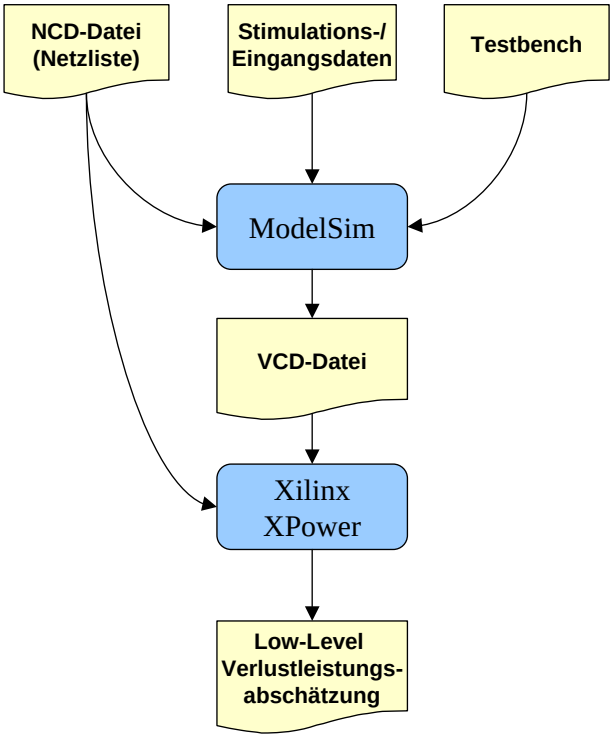

Abbildung 2.7: Flussdiagramm für die Durchführung der Low-Level Verlustleistungsabschätzung mit dem Simulationsprogramm ModelSim und dem Low-Level Abschätzprogramm XPower von Xilinx

Hsieh et al. [14] schlagen vor ein kürzeres synthetisches Ersatz-Applikationsprogramm zu erzeugen, um den Prozess der Verlustleistungsabschätzung durch eine Simulation auf Register-Transfer-Ebene zu beschleunigen. Dadurch können genaue Schätzungen schneller durchgeführt werden, jedoch ist eine Optimierung des Applikationsprogramms bezüglich der Verlustleistung nicht mehr so leicht möglich, da Leistungsspitzen durch ungünstige Befehlskombinationen nicht genau ermittelt werden können. Der Ansatz bezieht Pipeline-Stalls und falsche Sprungvorhersagen mit in die Berechnung des synthetischen Ersatz-Programms mit ein – nicht jedoch bei der Berechnung der Verlustleistung.

2.2.3 Modulbasierte Abschätzung

Für die modulbasierte Verlustleistungsabschätzung geht man nach dem „Teile und herrsche“ Prinzip vor. Hierbei unterteilt man den Prozessor in elementare Grundmodule, für welche die Verlustleistung einfacher bestimmt werden kann (Siehe auch Abb. 2.8). Die Grundmodule können von sehr feingranular, wie z.B. Schieberegister oder Logikgatter, bis grobgranular, wie z.B. eine Barrelshifter-Einheit oder eine komplette ALU, reichen. Um die Gesamtverlustleistung P_{ges} des Systems zu ermitteln, summiert man die Verlustleistungen der einzelnen Module für einen vorher festgelegten Betrachtungszeitraum auf:

$$P_{ges} = \sum_{i=1}^{z} P_i \tag{2.14}$$

Die Verlustleistung P_i muss dabei für jedes Modul der z vorhandenen Module einzeln über Verlustleistungsmodelle, die mit den Modul-Eingangswerten parametrisiert sind, bestimmt werden. Dies geschieht über Berechnungen nach der Formel (2.12), wobei für das gesamte Modul eine Ersatzkapazität C_{ersM} berechnet wird, und die Aktivitätsrate α aus den Aktivitätsraten der Eingangssignale des Moduls ermittelt wird.
Die Herausforderung bei dieser Verlustleistungsabschätzung ist es, genaue Aktivitätsraten für die Eingänge der Module zu finden und die mathematischen Modelle für jedes Modul optimal einzustellen.

Neben dem Vorteil der relativ hohen Genauigkeit, die man mit dieser Art der Verlustleistungsabschätzung erzielen kann, gibt es jedoch den Nachteil der Inkaufnahme zweier Voraussetzungen: Um die Makromodelle der Module zu erstellen und die Aktivitätsraten zu ermitteln, benötigt man

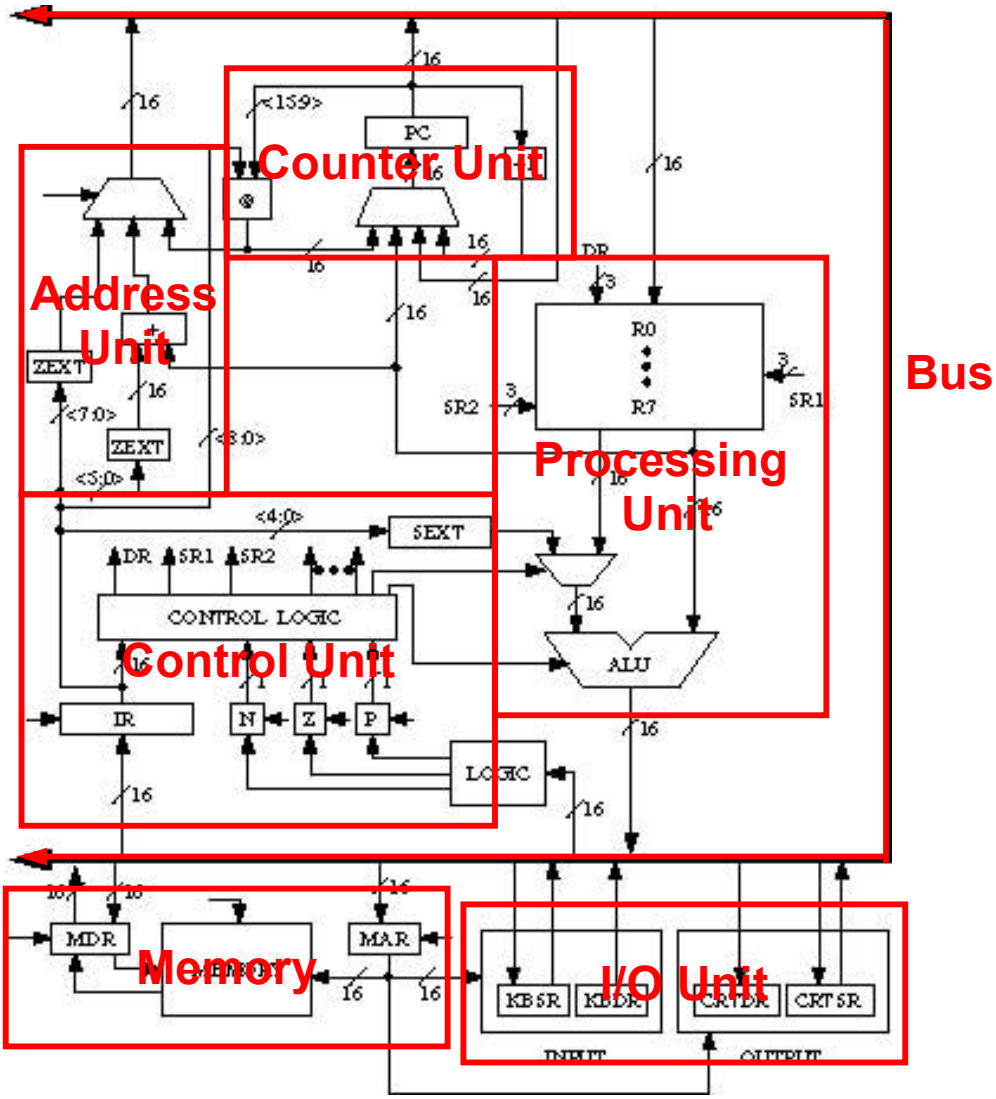

Abbildung 2.8: Für die modulbasierte Abschätzung wird ein Prozessor in funktional zusammengehörige Komponenten unterteilt, die separat untersucht werden (Ursprüngliches Bild aus [25], nachträglich bearbeitet)

Kenntnisse der inneren Strukturen der Module und der Prozessor muss sich gut in Module zerlegen lassen. Dies ist bei Benutzung von kommerziellen IP-Cores, die monolytisch angeboten werden, meist nicht möglich. Bei vielen Prozessoren, von denen die Hardware-Beschreibung als Quelltext vorliegt, kann eine Unterteilung des Prozessors in Module aber auch schwierig oder gar unmöglich sein, wenn dieser Quelltext nicht modular aufgebaut ist.
Mit dem Wattch-Framework [3] lässt sich die Verlustleistung von verschiedenen Prozessoren auf der Architekturebene modulbasiert abschätzen. Es setzt dabei auf den SimpleScalar Prozessorsimulator [1] auf und ist durch Funktionsmodule erweiterbar. Die hier eingesetzte modulbasierte Methode liefert Abschätzungen mit einer Abweichung von der Low-Level Simulation auf Schaltkreisebene, welche sich meist zwischen $\pm 10\%$ bewegt.
Ein weiteres Beispiel für eine sehr genaue (schon fast an eine Low-Level Simulation heranreichende) modulbasierte Verlustleistungsabschätzung ist das CAPET Framework [4] von IBM. Dieses schätzt die Verlustleistung von Transistorlevel-Makromodulen durch das Beobachten von Schalt- und Taktaktivitäten in einer Simulation auf Register-Transfer-Ebene ab. Ein Simulationsschritt wird für jeden Takt durchgeführt. Das Verfahren kann auf verschiedenen Ebenen des Entwurfsprozesses eingesetzt werden, ist durch die taktweise Low-Level Simulation jedoch sehr langsam und daher für große Applikationsprogramme nicht gut einsetzbar. Es wurde für den IBM Cell Prozessor [10] entwickelt und auch an diesem getestet.
Auf einer sehr viel höheren Ebene der Abstraktion basiert der Extended Queueing Networks basierte Ansatz in [23]. Hierbei wird ein Prozessor mit einem speziellen erweiterten Warteschlangennetzwerk, einem sogenannten EQN*, modelliert und dabei die Verlustleistung abgeschätzt. Die hier verwendeten Module sind grobgranular ausgelegt und entsprechen den einzelnen Funktionseinheiten des Prozessors. Die mittleren Verlustleistungen der Module werden durch Low-Level Simulation bestimmt, und als Aktivitätsrate wird ein binärer Wert verwendet. Während der Simulation wird nur betrachtet, ob ein Modul in einem Prozessortakt verwendet wird oder nicht. Der LC-3 Prozessor von Patt et al. [25] wurde mit dieser Methode untersucht und die Ergebnisse mit der Low-Level Abschätzung verglichen. Bedingt durch die hohe Abstraktionsebene, auf der dieser Ansatz die Verlustleistung ermittelt, ist dieser sehr schnell, liefert aber trotzdemnoch sehr gute Abschätzungen.
Ein Zwischenschritt zur instruktionsbasierten Methode wird von Hsieh et al. in [13] beschrieben. Mit Hilfe von Tokens, die den ausgeführten Instruktionen entsprechen und dabei Gatter durchlaufen, wird die Verlustleistung

eines Zilog Signalprozessors abgeschätzt. Die Gatter sind dabei recht feingranular, und die Verlustleistung wird über deren Kapazitäten und Aktivitätsraten bestimmt. Es wird dabei auf die Pipelining-Probleme für die Abschätzung eingegangen, und es werden Lösungen zur Berechnung von Pipeline-Stalls, Data-Forwarding und Pipeline-Flushes diskutiert.

2.2.4 Instruktionsbasierte Abschätzung

Bei der instruktionsbasierten Verlustleistungsabschätzung wird eine Datenbank mit allen Instruktionen angelegt, die der zu untersuchende Prozessor ausführen kann, Zu jeder Instruktion wird ein Energiewert gespeichert, der durch Messung oder in einer Low-Level Simulation für die jeweiligen Instruktion ermittelt wird.
Für ein bestimmtes Programm, das auf diesem Prozessor laufen soll, wird durch ein Software-Simulations-Programm ein Befehls-Trace, also eine Liste der für dieses Programm auszuführenden Instruktionen, aufgezeichnet. Anhand dieses Traces werden nun die Energiewerte der einzelnen Instruktionen aufsummiert und damit der gesamte Energieverbrauch und die durchschnittliche Verlustleistung berechnet (Siehe auch Abb. 2.9).

Zur Berechnung der gesamten verbrauchten Energie E_{ges} einer Befehlsfolge mit m Instruktionen werden die einzelnen Energiewerte $E_{asm}(i)$ der ausgeführten Instruktionen ($i\epsilon[1:m]$) einfach aufsummiert:

$$E_{ges} = \sum_{i=1}^{m} E_{asm}(i) \tag{2.15}$$

Um die durchschnittliche Verlustleistung P_{ds} eines Programms zu berechnen, wird der Mittelwert der Verlustleistungen P_{asm} der einzelnen Instruktionen gebildet:

$$P_{ds} = \sum_{x=1}^{m} \frac{P_{asm}(x)}{m} \tag{2.16}$$

Zur Charakterisierung des Prozessors, also der Ermittlung der Energiewerte der einzelnen Befehle, werden Messungen oder Low-Level Abschätzungen herangezogen. Dabei muss für jede Instruktion ein typischer Energiewert ermittelt werden. Hierzu wird der Befehl mehrmals hintereinander

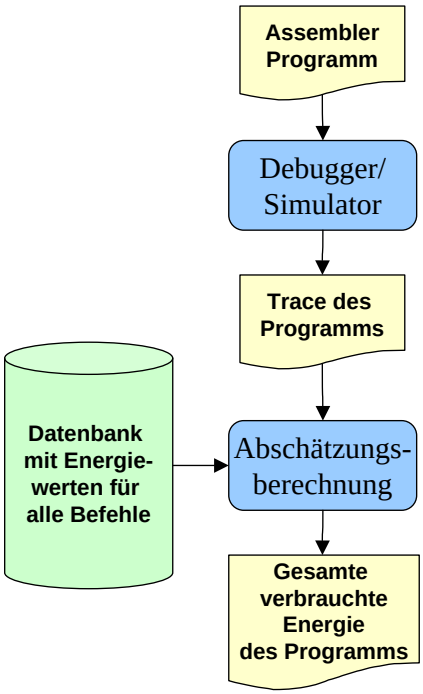

Abbildung 2.9: Flussdiagramm für die Durchführung der instruktionsbasierten Verlustleistungsabschätzung

mit unterschiedlichen Eingangsdaten ausgeführt, um Schwankungen der Verlustleistung durch Datenbitabhängigkeit auszugleichen und die mittlere Verlustleistung zu ermitteln.
Da diese Methode den Prozessor stark abstrahiert, liefert sie nicht so genaue Ergebnisse wie die modulbasierte Methode, ist aber für vergleichende Abschätzungen ausreichend. Ein Vorteil dieser Methode ist das Vermeiden der manchmal schwierigen oder sogar unmöglichen Modularisierung des Cores. Sie kann so auf jeden Prozessor angewandt werden, teilweise auch ohne Kenntnis der inneren Struktur.
Das SEA Framework [17] ist ein instruktionsbasiertes Verlustleistungsanalyse-Tool für IP-Cores, im speziellen für den MicroSparc II Core von Sun. Dieses Framework berücksichtigt bei der Analyse auch die Verlustleistung bei Pipeline-Stalls, ist jedoch nicht dafür ausgelegt, die Verlustleistung von Pipeline-Flushes bei der Ausführung bedingter Sprünge zu berechnen. Es werden Eingangsdatenabhängigkeiten berücksichtigt, indem ein maximaler, minimaler und durchschnittlicher Leistungsverbrauch ausgegeben wird. Jedoch werden keine Instruktionswortabhängigkeiten berücksichtigt, da diese bei dem getesteten Core zu gering gegenüber den Eingangsdatenabhängigkeiten sind. Das SEA-Framework ist ein Abschätz-Tool, das auf fertig synthetisierte und optimierte IP-Cores ausgerichtet ist. Nachteilig ist, dass für jede Veränderung am Prozessor eine neue Datenbank aufgebaut werden muss.
Für den MicroBlaze IP-Core Prozessor von Xilinx wurde in [24] eine Instruction-based Verlustleistungsanalyse-Methode vorgestellt. Die Befehle werden einzeln mit einer Low-Level-Simulation und dem Low-Level-Verlustleistungs-Abschätz-Tools XPower des FPGA Herstellers charakterisiert und eine Verlustleistungstabelle erstellt. Danach kann die Gesamtverlustleistung eines auf dem MicroBlaze ablaufenden Programms durch Profiling und die Verlustleistungstabelle ermittelt werden. Diese Methode lässt jedoch Probleme, die durch Pipeline-Hazards entstehen, komplett außer Acht.

2.2.5 Funktionsbasierte Abschätzung

Funktionsbasierte Abschätzungsmethoden betrachten nur grob verschiedene funktionelle Zustände eines Prozessors, in denen dieser eine unterschiedliche durchschnittliche Verlustleistung verbraucht.
Diese Methode liefert nur eine relativ ungenaue Abschätzung der Verlustleistung, eignet sich aber gut für eine schnelle Abschätzung des Energiever-

brauchs bei Prozessoren mit stark unterschiedlichem Leistungsverbrauch in verschiedenen Zuständen.
Der Focus des AEON-Tools [18] liegt bei der Verlustleistungsabschätzung von Sensornetzen und es setzt hierfür auf das Framework AVRORA [31] auf, welches ursprünglich für die Analyse von Atmel AVR Prozessoren entwickelt wurde. Die Verlustleistung des Prozessors wird als Durchschnittswert für die Zustände *Active*, *Idle* und *Standby* aus Messungen ermittelt.

2.3 Bewertung der Abschätzungsmethoden

Die genauen Methoden der Messung und für die Low-Level Simulation sind sehr aufwendig und zeitintensiv. Modulbasierte Methoden sind zwar etwas schneller, fordern aber immer eine genaue Kenntnis der inneren Strukturen. Die funktionsbasierte Methode ist für Optimierungen an Code und Hardware, auf das die Methode in dieser Arbeit abzielt, wegen der geringen Genauigkeit ungeeignet.
Der in den Kapiteln 4 und 5 vorgestellte Ansatz basiert auf der instruktionsbasierten Verlustleistungsabschätzung unter spezieller Berücksichtigung von Pipelining-Strukturen und Einflüssen des Pipelinings auf die Verlustleistung. Dadurch kann die Genauigkeit verbessert werden, ohne den Aufwand stark zu erhöhen. Speziell bei Veränderungen von der inneren Struktur des Prozessors oder bei einem Redesign muss nicht der gesamte Prozessor neu charakterisiert werden, sondern nur die von der Veränderung betroffenen Pipeline-Stufen. Außerdem wird mit der in dieser Arbeit vorgestellten Methode erstmals eine echte taktgenaue instruktionsbasierte Abschätzung bei Prozessoren mit Pipelining möglich. Auch wenn viele instruktionsbasierte Methoden angeblich „cycle-accurate“, also taktgenau, die Verlustleistung abschätzen, ist dies bei Prozessoren mit Pipelining nicht der Fall (Siehe auch Kap. 4).
Die bisherigen instruktionsbasierten Ansätze gehen gar nicht oder nur teilweise auf die Problematik der Verlustleistungsabschätzung ein, die durch Pipelining in Prozessoren entsteht. Warum das Pipelining für moderne Prozessoren so wichtig ist und wie es die Abschätzung der Verlustleistung beeinflusst, wird in dieser Arbeit dargelegt.
Hierzu wird im nächsten Kapitel das Pipelining mit seinen verschiedenen Strukturen und Problemen erklärt.

3 Pipelining

Beim Pipelining handelt es sich generell um eine Fließband-artige Bearbeitung einer Aufgabe, indem die Teilschritte parallelisiert werden, die für die Ausführung der Aufgabe notwendig sind. Dadurch kann der Durchsatz bei der Abarbeitung erhöht werden.
Bei Pipelining in integrierten Schaltungen handelt es sich speziell um ungetaktete Logik, die durch Register oder Flip-Flops in getaktete Pipeline-Stufen unterteilt wird. Pipelining hat normalerweise einen positiven Effekt auf die Verlustleistung von Schaltungen, da durch die eingefügten Register und die Taktung der Signale die Fortpflanzung und Aufsummierung von Glitches vermieden werden [5].
Das trifft prinzipiell auch auf das Befehls-Pipelining in Prozessoren zu, hier gibt es allerdings einige Randeffekte, die diese Verminderung der Verlustleistung nicht nur schmälern, sondern durch zusätzliche Verzögerungen auch den Leistungsverbrauch erhöhen können.

3.1 Befehls-Pipelining

Befehls-Pipelining ist eine spezielle Form der Prozessorparallelität. Durch Pipelining kann der Durchsatz eines Prozessors gesteigert werden, ohne dass man die Zeit zur Abarbeitung eines Befehls verkürzen muss.
Bei Prozessoren mit nur einer Recheneinheit werden die Befehle nacheinander abgearbeitet. Unterteilt man die Abarbeitung eines Befehls in mehrere unabhängige Schritte gleicher Länge bzw. Zeitdauer, so ist es möglich, schon den nächsten Befehl zu beginnen während der erste noch in Bearbeitung ist. Bis der erste Befehl die Pipeline verlässt, vergeht weiterhin dieselbe Zeit wie ohne Parallelität. Ist die Pipeline jedoch gefüllt, wird bei jedem Takt ein weiterer Befehl fertig (Siehe Abb. 3.1).
Eine typische fünfstufige Befehls-Pipeline besteht aus folgenden Stufen:

Instruction Fetch (IF): Der nächste Befehl wird aus dem Befehlsspeicher geholt

Instruction Decode (ID): Der Befehl wird dekodiert und die Operanden bereitgestellt

Execution (EX): Die ALU führt die eigentliche Rechenoperation des Befehls aus

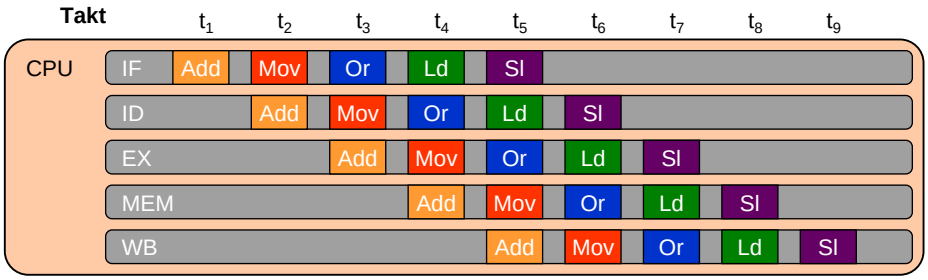

Abbildung 3.1: Die Befehle *Add*, *Mov*, *Or*, *Ld* und *Sl* durchlaufen nacheinander die fünf Stufen der Befehls-Pipeline

In jedem Takt wird hier nicht ein Befehl ausgeführt, sondern bis zu fünf Teile verschiedener Befehle gleichzeitig, die in den Stufen der Pipeline abgearbeitet werden. Für Takt t_5 z.B. wird das Ergebnis des *Add* Befehls zurückgeschrieben, der *Mov* Befehl greift auf den Speicher zu, das Ergebnis des *Or* Befehls wird in der EX Stufe berechnet, der *Ld* Befehl wird dekodiert und der *Sl* Befehl wird gerade aus dem Programmspeicher geladen.
Der Idealfall in Abb. 3.1 geht von der Annahme aus, dass jeder Befehl alle fünf Stufen der Pipeline durchläuft. Dies ist jedoch nicht immer der Fall. Ein *load* Befehl zum Beispiel wird in der *Write Back* Phase nichts zu tun haben (Siehe [29] S.429).
Durch Befehlspipelining erreicht man einen theoretisch um Faktor n höheren Durchsatz bei einer Unterteilung in n Schritte bzw. Stufen. Da die Pipeline jedoch erst gefüllt werden muss, was $n - 1$ Takte benötigt, liegt

der Beschleunigungsfaktor F bei m auszuführenden Befehlen bei:

$$F = \frac{n \cdot m}{m + n - 1} \tag{3.1}$$

Doch auch dieser Faktor wird in der Realität nur unter Idealbedingungen erreicht. Durch Konflikte in der Pipeline (Pipeline-Hazards, siehe auch Kapitel 3.2) kommt es zum kurzzeitigen Halten (Pipeline-Stall) oder zum Verwerfen der in der Pipeline geladenen Daten (Pipeline-Flush), nach dem die Pipeline erst wieder gefüllt werden muss. Hierdurch verringert sich der Durchsatzgewinn.

3.2 Pipeline-Hazards und Lösungen

Pipeline-Hazards sind Konflikte, die durch das Befehls-Pipelining in Prozessoren auftreten. Man unterscheidet drei Arten von Konflikten bei der Abarbeitung von Befehlen in einer Prozessor-Pipeline:

Daten-Hazards: Konflikte die beim Lesen vom oder Schreiben in den Speicher entstehen

Struktur-Hazards: Konflikte die an Ressourcen auftreten, auf die von mehr als einer Pipeline-Stufe zugegriffen wird

Branching-Hazards: Konflikte die durch Verzweigungen im Code, sogenannte „Branch“ Instruktionen, entstehen

Zur Vermeidung dieser Hazards gibt es verschiedene Lösungen sowohl auf Hardware-Ebene als auch auf Software-Ebene, bei denen die Pipeline in Teilen oder auch vollständig angehalten wird, bis der jeweilige Konflikt gelöst wurde. Dies beeinflusst sowohl die Performance als auch die Verlustleistung des Prozessors.

3.2.1 Daten-Hazards

Es gibt drei verschiedene Situationen in denen Daten-Hazards, also Konflikte beim Lesen und Schreiben von Daten, in Befehls-Pipelines auftreten können:

„**Read after Write**“: Daten werden von einem später in die Pipeline geschobenen Befehl in einer frühen Stufe gelesen, vor der früher ausgeführte Befehl die Daten in der am Ende der Pipeline liegenden Write-Back Stufe aktualisieren konnte. Somit rechnet der spätere Befehl eventuell mit einem veralteten Wert. (Siehe Abb. 3.2)

„**Write after Read**“: Wenn ein früherer Befehl sehr spät in der Pipeline ein Datum einliest, das ein darauf folgender Befehl in einer sehr frühen Pipeline-Stufe schon überschreibt, erhält der frühere Befehl beim Lesen eventuell das schon aktualisierte Datum, mit dem er aber gar nicht rechnen sollte.

„**Write after Write**“: Wenn zwei kurz aufeinander folgende Instruktionen auf dasselbe Datum schreiben wollen, muss gewährleistet sein, dass der später ausgeführte Befehl auch wirklich nach dem früheren Befehl das Ergebnis in den Speicher schreibt, da sonst der falsche Wert, nämlich das Ergebnis der früheren Instruktion im Speicher bestehen bleibt.

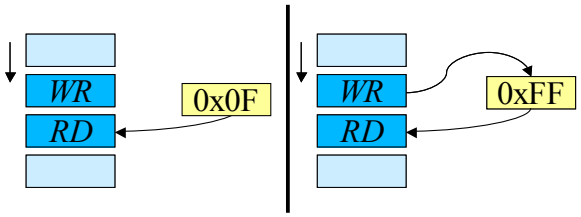

Abbildung 3.2: Bei einem „Read after Write“ Daten-Hazard werden die veralteten Daten gelesen, bevor die neuen in den Speicher geschrieben werden

Als Lösungsmöglichkeiten für Daten-Hazards werden folgende Techniken eingesetzt:

Pipeline-Stall

Zur Lösung dieses Konflikts gibt es z.B. den Pipeline-Stall, bei dem die ganze Pipeline angehalten wird, bis der Konflikt gelöst wurde. Hierbei werden alle Stufen der Pipeline angehalten, indem die Register zwischen den Stufen der Pipeline eingefroren werden.

Bubbling

In der Literatur wird das Bubbling oft als eine Sonderform des Pipeline-Stall behandelt. Zur besseren Unterscheidung wird hier aber Bubbling als eigene Lösungsart vorgestellt.
Vom Bubbling einer Pipeline spricht man, wenn nur die ersten Pipeline-Stufen bis hin zu der Stufe, die wegen des Konflikts warten muss, angehalten werden. In die folgenden Pipeline-Stufen werden die sogenannten

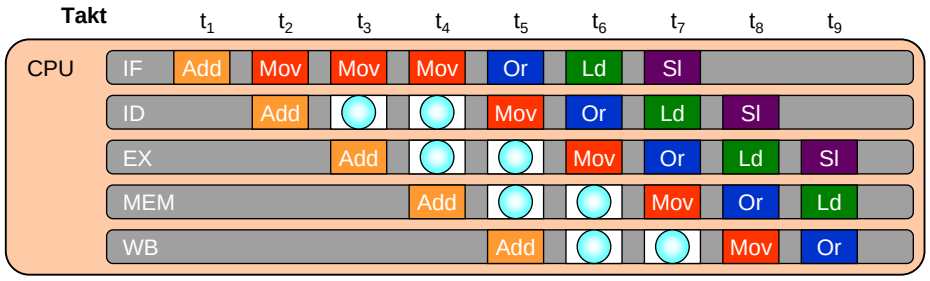

Abbildung 3.3: Bubbling einer fünfstufigen Pipeline nach dem *Add* Befehl

Ab Takt t3 muss der *Mov* Befehl warten und kann erst wieder in Takt t5 in die nächste Pipeline-Stufe vorrücken. Der *Mov* Befehl und alle darauf folgende Befehle werden somit um zwei Takte verzögert. Der *Add* Befehl (und alle vorhergehenden Befehle) wird weiter ohne Verzögerung ausgeführt. In die Stufen nach dem *Add* Befehl werden Bubbles eingefügt.
Nicht nur Konflikte können solch ein Verhalten der Pipeline erzwingen, auch Multi-Cycle Befehle, welche in einer bestimmten Stufe mehr als einen Takt benötigen (z.B. Multiplikationsinstruktionen oder Speicherzugriffe), können zu Bubbling in einer Pipeline führen.

Forwarding

Beim Forwarding werden Ergebnisse des letzten Befehls in der EX Stufe direkt an den nächsten Befehl in der EX Stufe weitergegeben, ohne

sie vorher in den Speicher zu schreiben und erneut auszulesen. Dadurch spart man Zeit für das Schreiben in und das Lesen aus dem Speicher, und vermeidet zusätzlich einen „Read after Write“ Daten-Hazard.

3.2.2 Struktur-Hazards

Struktur-Hazards treten auf, wenn zwei Instruktionen an unterschiedlichen Stellen der Pipeline dieselbe Ressource zur selben Zeit benötigen (Siehe Abb. 3.4).

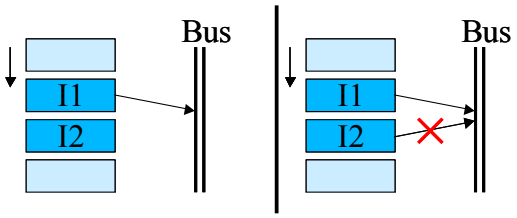

Abbildung 3.4: Wollen zwei Befehle in unterschiedlichen Pipeline-Stufen zur gleichen Zeit auf den Bus zugreifen, kommt es zu einem Struktur-Hazard

Um Struktur-Hazards zu lösen, werden meist Pipeline-Stalls oder die Bubbling Methode eingesetzt (Siehe Kapitel 3.2.1).

3.2.3 Branching-Hazards

Branching-Hazards, oder auch Control-Hazards genannt, entstehen bei Verzweigungen im Code, die durch bedingte Sprünge verursacht werden. Bei bedingten Sprüngen ist nicht bekannt, welche Instruktion als nächste geladen werden soll, bis der Sprung-Befehl die Bedingung und damit die Adresse des Sprungziels berechnet hat (Siehe Abb. 3.5).

Aber auch unbedingte Sprünge können einen solchen Konflikt erzeugen, wenn die Adresse des Sprungziels erst in einer späten Stufe der Pipeline berechnet wird, und damit in der IF Stufe der Pipeline bis dahin die Adresse des Befehls noch nicht bestimmt ist, welcher aus dem Programmspeicher geladen werden muss.
Zur Lösung von Branching-Hazards könnten natürlich mit Bubbling einfach die betroffenen Stufen angehalten werden, bis die Bedingung des

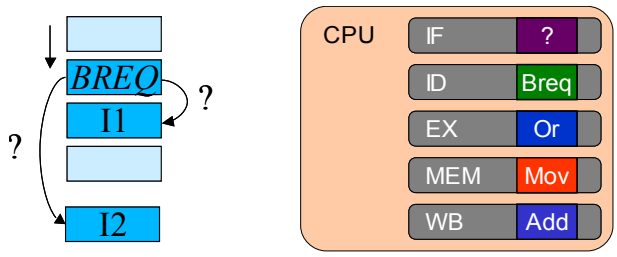

Abbildung 3.5: Bei einem bedingten Sprung (Hier: „Branch if equal“) muss zuerst die Bedingung berechnet werden, um die Adresse des nächsten zu ladenden Befehls herauszufinden

Sprungs berechnet ist und die Zieladresse bereitsteht. Meist wird jedoch eine Sprungvorhersage gemacht, wobei die Verzweigung spekulativ ausgeführt wird. Wenn die Vorhersage richtig ist, wird weiterhin sequentiell mit gefüllter Pipeline weitergearbeitet. Dabei treten keine Verzögerungen oder Veränderungen der Verlustleistung auf. Ist die Vorhersage jedoch falsch

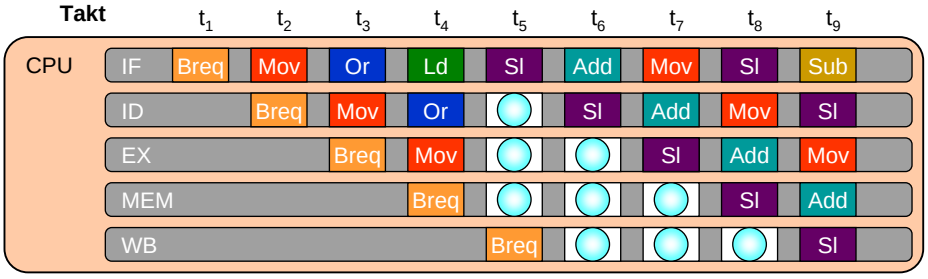

Abbildung 3.6: Die Befehle *Mov*, *Or* und *Ld* werden nach einer falschen Sprungvorhersage für den *Breq* Befehl durch einen Pipeline-Flush abgebrochen und die Pipeline muss wieder von vorne aufgefüllt werden

Bei der Sprungvorhersage werden statische und dynamische Ansätze unterschieden:

3.2.3.1 Statische Sprungvorhersage

Bei der statischen Sprungvorhersage wird einfach angenommen, dass immer gesprungen wird („Branch Always Taken“) oder, dass ein Sprung nie genommen wird („Branch Not Taken“ oder auch „Branch Never Taken“genannt). Gerade die „Branch Not Taken“ Methode ist quasi der Standard bei Mikroprozessoren für eingebettete Systeme (z.B. Atmel AVR, Motorola 68020, Sun Sparc, usw.). Hierbei wird nach einer Branch Instruktion immer der nächste Befehl im Speicher ausgeführt, als ob kein Sprung stattfinden würde. Kommt bei der Bearbeitung der Branch Instruktion jedoch heraus, dass doch gesprungen werden muss, kommt es zu einem Pipeline-Flush.
Eine etwas verbesserte Form der statischen Sprungvorhersage geht davon aus, dass bei Sprüngen, die auf eine höhere als die aktuelle Programmspeicheradresse zielen, also nach vorne springen, die „Branch Not Taken“-Methode angewandt wird, aber bei Sprüngen an eine niedrigere Adresse, also Rücksprüngen, die „Branch Taken“-Methode verwendet wird. Dies geschieht, da Rücksprünge oft auf *while* oder *for* Schleifen hindeuten und diese häufig wiederholt werden.
Trotzdem sind statische Sprungvorhersagen je nach ausgeführtem Programm nur zu 40%-80% richtig.

3.2.3.2 Dynamische Sprungvorhersage

Die Sprungvorhersage wird als dynamisch bezeichnet, wenn die Vorhersage zur Laufzeit aus dem bisherigen Sprungverhalten eines Sprungbefehls an einer bestimmten Befehlsadresse hergeleitet wird. Mit den dynamischen Sprungvorhersagemethoden erreicht man bei modernen Prozessoren Vorhersagen, die in bis zu 98% aller Sprünge zutreffen. Diese sind jedoch sehr aufwendig und werden deshalb nur bei Prozessoren mit sehr langen Pipelines eingesetzt, bei denen Flushe zu einer enormen Verzögerung führen.

3.2.3.2.1 Sprungvorhersagen mit State Machines:

Da die meisten bedingten Sprünge durch Schleifen verursacht werden, die mehrere Zyklen durchlaufen, kann man vorhersagen, dass der Sprung am Ende der Schleife mit sehr hoher Wahrscheinlichkeit erfolgen wird. Für jeden Sprung wird nun eine State-Machine implementiert, die, wenn der Sprung z.B. das letzte und vorletzte Mal erfolgte, vorhersagt, dass der

Sprung wieder ausgeführt wird. Sollte der Sprung die letzten beiden Male nicht erfolgt sein, wird er auch dieses Mal nicht ausgeführt (Siehe Abb. 3.7).

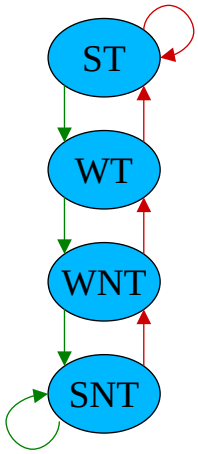

Abbildung 3.7: Einfache State Machine für eine Sprungvorhersage mit den States „Strong Taken“ (ST), „Weak Taken“ (WT), „Weak Not Taken“ (WNT) und „Strong Not Taken“ (SNT),

3.2.3.2.2 Sprungvorhersagen mit History Buffer:

Legt man für jeden Sprung zu einer bestimmten Adresse eine Tabelle mit „Branch Taken“ bzw. „Branch Not Taken“ Einträgen an, können aus dieser bestimmte Regeln für die Vorhersage des nächsten Sprungs an diese Adresse berechnet werden. Somit können auch wiederkehrende Schleifen, die z.B. immer 200-mal durchlaufen werden, richtig vorhergesagt werden.

3.2.3.3 Branch Delay Slots

Um die Zeit bis zur Auswertung eines bedingten Sprungbefehls zu nutzen oder den Pipeline-Flush bei einer falschen Vorhersage zu verkürzen, werden vom Compiler nach Branch Instruktionen in sogenannten „Branch Delay Slots“ Befehle eingeschoben, die unabhängig vom Ergebnis des Sprungs ausgeführt werden können. Solche Befehle lassen sich jedoch nicht immer finden. Dies führt oft dazu, dass einfach ein NOP-Befehl eingefügt wird.

3.3 Spezielle Pipeline-Strukturen

Flynn [6] unterteilt Pipeline-Strukturen von Prozessoren in folgende Klassen:

- statische Pipelines
- dynamische Pipelines

In einer statischen Pipeline durchläuft jeder Befehl alle Stufen nacheinander (In-Order). In einer dynamischen Pipeline (Out-of-Order) können hingegen Instruktionen, welche unabhängig von einem Befehl mit langer Wartezeit (z.B. Festplattenzugriff) sind, vor diesen vorgezogen werden. Befehle können dabei bestimmte von ihnen zur Ausführung nicht benötigte Stufen überspringen. Bei dynamischen Pipelines kommt es jedoch zu weiteren Konflikten, welche die Erhöhung des Durchsatzes wieder drosseln.
Da dies aber keinen direkten Effekt auf die Verlustleistung hat, wird für diese Arbeit eine mehr strukturell orientierte Einteilung benötigt. Die Pipeline-Strukturen werden hierbei in folgende Kategorien aufgeteilt:

- Einfache Pipelines (Bsp. i486; Abb. 3.8 li.)
- Superskalare Prozessoren (Bsp. Pentium, Abb. 3.8 re.)
- Parallel Branch Execution (Multiway Branching; Abb. 3.9)
- Vollständig parallele Pipelines (Multi-Core Systeme; Abb. 3.10)

3.3.1 Superskalare Prozessoren

Superskalare Prozessoren können mehrere Befehle aus einem Befehlsstrom gleichzeitig mit mehreren parallel arbeitenden Funktionseinheiten, wie z.B. ALU, Shifter oder Gleitkommarecheneinheit, parallel verarbeiten. Es handelt sich hierbei um Parallelität auf Befehlsebene, wobei die feingranulare, echte Nebenläufigkeit genutzt wird. Bei der Abarbeitung muss immer ein geeignetes Set von Mikroinstruktionen aus dem Befehls-Cache geladen und dekodiert werden. Die Funktionseinheiten können dabei wieder in Pipeline-Stufen unterteilt sein und auch unterschiedlich lange Pipelines besitzen.

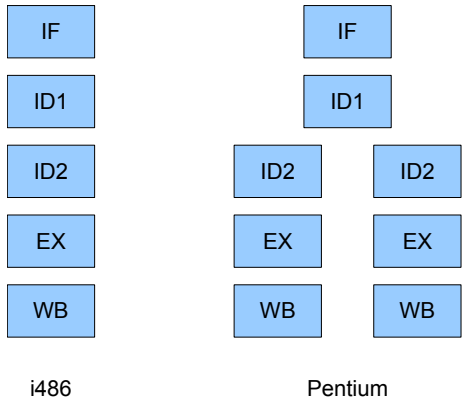

Abbildung 3.8: Links: Einfache Pipeline eines i486er Prozessors. Rechts: Bei der superskalaren Pipeline-Architektur des Intel Pentium wird ein Set an Instruktionen aus dem Speicher geholt, dieses wird dann dekodiert und auf die zwei Teil-Pipelines aufgeteilt

Man unterscheidet hierbei superskalare Prozessoren mit statischen und dynamischen Scheduling: Im ersten Fall werden die zu einem Set zu bündelnden Befehle vom Compiler ausgewählt und können während der Laufzeit nicht mehr verändert werden. Beim dynamischen Scheduling entscheidet der Prozessor zur Laufzeit, welche Befehle zu einem Set gebündelt werden.

3.3.2 Parallel Branch Execution

Um die Verzögerung eines Pipeline-Flushs zu vermeiden können bei einem Sprungbefehl durch das sogenannte Multiway Branching beide Verzweigungen eines Sprungs parallel ausgeführt werden. Beide Befehlspfade werden bis zu der Pipeline-Stufe parallel verfolgt, in der die Bedingung für den Sprung berechnet ist. Dann wird der richtige Befehlspfad weiterverfolgt und der falsche abgebrochen. Dies erfordert eine redundante Logik bis zu jener Pipeline-Stufe, die die Bedingung der Sprungbefehle berechnet (Siehe auch Abb. 3.9).

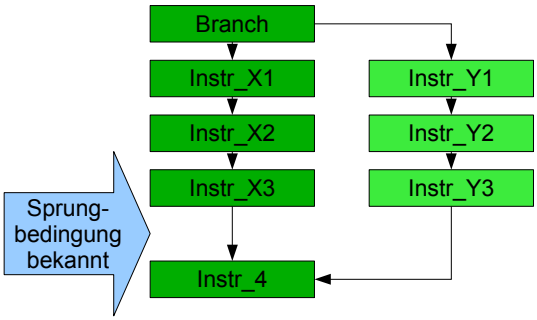

Abbildung 3.9: Beim Multiway Branching werden diejenigen Pipeline-Stufen eines Befehls-Flows parallel ausgeführt, von denen man noch nicht weiß, ob sie nach der Verzweigung ausgeführt werden oder nicht

3.3.3 Vollständig parallele Pipelines

In Multi-Core Prozessoren wie dem Core2Duo von Intel sind zwei oder mehr vollständig parallele Pipelines implementiert. Diese sind jedoch durch Daten- und Ressourcenabhängigkeiten nicht komplett voneinander unabhängig. Ein Beispiel für zwei parallele unabhängige Pipelines zeigt Abb. 3.10.

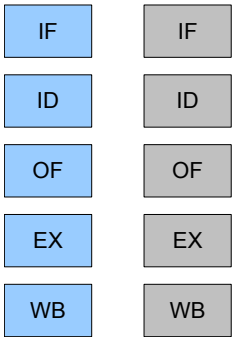

Abbildung 3.10: Zwei parallele unabhängige Pipelines

3.3.4 Pipeline-Länge

Auch die Anzahl der Stufen einer Pipeline (Siehe Abb. 3.11) spielt bei der Verlustleistung eine wichtige Rolle. Obwohl theoretisch eine längere

Pipeline den Durchsatz erhöhen müsste, kann eine zu lange Pipeline, wie etwa beim Pentium 4 Prescott mit über 30 Stufen, durch verlustreiche Pipeline-Flushes nachteilige Effekte auf die Performance und die Verlustleistung haben. Der Nachfolger dieses Prozessors, der Core Duo, hatte aus diesem Grund auch wieder nur eine 14-stufige Pipeline.

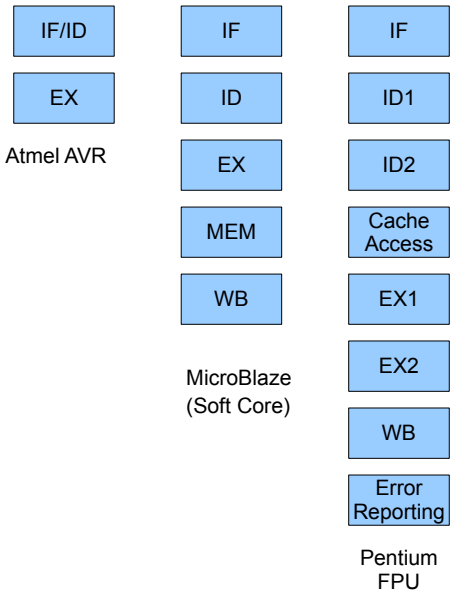

Abbildung 3.11: Einfache Pipelines unterschiedlicher Länge

In [11] und [12] wurden verschiedene Pipeline-Längen theoretisch und durch Simulationen untersucht. Hierbei wird sowohl auf die Performance als auch auf den Energiebedarf eingegangen. Der Performance-Gewinn durch eine längere bzw. tiefere Pipeline muss abgewogen werden gegen die höhere Verzögerung durch Hazards. Spielt der Energiebedarf eine Rolle, so bringen kurze Pipelines größere Effizienz mit sich. Bei Performance-Optimierung wird eher eine längere Pipeline (ca. 22 Stufen) empfohlen.

4 Methodik zur taktgenauen Berechnung der Verlustleistung

Eine taktgenaue Verlustleistungsanalyse unterstützt vor allem Hardware-Designern beim Entwurf verlustleistungsoptimaler Komponenten. Hierbei kann die Verlustleistung einzelner Befehle oder von Befehlskombinationen untersucht und Hardwarekomponenten ausgetauscht oder optimiert werden.
Während die modulbasierte Verlustleistungsabschätzung, die Low-Level Simulation und die Messung am Prototypen taktgenaue Ergebnisse liefern können, ist dies bei der instruktionsbasierten Verlustleistungsabschätzung generell nicht der Fall. Es wird zwar in vielen Veröffentlichungen behauptet „cycle-accurate“, also taktgenau, zu sein, jedoch wird für jeden Takt angenommen, dass genau ein Befehl ausgeführt wird. Dies ist bei Prozessoren ohne Pipeline meist auch richtig. Bei Prozessoren mit Pipelining werden hingegen, wie in Kapitel 3.1 beschrieben, in einem Takt mehrere Befehle gleichzeitig in den verschiedenen Stufen der Pipeline ausgeführt (Siehe auch Abb. 3.1).

4.1 Unterteilung nach Pipeline-Stufen

Für eine taktgenaue instruktionsbasierte Verlustleistungsabschätzung ist folglich eine genaue stufenweise Nachbildung der Ausführung der Befehle in der Pipeline erforderlich.
Hierbei wird nach dem „Teile und herrsche“ Prinzip vorgegangen, ähnlich dem, welches auch bei der modulbasierten Verlustleistungsabschätzungsmethode verwendet wird.
Zuerst wird die gesamte Verlustleistung in die statische Verlustleistung

P_{stat} und die dynamische Verlustleistung P_{dyn} unterteilt:

$$P_{ges} = P_{stat} + P_{dyn} \tag{4.1}$$

Obwohl in dieser Arbeit die Temperatur als konstant angenommen wird, und damit die statische Verlustleistung als konstant angesehen wird, spielt sie bei der Berechnung der gesamten verbrauchten Energie eines zu untersuchenden Programms eine Rolle.
Die dynamische Verlustleistung wird weiter in einen befehlsabhängigen (P_{ba}) und einen befehlsunabhängigen (P_{bu}) Teil unterteilt:

$$P_{dyn} = P_{bu} + P_{ba} \tag{4.2}$$

Die befehlsunabhängige Verlustleistung ist hierbei die Verlustleistung der Teileinheiten eines Prozessors, die nicht unmittelbar von dem jeweiligen ausgeführten Befehl abhängen. Hierzu zählen z.B. globale Kontrolllogik, die Taktgeneratoren, die aus dem Haupttakt verschiedene weitere Takte generieren, und Timer.
Zwar ist P_{bu} nicht von dem gerade ausgeführten Befehl abhängig, im Gegensatz zur statischen Verlustleistung hängt sie jedoch linear von der Taktfrequenz des Prozessors ab.

$$P_{bu} = U_{dd}^2 \cdot C_{bu} \cdot f_{clk} \cdot \alpha, \tag{4.3}$$

Die befehlsabhängige Verlustleistung ist der Teil der dynamischen Verlustleistung, der von dem gerade auf dem Prozessor ausgeführten Befehl abhängt. Sie setzt sich aus der Verlustleistung der Pipeline und der dazugehörigen Steuerung zusammen.
Da aber mehrere Befehle gleichzeitig in verschiedenen Stufen der Pipeline verarbeitet werden (Siehe auch Abb. 4.1), muss diese Verlustleistung wiederum weiter unterteilt werden in die Verlustleistungen P_{ps} der einzelnen Pipeline-Stufen. Diese sind für jede Pipeline-Stufe und für jede Instruktion unterschiedlich und beinhalten die Leistungsaufnahme der jeweiligen Pipeline-Stufe und ihrer Steuerungslogik. Die Verlustleistung $P_{ps}(I, J)$ ist somit die Verlustleistung, die verbraucht wird, wenn die Pipeline-Stufe J die Instruktion I verarbeitet.

Dabei ist die jeweilige Verlustleistung der einzelnen Pipeline-Stufen von dem gerade in diesen ausgeführten Befehl abhängig. Wird z.B. ein *Load*

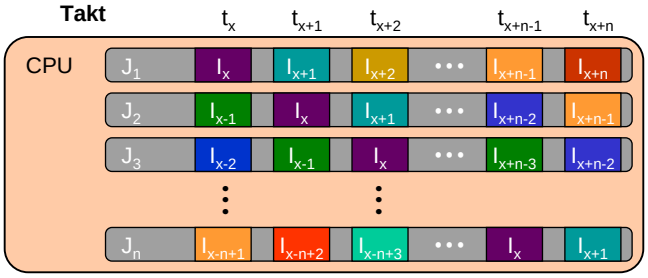

Abbildung 4.1: Ohne Konflikte durchläuft jeder Befehl I_x alle n Stufen J_1 bis J_n der Pipeline in n Takten

Befehl (Ld) in der EX Stufe der Pipeline ausgeführt, verbraucht diese Stufe in diesem Takt weniger Energie als bei einem Add Befehl. In der MEM Stufe der Pipeline wird der Ld Befehl dagegen mehr Energie verbrauchen als der Add Befehl.
Wenn nur ein Befehl von der Pipeline verarbeitet wird und man annimmt, dass alle anderen Pipeline-Stufen, die nicht genutzt werden, keine Energie verbrauchen, so kann die Verlustleistung P_{ba} der Pipeline für jeden Takt durch folgende Formeln bestimmt werden:

$$\begin{aligned} P_{ba}(t_1) &= P_{ps}(I, J_1) \\ P_{ba}(t_2) &= P_{ps}(I, J_2) \\ &\vdots \\ P_{ba}(t_n) &= P_{ps}(I, J_n) \end{aligned} \tag{4.4}$$

Dies gilt nur solange die anderen Stufen der Pipeline keine Leistung verbrauchen und keine Konflikte auftreten. Im normalen Betrieb ist die Pipeline jedoch gefüllt, und jede Pipeline-Stufe verarbeitet einen Teil einer Instruktion. Für die gesamte befehlsabhängige Verlustleistung P_{ba} einer n-stufigen vollständig gefüllten Pipeline zum Zeitpunkt t_x gilt:

$$P_{ba}(t_x) = \sum_{k=1}^{n} P_{ps}(I_{x-k+1}, J_k) \tag{4.5}$$

4.2 Beispiel: Fünfstufige Pipeline

Für das erweiterte Beispiel in Abb. 4.2, der fünfstufigen Pipeline aus Kapitel 3.1, ergibt sich für die gesamte Verlustleistung der Takte t_1, t_2 und t_3:

$$
\begin{aligned}
P_{ges}(t1) &= P_{stat} + P_{bu} + P_{WB}(Add) + P_{MEM}(Mov) \\
&\quad + P_{EX}(Or) + P_{ID}(Ld) + P_{IF}(Sl) \\
P_{ges}(t2) &= P_{stat} + P_{bu} + P_{WB}(Mov) + P_{MEM}(Or) \\
&\quad + P_{EX}(Ld) + P_{ID}(Sl) + P_{IF}(St) \\
P_{ges}(t3) &= P_{stat} + P_{bu} + P_{WB}(Or) + P_{MEM}(Ld) \\
&\quad + P_{EX}(Sl) + P_{ID}(St) + P_{IF}(Sub)
\end{aligned}
$$

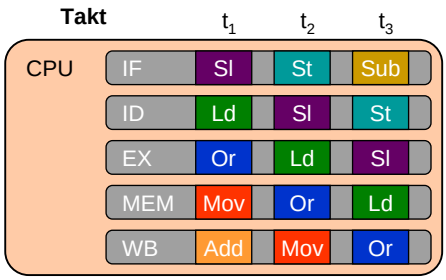

Abbildung 4.2: Drei Pipeline-Takte eines Befehlsablaufs in einer fünfstufigen Pipeline

Hierbei sind P_{stat} und P_{bu} jeweils die statische, bzw. befehlsunabhängige Verlustleistung für einen Takt. Diese sind für jeden Takt gleich, da die Abweichung durch unterschiedliche Belegung von Registern vernachlässigbar ist und einer zusätzlichen Mittelung unterliegt. Die befehlsabhängige Verlustleistung der einzelnen Pipeline-Stufen ist jedoch von Takt zu Takt verschieden. Für Takt t_1 z.B. setzt sie sich zusammen aus der Verlustleistung des *Add* Befehls in der WB Stufe, der Verlustleistung des *Mov* Befehls in der MEM Stufe, der Verlustleistung des *Or* Befehls in der EX Stufe, der Verlustleistung des *Ld* Befehls in der ID Stufe und der Verlustleistung des *Sl* Befehls in der IF Stufe zusammen. Im nächsten Takt ist der *Add* Befehl fertig bearbeitet und fällt aus der Pipeline heraus. Alle weiteren Befehle rücken eine Stufe weiter und in der IF Stufe kommt der

St Befehl dazu. Die Verlustleistungen des *Mov* Befehls in der WB Stufe, des *Or* Befehls in der MEM Stufe, des *Ld* Befehls in der EX Stufe, des *Sl* Befehls in der ID Stufe und des *St* Befehls in der IF Stufe addieren sich für Takt t_2 zur gesamten befehlsabhängigen Verlustleistung. Genauso wird im nächsten Takt t_3 verfahren.
Somit ist es nun möglich, für jeden Takt die Verlustleistung der Pipeline zu bestimmen und die Leistungsaufnahme für den gesamten Prozessor abzuschätzen. Diese Methode der Unterteilung der Verlustleistung erlaubt auch die Effekte, welche während der Abarbeitung der Befehle Konflikte in der Pipeline verursachen und sich damit auf die Verlustleistung auswirken, nachzubilden. Weiterhin kann für komplexere Pipeline-Strukturen die Verlustleistung abgeschätzt und die Struktur der Pipeline bezüglich der Leistungsaufnahme optimiert werden.

5 Allgemeines Modell zur Berechnung des Energieverbrauchs in Befehls-Pipelines

In diesem Kapitel wird veranschaulicht, wie die instruktionsbasierte Abschätzung der Verlustleistung für Programme oder Programmteile durch die Berücksichtigung der Pipeline-Stufen erheblich verbessert werden kann. Zur Berechnung der Verlustleistung werden allgemeine Formeln von Pipeline-Strukturen mit und ohne Hazard-Wirkung entwickelt.
Hierzu wird in Kapitel 5.2 näher auf die generellen Einflüsse durch Pipeline-Hazards auf die Verlustleistung eingegangen.
Für die in Kapitel 3.3 vorgestellten speziellen Pipeline-Strukturen werden dann in Kapitel 5.3 Modelle zur Berechnung der Verlustleistung beschrieben.

5.1 Berechnung der Verlustleistung ohne Pipeline-Hazards

Den folgenden Betrachtungen werden zunächst Prozessoren mit singulär n-stufigen Pipelines ohne Forwarding und mit uniformer Taktung zugrunde gelegt.
Auch wenn in jedem Prozessortakt mehrere Teilinstruktionen gleichzeitig in den verschiedenen Stufen der Pipeline bearbeitet werden, wird jeder Befehl I_i, der eine beliebige Instruktion aus der Menge W aller auf dem Prozessor ausführbaren Instruktionen sein kann und an der i-ten Stelle einer Befehlsfolge ausgeführt wird, im Normalfall (ohne Pipeline-Flushes oder Multi-Cycle Instruktionen) in jeder Stufe einen Takt lang bearbeitet

(Siehe auch Kapitel 4.1). Zur Berechnung der Verlustleistung von einer Befehlsfolge mit den Befehlen I_1 bis I_m werden die Verlustleistungen der einzelnen Stufen für jede Instruktion I_i aufsummiert, auch wenn diese eigentlich zeitlich versetzt ausgeführt werden.

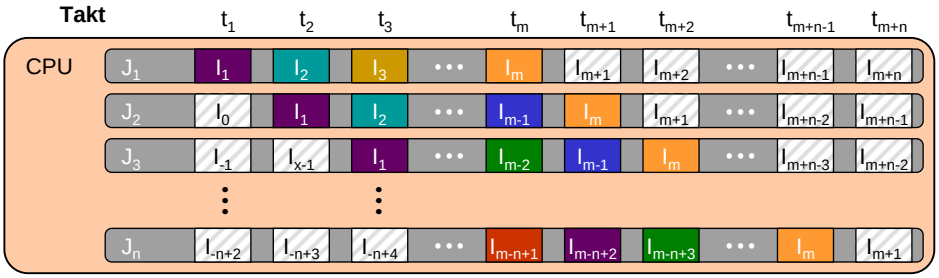

Abbildung 5.1: Alle Befehle I_1 bis I_m der Befehlsfolge durchlaufen alle n Stufen J_1 bis J_n der Pipeline

Für die Bearbeitung eines Programmabschnitts mit m Befehlen auf einer Pipeline mit n Stufen werden $m + n - 1$ Takte benötigt. Jedoch werden beim Füllen und Leeren der Pipeline auch Befehle ausgeführt, die nicht zu der Befehlsfolge gehören (Siehe Abb. 5.1). Will man nur die Verlustleistung dieser Befehlsfolge ermitteln, schiebt man die Befehle, die sich am Ende noch in der Pipeline befinden, an den Anfang an jene Stellen, welche normalerweise nicht zur Befehlsfolge gehörende Instruktionen verarbeiten und erhält damit die m Takte, die für die Abschätzung relevant sind.
Die Formel für die mittlere Verlustleistung P_{ges} dieser Befehlsfolge ergibt sich durch Mittelung über die befehlsabhängigen Verlustleistungen $P_{ba}(t_i)$ aller ausgeführter Instruktionen, addiert mit der mittleren statischen Verlustleistung P_{stat} und der mittleren befehlsunabhängigen Verlustleistung P_{bu}, zu:

$$P_{ges} = P_{stat} + P_{bu} + \frac{\sum_{i=1}^{m} P_{ba}(t_i)}{m}, \tag{5.1}$$

Wenn man nun für $P_{ba}(t_x)$ das Ergebnis aus Formel (4.5) einsetzt, erhält man:

$$P_{ges} = P_{stat} + P_{bu} + \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} P_{ps}(I_i, J_j)}{m} \tag{5.2}$$

$P_{ps}(I_i, J_j)$ ist dabei die Verlustleistung des ausgeführten Befehls I_i in der Pipeline-Stufe J_j ($I_i \in W$).
Energiewerte eignen sich besser für den Aufbau einer Datenbank, da sie, im Gegensatz zur Verlustleistung, nicht von der Frequenz abhängen. Nach Formel (2.5) ergibt sich die gesamte Energie E_{ges}, die durch diese Befehlsfolge verbraucht wird, zu:

$$\begin{aligned} E_{ges} &= P_{ges} \cdot t = P_{ges} \cdot \frac{m}{f_{clk}} \\ &= \left((P_{stat} + P_{bu}) \cdot m + \sum_{i=1}^{m}\sum_{j=1}^{n} P_{ps}(I_i, J_j) \right) \cdot \frac{1}{f_{clk}} \\ &= (E_{stat} + E_{bu}) \cdot m + \sum_{i=1}^{m}\sum_{j=1}^{n} E_{ps}(I_i, J_j), \end{aligned} \tag{5.3}$$

wobei f_{clk} die Taktfrequenz der Prozessor-Pipeline ist und E_{stat}, bzw. E_{bu} die statische, bzw. befehlsunabhängige Energie ist, die in einem Takt verbraucht wird. $E_{ps}(I_i, J_j)$ ist die dynamische Energie, die während eines Taktes, in dem der Befehl I_i in der Pipeline-Stufe J_j bearbeitet wird, von dieser Pipeline-Stufe verbraucht wird.

5.2 Einfluss der Pipeline-Hazards auf die Verlustleistungsberechnung

Die in Kapitel 4 und Kapitel 5.1 vorgestellten Methoden und auch die bisherigen instruktionsbasierten Verlustleistungsabschätzungsmethoden liefern nur solange relativ genaue Ergebnisse, solange keine Konflikte in der Pipeline auftreten, die Pipeline also immer gefüllt ist und ohne Verzögerung weiterläuft.

In diesem Kapitel wird nun der Einfluss der Pipeline-Hazards auf die Berechnung des Energieverbrauchs näher betrachtet.

5.2.1 Pipeline-Stall

Pipeline-Stalls, bei denen die komplette Pipeline angehalten wird, wirken sich nur auf den statischen Energieverbrauch aus, weil dabei alle Pipeline-Stufen angehalten werden und keine dynamischen Prozesse oder Schaltvorgänge in den Stufen ablaufen. Außerdem werden alle Register auf den aktuellen Werten gehalten, wodurch die Verarbeitung jederzeit fortgesetzt werden kann. In diesem Fall sinkt die durchschnittliche Verlustleistung des gesamten Programms durch die längeren Zeiten des Stillstands und der damit verbundenen niedrigeren Leistungsaufnahme. Die gesamte Energie, die ein Programm während der Laufzeit verbraucht, steigt jedoch wegen der längeren Ausführungsdauer und der dadurch zusätzlich verbrauchten statischen Energie an.

$$E_{ges} = (E_{stat} + E_{bu}) \cdot (m + s) + \sum_{i=1}^{m} \sum_{j=1}^{n} E_{ps}\left(I_i, J_j\right), \tag{5.4}$$

wobei s die Anzahl der Taktzyklen ist, während derer die Pipeline sich im „Stall“-Zustand befunden hat, also komplett angehalten wurde.
Passiert dieser Stall aufgrund einer Multi-Cycle Instruktion, muss die Verlustleistung der mehrere Takte ausführenden Stufe berücksichtigt werden:

$$\begin{aligned} E_{ges} = \; & (E_{stat} + E_{bu}) \cdot (m + mc) + \sum_{i=1}^{m} \sum_{j=1}^{n} E_{ps}\left(I_i, J_j\right) \\ & + E_{ps}\left(I_{mc}, J_{mc}\right) \cdot mc, \end{aligned} \tag{5.5}$$

wobei mc die Anzahl der Taktzyklen ist, die die Multi-Cycle Instruktion I_{mc} in der Stufe J_{mc} zusätzlich benötigt. $E_{ps}\left(I_{mc}, J_{mc}\right)$ ist dabei die Energie, die während eines Taktes von der Multi-Cycle Instruktion in dieser Stufe verbraucht wird. Hierbei wird davon ausgegangen, dass der Energieverbrauch des Multi-Cycle Befehls in dieser Stufe über alle Taktzyklen mc konstant ist.

5.2.2 Bubbling

Beim Bubbling muss zusätzlich der Energieverbrauch für die eingefügten Bubbles berücksichtigt werden. Es gibt zwei unterschiedliche Methoden, Bubbles zu realisieren.

Methode mit NOP Befehlen

Eine Methode ist es, *NOP* Befehle[1] für die durch Konflikte bedingten Wartezeiten einzufügen. Hierbei werden die Register vor den betroffenen Pipeline-Stufen durch das Steuerwerk oder durch eine spezielle *Clear*-Logik auf einen vordefinierten Wert gesetzt, der einem *NOP* Befehl entspricht. Diese zusätzliche Steuerungs- oder Clear-Logik erhöht dabei den befehlsunabhängigen Energieverbrauch E_{bu}.

$$\begin{aligned} E_{ges} = & \left(E_{stat} + E_{bu}\right) \cdot (m + s) + \sum_{i=1}^{m} \sum_{j=1}^{n} E_{ps}\left(I_i, J_j\right) \\ & + \left(\sum_{k=psx+1}^{n} E_{ps}\left(I_{NOP}, J_k\right) \right) \cdot s + E_{ps}\left(I_{psx}, J_{psx}\right) \cdot s, \end{aligned} \tag{5.6}$$

wobei s die Anzahl der Taktzyklen ist, die der Befehl, vor dem die Bubbles eingeschoben werden, warten muss, bis er weiter ausgeführt werden kann und in die nächste Pipeline-Stufe vorrücken darf. Dieser Befehl, der durch den Konflikt in der Pipeline-Stufe mit der „Folgenummer“ psx warten muss, wird trotzdem in dieser Stufe weiter ausgeführt und verbraucht dabei die für diese Pipeline-Stufe charakterisierte Energie.

Methode mit Halten der Stufen-Zustände

Bei der zweiten Methode werden die Register vor den betroffenen Stufen auf dem vorigen Wert eingefroren. Es wird sozusagen der vorherige Befehl in der Stufe gehalten, obwohl er auch gleichzeitig in den nächsten Stufen

[1] NOP (No Operation) Befehle sind neutrale Instruktionen, die den Prozessor anweisen, keine Rechenoperation auszuführen und auch keine Status- oder Zustandsänderungen vorzunehmen. Sie dienen als Verzögerungen

weiter propagiert wird. Doppelte Schreibvorgänge werden durch zusätzliche Logik verhindert.
Bei dieser Methode wird durch das Einfrieren der Stufen keine zusätzliche dynamische Energie für die Ausführung der NOP Befehle verbraucht. Dadurch kann die Formel (5.6) durch Streichen dieses Summenterms folgendermaßen vereinfacht werden:

$$\begin{aligned} E_{ges} = & \left(E_{stat} + E_{bu}\right) \cdot (m + s) + \sum_{i=1}^{m} \sum_{j=1}^{n} E_{ps}\left(I_i, J_j\right) \\ & + E_{ps}\left(I_{psx}, J_{psx}\right) \cdot s, \end{aligned} \tag{5.7}$$

Die Bubbling Methode mit dem Halten der Zustände ist dabei energetisch etwas günstiger als NOP Befehle in die Pipeline einzufügen, jedoch ist der Aufwand für die benötigte zusätzliche Logik geringfügig höher.
Da aber $E_{ps}\left(I_{NOP}, J\right)$ sehr klein ist, kann es meist vernachlässigt werden, und zur Vereinfachung auch im ersten Fall Formel (5.7) verwendet werden.

5.2.3 Pipeline-Flush

Besonders starken Einfluss auf die Verlustleistung haben die Verzweigungs- und Sprungbefehle (engl.: Branch, bzw. Jump) und die daraus folgenden Pipeline-Flushes. Einerseits ergibt sich dieser Einfluss durch das relativ häufige Auftreten von Pipeline-Flushes, andererseits durch die Ausführung von Befehlen, die durch den Flush wieder verworfen werden. Diese Befehle wurden in bisherigen instruktionsbasierten Verlustleistungsabschätzungen nicht berücksichtigt.
Eine Untersuchung des Assembler Codes von mehreren unterschiedlichen Programmen ergab, dass typischer Assembler Code durchschnittlich etwa 20% bis 25% Branch Instruktionen enthält, die bei jeder falschen Sprungvorhersage zu einem Pipeline-Flush führen. Bei statischer Sprungvorhersage liegt die Wahrscheinlichkeit einer falschen Sprungvorhersage bei 20% bis 60%. Dadurch kommt es im besten Fall in 4% aller abgearbeiteten Befehle zu einem Pipeline-Flush, im schlechtesten Fall jedoch in 15% aller ausgeführten Befehle.
Ist die Sprungvorhersage korrekt, wird keine zusätzliche Energie verbraucht und auch die Zahl der Taktzyklen ändert sich nicht. Bei einer falschen Sprungvorhersage werden Befehle jedoch teilweise ausgeführt und durch

einen Pipeline-Flush abgebrochen. Diese teilweise ausgeführten Befehle verursachen, neben der Verzögerung durch das Leeren und Neufüllen der Pipeline, auch einen höheren dynamischen Energieverbrauch.
Für die Berechnung der zusätzlich verbrauchten Energie durch die Pipeline-Flushs ist dabei nicht die Art der Sprungvorhersage oder Flush-Vermeidung von Bedeutung, sondern nur die Häufigkeit von solchen Pipeline-Flushs.

5.2.3.1 Berechnung mit Energieverbrauchswerten

Eine relativ einfache Methode, Pipeline-Flushes bei der instruktionsbasierten Verlustleistungsabschätzung zu berücksichtigen, besteht darin, für die Flushes einen festen zusätzlichen Energieverbrauch festzulegen. Dafür ist für jeden Sprungbefehl eine Charakterisierung mit folgendem Pipeline-Flush und eine Charakterisierung mit korrekter Sprungvorhersage durchzuführen.
Bei der Abschätzung muss anhand des Befehls-Traces herausgefunden werden, ob bei einem Sprung ein Pipeline-Flush stattgefunden hat oder nicht (Siehe dazu auch Kapitel 6.3.1). Die Effektivität und Genauigkeit dieser Methode wird in Kapitel 7 in Fallbeispiel 1 untersucht.
Diese Methode verbessert die Genauigkeit der instruktionsbasierten Verlustleistungsabschätzung, jedoch funktioniert sie nur für die Ermittlung des gesamten Energieverbrauchs, nicht für eine taktgenaue Analyse. Um die Genauigkeit noch zu steigern und den Energieverbrauch auch taktgenau abschätzen zu können, muss man die abgebrochenen Befehle bei einem Pipeline-Flush und deren teilweise Ausführung in den Stufen der Pipeline bei der Berechnung der Verlustleistung berücksichtigen.

5.2.3.2 Taktgenaue Abschätzung

Hierzu sind bei einem Pipeline-Flush zwei Implementationen zu unterscheiden: Bei der ersten Methode werden die fälschlich ausgeführten Befehle fertig ausgeführt und nur das Ergebnis verworfen, bei der zweiten werden sie abgebrochen.
In Pipelines, bei denen das Ergebnis der fälschlich ausgeführten Befehle einfach mit einem „nicht gültig“ Flag gekennzeichnet wird, werden die Befehle noch vollständig ausgeführt, aber das Zurückschreiben des Ergebnisses wird blockiert. Hierbei wird etwa die gleiche dynamische Verlust-

leistung verbraucht wie bei der normalen Ausführung der Befehle.
Die verbrauchte Energie E_{fl1} für einen solchen Pipeline-Flush, die zusätzlich zu der gesamten Energie aus Formel (5.3) des normalen Programmablaufs verbraucht wird, ergibt sich zu:

$$E_{fl1} = (E_{stat} + E_{bu}) \cdot f + \sum_{k=1}^{f} \sum_{l=1}^{n} E_{ps} \left(I_{F_k}, J_l \right), \tag{5.8}$$

wobei f die Anzahl der bei dem Flush fälschlich ausgeführten Befehle I_{F_1} bis I_{F_f} ist, die hier die komplette Pipeline von Stufe J_1 bis Stufe J_n durchlaufen.

Pipelines, bei denen bei einem Pipeline-Flush die Register zwischen den Stufen auf einen Nullzustand zurückversetzt werden, also NOP Befehle eingefügt werden, führen die fälschlich begonnen Befehle nicht zu Ende aus. Die Befehle werden stattdessen in der jeweiligen Stufe, in der sie sich gerade befinden, abgebrochen und benötigen deshalb nur einen Teil der dynamischen Verlustleistung, die sie normalerweise bei voller Ausführung benötigen würden.
Bei der Berechnung des Energieverbrauchs für einen solchen Pipeline-Flush mit einem Abbruch der Befehle, wie es in Abb. 3.6 für eine fünfstufige Pipeline dargestellt ist, darf nur die verbrauchte Energie der abgebrochenen Befehle in den Pipeline-Stufen berücksichtigt werden, die auch tatsächlich ausgeführt wurden.
Der erste Befehl I_{F_1}, der spekulativ nach dem Sprung ausgeführt wird, wird nach der Pipeline-Stufe J_x abgebrochen, durchläuft also die Stufen J_1 bis J_x. Der darauf folgende fälschlich ausgeführte Befehl I_{F_2} befindet sich zum selben Zeitpunkt noch in der vorherigen Pipeline-Stufe J_{x-1}, nach der dieser auch abgebrochen wird. Der letzte Befehl I_{F_f}, der durch den Pipeline-Flush abgebrochen wird, durchläuft nur noch die erste Stufe der Pipeline J_1.
Die zusätzlich verbrauchte Energie E_{fl2} für die zweite Methode ergibt sich dann zu:

$$\begin{aligned} E_{fl2} = \; & (E_{stat} + E_{bu}) \cdot f + \sum_{l=1}^{x} E_{ps} \left(I_{F_1}, J_l \right) + \sum_{l=1}^{x-1} E_{ps} \left(I_{F_2}, J_l \right) \\ & + \cdots + E_{ps} \left(I_{F_f}, J_1 \right) \end{aligned} \tag{5.9}$$

5.2.3.3 Berücksichtigung von Delay-Slots

Weiterhin werden Pipeline-Flushes zusätzlich durch die Verwendung von Delay Slots beeinflusst. Durch diese reduzieren sich die Befehle, die bei einem Flush abgebrochen werden müssen, da nach dem Sprung unabhängige Befehle ausgeführt werden, die fertig bearbeitet werden, auch wenn die Sprungvorhersage sich als falsch herausstellt. Da im ersten Fall in Formel (5.8) die Befehle sowieso komplett ausgeführt werden, muss diese Formel nicht verändert werden. Es ist jedoch zu beachten, dass die Anzahl der fälschlich ausgeführten Befehle f nicht die Delay Slot Befehle enthält, sie wird also reduziert um die Anzahl der Delay Slot Befehle d. Außerdem ist der Befehl I_{F_1} dann der erste fälschlich ausgeführte Befehl nach den Delay Slot Befehlen.
Im zweiten Fall muss zusätzlich noch beachtet werden, dass die abgebrochenen Befehle I_{F_1} bis I_{F_f}, da sie d Takte später ausgeführt werden, auch d weniger Pipeline-Stufen durchlaufen. Der Befehl I_{F_1} wird somit schon nach der Stufe J_{x-d} abgebrochen.
Die Formel für die durch den Pipeline-Flush zusätzlich verbrauchte Energie lautet dann:

$$\begin{aligned} E_{fl3} = & \left(E_{stat} + E_{bu}\right) \cdot f + \sum_{l=1}^{x-d} E_{ps}\left(I_{F_1}, J_l\right) + \sum_{l=1}^{x-d-1} E_{ps}\left(I_{F_2}, J_l\right) \\ & + \cdots + E_{ps}\left(I_{F_f}, J_1\right) \end{aligned} \tag{5.10}$$

5.3 Modelle für spezielle Pipeline-Strukturen

Die in den beiden vorhergehenden Kapiteln beschriebenen Modelle sind auf Prozessoren mit einer einfachen Pipeline ausgelegt. Für die in Kapitel 3.3 beschriebenen speziellen Pipeline-Strukturen müssen die Formeln teilweise angepasst werden.
Hier wird nun der Einfluss einiger sehr häufig vorkommender spezieller Pipeline-Strukturen auf die Berechnung des Energieverbrauchs untersucht.

5.3.1 Out-of-Order Execution

Bei der Out-of-Order Execution wird die normale Reihenfolge der Befehle verändert, um Pipeline-Stalls zu vermeiden. Hierbei werden Befehle, wenn dies durch Datenunabhängigkeit möglich ist, außerhalb der Reihe der Befehlsfolge ausgeführt.
Durch die Vermeidung von Pipeline-Stalls würde normalerweise der gesamte Energieverbrauch sinken, jedoch steigt der Energieverbrauch durch die zusätzliche Logik für den Out-of-Order-Ausführungs-Scheduler, der zur Ermittlung der unabhängigen Befehle nötig ist, stark an. Um sich diese Logik zu sparen, hat z.B. der Intel Atom Prozessor nur eine strikte In-Order Pipeline und kann damit seinen Energieverbrauch um etwa 26% senken [9].
Auf die Berechnung des Energieverbrauchs nach den Methoden aus den Kapiteln 4, 5.1 und 5.2 hat die Out-of-Order Ausführung keinen Einfluss. Die Formeln sind auch in diesem Fall gültig und können ohne Modifikationen angewandt werden. Jedoch muss die Reihenfolge der Instruktionen des Befehls-Traces, der der Berechnung zugrunde gelegt wird, auch der wirklichen Ausführungsreihenfolge im Prozessor entsprechen. Dafür muss das zu untersuchende Programm simuliert werden, und der Simulator muss auch den Out-of-Order-Ausführungs-Scheduler realitätsnah simulieren können. Da bei Out-of-Order Execution bestimmte Pipeline-Stufen von Befehlen auch übersprungen werden können, muss dies in der Berechnung des Energieverbrauchs berücksichtigt werden:

$$\begin{aligned} E_{ges} = & \left(E_{stat} + E_{bu}\right) \cdot (m + s - o) + \sum_{i=1}^{m} \sum_{j=1}^{n} E_{ps}\left(I_i, J_j\right) \\ & - \sum_{k=1}^{o} E_{ps}\left(I_{O_k}, J_{O_k}\right), \end{aligned} \tag{5.11}$$

wobei s die die Anzahl der Stall-Zyklen, o die Anzahl der durch Out-of-Order Execution vermiedenen Stall-Zyklen und I_{O_k} die k-te Out of Order Instruktion ist, die die Pipeline-Stufe J_{O_k} überspringt.

5.3.2 Superskalare Prozessoren

Bei superskalaren Prozessoren wird immer ein Set von Befehlen geladen und dekodiert und dann auf mehrere Funktionseinheiten, wie z.B. ALU, Shifter oder Gleitkommarecheneinheit, verteilt. Hierbei muss man die Pipeline in mehrere Unter-Pipelines unterteilen, und für jede dieser Pipelines separat den Energieverbrauch berechnen. Für eine superskalare Architektur mit zwei separaten Funktionseinheiten zerlegt man die Pipeline in drei Unter-Pipelines: Die Pipeline-Stufen für das Befehlholen und -dekodieren, die die Befehle noch als Befehls-Set verarbeiten, werden als erste Teil-Pipeline betrachtet. Die zweite und dritte Unter-Pipeline sind jeweils die Pipelines der zwei Funktionseinheiten und deren Vor- und Nachverarbeitung der Befehle.
Die gesamte Energie E_{ges} für einen superskalaren Prozessor mit zwei Funktionseinheiten U und Y berechnet sich dann nach folgender Formel:

$$\begin{aligned} E_{ges} = & \left(E_{stat} + E_{bu}\right) \cdot v + \sum_{i=1}^{v} \sum_{j=1}^{f} E_{ps}\left(I_{S_i}, J_{S_j}\right) \\ & + \sum_{i=1}^{u} \sum_{j=1}^{g} E_{ps}\left(I_{U_i}, J_{U_j}\right) + \sum_{i=1}^{y} \sum_{j=1}^{h} E_{ps}\left(I_{Y_i}, J_{Y_j}\right), \end{aligned} \tag{5.12}$$

wobei v die Anzahl der Takte ist, die das Programm zur Abarbeitung benötigt. Durch die teilweise parallele Verarbeitung der Befehle ist v ohne Simulation nicht trivial festzustellen, selbst wenn keine Pipeline-Hazards auftreten. v ist dabei die Anzahl der Befehls-Sets, die die erste Teil-Pipeline durchlaufen.
Die Befehls-Sets I_{S_1} bis I_{S_v} durchlaufen dabei die Pipeline-Stufen J_{S_1} bis J_{S_f} der ersten Teil-Pipeline, bevor sie aufgeteilt und auf die Funktionseinheiten U und Y verteilt werden. Hier wird der eine Teil der Befehle (hier durch I_{U_1} bis I_{U_u} dargestellt) durch die Pipeline-Stufen J_{U_1} bis J_{U_g} der Funktionseinheit U weiterverarbeitet, der andere Teil (hier I_{Y_1} bis I_{Y_y}) durchläuft die Pipeline-Stufen J_{Y_1} bis J_{Y_h} der Funktionseinheit Y.
Für mehr als zwei Funktionseinheiten lässt sich diese Formel einfach durch weitere Summenblöcke für die jeweiligen Funktionseinheiten erweitern. Bei Flushes und Stalls muss beachtet werden, dass sie teilweise nur einzelne Unter-Pipelines betreffen, dann aber nach demselben Prinzip wie in dem Kapitel 5.2 berechnet werden können.

5.3.3 Parallel Branch Execution

Der Energieverbrauch bei Prozessoren mit Parallel Branch Execution kann nach dem gleichen Prinzip wie bei Prozessoren mit einfachen Pipelines berechnet werden. Nur Sprungbefehle müssen hier extra behandelt werden. Bei Verzweigungen durch Sprünge werden beide Befehlspfade parallel verfolgt. Somit muss bei jedem Sprung die dynamische Energie von der richtigen und der falschen Verzweigung bis zu jener Pipeline-Stufe, in der die Bedingung für den Sprung ausgewertet ist, in die Berechnung mit einbezogen werden.
Die gesamte Energie für Pipelines mit Parallel Branch Execution kann dann nach Formel (5.13) berechnet werden.

$$\begin{aligned} E_{ges} = & \left(E_{stat} + E_{bu}\right) \cdot m + \sum_{i=1}^{m} \sum_{j=1}^{n} E_{ps}\left(I_i, J_j\right) \\ & + \sum_{h=1}^{s} \sum_{i=x_h}^{y_h} \sum_{j=1}^{x} E_{ps}\left(I_i, J'_j\right) \end{aligned} \tag{5.13}$$

Der im Vergleich zu Formel (5.3) dazugekommene Summenblock ist hier der Energieverbrauch der zusätzlichen Stufen J'_1 bis J'_x, die nach jedem der s Sprungbefehle die jeweiligen Befehle I_{x_h} bis I_{y_h} des alternativen Befehlspfad ausführen.
Für den statischen Energieverbrauch muss beachtet werden, dass es nach falschen Sprungvorhersagen keine Pipeline-Flushes und damit auch keine Verzögerungen mehr gibt. Die befehlsunabhängige und die statische Energie sind bei Prozessoren mit Multiway Branching aber generell höher durch die komplexere und mehr Fläche in Anspruch nehmende zusätzliche Logik.

5.3.4 Vollständig parallele Pipelines

Vollständig parallele Pipeline-Strukturen, wie sie in Multi-Core Prozessoren vorkommen, haben meist auch komplett getrennte Steuerungslogik und können somit jede für sich einzeln betrachtet werden. Die Formeln aus den Kapiteln 5.1 und 5.2 werden auf jede der Pipelines separat angewandt. Da hier der Befehlsfluss Out-Of-Order ist, ist es nicht möglich, die Anzahl

der benötigten Taktzyklen vorher abzuschätzen. Auch die Verteilung der Befehle auf die einzelnen Pipelines wird meist erst zur Laufzeit bestimmt. Diese Werte können aber meist simulativ erfasst werden.
Auch können vermehrt Daten und Struktur-Hazards auftreten, wenn etwa von zwei Pipeline-Stufen aus verschiedenen Pipelines auf dieselbe Ressource zugegriffen wird. Dies wird meist durch einen Pipeline-Stall in einer der beiden betroffenen Pipelines gelöst.

6 Umsetzung

Um die theoretischen Modelle aus den Kapiteln 4 und 5 in die Praxis umzusetzen, benötigt man eine Datenbank mit Energiewerten für jede Instruktion, unterteilt nach Pipeline-Stufen. Außerdem werden genaue Informationen über die ausgeführten Instruktionen in Form eines Befehls-Traces benötigt.
Dazu wird in Kapitel 6.1 ein Überblick über den Abschätzungsablauf gegeben, in Kapitel 6.2 die Charakterisierung der Instruktionen und die Erstellung der Datenbank beschrieben und in Kapitel 6.3.1 die Anforderungen und die Erstellung des benötigten Befehls-Traces aufgezeigt.
In Kapitel 6.4 wird die Validierung der Abschätzungsergebnisse behandelt, bei der diese mit Low-Level Abschätzungen taktweise verglichen werden.

6.1 Abschätzungsablauf

Den Kern des hier vorgestellten Abschätzungs-Frameworks bildet das nach der in dieser Arbeit entwickelten Methode benannte ACCPWR Programm, welches taktgenaue Abschätzungen und Abschätzungen mit Einbeziehung von Pipeline-Hazards liefert. Der normale Ablauf der Verlustleistungsabschätzung aus Kapitel 2.2.4 muss für die genauere Abschätzung mit Unterteilung nach Pipeline-Stufen etwas modifiziert werden (Siehe Abb. 6.1).

Wie bei der instruktionsbasierten Verlustleistungsabschätzung wird ein Assembler Programm oder ein in Assembler übersetztes Hochsprachen-Programm mit einem Debugger simuliert und durch diesen ein Befehls-Trace erzeugt.
Bei diesem veränderten Ablauf wird der Debugger mit einem Script gesteuert, um erweiterte Informationen zu erhalten, die nicht im normalen Befehls-Trace auftauchen. Diese werden benötigt, um Pipeline-Stalls und die Befehle, welche bei einem Pipeline-Flush verworfen werden, herauszufinden. Auf dies wird näher in Kapitel 6.3.1 eingegangen.

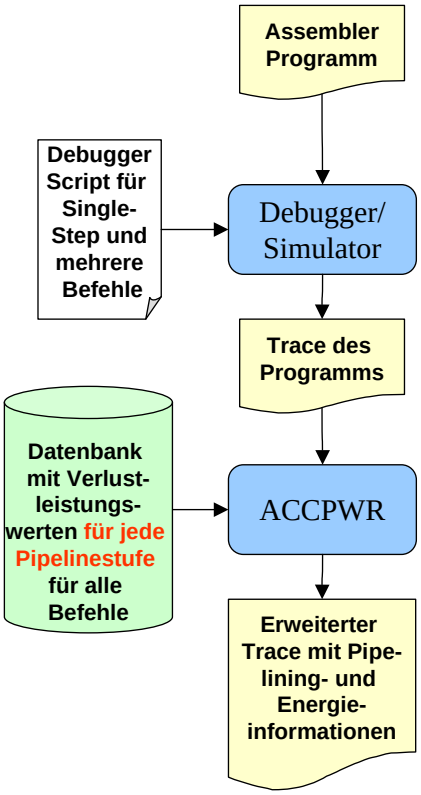

Abbildung 6.1: Flussdiagramm für die Durchführung der erweiterten Verlustleistungsabschätzung nach dem ACCPWR Verfahren

Dieser erweiterte Befehls-Trace wird dann von dem Verlustleistungsabschätzungsprogramm ACCPWR ausgewertet. Die für die Berechnung benötigten Energiewerte werden aus einer Datenbank geholt und die gesamte und taktweise Verlustleistung berechnet.
Die größte Veränderung betrifft aber die Datenbank, welche nicht mehr nur einen Energiewert für jeden Befehl enthält, sondern in der dieser Energiewert aufgeschlüsselt ist in die Energiewerte der Pipeline-Stufen. Somit wird jeder Pipeline-Stufe des Prozessors ein Energiewert für jede Instruktion zugewiesen. Wie die Charakterisierung dieser Energiewerte erfolgen kann und wie die Datenbank aufgebaut ist, wird in Kapitel 6.2 beschrieben.

6.2 Erstellung der Datenbank

Die Ermittlung der Energiewerte für die einzelnen Befehle zum Aufbauen einer Datenbank kann entweder durch Messung oder Low-Level Abschätzung erfolgen und wird als Charakterisierung bezeichnet. Diese Energiewerte werden in einer Datenbank gespeichert, die in Kapitel 6.2.2 genauer beschrieben wird.

6.2.1 Charakterisierung

Die Charakterisierung muss für jeden Prozessor oder jede Prozessorkonfiguration einmal durchgeführt werden. Hierbei wird die Verlustleistung für jede Instruktion ermittelt und damit eine Datenbank aufgebaut, die für die Verlustleistungsabschätzung von Befehlsfolgen oder auch ganzer Programme herangezogen werden kann.
Für „Hard-Core“ oder IP-Core Prozessoren kann die erforderliche Charakterisierung mit einer Unterteilung nach Pipeline-Stufen nur durch den Hersteller erfolgen. Im Projekt LEMOS (Siehe auch Kapitel 2.2.1) wurden von uns Methoden für die Charakterisierung von Halbleiterbauelementen beim Hersteller entwickelt und nachgewiesen. Dazu wurde der LC2 Prozessor in Module aufgeteilt, und diese wurden einzeln charakterisiert. Diese Werte wurden dann in einem Framework verwendet, das sich EQN* Netze bedient, um die Verlustleistung des Prozessors auf einer hohen Abstraktionsebene abzuschätzen [23].
Da Soft-Core Prozessoren aber selbst konfiguriert und für den Einsatz-

zweck vom Anwender verändert werden – daher in vielen verschiedenen Konfigurationen zum Einsatz kommen – muss hier die Charakterisierung vom Anwender selbst vorgenommen werden.
In den nächsten Kapiteln wird zunächst das generelle Verfahren zur Charakterisierung eines Prozessors beschrieben und dann die Erweiterungen, die notwendig sind, um eine Datenbank mit Energiewerten für die einzelnen Pipeline-Stufen aufzubauen.

6.2.1.1 Generelles Verfahren

Die Verlustleistungswerte werden hierbei durch eine Low-Level Simulation bestimmt oder direkt an einem Prototypen gemessen. In beiden Fällen muss der in einer Hardware-Beschreibungssprache, wie z.B. VHDL, vorliegende Prozessor synthetisiert und beim „Mapping“ auf die Zieltechnologie übertragen, sowie eine fertig platzierte und verdrahtete Netzliste (NCD-Datei) erstellt werden. Diese kann für Messungen auf ein FPGA übertragen werden, oder es kann für die Low-Level Simulation aus der NCD-Datei ein VHDL Simulationsmodell generiert werden.
Für jeden Befehl muss die durchschnittliche Energie ermittelt werden. Hierfür wird der Befehl mehrmals und mit unterschiedlichen Parametern ausgeführt, die jeweiligen Verlustleistungswerte ermittelt und über Formel (2.16) der Mittelwert gebildet. Für die Parameter der zu untersuchenden Instruktion gilt, dass sich durchschnittlich viele Bits in dem Instruktionswort ändern. Genauso müssen die zu verarbeitenden Datenwörter in den Registern so beschaffen sein, dass in etwa die Hälfte der durch den Befehl änderbaren Bits der Datenwörter sich von Befehl zu Befehl ändern.
Die Assembler Datei mit der zu untersuchenden Instruktion, die mehrmals hintereinander unterschiedlich parametriert ausgeführt wird, wird nun durch einen Compiler in eine binäre auf dem Zielprozessor ausführbare Datei (ELF- oder HEX-Datei) umgewandelt. Hierbei ist zu beachten, dass durch diese Umwandlung bei manchen Prozessoren vor dem eigentlichen Programm ein boot-up Code eingeschoben wird. Da dieser die Messung verändern könnte, muss die Messung der Verlustleistung deshalb entsprechend verzögert gestartet werden.
Bei der Low-Level-Simulation muss diese ausführbare Datei in eine Hardware-Beschreibungssprache wie VHDL umgewandelt werden, die den Speicher mit dem darin stehenden Programm simuliert.
Diese wird zusammen mit dem „Post Place and Route“ Simulationsmodell des Prozessors mit einem Low-Level Simulationsprogramm wie ModelSim

simuliert. Das Simulationsprogramm protokolliert bei der Simulation alle Signaländerungen an allen Knoten des Systems und speichert sie in eine „Value Change Dump“ Datei (VCD-Datei). Anhand der „Post Place and Route“ Netzliste und der „Value Change Dump“ Datei kann ein Low-Level Verlustleistungsanalyseprogramm die Verlustleistung des zu untersuchenden Prozessors während der Ausführung des betreffenden Programmabschnitts abschätzen.

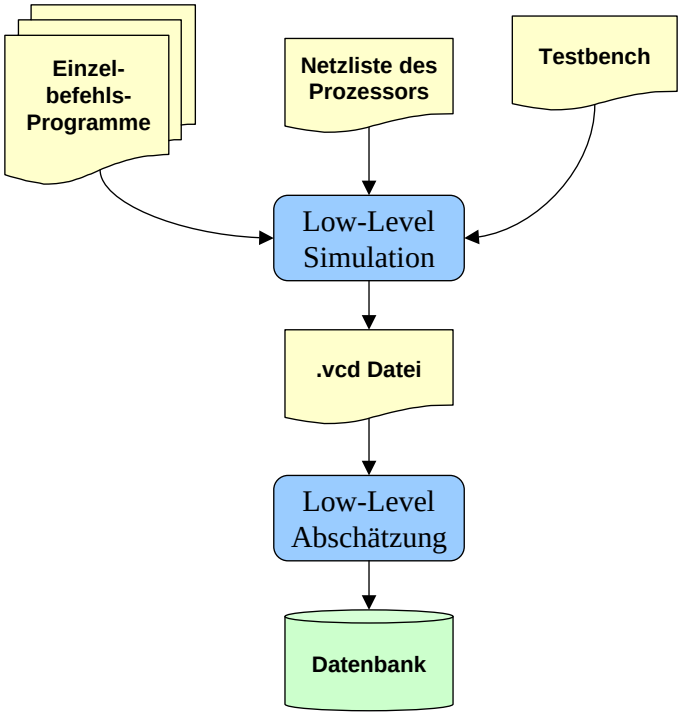

Abbildung 6.2: Statt der Netzliste für den gesamten Prozessor werden die Netzlisten der Pipeline-Stufen für jede Instruktion charakterisiert und diese Werte in der Datenbank gespeichert

Messung

Dieses Verfahren kann auch durch Messungen an einem Prototypen des zu untersuchenden Prozessors durchgeführt werden. Hierbei muss für jeden Befehl, bzw. für jede Gruppe ähnlicher Befehle, eine Assembler Datei geschrieben werden, in der derselbe Befehl mehrmals hintereinander ausgeführt und erst danach durch einen unbedingten Sprung zurück an

den Anfang verzweigt wird, wodurch eine Endlosschleife gebildet wird. Es wird empfohlen hier den Befehl mehr als 2000 mal auszuführen, bevor der Sprung erfolgt. Durch das Verhältnis von 2000 zu 1 ist gewährleistet, dass der Sprungbefehl die Messung der Verlustleistung der eigentlichen Instruktion nicht verfälscht. Auch hierbei muss die Instruktion mit verschiedenen Parametern ausgeführt werden, um einen typischen Wert für die Verlustleistung zu bekommen.
Bei der Messung wird in den Versorgungsstromkreis des Prozessors ein möglichst kleiner, hochgenauer Messwiderstand eingefügt. Bei den Versuchen für das LEMOS Projekt wurden gute Ergebnisse mit $0,2\Omega$ bis $0,5\Omega$ Widerständen erzielt. Aus der an diesem Widerstand R_{mess} abfallenden Spannung U_{mess} kann über Formel (6.1) der in den Prozessor fließende Strom I_{in} berechnet werden (Siehe auch Abb. 6.3).

$$I_{in} = \frac{U_{mess}}{R_{mess}} \tag{6.1}$$

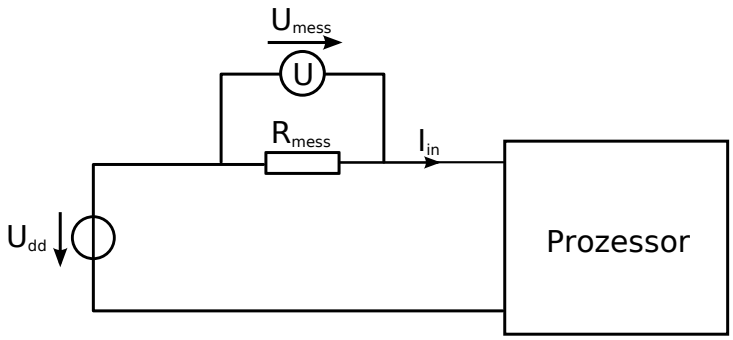

Abbildung 6.3: Schaltung zur Messung des durch den Prozessor fließenden Stroms über die Spannung an dem Messwiderstand R_{mess}

Da die zu messende Spannung U_{mess} sehr klein ist, muss ein entsprechend hochauflösendes Messgerät verwendet werden. Ein Vorteil ist jedoch, dass durch die Endlosschleife keine zeitlich genaue Messung erfolgen muss, sondern die durchschnittliche Spannung über den Messwiderstand als Messwert ausreicht. Diese durchschnittliche Spannung kann somit durch eine über das gesamte Zeitintervall für den Programmablauf integrierende Messung ermittelt werden.
Die Leistungsaufnahme P_{in} des Prozessors kann nun mit I_{in} und U_{dd} anhand von Formel (2.4) berechnet werden. Der Spannungsabfall U_{mess} an

dem Messwiderstand kann in diesem Falle vernachlässigt werden, da davon ausgegangen werden kann, dass U_{mess} sehr viel kleiner als U_{dd} ist.
Um die Differenz zwischen der gemessenen Leistungsaufnahme und der eigentlichen Verlustleistung gering zu halten, sollte darauf geachtet werden, dass der Prozessor während der Messung keine Ausgangssignale treiben muss. Manche Prozessoren und FPGAs, die mehrere Stromversorgungsanschlüsse aufweisen, bieten die Möglichkeit, den tatsächlichen Stromverbrauch des Prozessorkerns, bzw. des FPGA Cores direkt zu bestimmen, wenn die Versorgungsleitungen nach Hardware-Komponenten aufgegliedert herausgeführt sind. Da hier der Leistungsverbrauch der Ausgangsleitungstreiber abgekoppelt ist und über andere Leitungen versorgt wird, kann so die Messung durchgeführt werden, ohne dass z.B. Peripherie oder PAD-Logik das Ergebnis beeinflussen.

6.2.1.2 Erweiterte Charakterisierung für Energiewerte der einzelnen Pipeline-Stufen

Für die Verlustleistungsabschätzung mit der Unterteilung des Prozessors wie in Kapitel 4.1 beschrieben, muss eine Datenbank mit Energiewerten aufgebaut werden, in der jede Pipeline-Stufe für jede Instruktion charakterisiert ist. Das bedeutet, dass für jede Pipeline-Stufe einzeln die Verlustleistung gemessen werden muss und zwar für jede Instruktion (Siehe auch Abb. 6.1).

Für die Charakterisierung muss der VHDL Quellcode des Prozessors nach den Pipeline-Stufen modularisiert werden. Es muss daher, soweit noch nicht für jede Pipeline-Stufe ein Modul (in VHDL „Entity“ genannt) mit der zu dieser Stufe gehörenden Logik existiert, die Logik der Pipeline-Stufen in separate Module gekapselt, und diese Module in dem Hauptmodul des Prozessors eingebunden werden.
Dies geschieht, indem man alle Komponenten, Verbindungsleitungen und die auf die Pipeline-Stufe folgende Registerstufe im ursprünglichen Quellcode des Prozessors identifiziert, herauslöst und in ein eigenes Modul auslagert, welches an dessen Stelle im ursprünglichen Code als Modul eingebunden wird.
Bei der Synthese wird dieses modularisierte Design wieder zu einer einzigen Netzliste umgewandelt. Diese Modularisierung ändert somit weder die Funktion des Prozessors, noch das Timing- oder Energieverbrauchsverhalten. Von Vorteil ist jedoch, dass durch diese Modularisierung Kompo-

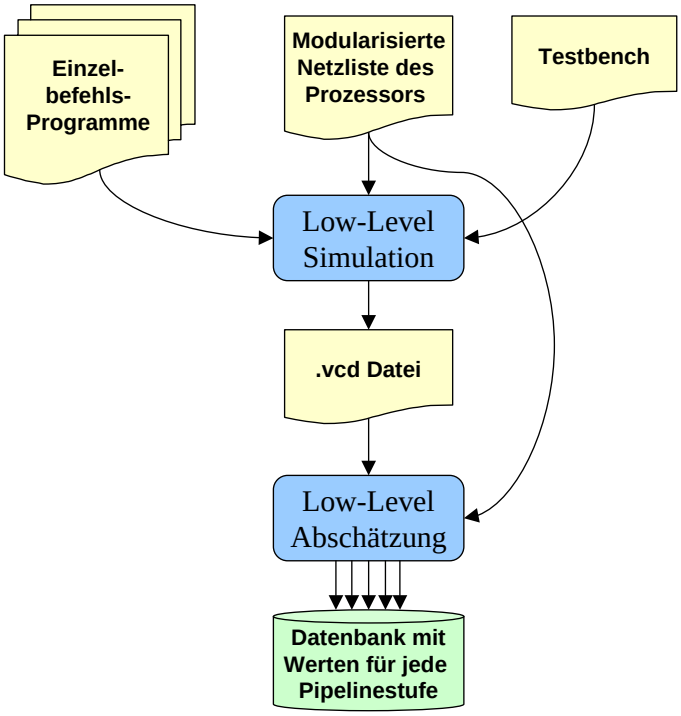

Abbildung 6.4: Die Netzliste des nach Pipeline-Stufen modularisierten Prozessors wird für jede Instruktion charakterisiert und die Werte aus der Low-Level Abschätzung für jede Pipeline-Stufe in der Datenbank gespeichert

nenten der einzelnen Pipeline-Stufen im fertig platzierten und verdrahteten Design leichter identifiziert und zugeordnet werden können. Bei vielen Low-Level Abschätzungsprogrammen, wie z.B. bei dem in dieser Arbeit verwendeten Xilinx XPower, geschieht diese Zuordnung dann sogar automatisiert und die Verlustleistung der einzelnen Module kann direkt ermittelt werden.

Messung

Die Low-Level Simulation ist hierbei die am besten geeignete Methode, um diese Abschätzungen durchzuführen, da sich bei diesem Verfahren die abzuschätzenden Komponenten am leichtesten einzeln bestimmen lassen. Außerdem kann die Verlustleistung der einzelnen Stufen auch durch echte Messungen ermittelt werden.
Für Soft-Cores kann dies über Messungen an einer prototypischen Realisierung auf einem FPGA durchgeführt werden, welcher allerdings nur mit der abzuschätzenden Komponente konfiguriert sein darf. Die Testumgebung muss hierfür die Eingangssignale der jeweiligen Pipeline-Stufe als Stimuli von außerhalb des FPGA bereitstellen, entweder durch ein zweites FPGA oder über einen externen Patterngenerator.
Messungen bei „Hard-Core“ Prozessoren sind hierbei nicht direkt möglich, da der Aufwand, nur eine einzelne Pipeline-Stufe zu produzieren oder Messleitungen für die Stromversorgungen der einzelnen Pipeline-Stufen aus dem Chip herauszuführen, zu groß ist.
Stattdessen können Low-Level Abschätzungen auf Logik- oder Schaltkreis-Ebene vom Hersteller durchgeführt werden, und diese Werte im Datenblatt des Prozessors veröffentlicht werden, ohne die genaue Struktur der Pipeline-Stufen preisgeben zu müssen.

Alternative Messmethode

Eine weitere Möglichkeit, durch Messungen an Prozessoren die Verlustleistung der einzelnen Pipeline-Stufen zu ermitteln, besteht darin, nur einen einzigen Befehl durch die Pipeline laufen zu lassen. Durch eine spezielle Befehlsfolge vor und nach der abzuschätzenden Instruktion kann im laufenden Programm eine Situation herbeigeführt werden, in der nur der abzuschätzende Befehl in der interessierenden Pipeline-Stufe ausgeführt wird und so stufenweise und präzise gemessen werden kann.

Diese Situation muss dabei folgende Eigenschaften haben: Durch einen Flush muss die Pipeline geleert werden. Eventuell können sich aber, wenn der Sprung sehr früh in der Pipeline evaluiert wird, noch Befehle, die vor dem Sprung ausgeführt wurden, in den letzten Stufen der Pipeline befinden. Mit einem zweiten provozierten Pipeline-Flush kann die Pipeline aber vollständig geleert werden.
Voraussetzung ist hierbei allerdings, dass bei einem Pipeline-Flush der Prozessor die Instruktionen auch wirklich abbricht und nicht nur die Ergebnisse verwirft (Siehe auch Kap. 5.2.3). Ersatzweise kann die Pipeline auch alternativ durch vor dem Befehl ausgeführte *NOP*-Instruktionen geleert werden.
Weiterhin muss verhindert werden, dass nach der abzuschätzenden Instruktion weitere Befehle in die Pipeline geladen werden. Dies kann entweder durch auf die Instruktion folgende *NOP*-Instruktionen oder provoziertes Bubbling in der Pipeline geschehen.
Die Messung muss bei dieser Methode – im Gegensatz zur normalen Messung, die keine zeitliche Genauigkeit aufweisen muss – taktgenau erfolgen. Bei hohen Frequenzen wird es jedoch schwierig, genaue Messungen durchzuführen, da bei den meisten Messgeräten mit kürzerer Messzeit die Messgenauigkeit proportional vermindert wird.

6.2.2 Aufbau der Datenbank

Aus den Messungen oder Low-Level Abschätzungen aus Kapitel 6.2.1.2 erhält man bei jeder Instruktion für jede Pipeline-Stufe einen mittleren Strom I_{in} pro Takt, der zusammen mit der festen Versorgungsspannung U_{dd} nach Formel (2.4) die mittlere Leistung P ergibt. Um die einzelnen Verlustleistungswerte unabhängig von der Taktfrequenz des Prozessors zu speichern, werden diese durch die Frequenz geteilt beziehungsweise mit der Periode T der Taktfrequenz multipliziert, und man erhält so nach Formel (2.5) Energiewerte für die einzelnen Instruktionen und Pipeline-Stufen.
Die Datenbank für ACCPWR ist dabei eine Tabelle, die als CSV Textdatei (Comma Separated Values) gespeichert wird. Ihr Aufbau ist in Tabelle 6.1 dargestellt.

Eine komplette Tabelle aller charakterisierten Instruktionen für den Jam CPU Soft-Core ist in Anhang A.2 aufgeführt.
Die Werte sind hier in Femtojoule (fJ)gespeichert um Nachkommastellen zu vermeiden und trotzdem mit sehr genauen Werten rechnen zu können.

Instruktion	ASMCODE	E_stat(fJ)	E_bu(fJ)	E_if(fJ)	E_id(fJ)	E_ex(fJ)	E_mem(fJ)	E_wb(fJ)
add	000000	175273	160000	102703	261266	411312	35570	4509
addd	000011	175273	160000	123019	203815	377918	31537	4431
⋮								

Tabelle 6.1: In der Datenbank werden die jeweilige Instruktion mit dem identifizierenden Teil des binären Mikrobefehlswortes (ASMCODE), den Werten für die statische Energie und die befehlsunabhängige Energie zusammen mit den Energiewerten der einzelnen Pipeline-Stufen aufgelistet

6.3 Abschätzungsverfahren

Zur Berechnung der Verlustleistung über das instruktionsbasierte Verfahren muss bekannt sein, welche Befehle vom Prozessor ausgeführt werden sollen. Dies muss simulativ am Rechner über einen Softwaresimulator oder Debugger erfolgen, der einen Befehls-Trace des Programms ausgibt.
Für die taktgenaue Berechnung des Energieverbrauchs reicht der normale Befehls-Trace nicht aus, da der genaue Fluss der Befehle durch die Pipeline nachgebildet werden muss.

6.3.1 Der Befehls-Trace

Der Befehls-Trace ist die Liste aller Assemblerbefehle, die während der Abarbeitung des abzuschätzenden Programms auf dem Prozessor ausgeführt werden. Er kann durch ein „Mitschreiben“ der einzelnen Befehle beim manuellen „durch-steppen“ im Debugger in eine Log-Datei aufgezeichnet werden. Manche Simulatoren bieten auch die Möglichkeit, den Befehls-Trace automatisch zu erstellen.
Er unterscheidet sich von dem Quellcode des ausgeführten Programms in der Hinsicht, dass er nur die wirklich ausgeführten Befehle enthält und diese in der richtigen Reihenfolge der Ausführung ausgibt. Außerdem wird durch die Simulation bei Schleifen die Anzahl der Durchläufe ermittelt und daher die Häufigkeit, mit der ein Befehl in einer Schleife aufgerufen wird, protokolliert. Ein Ausschnitt aus einer solchen Log-Datei des GNU Debuggers GDB, der hier den Xilinx MicroBlaze Debugger (XMD) steuert, ist im Listing 6.1 dargestellt.

```
0x00000110 in ?? ()
Dump of assembler code from 0x110 to 0x114:
0x00000110:     lwi     r15, r1, 0
End of assembler dump.
0x00000114 in ?? ()
Dump of assembler code from 0x114 to 0x118:
0x00000114:     rtsd    r15, 8
End of assembler dump.
0x00000490 in ?? ()
Dump of assembler code from 0x490 to 0x494:
0x00000490:     brlid   r15, -80        // 0x440
End of assembler dump.
0x00000440 in ?? ()
Dump of assembler code from 0x440 to 0x444:
0x00000440:     lwi     r3, r0, 1216    // 0x4c0
End of assembler dump.
```

Listing 6.1: Ausgabe des Debuggers GDB

Bereinigt von zusätzlichen Ausgaben des Debuggers sieht der Befehls-Trace dann folgendermaßen aus:

```
0x00000110:     lwi     r15, r1, 0
0x00000114:     rtsd    r15, 8
0x00000490:     brlid   r15, -80        // 0x440
0x00000440:     lwi     r3, r0, 1216    // 0x4c0
```

Listing 6.2: Standard Befehls-Trace

Dieser Trace enthält nur die vollständig ausgeführten Befehle. Instruktionen, die fälschlich nach einer nicht korrekten Sprungvorhersage begonnen, aber dann nach einer bestimmten Pipeline-Stufe abgebrochen wurden, werden hier nicht ausgegeben. Weiterhin fehlen bei vielen Simulatoren, wie z.B. dem Xilinx MicroBlaze Debugger, die Befehle, die in Delay Slots ausgeführt werden. Alle diese hier weggelassenen Befehle beanspruchen jedoch die Leistung verschiedener Pipeline-Stufen und müssen daher bei der Berechnung der Verlustleistung berücksichtigt werden.
Eine weitere fehlende Eigenschaft des normalen Befehls-Traces ist, dass man nicht erkennen kann, wann ein Pipeline-Stall auftritt. Diese Informationen werden aber für die Berechnung der statischen Verlustleistung und

für die taktgenaue Abschätzung benötigt.
Diese Eigenschaften müssen für die genaue Nachbildung des Befehlsflusses durch die Pipeline-Stufen des Prozessors entweder nachträglich aus dem Befehls-Trace oder direkt bei der Simulation ermittelt werden.

6.3.2 Taktgenaue Berechnung des Energieverbrauchs

Anhand der Informationen über den exakt nachgebildeten Befehlsfluss, durch den genau festgestellt werden kann, welcher Befehl zu einem bestimmten Zeitpunkt in den jeweiligen Pipeline-Stufen des Prozessors bearbeitet wird, und anhand der Charakterisierung aller Befehle mit Einzelenergiewerten für jede Pipeline-Stufe kann mit dem Verfahren aus Kapitel 4 der Energieverbrauch für jeden Takt und somit auch für die gesamte Befehlsfolge exakt berechnet werden.

6.4 Validierung

Um das Abschätzungsergebnis des hier vorgestellten Ansatzes validieren und die Genauigkeit der Abschätzung überprüfen zu können, werden die Schätzwerte von dem ACCPWR Programm mit einer Low-Level Abschätzung verglichen. Hierbei wird das gesamte Programm oder die interessierende Befehlsfolge mit einem Low-Level Simulator simuliert, die Signaländerungen in eine VCD-Datei protokolliert und diese mit einem Low-Level Abschätzungsprogramm ausgewertet.
Für einen taktweisen Vergleich der Abschätzungen muss, statt einer VCD-Datei für den gesamten Befehlsablauf, für jeden Takt eine eigene VCD-Datei erstellt werden (Siehe auch Abb. 6.5) und für diese wiederum einzeln eine Low-Level Abschätzung durchgeführt werden.

Hierzu wird das DO-Script des hier benutzten Low-Level Simulators ModelSim so modifiziert, dass für jeden Takt die Signaländerungen in eine neue VCD-Datei geschrieben werden. Für diese VCD-Dateien werden dann mit dem herstellereigenen Verlustleistungsabschätzungsprogramm *XPower* die jeweiligen Abschätzungen für die einzelnen Takte erstellt.
Diese taktgenauen Low-Level Abschätzungen werden nun taktweise mit den Ergebnissen aus der Verlustleistungsabschätzung derselben Befehlsfolge mit ACCPWR verglichen.

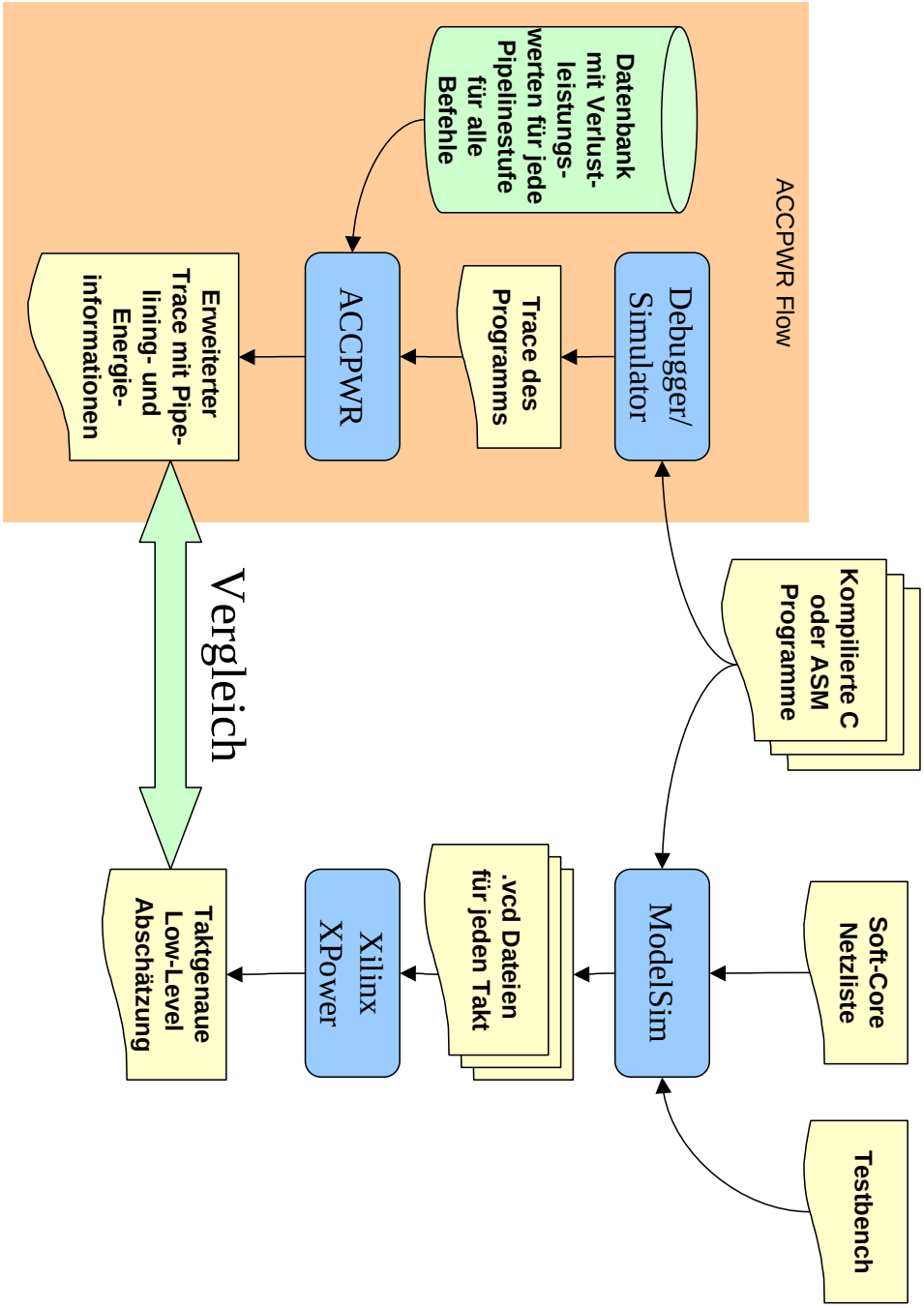

Abbildung 6.5: Die Ergebnisse der Berechnung mit ACCPWR werden mit den Low-Level Abschätzungen taktweise verglichen

7 Fallbeispiele

Die praktische Umsetzung erfordert weitere Anpassungen. Diese werden anhand von den folgenden drei Fallbeispielen genauer erläutert.
Fallbeispiel 1 zeigt an dem strukturell einfachen AVR-Core den Einfluss von Branch Instruktionen und speziell von Pipeline-Flushes auf den Energieverbrauch eines Prozessors.
Die genaue Charakterisierung eines nach Pipeline-Stufen modularisierten Prozessors und die stufenweise bzw. taktgenaue Abschätzung nach Kapitel 4 wird in Fallbeispiel 2 an dem Jam CPU Soft-Core gezeigt.
Für die Abschätzung eines IP-Cores, dessen innerer Aufbau nicht bekannt ist, wird in Fallbeispiel 3 anhand des MicroBlaze Prozessors eine Methode gezeigt, die trotzdem eine stufenweise Abschätzung erlaubt.
Die in den vorhergehenden Kapiteln beschriebenen Methoden können auf beliebige Prozessoren angewendet werden. Für die Fallbeispiele wurden jedoch Soft-Core Prozessoren verwendet, da für diese die Hardware-Beschreibungen frei verfügbar sind, und deshalb die genaue Untersuchung und Messung der einzelnen Prozessormodule oder Pipeline-Stufen möglich ist.

7.1 Fallbeispiel 1: Der Einfluss von Pipeline-Flushes auf die Verlustleistung

Dieses Fallbeispiel wurde durchgeführt, um den Effekt der Pipeline-Flushes auf die Verlustleistung zu untersuchen. Es wurden dabei Abschätzungen nach dem Standardverfahren der instruktionsbasierten Verlustleistungsabschätzung, ohne spezielle Behandlung von falschen Sprungvorhersagen, mit einem Abschätzverfahren verglichen, bei dem nach Sprüngen und bedingten Instruktionen berücksichtigt wurde, ob die Sprungvorhersage richtig oder falsch war.
Für dieses Fallbeispiel wurde der recht einfache, quelloffene Soft-Core „AVR-Core“ aus dem OpenCores Project verwendet, der im nächsten Kapitel kurz vorgestellt wird. Für die folgenden Untersuchungen wurde dieser Soft-Core synthetisiert und für einen Spartan-3 FPGA implementiert. Die Low-Level Simulation wurde auf einem daraus generierten Simulationsmodell mit dem Simulationsprogramm ModelSim bei einem Prozessortakt von 8 Megahertz durchgeführt.

7.1.1 Der „AVR-Core“ Soft-Core

Der AVR-Core ist ein freier Soft-Core, geschrieben von Ruslan Lepetenok in VHDL [21]. In der Funktion und Programmierung ist er dem ATmega103 von Atmel nachgebildet. Der AVR-Core ist ein 8-Bit RISC Prozessor mit 32 frei verwendbaren 8-Bit Registern und 23 Interrupt-Leitungen. Wie die Prozessoren aus Atmels AVR-Reihe ist er als Prozessor mit Harvard Architektur ausgelegt und kann somit bis zu 128 Kb Programmspeicher und bis zu 64 Kb Datenspeicher verwalten. Der AVR-Core besitzt eine einfache Pipeline mit zwei Stufen:

- Instruction Fetch/Decode
- Execute/Write Back

Als Sprungvorhersage bei bedingten Sprüngen verwendet er eine statische Vorhersage, die immer „Not-Taken“ vorhersagt, also spekulativ die nächsten im Programmspeicher stehenden Befehle ausführt, als ob nicht gesprungen würde. Die Evaluierung des Sprungbefehls ist am Ende der „Execute/Write Back“-Phase abgeschlossen. Somit wurde schon ein falscher Befehl geladen und dekodiert. Das bedeutet, dass der Prozessor bei einer

falschen Vorhersage für einen Pipeline-Flush einen zusätzlichen Takt benötigt.
Der Vorteil von diesem an einen realen Prozessor angelehnten Soft-Core ist, dass der Assembler Code sehr einfach und weit verbreitet ist, und es einen C-Compiler für diesen gibt.

7.1.2 Methodik

Für den Vergleich der Abschätzungsmethoden bezüglich des Einflusses von Pipeline-Flushes auf die Verlustleistung wurde eine leicht abgewandelte Methode verwendet. Statt einer Datenbank mit Energiewerten für jede Pipeline-Stufe wird die Standardmethode etwas erweitert, um Pipeline-Flushes zu erfassen. Hierfür wird zu der normalen Datenbank mit den Energiewerten für alle Instruktionen eine zusätzliche Datenbank für bedingte Sprungbefehle eingeführt. In dieser sind für jede bedingte Instruktion zwei unterschiedliche Verlustleistungswerte gespeichert. Diese entsprechen dem Energieverbrauch der Instruktion bei einer Ausführung mit korrekter Sprungvorhersage und dem Energieverbrauch bei einer falschen Sprungvorhersage, bei dem es zu einem Pipeline-Flush kommt. Die Energieaufnahme der fälschlich teilweise ausgeführten Instruktionen des Pipeline-Flushs nach dem Sprungbefehl wird dabei pauschal zu dem Wert für die Abarbeitung des Sprungbefehls mit falscher Sprungvorhersage hinzugerechnet.
Auch die längere Ausführungsdauer wird bei den Sprungbefehlen mit Pipeline-Flush genau wie bei Multi-Cycle Befehlen berücksichtigt, und die Anzahl der für die Ausführung benötigten Takte in der Datenbank zu den jeweiligen Instruktionen gespeichert. So ergeben sich für bedingte Instruktionen jeweils zwei Einträge in der Datenbank mit jeweils dem entsprechenden Energiewert und der Anzahl der benötigten Taktzyklen.

7.1.2.1 Charakterisierung mit Ausgleichsfaktor für Befehlswort-Bitänderungen

Die Charakterisierung wird nach dem generellen Charakterisierungsverfahren, wie in Kapitel 6.2.1.1 beschrieben, mittels Low-Level Simulation durch das Simulationsprogramm ModelSim und das Low-Level Abschätz-Tool XPower des FPGA Herstellers Xilinx durchgeführt. Jedoch wurden

für bedingte Sprungbefehle jeweils zwei Energiewerte charakterisiert. Einmal wurde die Verlustleistung für den Fall ermittelt, dass die Sprungvorhersage korrekt war, und einmal dafür, dass sie nicht korrekt war und die Pipeline mit einem Flush geleert werden musste.
Bei der Charakterisierung wurden auch Untersuchungen angestellt, inwiefern bestimmte Befehlskombinationen die Verlustleistung beeinflussen, und es wurde dabei eine Befehlsbitabhängigkeit festgestellt. Das bedeutet, dass der Energieverbrauch einer Befehlskombination nicht nur von dem Verbrauch der Ausführung dieser Befehle abhängt, sondern auch von der Veränderung der einzelnen Bits zweier aufeinander folgender Befehlswörter. Dieser Effekt wird hauptsächlich durch das Dekodieren des Befehls hervorgerufen und das Umladen der Register, in denen diese Befehlswörter gespeichert werden.
Daher wird hier ein Ausgleichsfaktor $AF(i)$ auf Basis der Hamming Distanz[1] $HD(i)$ einer Instruktion i und der darauf folgenden Instruktion eingeführt, der sich wie folgt errechnet:

$$AF(i) = (HD_{avg} - HD\,(i)) \cdot P_{sbc}, \tag{7.1}$$

wobei HD_{avg} die durchschnittliche Hamming Distanz aller möglichen Instruktionswortkombinationen ist. Bei dem hier verwendeten Prozessor, bei dem ein Instruktionswort eine Breite von 16 Bit aufweist, wird die durchschnittliche Hamming Distanz auf die Hälfte der Instruktionswortbreite und somit auf 8 festgelegt.
P_{sbc} (Single Bit Change) ist die Verlustleistung, die verbraucht wird, wenn sich genau ein Bit im Instruktionswort ändert. Dieser Wert wurde durch Low-Level Abschätzungen für den AVR-Core ermittelt und beträgt hier $0.22mW$.
Die Ergebnisse der Charakterisierung sind im Anhang A.1 in den Energiewerttabellen für unbedingte und bedingte Instruktionen gegeben.

7.1.2.2 Die Verlustleistungsabschätzung

Da hier nur Abschätzungen zu Vergleichszwecken vorgenommen werden und keine taktgenaue Analyse durchgeführt wird, wurde auch das Erstellen des Befehlspfades vereinfacht und durch ein simples Profiling ersetzt,

[1] Die Hamming Distanz entspricht der Anzahl der Bitstellen zweier binären Wörter, die sich voneinander unterscheiden

bei dem alle ausgeführten Instruktionen tabellarisch mit der Anzahl der jeweiligen Befehlsaufrufe und somit nicht in ihrer zeitlichen Reihenfolge ausgegeben werden. Dies wird durch das speziell für AVR Prozessoren entwickelte AVRORA-Analyseprogramm vorgenommen, welches ein sehr schnelles und genaues Profiling ermöglicht.
Dieses Programm ermittelt die während einer Befehlsfolge ausgeführten Befehle, deren Aufruf-Häufigkeit und die für diesen Befehl aufgewendeten Taktzyklen. Da hier bei den Sprungbefehlen die Information fehlt, ob der Sprung erfolgreich ausgeführt wurde oder mit einem darauf folgenden Pipeline-Flush abgebrochen wurde, muss dies aus den gegebenen Daten rekonstruiert werden.
Hierfür wurde ein C-Programm geschrieben, welches neben den Pipeline-Flush Berechnungen auch die komplette Energieverbrauchsabschätzung durchführt. Dieses Programm unterteilt die Verlustleistung noch nicht in ihre jeweiligen Stufen, sondern berücksichtigt Pipeline-Flushes nur mit einem abgeschätzten Mehrverbrauch an Energie. Daher wird diese Methode im folgenden als „einfaches“ ACCPWR Verfahren bezeichnet.
Zur Berechnung der Pipeline-Flushes stehen für jeden Befehl die Häufigkeit der Aufrufe $A_{I,ges}$ und die Anzahl der bei diesen Aufrufen insgesamt benötigten Taktzyklen $Z_{I,ges}$ zur Verfügung, die von dem Profiling-Programm ausgegeben werden. Da die Taktzyklen eines Pipeline-Flush hier einfach zum vorhergehenden Sprungbefehl hinzugerechnet werden, können folgende Formeln aufgestellt werden:

$$A_{BR,ges} = A_{BR,nf} + A_{BR,fl} \tag{7.2}$$

$$Z_{BR,ges} = A_{BR,nf} \cdot z_{BR,nf} + A_{BR,fl} \cdot z_{BR,fl} \tag{7.3}$$

Hierbei stellt $A_{BR,nf}$ die Anzahl der Aufrufe des Sprungbefehls BR dar, die richtig vorhergesagt wurden und daher keinen Pipeline-Flush verursachen. Weiterhin sind $Z_{BR,ges}$ die Anzahl der Taktzyklen die alle aufgerufenen BR-Befehle verbraucht haben und $A_{BR,ges}$ die Anzahl aller Aufrufe dieses Befehls in der untersuchten Befehlsfolge. $z_{BR,nf}$ und $z_{BR,fl}$ sind die Anzahl der Taktzyklen die ein einziger Aufruf des Befehls BR benötigt, wenn er ohne Pipeline-Flush abläuft, bzw. wenn er mit Pipeline-Flush ausgeführt wird.
Durch Einsetzen von $A_{BR,fl}$ und Auflösen nach $A_{BR,nf}$ lässt sich daraus die Häufigkeit der Sprünge mit richtiger Sprungvorhersage für den Sprungbefehl BR ermitteln:

$$A_{BR,nf} = \frac{Z_{BR,ges} - (A_{BR,ges} \cdot z_{BR,fl})}{z_{BR,nf} - z_{BR,fl}}, \tag{7.4}$$

Die Anzahl der Ausführungen des Befehls mit Pipeline-Flush $A_{BR,fl}$ errechnet sich dann folgendermaßen:

$$A_{BR,fl} = A_{BR,ges} - A_{BR,nf} \tag{7.5}$$

Für jeden Aufruf des Sprungbefehls, ohne und mit Pipeline-Flush, wird nun, wie auch für alle anderen Instruktionen, der jeweilige Energiewert für die einmalige Ausführung aus der Datenbank gelesen und zusammen mit den Energiewerten der restlichen Instruktionen zur gesamten verbrauchten Energie E_{ges} aufsummiert.
Es wurden fünf verschiedene Test-Programme mit verschiedenen Eingangsdaten untersucht (Siehe Tabelle 7.1).

Programm	**Beschreibung**
I/O	Ein Assembler Programm, das Daten auf die Ports des Prozessors ausgibt und Warteschleifen durchläuft
MUL	Ein Assembler Programm, welches eine Multiplikation zweier 8-Bit Zahlen ausführt
DIV	Ein Assembler Programm, welches eine 16-Bit Zahl durch eine 8-Bit Zahl dividiert
ACC	Ein Assembler Programm, bei dem Zahlen multipliziert und aufaddiert werden
GCD	Ein C-Programm, welches den größten gemeinsamen Teiler zweier Zahlen berechnet

Tabelle 7.1: Die Test-Programme zur Abschätzung der Verlustleistung und Vergleich mit der Low-Level Abschätzung

Um die Genauigkeit der Methode zu überprüfen, wurden die Ergebnisse für die Test-Programme mit Low-Level Abschätzungen, wie in Kapitel 6.4 beschrieben, verglichen. Außerdem wurden die Ergebnisse mit instruktionsbasierten Abschätzungen nach dem Standardverfahren verglichen, um den Einfluss der genaueren Analyse von bedingten Sprungbefehlen auf die Verlustleistungsabschätzung zu untersuchen.

7.1.3 Ergebnisse

Die instruktionsbasierte Verlustleistungsabschätzung hatte bei diesen Test-Programmen eine maximale Abweichung von 25% im Vergleich zu der Low-Level Abschätzung. Jedoch im Gesamtdurchschnitt aller Test-Programme lag die Abweichung bei 5.9%.

Test Programm	**Abweichung (ACCPWR)**	**Abweichung (Standard)**	**Differenz Absolutwerte**
I/O (1 Loop)	-6,26%	0,98%	-5,29%
I/O (2 Loops)	-4,11%	1,61%	-2,50%
I/O (6 Loops)	2,67%	5,68%	3,01%
I/O (7 Loops)	3,34%	6,03%	2,69%
I/O (8 Loops)	-0,76%	1,79%	1,03%
MUL (0xAA*0x55)	-3,35%	12,79%	9,43%
MUL (0x55*0xAA)	-4,51%	13,25%	8,74%
MUL (0x01*0xFF)	-9,10%	2,32%	-6,79%
MUL2 (0xAA*0x55)	-0,63%	15,12%	14,49%
DIV (0xAAAA/0x55)	1,67%	18,71%	17,04%
DIV (0xFFFF/0x0A)	1,90%	15,36%	13,46%
DIV (0x0001/0x55)	-13,20%	8,09%	-5,11%
DIV (0x0004/0x03)	-25,74%	-3,07%	-22,68%
DIV2 (0xAAAA/0x55)	-0,45%	18,35%	17,90%
ACC (1 Wert)	1,85%	15,19%	13,34%
ACC (5 Werte)	1,06%	12,12%	11,06%
ACC (10 Werte)	4,63%	14,77%	10,14%
GCD (1747, 488)	-21,29%	-25,44%	4,16%
Durchschnitt (Absolutwerte)	**5,92%**	**10,59%**	**4,67%**

Tabelle 7.2: Abweichungen der instruktionsbasierten Abschätzverfahren von der Low-Level Abschätzung in Prozent

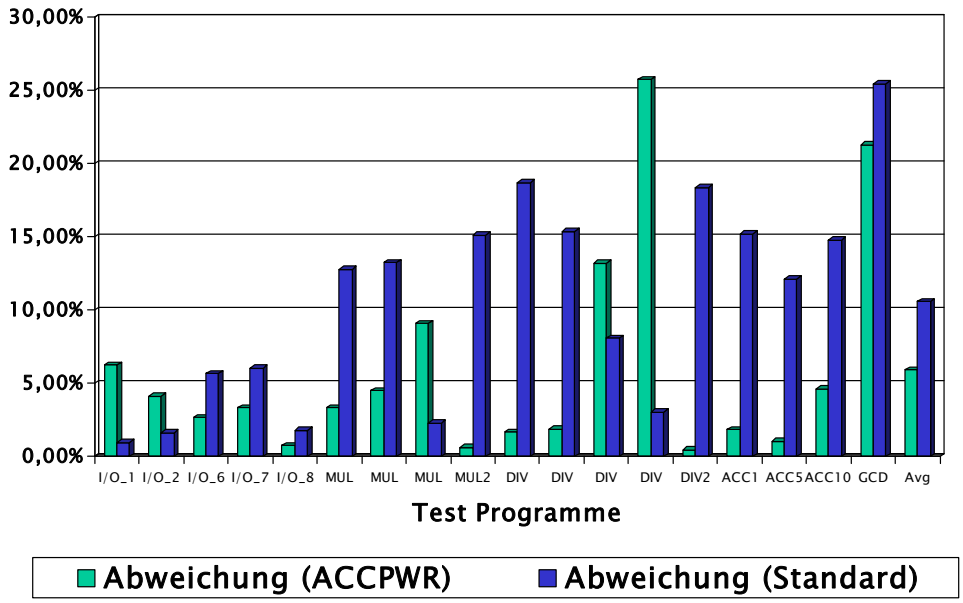

Abbildung 7.1: Die Beträge der prozentualen Abweichungen des einfachen ACCPWR Verfahrens im Vergleich mit denen des Standardabschätzverfahrens

7.1.3.1 Abhängigkeit der Verlustleistung von den Bitänderungen der Instruktionswörter

Bei dem Vergleich in Tabelle 7.2 des einfachen ACCPWR Abschätzprogramms mit dem Standardverfahren für instruktionsbasierte Verlustleistungsabschätzung wurde für beide Programme auf die gleiche Datenbank zugegriffen, für welche die Instruktionen nach dem oben beschriebenen Verfahren charakterisiert wurden. Der Einfluss der Befehlswort-Bitänderungen nach Formel (7.1) auf die Verlustleistung wurde dabei miteinbezogen.
Ohne die instruktionsspezifischen Ausgleichsfaktoren hätten beide Verfahren eine um etwa vier Prozentpunkte höhere Abweichung von der Low-Level Abschätzung. Diese Befehlsbitabhängigkeit kann durch diese Maßnahme hier reduziert werden, so dass sie gegenüber den Bitänderungen in den verarbeiteten Daten nicht mehr ins Gewicht fällt.

7.1.3.2 Abhängigkeit der Verlustleistung von den Datenbitänderungen

Der Einfluss der Parameter der Instruktionen auf die Leistungsaufnahme ist hierbei noch ein Unsicherheitsfaktor, der auch in dem hier vorgestellten Verfahren unberücksichtigt bleibt. Dass er jedoch einen Einfluss hat, zeigt sich anhand der Programme MUL2 und DIV2, in denen jeweils eine *add* Instruktion, bei der zwei Nullen addiert werden, um das Carry-Flag im Prozessor zu löschen, durch die spezielle *clc* Instruktion (Clear Carry Bit) ersetzt wurde. Die *clc* Instruktion wurde als eigenständiger Befehl charakterisiert und weist hierbei einen geringeren Energieverbrauch auf als der *add* Befehl, welcher mit durchschnittlichen Daten parametrisiert wurde. Dies zeigt, dass der *add* Befehl auf zwei Nullen ausgeführt sehr viel weniger Energie verbraucht, als auf andere Daten.
Dies erklärt auch die hohe Abweichung der instruktionsbasierten Abschätzungen gegenüber der Low-Level Abschätzung bei der Division von vier durch drei (DIV Beispiel 4), welche ein Extrembeispiel darstellt. Bei diesen Eingangsdaten wird in diesem Test-Programm sehr oft eine *rol* Instruktion (Rotate Left) auf einen Nullwert oder eine Binärzahl mit einer einzigen „1“ durchgeführt. Dies führt zu einer erheblich geringeren tatsächlichen Leistungsaufnahme als der mit dem instruktionsbasierten Verfahren abgeschätzten.
Diese Datenbitabhängigkeit der Verlustleistung ist jedoch nur schwer in die Abschätzung miteinzubeziehen.

7.1.3.3 Einfluss der Pipeline-Flushes auf den Energieverbrauch

Wie der Durchschnitt über die Differenz der Absolutwerte zeigt, lag die Abweichung bei dem Verfahren unter Einbeziehung der Pipeline-Flushes bei bedingten Befehlen gegenüber dem bisherigen instruktionsbasierten Standardabschätzverfahren etwa 4,7% näher an den Ergebnissen der Low-Level Abschätzung.
Zwar ergab sich bei der Division von 4 durch 3 die maximale Abweichung von 25% der einfachen ACCPWR Abschätzung von der Low-Level Abschätzung, jedoch stellt dies, wie schon oben beschrieben, einen Extremfall dar. Das Standardverfahren lieferte meist einen zu geringen Energieverbrauch und war daher in diesem Fall näher an der Low-Level Abschätzung. Für die meisten anderen Test-Programme war die einfache ACCPWR Abschätzung deutlich näher an den Vergleichsergebnissen.

Die Differenz der Abweichungen der beiden Verfahren hängt dabei nicht nur von der Anzahl der ausgeführten bedingten Instruktionen in den Test-Programmen ab, sondern auch von der Anzahl der falschen Sprungvorhersagen.
Der hier verwendete Prozessor besitzt nur eine zweistufige Pipeline, und es treten somit keine längeren Pipeline-Flushes auf. Bei längeren Pipelines, bei denen bei einem Flush mehrere Stufen verworfen werden müssen, ist die Auswirkung der Pipeline-Flushes auf die Verlustleistung entsprechend größer.
Sobald aber die Pipeline komplizierter und länger wird, und bei Sprüngen unterschiedlich viele Stufen bei einem Flush verworfen werden (z.B. durch Nutzung von Delay Slots oder bei Sprungbefehlen direkt nach falsch vorhergesagten Sprungbefehlen), reicht dieses einfache Verfahren nicht mehr aus.
Auch um die anderen Effekte des Pipelinings auf den Energieverbrauch miteinzubeziehen, muss die Pipeline für die Verlustleistungsanalyse in die einzelnen Stufen zerlegt werden.

7.2 Fallbeispiel 2: Abschätzungsverfahren mit Einzelenergiewerten für jede Pipeline-Stufe

Dieses Fallbeispiel beschreibt die Energieverbrauchsanalyse durch Aufteilung des gesamten Energieverbrauchs in die Einzelenergiewerte der Pipeline-Stufen, wie es in den Kapiteln 4 und 5 beschrieben wurde, und die Nachbildung der Abläufe in der Pipeline, insbesondere von Pipeline-Hazards wie Stalls, Bubbles oder Flushes.
Um dieses Abschätzungsverfahren zu veranschaulichen, wurde der quelloffene Soft-Core „Jam CPU“ der Chalmers University of Technology eingesetzt.
Dieser wird für den Spartan-3 FPGA von Xilinx synthetisiert und implementiert und anhand des daraus erstellten Timing-Modells für die Low-Level Abschätzungen simuliert.

7.2.1 Der „Jam CPU“ Soft-Core

Die Jam CPU [30] ist eine Weiterentwicklung eines RISC Prozessors mit dem Namen Concert'02 und ist ein quelloffener Soft-Core. Dieser in VHDL geschriebene Soft-Core besitzt eine Harvard Architektur und kann 4 KByte Befehls- und 4 KByte Datenspeicher ansprechen. Der Adressbus ist 19 Bit und der Datenbus 64 Bit breit.
Die Jam CPU hat eine Pipeline, die in folgende fünf Stufen Unterteilt ist:

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execute (EX)
- Memory Access (MEM)
- Write Back (WB)

Branch Befehle, also bedingte Sprünge werden bei der Jam CPU schon in der ID Stufe evaluiert, wodurch eine Verzögerung von nur einem Takt auftritt. Diese Verzögerung kann durch einen Branch Delay Slot vom Programmierer für eine vom bedingten Sprung unabhängige Instruktion genutzt werden. Illegale Befehle müssen jedoch selbst abgefangen werden, da hardwareseitig keine Kontrolle auf Befehle durchgeführt wird, die nicht im Delay Slot ausgeführt werden können. Somit treten bei Branch Befehlen

keine Pipeline-Flushes auf.
Bei den TRAP und JUMP Befehlen hingegen wird der Sprung erst in der MEM Stufe der Pipeline ausgeführt, und die nachfolgenden drei Befehle werden mit einem Pipeline-Flush verworfen. Der Pipeline-Flush geschieht durch Abbruch der fälschlich ausgeführten Instruktionen und durch Löschen der Register vor den betroffenen Pipeline-Stufen, was dem Einfügen von *nop* Instruktionen in die Pipeline entspricht (Siehe auch Kapitel 5.2.3.2).
Externe Interrupts werden als TRAPS in die Pipeline geleitet und verursachen damit auch einen Pipeline-Flush.
Die Jam CPU unterstützt Forwarding, so dass die aktuelle Instruktion das Ergebnis der vorherigen Instruktion direkt als Eingangsdaten verwenden kann, ohne dass es zuerst in ein Register zurückgeschrieben und neu eingelesen werden muss. Somit können die meisten durch Daten-Hazards verursachten Pipeline-Stalls vermieden werden.
Bei den Pipeline-Stalls wird der in den Registern gespeicherte Wert gehalten (Siehe auch Kapitel 5.2.2).
Bei der Jam CPU können drei verschiedene Arten von Pipeline-Stalls auftreten:

- Die Pipeline wird komplett angehalten, verursacht durch das Signal *mem_stalling*. Dieses tritt für die Dauer von einem Takt auf, wenn der „Store Word“ (*sw*) Befehl in die MEM Stufe gelangt, da der Speicherzugriff zwei Takte benötigt.

- Die Register vor der ID Stufe und vor der EX Stufe werden gehalten, und in das Register nach der EX Stufe werden Bubbles (*nop* Instruktionen) eingefügt, wenn das Signal *ex_stalling* aktiv wird. Dies tritt auf, wenn eine Multi-Cycle Instruktion, in diesem Fall die Multiplikation in der EX Stufe, ausgeführt wird. Eine weitere Ursache für dieses Signal kann ein Daten-Hazard sein, wenn die EX Stufe auf ein Register zugreifen möchte, das durch einen vorhergehenden „Load Word“ (lw) Befehl geschrieben werden soll. Da dieser wieder zwei Takte für den Speicherzugriff benötigt, liegt der zu verarbeitende Wert noch nicht vor und die EX Stufe muss einen Takt angehalten werden.

- Bei dem Signal *id_stalling* werden die Register vor der ID Stufe angehalten und in die Register vor der EX Stufe Bubbles eingefügt. Dieses Bubbling tritt auf, wenn einer der Operanden für die Bedingung des Branch Befehls noch nicht vorliegt. Durch das frühe Evaluieren

und Ausführen der Branch Instruktionen in der ID Stufe kann es vorkommen, dass ein benötigtes Ergebnis aus einer vorhergehenden Operation noch nicht vorliegt und so auch nicht per Forwarding in die ID Stufe weitergereicht werden kann.

Ein großer Vorteil der Jam CPU für diese Untersuchung ist der sehr gut strukturierte VHDL Quellcode mit einer klaren Einteilung auf die fünf Pipeline-Stufen (Siehe Abb. 7.2). Zwar sind diese Module für die Pipeline-Stufen im Quellcode Jam CPU Soft-Core noch nicht vorhanden, der VHDL Code des Prozessors ist jedoch schon so nach Pipeline-Stufen strukturiert, dass diese leicht in Module ausgegliedert werden können.

Nachteilig ist jedoch, dass es für diesen Prozessor keine Entwicklungsumgebung und keinen Compiler gibt. Die Ein-Befehl-Programme für die Charakterisierung und die Test-Programme, an denen die Abschätzung durchgeführt wird, mussten daher alle per Hand in Maschinensprache (Binärcode) geschrieben werden.

7.2.2 Methodik

Für das hier verwendete instruktionsbasierte Abschätzungsverfahren muss der Prozessor zuerst charakterisiert werden. Dazu muss der Quellcode des Prozessors nach Pipeline-Stufen modularisiert werden, um die Charakterisierung so durchführen zu können, dass man bei jeder Instruktion Energiewerte für jede Stufe der Pipeline erhält.
Die mit diesen Energiewerten aufgebaute Datenbank kann dann zur schnellen, taktgenauen Abschätzung des Energieverbrauchs des Prozessors verwendet werden.
Für alle Untersuchungen wurde der Prozessor in der Low-Level Simulation mit 50 Megahertz getaktet, was einer Taktdauer von 20 Nanosekunden entspricht.

7.2.2.1 Charakterisierung

Die Modularisierung und Charakterisierung wurden nach dem in Kapitel 6.2.1.2 beschriebenen Verfahren durchgeführt und somit die Energiewerte für jede Pipeline-Stufe und jede Instruktion ermittelt.
Die Synthese und Implementation des Soft-Cores sowie die Erstellung des Low-Level Simulationsmodells wurden mit der Xilinx ISE Software von

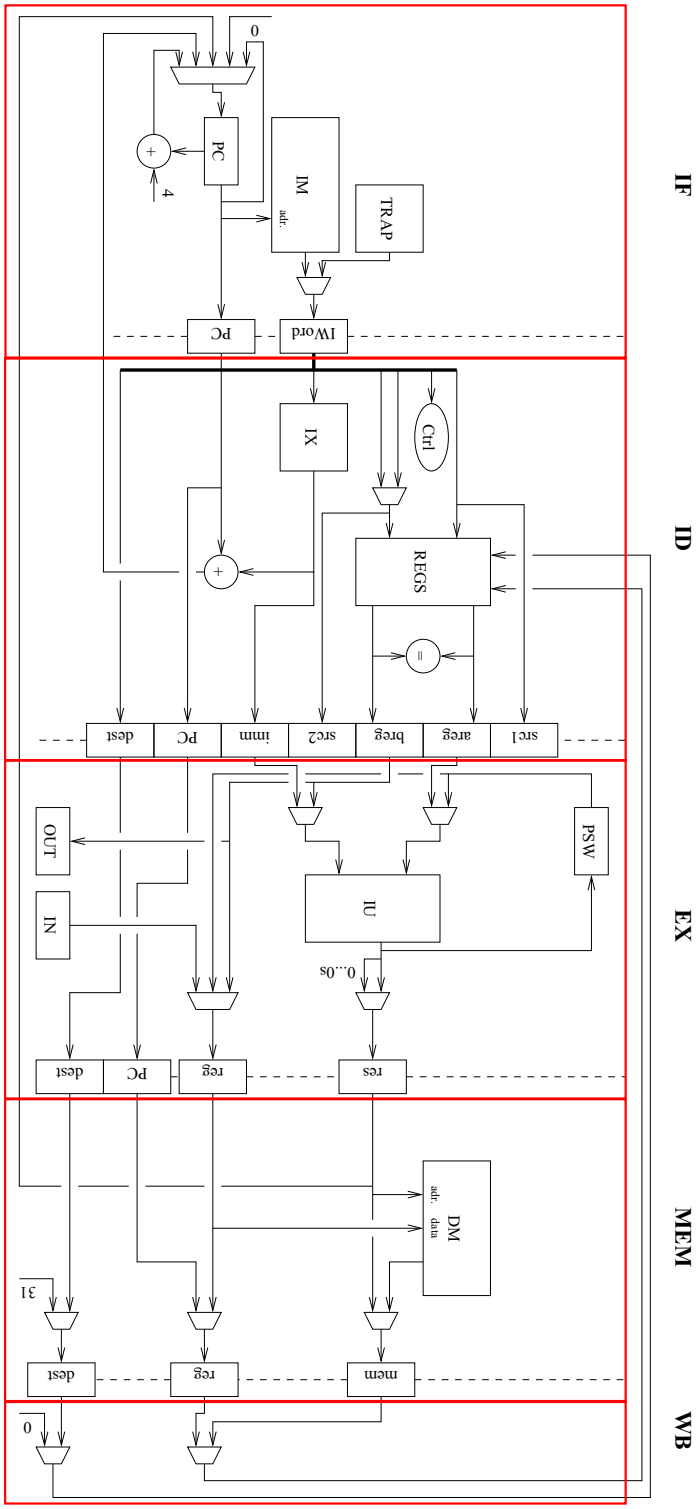

Abbildung 7.2: Schematische Darstellung der Jam CPU: Die Logik der Pipeline ist unterteilt in die fünf Stufen IF, ID, EX, MEM, und WB (Übernommen von [30] und bearbeitet)

dem FPGA Hersteller Xilinx ausgeführt. Als Low-Level Simulations Programm wurde ModelSim der Firma Mentor Graphics verwendet, und für die Low-Level Abschätzung das Programm XPower Analyzer von Xilinx, im folgenden einfach XPower genannt, eingesetzt.

7.2.2.1.1 Spannungsversorgung des Xilinx Spartan-3 FPGAs

Der Spartan-3 FPGA hat drei verschiedene Typen von Anschlüssen zur Spannungsversorgung des Chips, welche unabhängig voneinander mit verschiedenen Spannungen betrieben werden. Laut Spartan-3 Users Guide [37] liefern die V_{CCO} Eingänge die benötigte Spannung nur zum Treiben der Ausgänge des FPGAs und werden daher für die Verlustleistungsberechnung des Prozessors nicht berücksichtigt. Auch die V_{CCAUX} Eingänge, welche die Spannungsversorgung für Teile des Digital Clock Managers (DCMs), differentielle Leitungstreiber, Konfigurationsanschlüsse und das JTAG Interface bereitstellen, sind für die Abschätzung der Verlustleistung des Prozessors im laufenden Betrieb nicht von Bedeutung[2].
Als einzig für die Abschätzung relevanter Spannungsversorgungsanschluss bleibt V_{CCINT}, welcher die Hauptspannungsversorgung für die Logik und die internen Verbindungsleitungen darstellt. Für die folgenden Abschätzungen und Vergleiche wird also nur die Verlustleistung an diesem Anschlusstyp berücksichtigt.
Diese Verlustleistung wird nun in Energiewerte umgewandelt und in den statischen und den dynamischen Energieverbrauch unterteilt.

7.2.2.1.2 Statischer Energieverbrauch

Die statische Verlustleistung bei auf FPGAs laufenden Soft-Core Prozessoren hängt in hohem Maße von der Größe des verwendeten FPGAs ab. Jedoch belegt der Prozessor nur einen Teil der zur Verfügung stehenden programmierbaren Logik des FPGAs. Um eine gewisse Vergleichbarkeit zur dynamischen Verlustleistung herzustellen, und um die statische Verlustleistung unabhängig von der FPGA Größe zu machen, wird diese auf die benutzten Ressourcen des FPGA umgerechnet.
Bei dem hier verwendeten Spartan-3 (Genaue Bezeichnung: xc3s1000) hat

[2] Die dynamische Verlustleistung dieses Versorgungseingangs war laut XPower bei allen Experimenten immer Null, somit ist die hier verwendete Logik auch komplett unabhängig von diesem Eingang.

der gesamte mit der Jam CPU konfigurierte FPGA einen statischen Leistungsverbrauch von $45,05$ Milliwatt. Dies entspricht einem Energieverbrauch von $901,0$ Pikojoule pro Takt bei einer Frequenz von 50 Megahertz. Geringe Abweichungen von diesem Wert ergeben sich im Betrieb nur noch durch Temperaturänderungen im Chip, die hier jedoch nicht berücksichtigt werden.
Die Jam CPU benutzt 1494 Slices von den insgesamt 7680 zu Verfügung stehenden Slices des verwendeten FPGAs. Dies entspricht einer Ausnutzung von 19%. Umgerechnet auf diese Ausnutzung entspricht der statische Energieverbrauch der Jam CPU pro Takt somit $175,3$ Pikojoule.

7.2.2.1.3 Dynamischer Energieverbrauch

Die dynamische Verlustleistung wird, wie in Kapitel 4.1 beschrieben, in die befehlsunabhängige Verlustleistung P_{bu} und die Verlustleistungen der einzelnen Pipeline-Stufen (P_{IF}, P_{ID}, P_{EX}, P_{MEM} und P_{WB}) untergliedert. Diese erhält man, indem man in dem verwendeten Programm XPower die Ansicht „By Hierarchy“ wählt, in welcher die Verlustleistungen der vorher in Module ausgelagerten Pipeline-Stufen direkt abgelesen werden kann (Siehe Abb. 7.3).

P_{bu} ist die dynamische Verlustleistung, die bei einem Stall der kompletten Pipeline gemessen werden kann. Diesen kann man künstlich herbeiführen, indem man die Signalleitung *mem_stalling* in dem Simulationsprogramm auf logisch „1“ festlegt, und die dynamische Verlustleistung des FPGAs ermittelt.
Dieser Wert ist für alle charakterisierten Instruktionen gleich und wurde mit 8 Milliwatt ermittelt, was einem Energieverbrauch von 160 Pikojoule pro Takt entspricht.
Die befehlsabhängige Verlustleistung P_{ba} entspricht der Summe der Verlustleistungen P_{ps} ($ps \epsilon \{IF, ID, EX, MEM, WB\}$) der einzelnen Pipeline-Stufen und muss für jede Instruktion des Prozessors separat ermittelt werden. Dies geschieht über ein Programm, welches jeweils nur eine Instruktion mehrmals hintereinander ausführt. Dieses Programm muss für die Jam CPU mangels Compiler oder Assembler in Maschinencode geschrieben werden. Der Aufbau des Programms und der Ablauf der Charakterisierung geschehen nach dem Verfahren, das in den Kapiteln 6.2.1.1 und 6.2.1.2 beschrieben wurde.

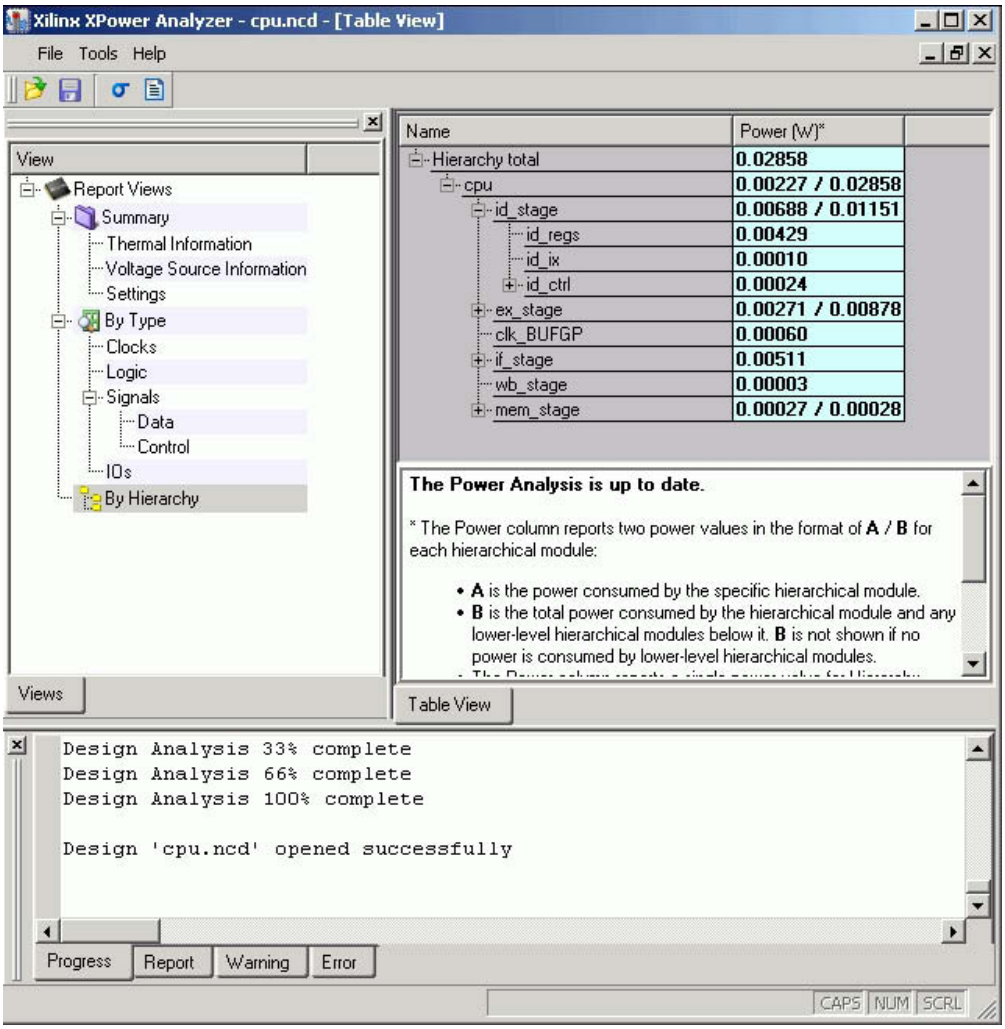

Abbildung 7.3: Die Darstellungsart „By Hierarchy“ des Low-Level Abschätz-Tools XPower des FPGA Herstellers Xilinx gibt detaillierte Verlustleistungsinformationen über die Module des Prozessors aus

P_{ba} ist dann die Verlustleistung, die im normalen Betrieb des Prozessors den Wert von P_{bu} übersteigt, also:

$$P_{ba} = P_{dyn} - P_{bu} \tag{7.6}$$

Der hierdurch ermittelte Wert ist jedoch etwas größer als die Summe der Werte, die die hierarchische Ansicht in XPower für die einzelnen Pipeline-Stufen liefert (Siehe Formel (7.7)). Wodurch diese Diskrepanz zustande kommt, konnte leider auch nach Kontakt mit Xilinx nicht geklärt werden.

$$P_{ba} > \sum_{ps=IF}^{WB} P'_{ps} \tag{7.7}$$

wobei P'_{ps} die Verlustleistungen der jeweiligen Pipeline-Stufen sind, welche aus der hierarchischen Ansicht übernommen wurden.
Um diesem Umstand aber gerecht zu werden, wird der überschüssige Teil der Verlustleistung auf die Verlustleistungen der Pipeline-Stufen aus der hierarchischen Ansicht prozentual gewichtet dazu addiert. Wie sich die anteilige Verlustleistung der einzelnen Pipeline-Stufen berechnet, wird in (7.8) beispielhaft für die Verlustleistung P_{IF} der IF Stufe der Pipeline gezeigt.

$$P_{IF} = P'_{IF} + \left(\frac{P'_{IF}}{\sum_{ps=IF}^{WB} P'_{ps}} \cdot \left(P_{ba} - \sum_{ps=IF}^{WB} P'_{ps} \right) \right), \tag{7.8}$$

7.2.2.1.4 Unterscheidung bei bedingten Sprüngen

Die Bedingungen für die bedingten Sprünge *bne* (Branch if not equal) und *beq* (Branch if equal) werden beim Jam CPU Prozessor in der ID Stufe evaluiert. Wenn die Bedingung für den Sprung erfüllt ist, muss im darauf folgenden Takt, während der Branch Befehl sich in der EX Stufe befindet, der in der IF Stufe inkrementierte Befehlszähler mit der Sprungzieladresse geladen werden, und der schon geladene nächste Befehl noch im selben Takt durch den Befehl an der Sprungzieladresse ersetzt werden. Dies hat einen höheren Energieverbrauch für den Branch Befehl in der EX Stufe zur Folge.
Die beiden bedingten Sprünge werden deshalb zweimal charakterisiert. Einmal wird der Energieverbrauch bei erfolgtem Sprung ermittelt und

einmal, wenn die Bedingung für den Sprung nicht erfüllt ist. Die Werte werden für den *bne* Befehl als „bne“ für den nicht erfolgten Sprung und „bne1“ für den erfolgten Sprung in der Datenbank abgelegt (Siehe auch Abb. 7.4). Genauso wird mit *beq* verfahren.

7.2.2.1.5 Energieverbrauch ausgewählter Instruktionen

Wie in Abb. 7.4 zu sehen ist, unterscheiden sich einzelne Instruktionen teilweise deutlich im Energieverbrauch. Auch während ein Befehl die verschiedenen Pipeline-Stufen durchläuft, unterscheidet sich sein Energieverbrauch von Stufe zu Stufe erheblich.

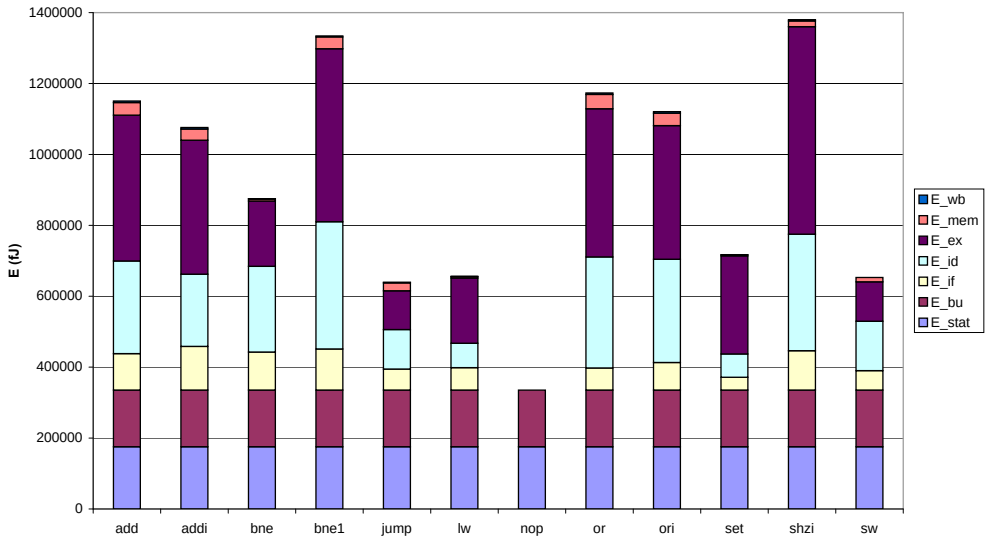

Abbildung 7.4: Der Energieverbrauch und dessen Verteilung auf die einzelnen Pipeline-Stufen bei der Bearbeitung ausgewählter Instruktionen auf dem Jam CPU Soft-Core

Zudem ist aber auch eine hohe Datenabhängigkeit festzustellen. Der *add* Befehl kann z.B. in der EX Stufe Werte von 0 Pikojoule annehmen, wenn sich keine Bits zum vorherigen Befehl ändern, also der exakt gleiche Befehl mit den gleichen Daten noch einmal dieselbe Stufe durchläuft, bis zu 891 Pikojoule bei umfangreichen Additionen mit vielen sich ändernden Bitstellen in Ein- und Ausgangsdaten.

Eine Liste aller charakterisierten Instruktionen des Jam CPU Soft-Cores kann Anhang A.2 entnommen werden.

7.2.2.2 Abschätzung

Für die taktgenaue Abschätzung des Energieverbrauchs ist es notwendig, den Fluss der Befehle in der Pipeline nachzubilden und dann für jeden Takt die jeweiligen Energiewerte aus der Datenbank zu den aktuellen Befehlen in den Pipeline-Stufen nachzuschlagen. Diese Werte werden für den jeweiligen Takt und für die gesamte zu untersuchende Befehlsfolge aufsummiert und man erhält so eine taktgenaue Energieverbrauchsabschätzung sowie eine Abschätzung der Energieaufnahme der abgelaufenen Befehlsfolge.
Das Abschätz-Programm ACCPWR, das diese Aufgaben ausführt, ist in diesem Fall ein Programm in der Script-Sprache „Perl“ und kann so auf unterschiedlichen Hardware-Systemen und Betriebssystemen ausgeführt werden.

7.2.2.2.1 Erstellung des Befehls-Trace

Da für diesen Prozessor kein Debugger oder Simulator verfügbar war, wurden die Befehls-Traces der Test-Programme manuell über eine Simulation des Prozessors auf der Verhaltensebene (behavioral level) ermittelt. Dies erfordert keine genaue Timing-Analyse und ist daher sehr viel schneller als eine Low-Level Simulation.
Hierfür wird das Verhaltensmodell des Prozessors zusammen mit einem Modell des ROM-Speichers und eines darin enthaltenen Test-Programms in ModelSim simuliert. Dabei werden das aktuelle 32-Bit Instruktionswort (*if_iword*) und der Befehlszähler (*if_pc*) des Prozessors für jeden Takt zusammen mit den Pipeline-Stall (*mem_stalling, ex_stalling, id_stalling*) und Pipeline-Flush (*mem_clear*) Signalen[3] in einer Log-Datei aufgezeichnet.
Diese Log-Datei enthält die Zeit in Nanosekunden, das binäre 32-Bit Wort des aktuellen Befehlszählerstands, das binäre 32-Bit Mikroinstruktionswort der IF Stufe und die oben beschriebenen, jeweils ein Bit breiten, Stall- und Flush-Signale. Zu jedem Zeitpunkt, an dem sich eines dieser Werte verändert, wird eine neue Zeile mit der entsprechenden Zeit und

[3] Die Abschätzung mit ACCPWR kann auch ohne diese internen Signale funktionieren. Dies wird im Fallbeispiel 3 am MicroBlaze Prozessor gezeigt.

den aktuellen Werten in die Log-Datei geschrieben.
Dieser Befehls-Trace wird von dem Abschätz-Programm ACCPWR anschließend ausgewertet und daraus der Fluss der Befehle durch die Pipeline berechnet.

7.2.2.2.2 Nachbildung von Stall- und Flush-Verhalten des Prozessors

Außer bei dem Pipeline-Stall durch das Signal *mem_stalling*, bei dem die komplette Pipeline angehalten wird, handelt es sich bei den Pipeline-Stalls der Jam CPU um Bubbling. Bei dieser Art von Pipeline-Stall wird der Befehl in der den Stall verursachende Pipeline-Stufe mit meist anderen Daten als im vorhergehenden Takt ausgeführt. Dadurch wird in dieser Stufe die Energie für diesen Befehl in dieser Pipeline-Stufe ein zweites Mal verbraucht.
Die Stufen vor dieser Pipeline-Stufe werden angehalten und nehmen somit keine dynamische Leistung auf.
In die Stufe, die direkt nach der Stall verursachenden Pipeline-Stufe folgt, wird ein *nop* Befehl eingefügt und die entsprechende Leistung verbraucht (Siehe auch „Sonderbehandlung des *nop* Befehls“). In allen darauf folgenden Stufen werden die Befehle aus den Registern der jeweils vorherigen Pipeline-Stufe normal verarbeitet und der Energieverbrauch berechnet sich für diese Stufen so, wie auch beim normalen Betrieb der Pipeline.

T	PC	IF	ID	EX	MEM	WB
100	5	beq	and	addi	addi	addi
120	6	add	beq	and	addi	addi
140	6	add(stalled)	beq(working)	nop	and	addi
160	14	set	add	beq1	nop	and
180	15	sw	set	add	beq1	nop

Tabelle 7.3: Da ein Register für die Bedingung des Branch Befehls noch nicht vorliegt, wird ein Teil der Pipeline angehalten und in der EX Stufe ein Bubble (*nop* Befehl) eingefügt

Bei einer *jump* oder *trap* Instruktion wird, wenn einer dieser Befehle die MEM Stufe erreicht, ein Pipeline-Flush ausgelöst und die Befehle in den Pipeline-Stufen IF, ID und EX durch *nop* Befehle ersetzt. Da diese verworfenen Befehle in diesem Takt zuerst noch ausgeführt werden, bis sie

durch die *nop* Befehle ersetzt werden, verbrauchen sie noch ihre normale für diese Pipeline-Stufen charakterisierte Energie.
Im nächsten Takt wird dann die Instruktion der Zieladresse in der IF Stufe geladen, die drei *nop* Instruktionen und die den Pipeline-Flush auslösende Instruktion werden in den jeweilig nächsten Pipeline-Stufen bearbeitet und verbrauchen die entsprechende Energie.

T	PC	IF	ID	EX	MEM	WB
40	4	add	addi	jump	addi	ori
60	5	lw	add	addi	jump	addi
80	14	addi	nop	nop	nop	jump
100	15	sw	addi	nop	nop	nop

Tabelle 7.4: Bei einem Pipeline-Flush werden die fälschlich ausgeführten Befehle abgebrochen und durch *nop* Befehle ersetzt. Dann wird die Pipeline mit den Befehlen der Sprungadresse (in diesem Fall 14) neu gefüllt

7.2.2.2.3 Sonderbehandlung des „nop“ Befehls

Der Instruktionssatz der Jam CPU verfügt eigentlich über keinen echten *nop* Befehl. So wird als *nop* Befehl das Instruktionswort $0x0000$ verwendet, welches einem *add* Befehl des „Zero“-Registers R0 mit sich selbst entspricht. Dieser Befehl wird nicht abgefangen, sondern es wird wirklich eine „0“ mit einer „0“ addiert und das Ergebnis verworfen, da das Zielregister, ebenfalls R0, schreibgeschützt ist.
Der befehlsabhängige Energieverbrauch in allen Pipeline-Stufen dieses *nop* Befehls ist nahezu Null. Dies ist aber nur der Fall bei konsekutiver Ausführung derselben Instruktion. Dies gilt zwar für fast alle Instruktionen außer Sprüngen. Normalerweise treten vollkommen gleiche Instruktionen nacheinander nur selten auf, außer eben bei *nop* Befehlen, welche häufig auch mehrfach hintereinander auftreten. Deshalb lohnt sich hier eine Sonderbehandlung.
Hierbei wird bei einem einzeln auftretenden *nop* Befehl der Energieverbrauch der normalen add Instruktion angesetzt. Auch für den ersten *nop* Befehl in einer Reihe hintereinander auftretender *nop* Befehle wird dieser Energiewert bei der Berechnung verwendet. Alle auf diesen ersten *nop* Befehl folgenden *nop* Befehle werden jedoch mit dem Energiewert „0“ berechnet.

T	PC	E_{stat}	E_{bu}	E_{ps1}	E_{ps2}	E_{ps3}	E_{ps4}	E_{ps5}
40	2	175273	160000	102703	203815	109416	31537	4352
60	3	175273	160000	63155	261266	377918	22210	4431
80	14	175273	160000	123019	261266	411312	35570	2104
100	15	175273	160000	54685	203815	0	0	4509

Tabelle 7.5: Energiewerte zu dem Beispiel aus Tabelle 7.4: Die Ausführung von *nop* Befehlen verbraucht generell die Energie eines *add* Befehls, außer es werden mehrere *nop* Befehle hintereinander in einer Stufe ausgeführt

7.2.3 Ergebnisse

Als Test-Programme wurden zwei Programme verwendet, die speziell zum Testen der Funktionen der Jam CPU entworfen wurden. Das erste Programm *p1* eignet sich dabei sehr gut für die Abschätzung durch ACCPWR, da es viele verschiedene Pipeline-Stalls und Flushes verursacht. Je mehr solche Pipeline-spezifischen Besonderheiten in einem Programm vorkommen, desto ungenauer wird die instruktionsbasierte Verlustleistungsabschätzung nach dem Standardverfahren, da sie nur die Verlustleistungen der Befehle im normalen Befehls-Trace aufsummiert und diese Besonderheiten nicht in die Rechnung miteinbeziehen kann.
Das zweite Programm p2 entspricht dabei eher einem Standardprogramm für einen Mikroprozessor. Es akkumuliert Werte in einer Schleife und schreibt die Zwischenergebnisse in den Speicher, bis das Ergebnis der Akkumulation einen bestimmten Wert überschreitet.

Test-Programm	**Low-Level Abschätzung**	**ACCPWR Abschätzung**	**Abweichung (ACCPWR)**
p1	52392pJ	53353pJ	**1,8%**
p2	79985pJ	74545pJ	**-6,8%**

Tabelle 7.6: Die Energiewerte der Abschätzungen und die jeweilige prozentuale Abweichung der Abschätzung von ACCPWR gegenüber der Low-Level Abschätzung

Die prozentuale Abweichung von der Low-Level Abschätzung, die hier als Referenzwert oder als so genannter „richtiger“ Wert angesehen wird, kann jeweils für das gesamte Test-Programm aus Tabelle 7.6 entnommen wer-

den.
Bei der taktgenauen Abschätzung wurde für jeden Takt der Energieverbrauch abgeschätzt und mit dem Wert der taktweisen Low-Level Abschätzung (Siehe Kapitel 6.4) verglichen. Abbildung 7.5 stellt diese Werte für eine kurze Befehlsfolge aus dem Test-Programm p2 einander gegenüber.

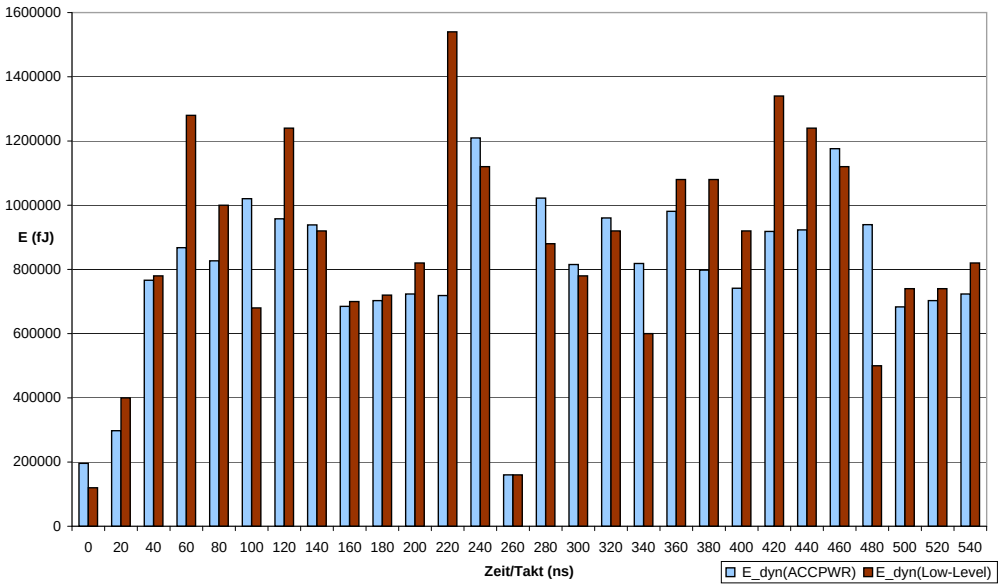

Abbildung 7.5: Die taktweisen Energiewerte der ACCPWR Abschätzung einer Befehlsfolge aus dem Test-Programm p2 im Vergleich zu den Werten der taktweisen Low-Level Abschätzung

7.2.3.1 Vergleich mit Standardverfahren

Normale instruktionsbasierte Verlustleistungsabschätzungsverfahren bilden die komplexen Abläufe bei Pipeline-Stalls und Flushes nicht genau nach und liefern daher nicht so genaue Abschätzungen wie das in dieser Arbeit vorgestellte Verfahren.
Bei den hier angestellten Vergleichsabschätzungen wurden dieselben Energiewerte aus der Charakterisierung des Prozessors verwendet, jedoch nur unterteilt in die Energiewerte E_{stat} und E_{dyn}, wovon letzterer von dem ausgeführten Befehl abhängt. Diese Werte wurden für das jeweilige untersuchte Test-Programm anhand des einfachen Befehls-Traces aufsummiert und mit dem Ergebnis der Low-Level Abschätzung verglichen.

Test-Programm	Low-Level Abschätzung	Standard Abschätzung	Abweichung (Standard)
p1	52392pJ	38661pJ	**-26,2%**
p2	79985pJ	65564pJ	**-18,0%**

Tabelle 7.7: Die Energiewerte der Abschätzungen und die prozentuale Abweichung gegenüber der Low-Level Abschätzung bei der Abschätzung nach dem instruktionsbasierten Standardverfahren

Die ACCPWR Abschätzung war in beiden Programmen deutlich näher an dem erwarteten Wert der Low-Level Abschätzung.
Da beim normalen instruktionsbasierten Verfahren keine Unterteilung in Pipeline-Stufen erfolgt und somit für den Takt, in dem die Instruktion in der IF Stufe der Pipeline bearbeitet wird, den gesamten Energieverbrauch dieses Befehls ausgibt, fällt bei der taktgenauen Abschätzung das Ergebnis noch deutlicher aus (Siehe Abb. 7.6).

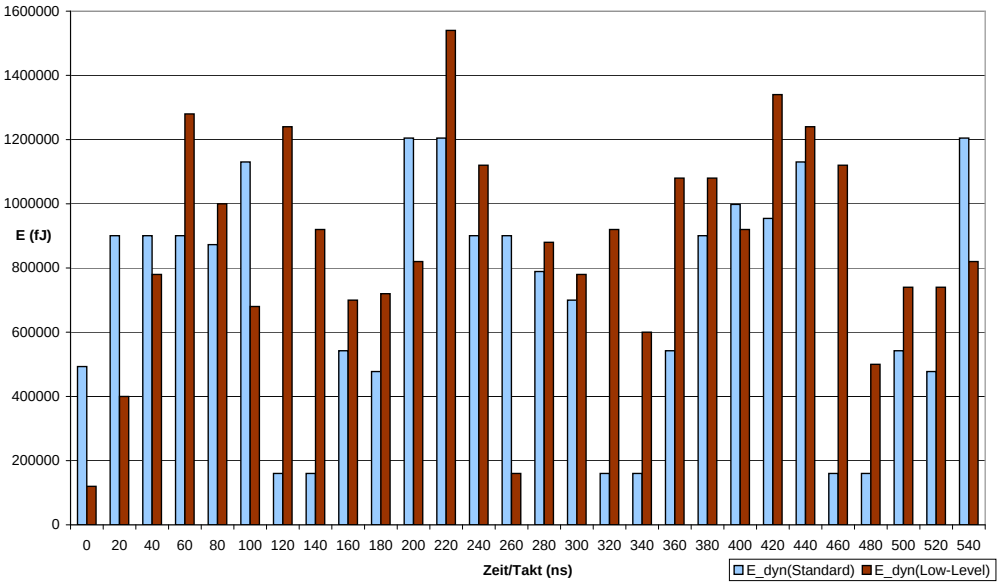

Abbildung 7.6: Die taktweisen Energiewerte der normalen instruktionsbasierten Abschätzung der gleichen Befehlsfolge wie in Abb. 7.5, wieder im Vergleich zu den Werten der taktweisen Low-Level Abschätzung

Als Vergleichswert wird der Mittelwert der Beträge der taktweisen prozen-

tualen Abweichungen berechnet. Tabelle 7.8 liefert hierfür eine Übersicht und zeigt auch die jeweiligen maximalen Abweichungen von der Low-Level Abschätzung beim taktweisen Vergleich.

Test-Programm	Proz. Ab-(weichung) (ACCPWR)	Proz. Abweichung (Standard)	Max. Abweichung (ACCPWR)	Max. Abweichung (Standard)
p1	16%	40%	524pJ	1020pJ
p2	18%	45%	841pJ	1080pJ

Tabelle 7.8: Der Mittelwert der Beträge der prozentualen Abweichung bei der taktgenauen Abschätzung und die maximale Abweichung von der taktweisen Low-Level Abschätzung

7.2.3.2 Daten- und Befehlsbitabhängigkeit

Die Abhängigkeit des Energieverbrauchs von der Veränderung von Bits in den zu verarbeitenden Daten ist bei allen Verfahren der instruktionsbasierten Verlustleistungsabschätzung nur sehr aufwendig modellierbar und sie ist deshalb auch der Hauptfaktor für Abweichungen der ACCPWR Abschätzungen von der Low-Level Abschätzung.
Die Abschätzung der Befehlsfolgen wurde bei Erreichen der abschließenden Endlosschleife aus den Befehlen „*beq r0,r0,r0*“ und „*add r0,r0,r0*“ (im Branch Delay Slot) abgebrochen. Bei dieser Endlosschleife jedoch verbraucht der Prozessor weit weniger Energie als die Abschätzung berechnet. Bei Vergleichen ergaben sich dabei Differenzen von bis zu 712 Pikojoule pro Takt.
Hier zeigt sich die Abhängigkeit der verbrauchten Energie von den tatsächlich verarbeiteten Daten. Um die Abschätzung für diesen Bereich zu verbessern, müsste dies durch einen weiteren Sonderfall für solche Endlosschleifen bei der Berechnung der Abschätzung abgefangen werden.
Noch deutlicher wird die Datenbitabhängigkeit bei der taktgenauen Energieabschätzung. Dies ist ersichtlich anhand der teils hohen Ausreißer in Abb. 7.5, z.B. in dem Takt bei 220 Nanosekunden. Die Datenbitabhängigkeit nachzubilden ist noch eine Schwäche aller instruktionsbasierten Abschätzungsverfahren. Bei der Abschätzung mit Modularisierung nach Pipeline-Stufen treten diese Abweichungen durch die Datenbitabhängigkeit vor allem in den Stufen EX, MEM und WB auf.

Eine Befehlsbitabhängigkeit wie bei dem AVR-Core Prozessor ist bei diesem Soft-Core für bestimmte Stufen (IF, ID) noch gegeben, aber im gesamten Energieverbrauch ist sie nur noch von sehr geringer Bedeutung.

7.2.3.3 Geschwindigkeit

Für ein 155 Takte laufendes Test-Programm brauchte die Abschätzung mit ACCPWR inklusive der Verhaltenssimulation in ModelSim nur 4,03 Sekunden. Die taktgenaue Low-Level Abschätzung mit ModelSim und XPower brauchte automatisiert 12 Minuten und 11 Sekunden. Für die beim Low-Level Verfahren zusätzlich nötige Synthese, Implementierung und Generierung des Timing Modells werden weitere 10 Minuten benötigt.
Gegenüber dem normalen instruktionsbasierten Verfahren ist ein höherer Zeitbedarf nicht zu ermitteln, da die etwas aufwendigeren Berechnungen auf heutigen Computern nicht ins Gewicht fallen und sich der zusätzliche Zeitaufwand für diese Berechnungen im Bereich von wenigen Tausendstelsekunden bewegt. Jedoch ist zu berücksichtigen, dass die Charakterisierung bei dem hier vorgestellten Verfahren auch deutlich komplizierter und damit zeitaufwendiger ist als bei den bisherigen instruktionsbasierten Verfahren.

7.3 Fallbeispiel 3: Abschätzungsverfahren bei IP-Core Prozessoren

Wenn die interne Hardware-Beschreibung des Prozessors nicht zugänglich ist, wie z.B. bei Hard-Core Prozessoren oder IP-Cores, bei denen der Hersteller den strukturellen Aufbau der Pipeline nicht offenlegt oder keine Daten über die Verlustleistung der einzelnen Pipeline-Stufen veröffentlicht, kann die genaue Verteilung der Verlustleistung auf die einzelnen Pipeline-Stufen nicht bestimmt werden.
An dem Beispiel des MicroBlaze IP-Cores von Xilinx wird hier gezeigt, wie dieses Verfahren auch auf Prozessoren angewendet werden kann, deren innerer Aufbau weitgehend unbekannt ist.

7.3.1 Der MicroBlaze IP-Core

Der Xilinx MicroBlaze [36] ist ein kommerzieller IP-Core von Xilinx, der speziell auf die Xilinx FPGAs zugeschnitten ist. Es handelt sich dabei um einen 32-Bit RISC Prozessor in Harvard Architektur. Das Grunddesign umfasst:

- 32 frei verwendbare Register
- eine ALU
- eine Shift Unit
- 2 Interruptstufen

Das Grunddesign kann über die Konfigurationssoftware Embedded Development Kit (EDK) mit verschiedenen Funktionseinheiten erweitert werden:

- Barrelshifter / Multiplizierer / Dividierer
- Memory Management Unit / Memory Protection Unit
- Floating Point Unit
- Caches
- Exception Handling
- Debug-Logik

Die Pipeline des MicroBlaze ist ähnlich aufgebaut wie die des Jam CPU Soft-Cores und umfasst normalerweise folgende fünf Pipeline-Stufen:

- Fetch (IF)
- Decode (OF)
- Execute (EX)
- Access Memory (MEM)
- Writeback (WB)

Xilinx bietet über die EDK Software die Option an, die Pipeline mit nur drei Stufen zu konfigurieren und dadurch Platz auf dem FPGA einzusparen. Dies hat jedoch einen Performance-Verlust zur Folge. Der dreistufigen Pipeline fehlen dabei die „Access Memory“ und „Writeback“ Stufen.
In diesem Fallbeispiel wird die auf Performance optimierte Variante mit fünf Pipeline-Stufen verwendet und auf jegliche zusätzliche Funktionseinheiten verzichtet.
Auch die Sprungvorhersage des MicroBlaze ist ähnlich wie bei den Prozessoren der vorherigen Fallbeispiele. Er nutzt keine komplexe dynamische Branch Prediction, sondern geht immer davon aus, dass nicht gesprungen wird. Er besitzt also eine statische „never-branch“ Prediction.
Im Gegensatz zur Jam CPU werden jedoch Sprünge in der dritten Pipeline-Stufe (EX) evaluiert. Somit kommt es zu zwei zusätzlichen Takten Verzögerung bei einem falsch vorhergesagten Sprung. Der Prozessor verfügt jedoch über einen Delay Slot, der vom Compiler für eine unabhängige Instruktion genutzt werden kann. Wird dieser verwendet, so muss nur der darauf folgende zweite Befehl in der IF Stufe nach dem Sprung durch einen Flush geleert und neu geladen werden.
Somit ergibt sich für bedingte Befehle eine zusätzliche Verzögerung von

- 0 Takten, wenn der Sprung nicht genommen wird, bzw. die Vorhersage korrekt war,
- 1 Takt, wenn der Sprung ausgeführt wird und ein Delay Slot genutzt wird,
- 2 Takten, wenn der Sprung ausgeführt wird und kein Delay Slot genutzt wird.

Pipeline-Stalls treten in Form von Bubbling bei Daten-Hazards auf, wenn ein Register in der EX Stufe benötigt wird, aber noch nicht zur Verfügung

steht. Dies kann z.B. durch einen noch nicht beendeten „Load Word“ Befehl (*lw*) verursacht werden, der das betreffende Register als Zielregister enthält. Diese Art des Bubbling, welches dem durch das *ex_stalling* Signal aus Fallbeispiel 2 verursachten Bubbling sehr ähnlich ist, ist die einzige Form von Pipeline-Stalls die bei diesem Prozessor auftreten können.

7.3.2 Methodik

Für die Untersuchungen wurde der MicroBlaze Prozessor in einer minimalen Konfiguration mit dem Xilinx Platform Studio (XPS) erzeugt. Die Low-Level Simulation basiert auf einem für den Spartan-3 FPGA synthetisierten und implementierten Simulationsmodell des Prozessors. Als Ziel-FPGA wurde hierbei der xc3s200, ein kleinerer Chip aus derselben Spartan-3 Reihe, gewählt, der jedoch dieselben Spannungsversorgungseigenschaften wie der FPGA in Fallbeispiel 2 aufweist.
Für die Low-Level Simulation wurde eine Prozessorfrequenz von 50 Megahertz verwendet, was zu einer Taktdauer von 20 Nanosekunden führt.

7.3.2.1 Charakterisierung

Die Charakterisierung erfolgt für diesen Prozessor nach dem Standardverfahren aus Kapitel 6.2.1.1 ohne die Unterteilung nach Pipeline-Stufen, da der innere Aufbau des IP-Cores und damit die Aufteilung der Logik auf die Pipeline-Stufen nicht bekannt ist.
Man erhält also für jeden Befehl einen Wert für die dynamische Verlustleistung und einen für die statische Verlustleistung. Die statische Verlustleistung des hier verwendeten kleineren Spartan-3 FPGAs beträgt 12,7 Milliwatt, was einem Energieverbrauch von 254,6 Pikojoule pro Takt entspricht. Dieser wird wie in Kapitel 7.2.2.1 auf benutzte FPGA Fläche umgerechnet und ergibt für die verwendete Konfiguration des MicroBlaze, die mit 1026 Slices 53% der Ressourcen des Spartan-3 (xc3s200) nutzt, einen Wert von 135,0 Pikojoule.
Der befehlsunabhängige Energieverbrauch E_{bu} kann hier nicht über einen künstlich erzeugten Pipeline-Stall herausgefunden werden. Jedoch gibt es bei dem Prozessor eine relativ lange Setup Phase von etwa 89 Takten nach dem Reset. Da in dieser Zeit keine Befehle in der Pipeline verarbeitet werden, entspricht der dynamische Energieverbrauch zu diesem Zeitpunkt in

etwa dem befehlsunabhängigen Energieverbrauch. Der hierbei für den MicroBlaze festgestellte befehlsunabhängige Energieverbrauch E_{bu} liegt bei 1084,1 Pikojoule.
Dieser Wert ist im Vergleich zur Jam CPU sehr hoch. Der Grund dafür ist, dass die EDK Software von Xilinx, mit der der MicroBlaze konfiguriert und implementiert wird, alle nötigen Komponenten für den Betrieb des Prozessors schon fest im Design für den FPGA implementiert. Der größte Anteil der befehlsunabhängigen Energie mit etwa 744 Pikojoule wird für die Generierung der Taktsignale durch den Digital Clock Manger (DCM) benötigt.
Eine weitere Möglichkeit den E_{bu} zu ermitteln, wäre die Messung des Energieverbrauchs bei Ausführung eines Programms mit mehreren *nop* Befehlen hintereinander.
Die befehlsabhängige Energieverbrauch E_{ba} ist die Differenz von E_{dyn} und E_{bu} (Siehe auch Formel (7.6)). Dieser ist für jede Instruktion unterschied-

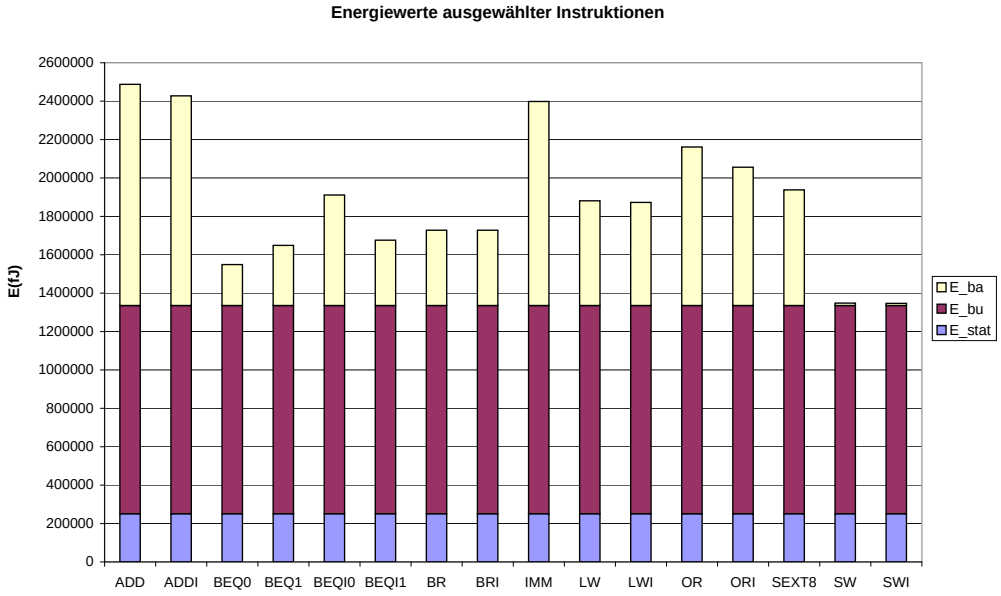

Abbildung 7.7: Der Energieverbrauch ausgewählter Instruktionen des MicroBlaze Prozessors unterteilt in die statische, befehlsunabhängige und befehlsabhängige Energie

Für die erweiterte Abschätzung muss die befehlsabhängige Energie E_{ba} in die Energiewerte der Pipeline-Stufen unterteilt werden. Hierfür gibt es mehrere Möglichkeiten. Zwei davon werden hier untersucht:

7.3.2.1.1 Verteilung auf alle Stufen gleichmäßig

Bei einer Gleichverteilung auf die hier vorliegenden fünf Pipeline-Stufen ergibt das für die Energiewerte der einzelnen Stufen, dass sie alle 20% des gesamten befehlsabhängigen Energieverbrauchs E_{ba} betragen.

	E_{ba}	E_{IF}	E_{OF}	E_{EX}	E_{MEM}	E_{WB}
Verteilung:	100%	20%	20%	20%	20%	20%
Beispiel:						
addi	1092pJ	218,4pJ	218,4pJ	218,4pJ	218,4pJ	218,4pJ

Tabelle 7.9: Die Verteilung des befehlsabhängigen Energieverbrauchs auf die Pipeline-Stufen durch Gleichverteilung

Der Energieverbrauch jedes charakterisierten Befehls wird nach diesem Schema aufgeteilt und in eine neue Datenbank geschrieben, die genauso aufgebaut ist wie in Fallbeispiel 2.

7.3.2.1.2 Verteilung anhand Erfahrungswerten eines Prozessors mit ähnlichem Pipeline-Aufbau

Da diese Gleichverteilung des Energieverbrauchs eher unwahrscheinlich ist, können auch Erfahrungswerte eines ähnlich aufgebauten Prozessors verwendet werden. Der Jam CPU Soft-Core hat einen ähnlichen Pipeline-Aufbau wie der MicroBlaze Prozessor und wird hier als Vergleichsprozessor verwendet.

	E_{ba}	E_{IF}	E_{OF}	E_{EX}	E_{MEM}	E_{WB}
Verteilung:	100%	15%	30%	50%	4%	1%
Beispiel:						
addi	1092pJ	163,8pJ	327,6pJ	546pJ	43,68pJ	10,92pJ

Tabelle 7.10: Die Verteilung des befehlsabhängigen Energieverbrauchs auf die Pipeline-Stufen nach Erfahrungswerten von dem Jam CPU Soft-Core

Die Verteilung wird hierbei aus den Mittelwerten der Verteilungen aller Befehle des Jam CPU Prozessors ermittelt und nach dem Schema aus Tabelle 7.10 auf alle Befehle des MicroBlaze angewendet.

7.3.2.2 Abschätzung

Für den MicroBlaze IP-Core liefert der FPGA-Hersteller Xilinx eine auf diesen Prozessor angepasste Version der GNU Compiler Collection (GCC) Entwicklungsumgebung. Die Test-Programme, die hier in der Programmiersprache C vorliegen, werden mit dem für den MicroBlaze modifizierten GNU C-Compiler *mb-gcc* kompiliert und in eine auf dem MicroBlaze ausführbare ELF-Datei umgewandelt. Diese kann direkt mit dem GDB simuliert werden und daraus, wie in Kapitel 6.3.1 beschrieben, der Befehls-Trace erzeugt werden.
Für die Low-Level Simulation wird aus der ELF-Datei über die Hilfsprogramme *data2mem* und *ucf2vhdl.pl* von Xilinx eine VHDL-Datei erzeugt, die den Programmspeicher mit dem darin enthaltenen Programm simuliert.
Für die eigentliche Abschätzung wurde das ACCPWR Perl Script aus Fallbeispiel 2 zur Abschätzung des MicroBlaze angepasst. Hierbei musste hauptsächlich die Nachbildung der Pipeline-Stalls und Flushes geändert werden, die aus dem Befehls-Trace ermittelt werden sollen.
Der Befehls-Trace wird bei dem MicroBlaze Prozessor über den Xilinx Microprozessor Debugger (XMD) und über ein Batch Script für den GNU Debugger (GDB), wie in Kapitel 6.3.1 beschrieben, erstellt.
Die Pipeline-Stalls und Flushes werden in diesem Fallbeispiel aus dem Befehls-Trace des GDB ohne Zuhilfenahme der Überwachung von internen Signalleitungen des Prozessors berechnet, da diese Signale hier nicht zugänglich sind.
Die Pipeline-Stalls durch Daten-Hazards lassen sich dabei aus den Parametern der ausgeführten Instruktionen und der Verwendung der Register berechnen. Für die teilweise ausgeführten Befehle nach einem falsch vorhergesagten Sprung und für die Befehle, die in Delay-Slots ausgeführt werden, benötigt man aber zusätzliche Angaben. Deshalb muss der Debugger über ein spezielles Script so gesteuert werden, dass er zusätzliche Ausgaben in die Log-Datei schreibt, über die dann die fehlenden Informationen berechnet werden können.
Hierzu müssen bei einer falschen Vorhersage „Branch Not Taken“ zusätzlich die Befehle, die im Speicher direkt nach dem Sprungbefehl stehen ausgegeben werden, da diese fälschlich ausgeführt werden. Bei einer falschen Vorhersage „Branch Taken“, also wenn an eine Adresse gesprungen wird und sich später herausstellt, dass der Sprung nicht ausgeführt werden hätte sollen, müssen die Befehle nach der Sprungzieladresse im Befehlsspeicher

zusätzlich ausgegeben werden.
Das Script zur Steuerung des Debuggers für die Erstellung des erweiterten Befehls-Trace beim MicroBlaze Prozessor ist in Listing 7.1 dargestellt. Da dieser Prozessor eine statische Sprungvorhersage mit „Branch Not Taken“ besitzt, kann bei jedem Step der nächste zu ladende Befehl und die zwei nächsten auf diesen im Befehlsspeicher folgenden Befehle ausgegeben werden. Die Untersuchung, ob ein Sprungbefehl vorliegt oder nicht, wird im Abschätzprogramm ACCPWR vorgenommen.

```
# Aufruf dieses Scripts:
# gdb -batch -x trace.gdb

target remote localhost:1234
set architecture MicroBlaze
set logging file trace_gdb.log
set logging on

#Lade das zu untersuchende Test-Programm
load test_prog.elf

# Setze letzte Befehlsadresse
# Für mit mb-gcc kompilierten Code ist dies 0x6c
set $exitpc=0x6c

# Gib die aktuell ausgeführte Instruktion und die
# 3 im Speicher folgenden Instruktionen aus
while ($rpc != $exitpc)
  disassemble ($rpc) ($rpc+12)
  printf "--\n"
  stepi
end
disassemble ($rpc) ($rpc+12)
```

Listing 7.1: GDB Script zur Ausgabe von je drei Befehlen pro Step

Für den in Listing 6.2 aufgeführten Programmausschnitt veranlasst dieses Script den Debugger folgende Ausgaben in die Log-Datei zu schreiben:

```
0x00000110 in ?? ()
Dump of assembler code from 0x110 to 0x11c:
0x00000110:     lwi     r15, r1, 0
0x00000114:     rtsd    r15, 8
0x00000118:     addik   r1, r1, 28
End of assembler dump.
--
0x00000114 in ?? ()
Dump of assembler code from 0x114 to 0x120:
0x00000114:     rtsd    r15, 8
0x00000118:     addik   r1, r1, 28
0x0000011c:     addi    r1, r1, -20
End of assembler dump.
--
0x00000490 in ?? ()
Dump of assembler code from 0x490 to 0x49c:
0x00000490:     brlid   r15, -80        // 0x440
0x00000494:     or      r0, r0, r0
0x00000498:     lw      r15, r0, r1
End of assembler dump.
--
0x00000440 in ?? ()
Dump of assembler code from 0x440 to 0x44c:
0x00000440:     lwi     r3, r0, 1216 // 0x4c0
0x00000444:     addik   r1, r1, -32
0x00000448:     swi     r19, r1, 28
End of assembler dump.
```

Listing 7.2: Ausgabe des Debuggers mit je drei Befehlen pro Step

Diese Ausgabe des Debuggers enthält nun alle nötigen Informationen, damit das Programm ACCPWR einen erweiterten Befehls-Trace mit Pipeline-Stalls, nur teilweise ausgeführten Befehlen und Delay-Slot Befehlen berechnen kann. Der fertige erweiterte Befehls-Trace für das obige Beispiel ist in Listing 7.3 dargestellt.

```
0x00000110 lwi
xxxxxxxxxx stall (Pipeline stall because of register r15)
xxxxxxxxxx stall (Pipeline stall because of register r15)
0x00000114 rtsd
0x00000118 addik (Delay Slot)
xxxxxxxxxx addi (Wrongly executed and aborted)
0x00000490 brlid
0x00000494 or (Delay Slot)
xxxxxxxxxx lw (Wrongly executed and aborted)
0x00000440 lwi
```

Listing 7.3: Erweiterter Befehls-Trace mit Pipeline-Stalls, teilweise ausgeführten Befehlen und Delay-Slot Befehlen

Die beiden Pipeline-Stalls am Anfang von Listing 7.3 entstehen durch das Laden des Registers R15, welches im darauf folgenden „*Return from Subroutine with Delay Slot*“-Befehl (*rtsd*) als Sprungziel benötigt wird. Die beiden Befehle *addi* und *lw* sind Befehle die fälschlich teilweise ausgeführt werden. Die Befehle *addik* und *or* sind Befehle, die im Delay-Slot nach einem Sprung vollständig ausgeführt werden.

7.3.3 Ergebnisse

Als Test-Programme wurden zwei einfache in der Programmiersprache C geschriebene Programme verwendet. Das *Accumulate* Programm multipliziert eine Reihe von Zahlen und summiert die Ergebnisse auf. Es benötigt dafür 255 Takte inklusive 125 Takte für die Softwareinitialisierung.
Das zweite Test-Programm *GCD* berechnet den größten gemeinsamen Teiler zweier Zahlen, in diesem Fall 1747 und 488, und benötigt hierfür auf dem MicroBlaze 766 Takte inklusive 138 Takte für die Softwareinitialisierung.
Die Befehls-Traces der Test-Programme wurden von dem ACCPWR Programm, wie in Kapitel 7.3.2.2 beschrieben, analysiert, und es wurde für jeden Takt eine Energieabschätzung ausgegeben. Diese Einzelabschätzungen wurden für jedes Test-Programm aufsummiert und mit der Low-Level Abschätzung des gesamten Programms verglichen. Zudem wurden die Einzelenergiewerte auch mit den Werten der taktweisen Low-Level Abschätzung verglichen.
Dieses Verfahren wurde zum einen für die Energiewert-Datenbank mit

gleicher Verteilung von E_{ba} auf die Pipeline-Stufen und zum anderen für eine ebensolche mit einer Verteilung aus Erfahrungswerten durchgeführt. Zudem wurde aus den Befehls-Traces noch eine instruktionsbasierte Energieverbrauchsabschätzung nach dem Standardverfahren erstellt.

Test Programm	**ACCPWR (Erfahrungsw.)**	**ACCPWR (Gleichvert.)**	**Standard-verfahren**
Accumulate			
Gesamt	5,2%	5,8%	12,3%
Taktweise	16,0%	14,8%	23,9%
GCD			
Gesamt	6,1%	8,5%	13,4%
Taktweise	17,0%	17,1%	24,4%

Tabelle 7.11: Übersicht über die prozentualen Abweichungen der verschieden instruktionsbasierten Abschätzungsmethoden von der Low-Level Abschätzung: Einerseits ermittelt über das gesamte Programm, andererseits über die Mittelwerte der taktweisen Abweichungen

Die geringe Abweichung der taktweisen Abschätzung in Tabelle 7.11 kann jedoch nicht direkt mit den Ergebnissen aus Fallbeispiel 2 verglichen werden, da durch den höheren befehlsunabhängigen Teil der feste Anteil des Energieverbrauchs beim MicroBlaze höher und damit die prozentuale Dynamik geringer als bei der Jam CPU ist.
Der Vergleich der maximalen absoluten Abweichungen ist hier ein genauerer Indikator (Siehe Tabelle 7.12). Diese Abweichungen aus dem taktweisen Vergleich der Abschätzungen auf Verhaltensebene mit der Low-Level Abschätzung fallen, im Vergleich zu den Werten aus Tabelle 7.8 aus dem Fallbeispiel 2, etwa doppelt so hoch aus.

Test Programm	**ACCPWR (Erfahrungsw.)**	**ACCPWR (Gleichvert.)**	**Standard-verfahren**
Accumulate	1190pJ	1190pJ	1962pJ
GCD	1165pJ	1250pJ	1696pJ

Tabelle 7.12: Maximale absolute Abweichungen der Abschätzverfahren beim taktweisen Vergleich mit der Low-Level Abschätzung

Diese Abweichungen resultieren wie schon bei dem Jam CPU Prozessor hauptsächlich aus der Datenbitabhängigkeit. Zusätzliche Abweichungen von den Energieverbrauchswerten der Low-Level Abschätzung kommen hier aber noch durch die nicht so genaue Unterteilung der Energiewerte auf die einzelnen Pipeline-Stufen hinzu.
Bei der Charakterisierung wurden auch Untersuchungen bezüglich der Bitänderungen in den Instruktionswörtern zweier aufeinander folgender Befehle durchgeführt, jedoch war die Abhängigkeit bei diesem Prozessor, genau wie in Fallbeispiel 2, nur minimal.

7.3.3.1 Vergleich der Verfahren zur Energieverteilung auf die Pipeline-Stufen

Da der genaue innere Aufbau des MicroBlaze Prozessors unbekannt ist, sind die beiden hier betrachteten Energieverteilungen – die Gleichverteilung und die Verteilung nach Erfahrungswerten des Jam CPU Prozessors – nur Schätzungen. Bei der Abschätzung des Energieverbrauchs der gesamten Programme erhält man mit der Verteilung nach Erfahrungswerten bei beiden Test-Programmen ein etwas genaueres Ergebnis. Dies deutet darauf hin, dass diese Form der Unterteilung etwas besser der wahren Verteilung des Energieverbrauchs auf die Pipeline-Stufen entspricht, beziehungsweise, dass der Hauptteil des Energieverbrauchs auf den ersten drei Pipeline-Stufen liegt.
Beim taktgenauen Vergleich hat die Verteilung mit Erfahrungswerten beim *GCD* Programm nur einen geringen Vorteil. Beim *Accumulate* Programm ist der Mittelwert der prozentualen Abweichungen bei dieser Verteilung sogar größer. Eine Erklärung könnte sein, dass bei dieser Verteilung die EX Stufe mit 50% zu hoch angesetzt ist, und sich deshalb der Fehler durch die Datenbitabhängigkeit stärker bemerkbar macht. Das bedeutet, dass in Extremfällen, bei denen der Energieverbrauch aufgrund der zu verarbeitenden Daten sehr gering ist, der Verbrauch der EX Stufe mit 50% eine höhere Abweichung liefert als bei der Gleichverteilung, wo er mit 20% angesetzt wird.

7.3.3.2 Vergleich mit Standardverfahren

Die Ergebnisse zeigen, dass die Abschätzverfahren selbst mit einer geschätzten Unterteilung in die Pipeline-Stufen deutlich genauere Werte lie-

fern als die Abschätzung nach dem Standardverfahren, bei dem die Besonderheiten der Befehlsausführung in Pipelines nicht berücksichtigt werden. Das Standardverfahren der instruktionsbasierten Verlustleistungsabschätzung kann unter normalen Bedingungen beim taktgenauen Vergleich nicht mithalten, da der Trace des GDB nicht für jeden Takt einen Befehl ausgibt und einige Befehle und Pipeline-Stalls weglässt. Um einen einigermaßen fairen Vergleich mit dem Standardverfahren zu gewährleisten, werden diese Taktverschiebungen und weggelassenen Befehle auch beim Standardverfahren in die Berechnung einbezogen. Die Stall-Zyklen und die zwei Takte nach Branch Instruktionen (ausgenommen der Delay Slot) werden mit einem befehlsabhängigen Energieverbrauch von 0 angesetzt. Der statische und befehlsunabhängige Teil des Gesamtverbrauchs gehen für diese Takte aber in die Berechnung ein.
Das ACCPWR Abschätzungsverfahren ist trotzdem genauer als die normale instruktionsbasierte Verlustleistungsabschätzung. Sowohl bei dem gesamten Energieverbrauch, wie auch insbesondere bei der taktgenauen Analyse zeigen sich deutliche Verbesserungen der Abweichungen von dem Referenzwert der Low-Level Abschätzung.

7.3.3.3 Geschwindigkeit

Die Zeit zur Verlustleistungsabschätzung mit ACCPWR war bei beiden Energiewert-Verteilungsverfahren erwartungsgemäß gleich. Ein Vergleich der Zeiten der Low-Level Abschätzung mit ModelSim und XPower, der ACCPWR Abschätzung und der normalen instruktionsbasierten Abschätzung ist in Tabelle 7.13 gegeben.

Test-Programm	**Low-Level Abschätzung**	**ACCPWR**	**Standard-verfahren**
Accumulate	18m20s	1m14s	1m14s
GCD	54m59s	3m32s	3m32s

Tabelle 7.13: Die jeweilige Ausführungsdauer der einzelnen Abschätzungsmethoden für die Test-Programme

Da das hier verwendete Verfahren zur Erstellung der Befehls-Trace über das Stepping Verfahren mit dem GDB sehr viel Zeit in Anspruch genommen hat, ist der Geschwindigkeitsvorteil gegenüber der Low-Level Abschätzung nicht so groß wie bei Fallbeispiel 2, jedoch dauert diese immer

noch etwa 17 mal so lange.
Die Zeit zum Kompilieren des C-Programms und für die eigentliche Berechnung der Verlustleistung aus dem Befehls-Trace bleibt dabei jeweils unter einer Sekunde. Der Rest der Zeit wird zur Erstellung des Befehls-Traces durch den GDB und den XMD benötigt.
Auch hier war gegenüber der normalen instruktionsbasierten Abschätzung kein Nachteil bei der Ausführungsdauer feststellbar, da diese auch auf dem Befehls-Trace von GDB und XMD basiert.
Eine Verbesserung der Geschwindigkeit wäre hier über ein schnelleres Simulations- oder Trace-erzeugendes Profiling-Programm möglich.

8 Diskussion der Ergebnisse

Aus den drei Fallbeispielen wurden verschiedene generelle Erkenntnisse gewonnen, die hier nochmal genauer betrachtet werden.
Allgemein hat sich gezeigt, dass die hier vorgestellte Methode, mit der Unterteilung der Verlustleistung auf die Pipeline-Stufen, bei der Genauigkeit sehr nah an aufwendigere Methoden wie die modulbasierte oder die Low-Level Verlustleistungsabschätzung herankommt. Auch taktgenaue Abschätzungen sind hierdurch möglich. Die Geschwindigkeit der Abschätzung entspricht dabei der schnellen instruktionsbasierten Methode.

8.1 Effekt des Pipelinings auf die Verlustleistung

Generell verringert sich die Verlustleistung pro ausgeführter Instruktion für Prozessoren mit längerer Pipeline, weil die Fortpflanzung und Aufsummierung von Glitches durch das Pipelining verhindert wird. Jedoch kommen bei längeren Pipelines auch die Pipeline-Stalls und Flushes mehr zum tragen und erhöhen so die aufgenommene Leistung wieder.
Dass dies einen nicht zu vernachlässigenden Einfluss auf die Gesamtverlustleistung hat, wurde mit den drei Fallbeispielen deutlich gezeigt. Auch bei dem AVR-Core Prozessor mit einer einfachen zweistufigen Pipeline hat dieser Effekt, der hier durch die Pipeline-Flushes nach falsch vorhergesagten Sprüngen verursacht wird, einen deutlichen Einfluss.
Bei den komplexeren Pipelines mit zusätzlichen unterschiedlichen Arten von Pipeline-Stalls und Bubbling, bei denen auch fälschlich ausgeführte Befehle länger bearbeitet werden, bevor diese durch einen Pipeline-Flush abgebrochen werden, wird dieser Einfluss auf die Verlustleistung des Prozessors noch deutlicher und muss folglich bei der Abschätzung berücksichtigt werden.
Es ist sicher möglich, durch eine geschickte Charakterisierung diese Effekte zum Teil zu berücksichtigen, indem man von vornherein für Branch-

Befehle eine bestimmte Flush-Quote in den Energiewert dieser Instruktionen miteinberechnet. Damit kann man aber bestenfalls einen etwas genaueren Gesamtenergieverbrauch berechnen. Dieser liefert aber nur für solche Programme eine höhere Genauigkeit, die in etwa diese Quote an Pipeline-Flushes erfüllen. Für die taktgenaue Abschätzung bringt dieses Verfahren jedoch keine Vorteile.

8.2 Abschätzgenauigkeit der verschiedenen Methoden

Bei einfachen Prozessoren mit kurzen Pipelines, wenigen Pipeline-Stalls und einer statischen Sprungvorhersage kann das Verfahren aus Fallbeispiel 1 eine gute Verbesserung der Genauigkeit liefern. Ohne hohen Aufwand wurde im vorliegenden Fall fast eine Halbierung des Fehlers erreicht. So müssen nur die Branch-Instruktionen bei der Charakterisierung doppelt erfasst, und die zusätzliche Energie für den Pipeline-Flush bei solchen Branch Instruktionen hinzugerechnet werden, für welche der Sprung falsch vorhergesagt wird. Bei der Abschätzung muss außerdem zusätzlich die Anzahl der falsch vorhergesagten Sprünge bekannt sein.
Um bei komplexeren Pipeline-Strukturen noch gute Abschätzungen zu erreichen, ist jedoch eine volle Berücksichtigung des Pipelining unumgänglich. Der Aufwand bei der Charakterisierung und der damit verbundenen Modularisierung ist hier höher, und die Nachbildung des Ablaufs der Befehle in der Pipeline erfordert eine genaue Betrachtung des Prozessors.
Damit erreicht diese Methode bei der Abschätzung des gesamten Energieverbrauchs eine Reduktion der Abweichung, die bei allen untersuchten Programmen mehr als 60% gegenüber der normalen instruktionsbasierten Abschätzung betrug. Weiterhin liegt sie mit einer durchschnittlichen Abweichung von 17% bei dem taktweisen Vergleich auch in einem Bereich, der für die Abschätzung von „Hot-Spots“ im Code oder zur Optimierung einzelner Pipeline-Stufen gut geeignet ist.
Die Modularisierung des Prozessors ist teilweise jedoch nur schwer möglich oder, wie bei IP-Cores oder Prozessoren, welche in Hardware beim Anwender vorliegen, gänzlich unmöglich. In diesem Fall kann die Methode aus Fallbeispiel 3 eine Alternative bieten, die von der Genauigkeit zwar nicht an die modularisierte Charakterisierung herankommt, jedoch die Abweichung in der Gesamtenergieabschätzung immer noch mehr als halbiert.

Die Verteilung des befehlsabhängigen Energieverbrauchs auf die Pipeline-Stufen spielt dabei eine wichtige Rolle. Beide untersuchten Unterteilungen lieferten hier gute Ergebnisse, mit leichten Vorteilen bei der Unterteilung nach Erfahrungswerten. Für eine noch genauere Unterteilung der Energiewerte auf die Pipeline-Stufen könnten Hersteller von Prozessoren oder IP-Cores in den Spezifikationen Prozentwerte herausgegeben, ohne dass Informationen über den internen Aufbau der Stufen veröffentlicht werden müssen.

8.2.1 Befehlsbitabhängigkeit

Mit dem Korrekturfaktor für die zusätzliche Verlustleistung durch Bitänderungen der Instruktionswörter konnte für den strukturell einfachen AVR-Core Prozessor eine Verbesserung der Genauigkeit von etwa 40% erreicht werden. Bei den komplexeren Prozessoren konnte die Genauigkeit hierdurch jedoch nicht wesentlich gesteigert werden, da hier die Befehlsbitabhängigkeit nur die ersten Pipeline-Stufen betrifft, und der Energieverbrauch der restlichen Logik überwiegt.

8.2.2 Datenbitabhängigkeit

Die Bestimmung der Verlustleistung in Abhängigkeit der vom Prozessor verarbeiteten Daten ist für die instruktionsbasierten Abschätzungsverfahren noch eine Herausforderung. Deutlich zeigt dies das Fallbeispiel 1 mit dem Test-Programm *DIV*. Hier ist die Abhängigkeit der Verlustleistung von den Eingangsdaten am deutlichsten zu beobachten.
Auch bei Fallbeispiel 2 werden die Abweichungen beim taktgenauen Vergleich hauptsächlich durch den unterschiedlichen Energieverbrauch bei der Verarbeitung der Daten, mit denen die Instruktionen parametrisiert sind, verursacht.
Um diese Abweichungen möglichst gering zu halten, ist bei allen instruktionsbasierten Verlustleistungsabschätzverfahren eine besonders ausgewogene und eventuell an die Zielapplikation angepasste Charakterisierung der Instruktionen erforderlich.
Eine weitere Möglichkeit wäre es einen Min/Max/Avg Ansatz zu verwenden. Hierbei werden zusätzlich zum Durchschnittswert für jeden Takt der minimal und der maximal mögliche Wert berechnet. So erhält man einen

„Korridor“ aller rein theoretisch möglichen taktweisen Energieverbrauchswerte, der bei einer genaueren Analyse helfen kann.

8.3 Geschwindigkeit der Abschätzung

Die Berechnungszeit der normalen instruktionsbasierten Verlustleistungsanalyse ist mit dem hier vorgestellte Verfahren zur Berechnung des Energieverbrauchs nahezu identisch.
Gegenüber der Low-Level Abschätzung ist, sobald die Charakterisierung einmal durchgeführt wurde, der Geschwindigkeitsgewinn enorm. In Fallbeispiel 2 war die Abschätzung mit dem ACCPWR Verfahren etwa 180 mal schneller als die Low-Level Abschätzung.

8.4 Aufwand für die Charakterisierung und Anpassung

Der Aufwand für die Charakterisierung jeder Instruktion für alle Pipeline-Stufen ist aufwendiger als bei der normalen instruktionsbasierten Abschätzung – einerseits durch die Modularisierung des Quellcodes der Hardware und andererseits dann, wenn die Pipeline-Stufen einzeln charakterisiert werden müssen. Diese Charakterisierung muss aber nur einmal für jede Prozessorkonfiguration durchgeführt werden.
Eine Alternative ist die Methode aus Fallbeispiel 3. Durch die geschätzte Aufteilung der befehlsabhängigen Energie kann dieser Aufwand mit dem Nachteil der geringeren Genauigkeit praktisch vermieden werden. Die Ergebnisse zeigen, dass die Abschätzung mit diesem Verfahren immer noch deutlich genauer als das Standardverfahren ist. Für eine taktgenaue Abschätzung ist aber eine Modularisierung und genaue Charakterisierung nach Pipeline-Stufen empfehlenswert.
Auch muss der Mehraufwand für die Anpassung des Abschätzungsprogramms an die Pipeline-Struktur des Prozessors zur Nachbildung des Befehlsflusses berücksichtigt werden. Hier könnte jedoch das bisherige kommandozeilenbasierte ACCPWR Programm angepasst werden, so dass über eine grafische Oberfläche verschiedene Programmoptionen konfiguriert werden können. Diese Konfiguration könnte etwa umfassen, ab welcher Stufe Bubbling eingefügt wird und welcher Konflikt dieses verursacht, oder in

welcher Stufe Sprünge evaluiert werden und welche Methode der Sprungvorhersage verwendet wird.

8.5 Einsatzbereiche der neuen Abschätzmethode

Die Standardaufgabe von schnellen Energieverbrauchsabschätzungen ist die Bestimmung der Energie-Effizienz eines Programms oder einer Befehlsfolge, um diese eventuell auszutauschen oder optimieren zu können. Für diese Aufgabe eignet sich das ACCPWR Verfahren besser als die bisherigen Standardverfahren, da durch die taktgenaue Analyse genau festgestellt werden kann, welche Code-Segmente oder Funktionen sich für die Optimierung eignen.
Bei sehr häufigen und verlustreichen Pipeline-Flushes können durch Gegenmaßnahmen, wie z.B. dem *Loop Unrolling*, bei dem Schleifen beim Kompilieren so weit wie möglich aufgelöst werden, Sprünge und damit einhergehende Pipeline-Flushes reduziert werden.
Mit diesem Verfahren ist es auch möglich, so genannte „Hot-Spots“ im Code zu ermitteln, also Befehlsfolgen, die besonders viel Energie verbrauchen. Wenn man diese durch eingefügte *nop* Befehle oder erzwungene Pipeline-Stalls unterbricht, kann man eine Erhitzung des Prozessors während dieser Programmabschnitte verhindern und damit zum Beispiel auf zusätzliche Kühlungsmaßnahmen verzichten.
Dieses Verfahren ist auch für Hardwarehersteller interessant, um den Energieverbrauch der Hardware eines Prozessors an bestimmte Einsatzgebiete anpassen zu können. So kann sich bei Prozessoren, die hauptsächlich „idle“ sind und auf kurze Ereignisse warten, auf die Minimierung der befehlsunabhängigen Verlustleistung konzentriert werden. Bei Prozessoren, die meist unter Volllast arbeiten, kann der befehlsabhängige Energieverbrauch optimiert werden.
Durch das neue Abschätzverfahren können verschiedene Pipeline-Strukturen allgemein oder für eine bestimmte Aufgabe untersucht werden. Dies kann zur Anpassung von Bubbling Verfahren oder zur Optimierung von Pipeline-Flushes von Nutzen sein, indem zum Beispiel Sprünge in der Pipeline in einer früheren Stufe evaluiert werden. Weiterhin ist es möglich, die Stufen einer Pipeline auf bestimmte Befehlssätze hin zu optimieren, oder einzelne Pipeline-Stufen gegen Stufen anderer Architektur auszutau-

schen.
Durch Aufsummierung der Energiewerte für nur eine einzige Pipeline-Stufe über den Verlauf einer Befehlsfolge hinweg kann für jede Pipeline-Stufe separat der Energieverbrauch ermittelt werden. So kann etwa Logik aus einer Pipeline-Stufe in eine spätere Stufe verschoben werden, um den Energieverlust bei Pipeline-Flushes zu minimieren. Außerdem kann der Einfluss des Hinzufügens von anderen Hardware-Funktionseinheiten zu den einzelnen Stufen separat untersucht werden.
Für das Hardware/Software-Codesign kann diese Methode auch als Entscheidungshilfe dienen, welche Teile eines Systems in Hardware und welche in Software ausgeführt werden sollten, um einen vorgegebenen Energieverbrauch zu erreichen.
Gerade die Aufgaben zur Hardwareoptimierung waren bisher immer Low-Level Abschätzungen vorbehalten. Durch das ACCPWR Verfahren sind hier auch schon in frühen Design-Stadien schnelle Untersuchungen möglich, ohne viel an Genauigkeit einbüßen zu müssen.

9 Zusammenfassung und Ausblick

9.1 Zusammenfassung

In dieser Arbeit wurde eine neue Methode zur Abschätzung der Verlustleistung von Prozessoren mit Pipelines vorgestellt. Diese Methode verbessert die Genauigkeit der instruktionsbasierten Abschätzung deutlich, ohne jedoch die Zeit der Abschätzung zu verlängern. Ebenso ist es mit ihr möglich, taktgenaue Energieverbrauchsabschätzungen von Prozessoren mit Pipelines durchzuführen.
Hierfür wurden in Kapitel 2 die Grundlagen der Entstehung der Verlustleistung und deren Abschätzung beschrieben. Es wurden die bisher verwendeten Verfahren zur Abschätzung der Verlustleistung von Prozessoren vorgestellt, ein Überblick über die aktuelle Forschung dazu gegeben und diese bezüglich ihrer Tauglichkeit zur Abschätzung der Verlustleistung von Pipeline-Strukturen bewertet.
In Kapitel 3 wurde das Pipelining genauer untersucht und dabei speziell die besonderen Folgen von Pipeline-Hazards und möglichen Gegenmaßnahmen auf den Befehlsfluss in der Pipeline analysiert. Im Fokus standen hier die für die Verlustleistungsabschätzung bedeutenden Besonderheiten wie Pipeline-Stalls, Bubbling und Pipeline-Flushes. Weiterhin wurden verschiedene Strukturen von Pipelines vorgestellt, welche in gängigen Prozessoren vorkommen.
Für die in dieser Arbeit vorgestellte Methode wurde die Verlustleistung der Pipeline in die Verlustleistungen der Pipeline-Stufen zerlegt. Wie dies geschieht, und wie daraus wieder die gesamte Verlustleistung einer Prozessor-Pipeline mit einem normalen Befehlsfluss berechnet werden kann, wurde in Kapitel 4 vorgestellt.
In Kapitel 5 entstand daraus dann ein allgemein gültiges Modell zur Berechnung des Energieverbrauchs eines solchen Prozessors. Es wurden die verschiedenen Sonderfälle, die durch Pipelining von Prozessoren bei der

Befehlsabarbeitung auftreten, und unterschiedliche Pipeline-Strukturen behandelt. Für diese wurden allgemeine Formeln für die Energieberechnung aufgestellt.
Welche Schritte zur realen Durchführung einer Verlustleistungsabschätzung nach dieser neuen Methode notwendig sind, wurde in Kapitel 6 gezeigt. Dazu wurden der Aufbau und die Erstellung der dafür notwendigen Datenbank mit Energiewerten für jede Pipeline-Stufe beschrieben. Um das Verfahren zur Abschätzung der Verlustleistung zu zeigen, wurde die Erstellung eines erweiterten Befehls-Traces und die Nachbildung des Befehlsflusses in der Pipeline erklärt. Zusätzlich wurde aufgezeigt, welches Verfahren zur Validierung der Ergebnisse aus den Fallbeispielen verwendet wurde.
Die Durchführung von drei Fallbeispielen zum Nachweis der in dieser Arbeit vorgestellten Methode wurde dann in Kapitel 7 beschrieben. Fallbeispiel 1 zeigte an dem AVR-Core Prozessor den Einfluss von Branch Instruktionen und speziell den Einfluss von Pipeline-Flushes auf den Energieverbrauch des Prozessors. Die genaue Charakterisierung eines nach Pipeline-Stufen modularisierten Prozessors und die stufenweise, bzw. taktgenaue Abschätzung wurde in Fallbeispiel 2 an dem Jam CPU Prozessor gezeigt. Die Abschätzung eines IP-Cores, dessen innerer Aufbau nicht bekannt ist, konnte mit einer Methode, wie sie in Fallbeispiel 3 anhand des MicroBlaze Prozessors beschrieben wurde, durchgeführt werden.
Die Ergebnisse aus diesen Fallbeispielen wurden in Bezug auf Genauigkeit und Geschwindigkeit der Abschätzung in Kapitel 8 diskutiert. Es hat sich gezeigt, dass die Genauigkeit der neuen Methode gegenüber der bisherigen instruktionsbasierten Methode deutlich verbessert werden konnte. Auch die Abweichung bei der taktgenauen Abschätzung wurde durch die vorgestellte Methode in einen Bereich gebracht, der eine genauere Analyse der Energieverbrauchsentwicklung über den Ablauf einer Befehlsfolge auf einem Prozessor hinweg ermöglicht und so für die meisten Anwendungsbereiche gut geeignet ist. Einbußen bei der Geschwindigkeit der Abschätzung konnten dabei nicht festgestellt werden.
Weiterhin wurden noch potentielle Einsatzbereiche der in dieser Arbeit vorgestellten Methode aufgezeigt und dabei die Vorteile gegenüber den bisherigen Methoden hervorgehoben.

9.2 Ausblick

Das hier vorgestellte Verfahren hat mit den Low-Level charakterisierten Werten im Vergleich mit der Low-Level Abschätzung der untersuchten Befehlsfolgen stets nur geringe Abweichungen gezeigt. Interessant wären hier Untersuchungen mit echten Messungen, wobei auch die Charakterisierung durch Messung nach den in Kapitel 6.2.1.2 vorgeschlagenen Methoden erfolgen könnte.
Für das Fallbeispiel 3, bei dem die Aufteilung der Logik des Prozessors auf die Pipeline-Stufen unbekannt war, könnten Methoden entwickelt werden, mit denen man auf eine genauere Verteilung des Energieverbrauchs auf die Pipeline-Stufen schließen könnte. Es wäre z.B. durch taktweise Messungen eines einzelnen, die Pipeline durchlaufenden, Befehls möglich, diese Einzelwerte der Pipeline-Stufen zu charakterisieren.
Schließlich wäre eine Übertragung dieses Verfahrens auf andere Pipelines denkbar. Hier kämen z.B. Pixel-Pipelines in Betracht, da mittlerweile gerade Grafikkarten den Prozessor beim Energieverbrauch zum Teil schon überholen. Auch ASICs, z.B. zur Dekodierung von hochauflösenden Videos in Handys, würden sich gut für eine Optimierung mit diesem Verfahren eignen.

Abkürzungs- und Variablenverzeichnis

$A_{BR,fl}$	Anzahl der Aufrufe des Sprungbefehls BR in einer Befehlsfolge, welche falsch vorhergesagt wurden und einen Pipeline-Flush verursachen
$A_{BR,ges}$	Anzahl aller Aufrufe des Sprungbefehls BR in einer Befehlsfolge
$A_{BR,nf}$	Anzahl der Aufrufe des Sprungbefehls BR in einer Befehlsfolge, welche richtig vorhergesagt wurden und keinen Pipeline-Flush verursachen
ACCPWR	einerseits die in dieser Arbeit vorgestellte neuartige Methode zur Abschätzung der Verlustleistung; andererseits das Programm zur Berechnung der Verlustleistung nach der ACCPWR Methode
ASIC	Application Specific Integrated Circuit, für eine bestimmte Anwendung ausgelegte integrierte Schaltung
CMOS	Complementary Metal–Oxide–Semiconductor, setzt sich zusammen aus p-Kanal Transistoren (p-MOS) und n-Kanal Transistoren (n-MOS)
DO-Script	Batch Datei zur automatisierten Steuerung des Simulationsprogramms ModelSim
E_{ba}	befehlsabhängiger Teil der dynamischen Energie, welche vom Prozessors für die Abarbeitung einer Befehlsfolge in der Zeit t verbraucht wird
E_{bu}	befehlsunabhängiger Teil der dynamischen Energie, welche vom Prozessors für die Abarbeitung einer Befehlsfolge in der Zeit t verbraucht wird
E_{dyn}	dynamische Energie, die vom Prozessors für die Abarbeitung einer Befehlsfolge in der Zeit t verbraucht wird
E_{ges}	gesamte Energie, die vom Prozessors für die Abarbeitung einer Befehlsfolge in der Zeit t verbraucht wird

$E_{ps}(I_i, J_j)$	dynamische Energie, welche in der j-ten Pipeline-Stufe in einem Takt verbraucht wird, wenn der Befehl I_i ausgeführt wird
E_{stat}	statische Energie, die vom Prozessors für die Abarbeitung einer Befehlsfolge in der Zeit t verbraucht wird
EDK	Embedded Development Kit von Xilinx, Entwicklungsumgebung zur Konfiguration, Synthese, Implementierung und Simulation des Xilinx MicroBlaze IP-Core Prozessors
ELF-Datei	eine ausführbare Datei, welche vom Compiler für den Zielprozessor aus dem Quellcode erzeugt wird
EX Stufe	Execute Stufe einer Pipeline
f_{clk}	Taktfrequenz der Pipeline-Struktur
FPGA	Field Programmable Gate Array, integrierter Halbleiterbaustein, dessen Logik und Verdrahtung noch im Endgerät vom Anwender verändert werden können
GDB	GNU Debugger, Teil der GNU Compiler Collection (GCC)
I	Instruktion
I_i	Befehl I einer Instruktionsfolge an der Stelle i
I_s	Sättigungsstrom einer Diode in Sperrrichtung
ID Stufe	Instruction Decode Stufe einer Pipeline
IF Stufe	Instruction Fetch Stufe einer Pipeline
IP-Core	Intellectual Property Core, ein Prozessor, der als eine fertige Netzliste für eine Ziel-Hardware vorliegt und nur beschränkt verändert werden kann
ISE	(eigentlich Xilinx ISE) Entwicklungsumgebung vom FPGA Hersteller Xilinx zur Synthese, Implementierung und Simulation von VHDL oder Verilog Dateien
J	Pipeline-Stufe
J_j	j-te Pipeline-Stufe
JTAG	Hardware-Schnittstelle zum Programmieren und Debuggen eines Prozessors
k	Boltzmann-Konstante
m	Anzahl der Befehle, die auf der Pipeline-Struktur abgearbeitet werden

mc	Anzahl der Taktzyklen, die eine Multi-Cycle Instruktion zusätzlich benötigt
MEM Stufe	Memory Access Stufe einer Pipeline
MOS-FET	metal oxide semiconductor field-effect transistor, Feldeffekttransistor, der hauptsächlich in integrierten Schaltungen verwendet wird
n	Anzahl der Pipeline-Stufen
NCD-Datei	Netzliste, welche durch Synthetisieren und Implementieren eines Quellcodes in einer hardware-Beschreibungssprache erzeugt wird
P_{ba}	befehlsabhängiger Teil der dynamischen Verlustleistung
P_{bu}	befehlsunabhängiger Teil der dynamischen Verlustleistung
P_{dyn}	mittlere dynamische Verlustleistung
P_{ges}	mittlere Verlustleistung des Prozessors für die Abarbeitung einer Befehlsfolge
$P_{ps}(J_j, I_i)$	Verlustleistung des ausgeführten Befehls I_i in der Pipeline-Stufe J_j
P'_{ps}	die Verlustleistung der jeweiligen Pipeline-Stufe ps ($ps \epsilon \{IF, ID, EX, MEM, WB\}$), welche aus der hierarchischen Ansicht des Programms XPower übernommen wurde
P_{stat}	mittlere statische Verlustleistung
PC	programm counter, Befehlszähler, Register welches die Adresse des aktuell zu ladenden Befehls enthält
psx	Folgenummer der Pipeline-Stufe die den Befehl verarbeitet, welcher einen Konflikt auslöst und deshalb warten muss. Nach dieser Pipeline-Stufe werden Bubbles eingefügt
RC-Glied	Ein RC-Glied ist eine Schaltung aus einem Widerstand R und einem Kondensator C, die zusammen einen Tiefpass 1. Ordnung bilden
s	Anzahl der Taktzyklen, während derer sich die Pipeline im „Stall“-Zustand befindet
Soft-Core	Prozessor, welcher als Quellcode in einer Hardware-Beschreibungssprache vorliegt, somit verändert und für eine beliebige Hardware-Implementierung verwendet werden kann

t	Zeit für die Abarbeitung einer Befehlsfolge $I_{1:m}$ mit der Taktfrequenz f_{clk}
U_{gs}	Spannung zwischen Gate- und Source-Anschluss eines MOS-FET Transistors
U_{th}	Schwellspannung, ab der ein Transistor den Kanal zwischen Drain- und Source-Anschluss öffnet, bzw. schließt
VCD-Datei	Value Change Dump Datei, wird bei der Low-Level Simulation erzeugt und enthält alle Signaländerungen, welche in einem Hardware-Design während der Simulation auftreten
Verilog	Hardware-Beschreibungssprache
VHDL	Hardware-Beschreibungssprache
W	Menge der Instruktionen, die auf dem Prozessor ausführbar sind
WB Stufe	Write Back Stufe einer Pipeline
XMD	Xilinx MicroBlaze Debugger, Debugging-Programm von Xilinx, welches den internen Befehlsfluss des Xilinx MicroBlaze IP-Core Prozessors simuliert
XPS	Xilinx Plattform Studio, Hauptprogramm des Xilinx EDK zur Konfiguration des Xilinx MicroBlaze IP-Core Prozessors
$Z_{BR,ges}$	Anzahl der Taktzyklen die alle aufgerufenen *BR*-Befehle in einer Befehlsfolge benötigt haben
$z_{BR,nf}$	Anzahl der Taktzyklen die ein einziger Aufruf des Befehls *BR* benötigt, wenn er ohne Pipeline-Flush abläuft
$z_{BR,fl}$	Anzahl der Taktzyklen die ein einziger Aufruf des Befehls *BR* benötigt, wenn er mit einem Pipeline-Flush ausgeführt wird.

Literaturverzeichnis

[1] Austin, Todd, Eric Larson und Dan Ernst: *SimpleScalar: an infrastructure for computer system modeling.* IEEE Computer, 35(2):59–67, Feb 2002.

[2] Barroso, Luiz Andre und Urs Hölzle: *The Case for Energy-Proportional Computing.* IEEE Computer, 40(12):33–37, 2007.

[3] Brooks, David, Vivek Tiwari und Margaret Martonosi: *Wattch: a framework for architectural-level power analysis and optimizations.* In: *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, Seiten 83–94, New York, NY, USA, 2000. ACM.

[4] Chaudhry, Rajat, Daniel Stasiak, Stephen Posluszny und Sang Dhong: *A cycle accurate power estimation tool.* In: *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, Seiten 867–870, New York, NY, USA, 2006. ACM Press.

[5] Fischer, Robert, Klaus Buchenrieder und Ulrich Nageldinger: *Reducing the Power Consumption of FPGAs through Retiming.* In: *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, Seiten 89–94, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Flynn, Michael J.: *Computer Architecture: Pipelined and Parallel Processor Design.* Jones and Bartlett Publishers, Inc., USA, 1995.

[7] Glökler, Tilman und Heinrich Meyr: *Design of Energy-Efficient Application-Specific Instruction Set Processors.* Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[8] Gorath, Torsten: *LEMOS: Low-Power - Entwurfsmethoden für mobile Systeme*, August 2007. `http://lemos.offis.de`.

[9] Gorey, Aengus: *Einblicke in Intels Atom-Low-Power-Architektur*. Elektronik-Praxis, Feb. 2009. `http://www.elektronikpraxis.vogel.de/themen/hardwareentwicklung/mikrocontrollerprozessoren/articles/167946/`.

[10] Gschwind, Michael: *The Cell project at IBM Research*, 2005. `http://www.research.ibm.com/cell/`.

[11] Hartstein, Allan und Thomas R. Puzak: *The optimum pipeline depth for a microprocessor*. In: *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, Seiten 7–13, Washington, DC, USA, 2002. IEEE Computer Society.

[12] Hartstein, Allan und Thomas R. Puzak: *Optimum Power/-Performance Pipeline Depth*. In: *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, Seite 117, Washington, DC, USA, 2003. IEEE Computer Society.

[13] Hsieh, Cheng-Ta, Lung sheng Chen und Massoud Pedram: *Microprocessor power analysis by labeled simulation*. In: *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, Seiten 182–189, Piscataway, NJ, USA, 2001. IEEE Press.

[14] Hsieh, Cheng-Ta und Massoud Pedram: *Microprocessor power estimation using profile-driven program synthesis*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17(11):1080–1089, Nov. 1998. uses a synthesized replacement program to estimate the Power; takes pipeline stalls into account; uses superscalar pipelined architecture. Does not cover speculative execution (yet).

[15] Hwang, Kai: *Advanced Computer Architecture: Parallelism, Scaability, Programmability*. McGraw-Hill, 1993.

[16] Jiang, Xiangfeng: *Power Modelling of generic FPGA-Architectures*. Diplomarbeit, RWTH Aachen, 2006.

[17] Kalla, Praveen, Jörg Henkel und Xiaobo Sharon Hu: *SEA: fast power estimation for micro-architectures*. In: *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*, Seiten 600–605, New York, NY, USA, Jan. 2003. ACM Press.

[18] Landsiedel, Olaf, Klaus Wehrle und Stefan Götz: *Accurate prediction of power consumption in sensor networks*. In: *EmNets*

'05: Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors, Seiten 37–44, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Lee, Hyung Gyu, Sungyuep Nam und Naehyuck Chang: *Cycle-accurate Energy Measurement and High-Level Energy Characterization of FPGAs.* In: *ISQED '03: Proceedings of the 4th International Symposium on Quality Electronic Design*, Seite 267, Washington, DC, USA, 2003. IEEE Computer Society.

[20] Lee, Wen-Chin und Chenming Hu: *Modeling CMOS tunneling currents through ultrathin gate oxide due to conduction- and valence-band electron and hole tunneling.* IEEE Transactions on Electron Devices, 48(7):1366–1373, Jul 2001.

[21] Lepetenok, Ruslan: *AVR-Core.* OpenCores.org, 2008. `http://www.opencores.org/project,avr_core`.

[22] Matthiesen, Nils: *Der Stromfresser unterm Schreibtisch.* Spiegel online, März 2007. `http://www.spiegel.de/netzwelt/tech/0,1518,469622,00.html`.

[23] Musovic, Adisa, Robert Fischer, Axel Lehmann, Klaus Buchenrieder und Ulrich Nageldinger: *An EQN* Based Approach for High-Level Performance and Power Estimation.* In: *CMS 2005: Conceptual Modeling and Simulation Conference*, Seiten 193–198, 2005.

[24] Ou, Jingzhao und Viktor K. Prasanna: *Rapid energy estimation of computations on FPGA based soft processors.* In: *IEEE International SoC Conference*, 2004.

[25] Patt, Yale N. und Sanjay J. Patel: *Introduction to computing systems from bits and gates to C and beyond.* McGraw-Hill, Boston, 2. Auflage, 2004.

[26] Reimer, Axel, Arne Schulz und Wolfgang Nebel: *Modelling macromodules for high-level dynamic power estimation of FPGA-based digital designs.* In: *ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design*, Seiten 151–154, New York, NY, USA, 2006. ACM.

[27] Schütz, Martin: *Presseinformation der Bayerischen Akademie der Wissenschaften vom 21.07.2006.* `http://www.`

lrz-muenchen.de/wir/einweihungsfeier/medienberichte/PM_BAdW_LRZ-Einweihung.pdf.

[28] Shang, Li, Alireza S. Kaviani und Kusuma Bathala: *Dynamic power consumption in Virtex-II FPGA family.* In: *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, Seiten 157–164. ACM Press, 2002.

[29] Stallings, William: *Computer Organization and Architecture: Designing for Performance.* Pearson Prentice Hall, USA, 7th Auflage, 2006.

[30] Thelin, Johan Eriksson, Anders Lindström und Michael Nordseth: *Concert'02 Architecture Specification and Implementation.* Chalmers University of Technology, März 2002. http://digitalfanatics.org/index.php?title=JAM_CPU.

[31] Titzer, Ben L., Daniel K. Lee und Jens Palsberg: *Avrora: scalable sensor network simulation with precise timing.* In: *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, Seite 67, Piscataway, NJ, USA, 2005. IEEE Press.

[32] Tsividis, Yannis: *Operation and modeling of the MOS transistor.* McGraw-Hill series in electrical engineering : electronics and electronic circuits. McGraw-Hill, New York [u.a.], 3.print. Auflage, 1988.

[33] Weiss, Karlheinz, Carsten Oetker, Igor Katchan, Thorsten Steckstor und Wolfgang Rosenstiel: *Power estimation approach for SRAM-based FPGAs.* In: *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, Seiten 195–202. ACM Press, 2000.

[34] Weste, Neil H. E. und Kamran Eshraghian: *Principles of CMOS VLSI design: a systems perspective.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[35] Windschiegl, Armin: *Prognose von Leitungskapazitäten zur Verlustleistungsanalyse auf Logikebene.* Doktorarbeit, TU München, Nov. 2004.

[36] Xilinx: *MicroBlaze Processor Reference Guide*, v8.0 Auflage, 2007. http://www.xilinx.com/support/documentation/sw_manuals/edk81i_mb_ref_guide.pdf.

[37] Xilinx: *Spartan-3 Generation FPGA User Guide (UG331)*. v1.5 Auflage, Jan. 2009. `http://www.xilinx.com/support/documentation/user_guides/ug331.pdf`.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Anhang A: Charakterisierungsergebnisse

A.1 AVR-Core

Instruk-tion	nb/ br	E (pJ)	Takte	Instruk-tion	nb/ br	E (pJ)	Takte
BRBC	nb	2,67	1	BRLO	nb	2,66	1
BRBC	br	4,96	2	BRLO	br	4,87	2
BRBS	nb	2,66	1	BRLT	nb	2,66	1
BRBS	br	4,87	2	BRLT	br	4,87	2
BRCC	nb	2,67	1	BRMI	nb	2,66	1
BRCC	br	4,96	2	BRMI	br	4,87	2
BRCS	nb	2,66	1	BRNE	nb	2,67	1
BRCS	br	4,87	2	BRNE	br	4,96	2
BREQ	nb	2,66	1	BRPL	nb	2,67	1
BREQ	br	4,87	2	BRPL	br	4,96	2
BRGE	nb	2,67	1	BRSH	nb	2,67	1
BRGE	br	4,96	2	BRSH	br	4,96	2
BRHC	nb	2,67	1	BRTC	nb	2,67	1
BRHC	br	4,96	2	BRTC	br	4,96	2
BRHS	nb	2,66	1	BRTS	nb	2,66	1
BRHS	br	4,87	2	BRTS	br	4,87	2
BRID	nb	2,67	1	BRVC	nb	2,67	1
BRID	br	4,96	2	BRVC	br	4,96	2
BRIE	nb	2,66	1	BRVS	nb	2,66	1
BRIE	br	4,87	2	BRVS	br	4,87	2

Tabelle A.1: Die Energiewerte der bedingten Instruktionen für „no branch“ (nb), bei dem kein Pipeline-Flush erfolgt und für „branch“ (br), der zusätzlich noch die Energie für einen Pipeline-Flush beinhaltet

Instruktion	**E(pJ)**	**Takte**	**Instruktion**	**E(pJ)**	**Takte**
ADD	5,50	1	MOV	3,82	1
ADC	5,57	1	NEG	4,63	1
ADIW	8,02	2	NOP	2,49	1
AND	3,87	1	OR	5,30	1
BCLR	2,67	1	OUT	3,29	1
BSET	2,67	1	POP	3,40	2
CALL	4,42	4	PUSH	3,40	2
CLC	2,67	1	RCALL	4,10	3
CLH	2,67	1	RET	4,10	4
CLI	2,67	1	RJMP	7,32	2
CLN	2,67	1	ROL	6,22	1
CLR	4,57	1	ROR	3,90	1
CLS	2,67	1	SBC	6,91	1
CLT	2,67	1	SBIW	10,81	2
CLV	2,67	1	SEC	2,67	1
CLZ	2,67	1	SEH	2,67	1
CP	6,02	1	SEI	2,67	1
CPC	5,44	1	SEN	2,67	1
CPI	3,82	1	SER	4,04	1
DEC	5,47	1	SES	2,67	1
ELPM	5,75	3	SET	2,67	1
EOR	4,57	1	SEV	2,67	1
IN	3,94	1	SEZ	2,67	1
INC	4,68	1	ST	5,48	2
JMP	4,80	3	STD	5,05	2
LD	6,76	2	SUB	6,80	1
LDD	5,99	2	SUBI	5,55	1
LDI	4,04	1	TST	4,34	1
LPM	5,75	3			

Tabelle A.2: Die Energiewerte der charakterisierten unbedingten Instruktionen des AVR-Core Prozessors

A.2 Jam CPU

Instruktion	E_bu(fJ)	E_if(fJ)	E_id(fJ)	E_ex(fJ)	E_mem(fJ)	E_wb(fJ)
add	160000	102703	261266	411312	35570	4509
addd	160000	123019	203815	377918	31537	4431
addi	160000	123019	203815	377918	31537	4431
addx	160000	123019	203815	377918	31537	4431
and	160000	53222	330995	311487	15267	1908
beq	160000	115307	347599	472974	31643	2877
beq1	160000	94456	250524	190201	6655	644
bne	160000	107296	241679	184356	5879	630
bne1	160000	115977	358895	487785	33380	2924
cmp	160000	59404	197795	520779	16281	220
cmpd	160000	57099	150143	420646	1027	205
cmpi	160000	57099	150143	420646	1027	205
cmpx	160000	57099	150143	420646	1027	205
jump	160000	59150	111520	109416	22210	2104
jumpd	160000	59150	111520	109416	22210	2104
jumpx	160000	59150	111520	109416	22210	2104
lw	160000	63155	68862	183948	4375	380
mem_stall	160000	0	0	0	0	0
nop	160000	0	0	0	0	0
or	160000	62160	313284	417961	40528	4227
ori	160000	77819	291054	377065	35070	4352
reset	160000	36072	65755	276829	3092	412
reseti	160000	35945	14504	279358	2943	210
resetx	160000	35945	14504	279358	2943	210
set	160000	36072	65755	276829	3092	412
seti	160000	35945	14504	279358	2943	210
setx	160000	35945	14504	279358	2943	210
shzi	160000	111032	328699	585281	16050	3738
sw	160000	54685	139392	111299	12224	0

Tabelle A.3: Die charakterisierten Instruktionen der Jam CPU auf einem Spartan-3 FPGA

A.3 MicroBlaze

Instr.	E_bu(fJ)	E_if(fJ)	E_id(fJ)	E_ex(fJ)	E_mem(fJ)	E_wb(fJ)	E_ba(fJ)
ADD	1084080	230496	230496	230496	230496	230496	1152480
ADDC	1084080	230496	230496	230496	230496	230496	1152480
ADDI	1084080	218544	218544	218544	218544	218544	1092720
ADDIC	1084080	218544	218544	218544	218544	218544	1092720
ADDIK	1084080	218544	218544	218544	218544	218544	1092720
ADDIKC	1084080	218544	218544	218544	218544	218544	1092720
ADDK	1084080	230496	230496	230496	230496	230496	1152480
ADDKC	1084080	230496	230496	230496	230496	230496	1152480
AND	1084080	165264	165264	165264	165264	165264	826320
ANDI	1084080	144144	144144	144144	144144	144144	720720
ANDN	1084080	165264	165264	165264	165264	165264	826320
ANDNI	1084080	144144	144144	144144	144144	144144	720720
BEQ	1084080	42768	42768	42768	42768	42768	213840
BEQD	1084080	42768	42768	42768	42768	42768	213840
BEQI	1084080	115248	115248	115248	115248	115248	576240
BEQID	1084080	115248	115248	115248	115248	115248	576240
BGE	1084080	42768	42768	42768	42768	42768	213840
BGED	1084080	42768	42768	42768	42768	42768	213840
BGEI	1084080	115248	115248	115248	115248	115248	576240
BGEID	1084080	115248	115248	115248	115248	115248	576240
BGT	1084080	42768	42768	42768	42768	42768	213840
BGTD	1084080	42768	42768	42768	42768	42768	213840
BGTI	1084080	115248	115248	115248	115248	115248	576240
BGTID	1084080	115248	115248	115248	115248	115248	576240
BLE	1084080	42768	42768	42768	42768	42768	213840
BLED	1084080	42768	42768	42768	42768	42768	213840
BLEI	1084080	115248	115248	115248	115248	115248	576240
BLEID	1084080	115248	115248	115248	115248	115248	576240
BLT	1084080	42768	42768	42768	42768	42768	213840
BLTD	1084080	42768	42768	42768	42768	42768	213840
BLTI	1084080	115248	115248	115248	115248	115248	576240
BLTID	1084080	115248	115248	115248	115248	115248	576240
BNE	1084080	42768	42768	42768	42768	42768	213840
BNED	1084080	42768	42768	42768	42768	42768	213840
BNEI	1084080	115248	115248	115248	115248	115248	576240
BNEID	1084080	115248	115248	115248	115248	115248	576240
BR	1084080	78576	78576	78576	78576	78576	392880
BRA	1084080	78576	78576	78576	78576	78576	392880
BRAD	1084080	78576	78576	78576	78576	78576	392880
BRAI	1084080	78576	78576	78576	78576	78576	392880
BRAID	1084080	78576	78576	78576	78576	78576	392880
BRALD	1084080	78576	78576	78576	78576	78576	392880
BRALID	1084080	78576	78576	78576	78576	78576	392880
BRD	1084080	78576	78576	78576	78576	78576	392880
BRI	1084080	78576	78576	78576	78576	78576	392880
BRID	1084080	78576	78576	78576	78576	78576	392880
BRLD	1084080	78576	78576	78576	78576	78576	392880
BRLID	1084080	78576	78576	78576	78576	78576	392880
CMP	1084080	230496	230496	230496	230496	230496	1152480
CMPU	1084080	230496	230496	230496	230496	230496	1152480
IMM	1084080	212640	212640	212640	212640	212640	1063200
LBU	1084080	109200	109200	109200	109200	109200	546000

wird fortgesetzt

Fortsetzung

Instr.	E_bu(fJ)	E_if(fJ)	E_id(fJ)	E_ex(fJ)	E_mem(fJ)	E_wb(fJ)	E_ba(fJ)
LBUI	1084080	107472	107472	107472	107472	107472	537360
LHU	1084080	109200	109200	109200	109200	109200	546000
LHUI	1084080	107472	107472	107472	107472	107472	537360
LW	1084080	109200	109200	109200	109200	109200	546000
LWI	1084080	107472	107472	107472	107472	107472	537360
MFS	1084080	0	0	0	0	0	0
MSRCLR	1084080	0	0	0	0	0	0
MSRSET	1084080	0	0	0	0	0	0
MTS	1084080	0	0	0	0	0	0
OR	1084080	165264	165264	165264	165264	165264	826320
ORI	1084080	144144	144144	144144	144144	144144	720720
RSUB	1084080	230496	230496	230496	230496	230496	1152480
RSUBC	1084080	230496	230496	230496	230496	230496	1152480
RSUBI	1084080	218544	218544	218544	218544	218544	1092720
RSUBIC	1084080	218544	218544	218544	218544	218544	1092720
RSUBIK	1084080	218544	218544	218544	218544	218544	1092720
RSUBIKC	1084080	218544	218544	218544	218544	218544	1092720
RSUBK	1084080	230496	230496	230496	230496	230496	1152480
RSUBKC	1084080	230496	230496	230496	230496	230496	1152480
RTSD	1084080	78576	78576	78576	78576	78576	392880
SB	1084080	2592	2592	2592	2592	2592	12960
SBI	1084080	2352	2352	2352	2352	2352	11760
SEXT16	1084080	120528	120528	120528	120528	120528	602640
SEXT8	1084080	120528	120528	120528	120528	120528	602640
SH	1084080	2592	2592	2592	2592	2592	12960
SHI	1084080	2352	2352	2352	2352	2352	11760
SRA	1084080	0	0	0	0	0	0
SRC	1084080	0	0	0	0	0	0
SRL	1084080	0	0	0	0	0	0
SW	1084080	2592	2592	2592	2592	2592	12960
SWI	1084080	2352	2352	2352	2352	2352	11760
XOR	1084080	165264	165264	165264	165264	165264	826320
XORI	1084080	144144	144144	144144	144144	144144	720720

Tabelle B.4.: Die dynamischen Energiewerte der charakterisierten Instruktionen des MicroBlaze Prozessors mit P_{ba} gleichverteilt auf die Pipeline-Stufen

Instr.	E_bu(fJ)	E_if(fJ)	E_id(fJ)	E_ex(fJ)	E_mem(fJ)	E_wb(fJ)	E_ba(fJ)
ADD	1084080	172872	345744	576240	46099	11525	1152480
ADDC	1084080	172872	345744	576240	46099	11525	1152480
ADDI	1084080	163908	327816	546360	43709	10927	1092720
ADDIC	1084080	163908	327816	546360	43709	10927	1092720
ADDIK	1084080	163908	327816	546360	43709	10927	1092720
ADDIKC	1084080	163908	327816	546360	43709	10927	1092720
ADDK	1084080	172872	345744	576240	46099	11525	1152480
ADDKC	1084080	172872	345744	576240	46099	11525	1152480
AND	1084080	123948	247896	413160	33053	8263	826320
ANDI	1084080	108108	216216	360360	28829	7207	720720
ANDN	1084080	123948	247896	413160	33053	8263	826320

wird fortgesetzt

Fortsetzung

Instr.	E_bu(fJ)	E_if(fJ)	E_id(fJ)	E_ex(fJ)	E_mem(fJ)	E_wb(fJ)	E_ba(fJ)
ANDNI	1084080	108108	216216	360360	28829	7207	720720
BEQ	1084080	32076	64152	106920	8554	2138	213840
BEQD	1084080	32076	64152	106920	8554	2138	213840
BEQI	1084080	86436	172872	288120	23050	5762	576240
BEQID	1084080	86436	172872	288120	23050	5762	576240
BGE	1084080	32076	64152	106920	8554	2138	213840
BGED	1084080	32076	64152	106920	8554	2138	213840
BGEI	1084080	86436	172872	288120	23050	5762	576240
BGEID	1084080	86436	172872	288120	23050	5762	576240
BGT	1084080	32076	64152	106920	8554	2138	213840
BGTD	1084080	32076	64152	106920	8554	2138	213840
BGTI	1084080	86436	172872	288120	23050	5762	576240
BGTID	1084080	86436	172872	288120	23050	5762	576240
BLE	1084080	32076	64152	106920	8554	2138	213840
BLED	1084080	32076	64152	106920	8554	2138	213840
BLEI	1084080	86436	172872	288120	23050	5762	576240
BLEID	1084080	86436	172872	288120	23050	5762	576240
BLT	1084080	32076	64152	106920	8554	2138	213840
BLTD	1084080	32076	64152	106920	8554	2138	213840
BLTI	1084080	86436	172872	288120	23050	5762	576240
BLTID	1084080	86436	172872	288120	23050	5762	576240
BNE	1084080	32076	64152	106920	8554	2138	213840
BNED	1084080	32076	64152	106920	8554	2138	213840
BNEI	1084080	86436	172872	288120	23050	5762	576240
BNEID	1084080	86436	172872	288120	23050	5762	576240
BR	1084080	58932	117864	196440	15715	3929	392880
BRA	1084080	58932	117864	196440	15715	3929	392880
BRAD	1084080	58932	117864	196440	15715	3929	392880
BRAI	1084080	58932	117864	196440	15715	3929	392880
BRAID	1084080	58932	117864	196440	15715	3929	392880
BRALD	1084080	58932	117864	196440	15715	3929	392880
BRALID	1084080	58932	117864	196440	15715	3929	392880
BRD	1084080	58932	117864	196440	15715	3929	392880
BRI	1084080	58932	117864	196440	15715	3929	392880
BRID	1084080	58932	117864	196440	15715	3929	392880
BRLD	1084080	58932	117864	196440	15715	3929	392880
BRLID	1084080	58932	117864	196440	15715	3929	392880
CMP	1084080	172872	345744	576240	46099	11525	1152480
CMPU	1084080	172872	345744	576240	46099	11525	1152480
IMM	1084080	159480	318960	531600	42528	10632	1063200
LBU	1084080	81900	163800	273000	21840	5460	546000
LBUI	1084080	80604	161208	268680	21494	5374	537360
LHU	1084080	81900	163800	273000	21840	5460	546000
LHUI	1084080	80604	161208	268680	21494	5374	537360
LW	1084080	81900	163800	273000	21840	5460	546000
LWI	1084080	80604	161208	268680	21494	5374	537360
MFS	1084080	0	0	0	0	0	0
MSRCLR	1084080	0	0	0	0	0	0
MSRSET	1084080	0	0	0	0	0	0
MTS	1084080	0	0	0	0	0	0
OR	1084080	123948	247896	413160	33053	8263	826320
ORI	1084080	108108	216216	360360	28829	7207	720720
RSUB	1084080	172872	345744	576240	46099	11525	1152480
RSUBC	1084080	172872	345744	576240	46099	11525	1152480
RSUBI	1084080	163908	327816	546360	43709	10927	1092720

wird fortgesetzt

Fortsetzung							
Instr.	**E_bu(fJ)**	**E_if(fJ)**	**E_id(fJ)**	**E_ex(fJ)**	**E_mem(fJ)**	**E_wb(fJ)**	**E_ba(fJ)**
RSUBIC	1084080	163908	327816	546360	43709	10927	1092720
RSUBIK	1084080	163908	327816	546360	43709	10927	1092720
RSUBIKC	1084080	163908	327816	546360	43709	10927	1092720
RSUBK	1084080	172872	345744	576240	46099	11525	1152480
RSUBKC	1084080	172872	345744	576240	46099	11525	1152480
RTSD	1084080	58932	117864	196440	15715	3929	392880
SB	1084080	1944	3888	6480	518	130	12960
SBI	1084080	1764	3528	5880	470	118	11760
SEXT16	1084080	90396	180792	301320	24106	6026	602640
SEXT8	1084080	90396	180792	301320	24106	6026	602640
SH	1084080	1944	3888	6480	518	130	12960
SHI	1084080	1764	3528	5880	470	118	11760
SRA	1084080	0	0	0	0	0	0
SRC	1084080	0	0	0	0	0	0
SRL	1084080	0	0	0	0	0	0
SW	1084080	1944	3888	6480	518	130	12960
SWI	1084080	1764	3528	5880	470	118	11760
XOR	1084080	123948	247896	413160	33053	8263	826320
XORI	1084080	108108	216216	360360	28829	7207	720720

Tabelle B.5.: Die dynamischen Energiewerte der charakterisierten Instruktionen des MicroBlaze Prozessors mit P_{ba} unterteilt auf die Pipeline-Stufen mittels Erfahrungswerte