Technische Universität Dresden

Softwaremethoden zur Senkung der Verlustenergie in Microcontrollersystemen

Ralf Hildebrandt

der Fakultät Elektrotechnik und Informationstechnik der Technischen Universität Dresden

zur Erlangung des akademischen Grades eines

Doktoringenieurs

(Dr.-Ing.)

genehmigte Dissertation

Vorsitzender:	Prof.Dr.techn. Janschek		
Gutachter:	Prof.Dr.-Ing.habil. Fischer	Tag der Einreichung:	05.05.2006
	Prof.Dr.-Ing.habil. Schüffny	Tag der Verteidigung:	16.07.2007
	Prof.Dr.-Ing. Elst		

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr.-Ing. habil. W. J. Fischer für die Themenstellung und Unterstützung, sowie bei Herrn Dr.-Ing. H. Grätz vom Fraunhofer-Institut für Photonische Mikrosysteme (IPMS) für die hilfreichen Diskussionen während der Arbeit bedanken.

Ralf Hildebrandt

Inhaltsverzeichnis

Abkürzungen

Tabelle 1: Verwendete Namen und Bezeichner

ADC	Analog to Digital Converter
ALU	Arithmetic and Logic Unit
API	Application Programming Interface
ASM	Assembler
CISC	Complex Instruction Set Computer
CLA	Carry-Lookahead-Adder
CPA	Carry Propagate Adder
CPU	Central Processing Unit
CRA	Carry-Ribble-Adder
CSA	Carry-Save Array
CSTLFA	Carry-Save Tree / Leapfrog-Array
DAC	Digital to Analog Converter
DFS	Dynamic Frequency Scaling
DMA	Direct Memory Access
DSP	Digital Signal Processor
DVS	Dynamic Voltage Scaling
DVFS	Dynamic Voltage and Frequency Scaling
FFT	Fast Fourier Transform
FLL	Frequency Locked Loop
FPU	Floating Point Unit
HDL	Hardware Description Language
ISR	Interrupt Service Routine
LFCSA	Leapfrog-CSA
MIPS	Million Instructions Per Second
MTB	Master Toolbox [45]
PC	Program Counter
PPG	Partial Product Generation
PPR	Partial Product Reduction
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SDF	Standard Delay Format
SoC	System on Chip
SP	Stack Pointer
VHDL	VHSIC HDL
VHSIC	Very High Speed Integrated Circuit

Kurzfassung

Die Reduktion von Verlustleistung und Verlustenergie digitaler Schaltungen durch Optimierung von Hardware wie auch Software ist ein wesentliches Ziel beim Entwurf energieoptimierter Signalverarbeitungssysteme. Um dieses zu erreichen, werden Methoden zur Analyse und Bewertung von Realisierungen benötigt.

In dieser Arbeit werden Analysemethoden vorgestellt und diskutiert, die den Entwurf von Hardware und Software für energieoptimierte Microcontrollersysteme unterstützen. Schwerpunkte sind die schnelle und einfache Bewertung schaltungstechnischer Methoden beim Hardwareentwurf, die Quantisierung der Verluste bei Interaktion von Software und Hardware, sowie die detaillierte Analyse von Softwareprogrammen und von für deren Effizienz wichtiger Parameter.

Abstract

The reduction of power and energy dissipation of digital circuits by optimisation of hardware as well as software is a main target during the design of low power systems for signal processing. To achieve this objective methods for analysis and evaluation of implementations are necessary.

In this work techniques for analysis are presented and discussed, that assist the design of hardware and software for low power microcontroller systems. Main focuses are the fast and simple evaluation of wiring methods during hardware design, the quantisation of power and energy dissipation when software interacts with hardware as well as the detailed analysis of software programs and important efficiency related parameters.

1. Einleitung

1.1. Motivation

Beim Betrieb von digitalen Schaltungen wird elektrische Energie in nicht mehr nutzbare thermische Energie umgesetzt. Diese unerwünschte Verlustenergie stellt eine wesentliche Größe dar, welche die Möglichkeiten beim Einsatz der Schaltung beschränkt oder zusätzliche Maßnahmen zum Betrieb notwendig macht.

In Systemen mit begrenztem Energiebudget, wie z. B. batteriebetriebenen Systemen, limitiert die Verlustenergie die Betriebsdauer. Kann in einem solchen System nur 5% der Verlustenergie gespart werden, bedeutet das bei einer Betriebsdauer von 3 Jahren eine zusätzliche Laufzeit von rund 55 Tagen.

Die Umsetzung großer Mengen von Verlustenergie in kurzer Zeit, also eine hohe Verlustleistung, führt in Hochgeschwindigkeitssystemen zu einem ernsthaften thermischen Problem bei der Kühlung. Steht in Systemen nur eine Versorgungsquelle mit begrenzter Leistung (wie z. B. Solarzellen) zur Verfügung, kann eine zu hohe Verlustleistung den Betrieb generell verhindern oder zumindest eine zusätzliche Energiequelle (z. B. in Form von einer Hilfsbatterie) erforderlich machen.

Signalverarbeitung geschieht in zunehmendem Maße dezentral in autonomen, komplexen, mobilen Systemen, bei denen Verlustleistung und Verlustenergie ebenso wichtige Designgrößen wie Größe und Geschwindigkeit sind - vielleicht sogar die wichtigsten. Solche Systeme nehmen Signale aus der Umwelt über Sensoren auf und reagieren mit Steuerung von Aktoren. Kommunikation mit anderen Systemen oder einer steuernden Instanz geschieht oft nur temporär, sodass wesentliche Schritte der Signalverarbeitung innerhalb eines solchen Systems ausgeführt werden müssen. Diese Verlagerung der Signalverarbeitung von einer steuernden Instanz in ein Sensor- und Aktorsystem hinein ermöglicht Mobilität und kann Vorgänge beschleunigen sowie steuernde Instanzen vereinfachen oder sogar unnötig machen. Auch kann ein Verbund aus homogenen wie heterogenen Sensor- und Aktorsystemen gut an eine gegebene Problemstellung angepasst werden und mit steigender Komplexität der Anforderungen mitwachsen.

An mobile Systeme für Signalverarbeitung werden mannigfaltige Anforderungen gestellt. Neben einer langen Laufzeit bei geringem Energiebudget bzw. der Möglichkeit zum Betrieb mit einer Versorgungsquelle mit begrenzter Leistung wird hohe Flexibilität, große Unabhängigkeit von steuernden Instanzen und hohe Rechenleistung gefordert. Um die notwendige Eigenintelligenz und den numerischen Durchsatz zu erreichen, müssen ausreichend leistungsfähige Prozessoren und Verarbeitungseinheiten eingesetzt werden, während aufgrund der gewünschten Mobilität die räumlichen Dimensionen und das Gewicht der Versorgungsquelle deren Leistungsfähigkeit und Energiebudget limitiert. Um ein solches System zu realisieren, ist eine Reduktion von Verlusten unerlässlich. Diese Reduktion ist sowohl ein Problem beim Entwurf von Hardware als auch beim Softwareentwurf. Da spezialisierte Hardware eine Aufgabe oft schneller bei Umsetzung einer geringeren Menge Verlustenergie lösen kann als allgemeine und frei programmierbare Prozessoren, werden in mobilen Systemen oft mehrere spezialisierte Hardwarebaugruppen eingesetzt. Um eine hohe Effizienz des Gesamtsystems zu erreichen, müssen sowohl

solche Baugruppen leistungsoptimiert und angepasst an Softwarebedürfnisse als auch Software angepasst an die Hardwarebedürfnisse entworfen werden. Erst die Verzahnung von Soft- und Hardware führt zu einem effizienten System.

Die in Hardwarebaugruppen umgesetzte Verlustenergie hängt maßgeblich von der die Baugruppen steuernden Software ab. Eine Abschätzung der Verlustenergie ist sowohl beim Entwurf der Hardwarebaugruppen selber als auch beim Entwurf der steuernden Software von Interesse, um verschiedene Realisierungsmöglichkeiten zu vergleichen. Wünschenswert ist eine möglichst schnelle und genaue Abschätzung, die einfach in ihrer Anwendung und angepasst an die Problemstellung ist. Ein gleichzeitiges Optimum in allen Eigenschaften ist nicht erreichbar und so existiert eine Vielzahl von Analysemethoden mit spezifischen Vor- und Nachteilen. Trotz der Fülle von Methoden ist ein optimaler, allgemeiner Kompromiss bei den Eigenschaften noch nicht gefunden worden und weitere Schritte zur Verfeinerung der Methoden werden benötigt.

Die begrenzten Ressourcen mobiler Systeme erfordern hoch optimierte Software. Eine Abschätzung der durch ein Programm verursachten Verluste allein ist wünschenswert, bietet aber keine ausreichenden Informationen für die Programmoptimierung. Benötigt wird neben einer Verlustabschätzung auch eine Analyse von Programmen. Die heute in Entwicklungsumgebungen bereitgestellten Analysemethoden für hoch komplexe Programme, wie sie u. a. bei Personal Computern auftreten, sind nur teilweise geeignet für die Analyse von Software für kleine mobile Systeme, da diese Methoden zu grobkörnige Aussagen liefern. Der geforderte hohe Optimierungsgrad kann aber nur mit Hilfe von feinkörnigeren Analyseergebnissen erreicht werden.

Insbesondere für mobile Systeme wird häufig die Meinung vertreten, dass hoch optimierte Software mit Programmierung in Assembler gleichzusetzen ist. Dank ausgereifter Compiler ist es aber auch möglich, in einer Hochsprache effiziente Software zu entwickeln. Die Abstraktion von Hochsprachen erlaubt einen größeren Überblick, sodass algorithmische und strukturale Optimierungen viel leichter zu erkennen sind als beim Einsatz von Assembler. Gerade Optimierungen auf solch hohem Niveau haben besonders große Auswirkungen auf die Effizienz von Programmen. Ungünstige Konstrukte in Hochsprachen können jedoch eine Lösung ineffizient werden lassen. Zum Aufspüren und Vermeiden solcher Konstrukte wird eine detaillierte Programmanalyse benötigt.

Die Analyse von Software beschränkt sich bei vielen Analysemethoden auf die Charakterisierung nach einer Metrik. Es bleibt dem Anwender überlassen, aus dem anschließenden Vergleich verschiedener Realisierungen Schlussfolgerungen zu ziehen. Auch wenn durch eine automatisierte Analyse kaum eine genaue Lokalisierung von Gründen für ein konkretes Verhalten angegeben werden kann, würde eine Analyse von besonders wichtigen Parametern doch gezieltere und einfachere Schlussfolgerungen ermöglichen. Eine Analyse sollte also über die klassischen Metriken, wie z. B. Geschwindigkeit, hinausgehen und das Verständnis des Verhaltens von Realisierungen fördern. Dies kann auch feinkörnige Softwareoptimierungen ganz ohne Assemblerkenntnisse ermöglichen. Ziel muss es demnach sein, solch erweiterte Analysemethoden zu finden.

1.2. Überblick und Organisation

Mobile Sensor- und Aktorsysteme werden häufig mit Microcontrollern realisiert, da sowohl allgemein und flexibel programmierbare Prozessoren als auch hochspezialisierte und effiziente Baugruppen in Microcontrollern zur Verfügung stehen bzw. integriert werden können. Daher soll der Fokus der Betrachtungen in dieser Arbeit auf Microcontroller gerichtet werden.

Während des Hardwareentwurfs kann durch eine Synthese eine Netzliste erzeugt werden, welche eine Basis für Verlustabschätzungen bereitstellt. Die Verfügbarkeit einer synthesefähigen Hardwarebeschreibung bzw. einer synthetisierten Netzliste ist auch ein großer Vorteil für die Abschätzung von Verlusten bei der Interaktion von Hard- und Software. Ein wesentlicher Schritt zur Vereinfachung ist die Integration der Verlustabschätzung in eine vorhandene Hardwaresimulationsumgebung. Somit kann ein externes Werkzeug für die Abschätzung entfallen. Dies reduziert den Arbeitsaufwand für den Aufbau der Simulationsumgebung erheblich und ermöglicht den funktionalen Test parallel zur Verlustabschätzung. Um auch aufwändigere Interaktion mit Software simulieren zu können, ist eine hohe Simulationsgeschwindigkeit gefordert. Ein Kompromiss zwischen Genauigkeit bei der Abschätzung und den genannten Anforderungen wird mit einer Realisierung der Verlustabschätzung innerhalb einer VHDL-Simulationsumgebung in Kap. 2 vorgestellt.

Wesentlichen Anteil an der Effizienz des Gesamtsystems hat die Software. Diese muss an Microcontroller aufgrund ihrer spezifischen Eigenschaften angepasst sein. Eine solche Anpassung ist auch mit einer Hochsprache, wie ANSI C möglich, wenn geeignete Methoden zur systematischen Optimierung auf Hochsprachenebene gefunden werden. Wesentlich ist dabei die Überprüfung der Qualität der Optimierungsmethoden. Der abstrakte Charakter einer Hochsprache erlaubt neben einer reinen Effizienzbewertung auch eine Analyse der verwendeten Konstrukte der Hochsprache, was die Entdeckung von Ansätzen zur Optimierung erleichtert. Eine Analysemethode, die dank derartiger Mechanismen die Lücke zwischen Hochsprache und Assembler verkürzt und Softwareoptimierungen ohne Assemblerkenntnisse auf Hochsprachenniveau erlaubt, ist daher von großem Interesse. Neben einer solchen Methode muss auch die Software für Microcontroller als Gesamtsystem betrachtet werden. Da in den meisten Fällen kein Betriebssystem auf Microcontrollern eingesetzt wird, sind die Besonderheiten eines interruptgesteuerten Programmflusses zu beachten. Den dargestellten Problemen bei der Softwareoptimierung widmet sich Kap. 3.

Eine abschließende Zusammenfassung und Wertung der Arbeit in Bezug auf die hier formulierten Problemstellungen folgen in Kap. 4. Über die in den Kapiteln benötigten Grundlagen soll im Folgenden ein kurzer Überblick gegeben werden.

1.3. Grundlagen

Verluste (Verlustenergie und Verlustleistung) in digitalen Schaltungen setzen sich aus zwei Komponenten zusammen: einer dynamischen und einer statischen. Die dynamische Komponente umfasst die Verluste, die durch Umschaltvorgänge ausgelöst werden,

während die statische Komponente Verluste, ausgelöst durch Leckströme bezeichnet.

1.3.1. Metriken

Es existieren mehrere Metriken zur Charakterisierung von Verlusten und es hängt von der konkreten Anwendung ab, welche am besten geeignet ist. Verschiedene Beispiele dazu sind u. a. in [1] zu finden.

Sehr häufig wird die Verlustleistung $P = UI$ als Metrik verwendet. In erster Näherung kann davon ausgegangen werden, dass die Spannung U konstant ist. Statische Verluste werden ausgelöst durch nahezu konstante Leckströme, sodass auch die statische Verlustleistung in Näherung als konstant angesehen werden kann. Während eines Umladevorganges fließt dagegen ein variabler Strom $I = i(t)$, sodass die dynamische Verlustleistung zeitabhängig ist. Oft wird daher eine mittlere Verlustleistung angegeben.

Jedem Umladevorgang kann aber auch eine Verlustenergie $W = \int_T P\,dt$ zugeordnet werden. Da $P = P_{stat} + P_{dyn}$ die Summe aus statischer und dynamischer Verlustleistung ist, fließen beide Komponenten auch in die Verlustenergie ein. Eine Trennung von statischer und dynamischer Verlustenergie ist möglich, wenn die Größe der statischen Verlustleistung bekannt ist.

Für die Dimensionierung von Versorgungsquellen bezüglich der Leistungsabgabe und für Kühlmechanismen ist die Verlustleistung das geeignete Maß. Da diese stark schwanken kann, muss nicht nur eine mittlere, sondern auch eine maximale Verlustleistung angegeben werden. Da das theoretische Maximum in der Realität nicht immer erreicht wird, weil nicht alle Komponenten eines Schaltkreises in der Realität gleichzeitig arbeiten, wird oft ein realistisches Maximum angegeben. Intel gibt beispielsweise für die x86-Prozessoren die so genannte „Thermal Design Power“, AMD eine „Maximal Power“ an. Eine Zusammenstellung ist in [8] enthalten. Die umgesetzte Verlustenergie ist für batteriebetriebene Systeme dagegen das entscheidende Maß für deren Laufzeit.

Hybride mobile Systeme verfügen über eine Batterie und eine Energiequelle mit begrenzter Leistung. Somit sind sowohl Leistung als auch Energie bestimmende Metriken. Ist die Verlustleistung zeitweise größer als die durch die Energiequelle bereitgestellte Leistung, muss die Batterie unterstützend eingreifen. Ein prominentes Beispiel dafür ist der Mars Pathfinder, dessen Leistungsoptimierung in [10] beschrieben wird.

Es kann nicht endlos Verlustleistung gegen Rechenzeit eingetauscht werden, da auch Anforderungen an die Arbeitsgeschwindigkeit existieren [11]. Auch ist die Energie als Metrik nicht immer ausreichend, um eine Aussage darüber zu treffen, wie effizient ein System diese nutzt. Hierfür werden komplexe Metriken verwendet, die stark auf den konkreten Anwendungsfall abgestimmt sind. Häufig zur Anwendung kommen das Produkt aus Energie und Verzögerungszeit („Energy-Delay Product“) bzw. das Produkt aus Leistung und Verzögerungszeit („Power-Delay Product“) bzw. deren Quadrate oder auch „Leistung pro MIPS“ (Million Instructions Per Second). Die genannten Metriken werden häufig verwendet, wenn neben geringen Verlusten auch eine hohe Verarbeitungsgeschwindigkeit eine entscheidende Anforderung an das Design darstellt.

Nicht immer sind einzelne Metriken ausreichend, um eine Aussage über die „Fitness“ eines Systems zu treffen. In einem komplexen System gibt es zudem Abhängigkeiten

der Komponenten untereinander, die ebenfalls Auswirkungen auf die Verlustleistung haben. Wird beispielsweise der Prozessor eines Systems auf geringen Energieverbrauch und geringe Leistungsaufnahme optimiert und werden dabei längere Verarbeitungszeiten zugelassen, so können sich die längeren Verarbeitungszeiten auf von dem Prozessor gesteuerte Komponenten auswirken, die dadurch längere Zeit aktiv sein müssen [10].

1.3.2. Quellen für Verluste

Dynamische Verluste werden durch Umschaltvorgänge im Schaltkreis hervorgerufen und hauptsächlich durch das Auf- und Entladen von kapazitiven Lasten bestimmt. Die kapazitive Last am Ausgang einer Zelle C_l setzt sich zusammen aus der Eingangskapazität der folgenden Zelle C_{in} und der Lastkapazität der Verbindungleitung: $C_l = C_{in} + C_w$.

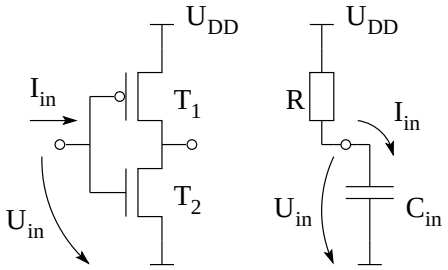

Abbildung 1: CMOS-Inverter und Ersatzschaltung beim Laden der Eingangskapazität

Die Eingangskapazität z. B. eines CMOS-Inverters (Bild 1) entspricht hauptsächlich der Gate-Source-Kapazität des nMOS-Transistors T_2. Mit der Lastkapazität C_l kann die mittlere dynamische Verlustleistung digitaler Schaltungen abgeschätzt werden mit

$$\overline{P_{dyn}} \approx \frac{\alpha}{2} C_l U_{DD}^2 f \tag{1}$$

Dabei sind U_{DD} die Versorgungsspannung, f die Taktfrequenz und α ein Aktivitätsfaktor, welcher angibt, wie häufig Umladevorgänge ausgelöst werden. Schaltet eine Zelle in jedem Taktschritt einmal um, so ist $\alpha = 1$, schaltet sie seltener, so ist $\alpha < 1$ und beim Auftreten von Hazards kann $\alpha > 1$ sein. Energie wird lediglich beim Laden von C_l, also bei $\alpha = 1$ mit jedem zweiten Taktschritt von der Versorgungsquelle entnommen.

Es fließt weiterhin ein „Kurzschlussstrom“, wenn kurzzeitig die Transistoren T_1 und T_2 während des Umschaltvorganges geöffnet sind, was einen zusätzlichen Beitrag zur dynamischen Verlustleistung darstellt. Die dabei umgesetzte Energie kann ebenfalls nicht mehr genutzt werden, aber sie ist (bei sorgfältigem Gatterentwurf) viel kleiner als die Energie, die beim Laden und Entladen von C_l umgesetzt wird. [2]

Neben den dynamischen Verlusten existiert eine statische Komponente, welche durch Leckströme verursacht wird. Diese Leckströme entstehen unter anderem durch Ströme in gesperrten PN-Übergängen, Subthreshold-Ströme und den Gateoxyd-Tunneleffekt. Weitere Details sind [6] angegeben.

Zur beispielhaften Veranschaulichung kann ein Modell für den Leckstrom in NMOS-Transistoren im Weak Inversion Bereich wie folgt angegeben werden [7]:

$$I_S = I_0\, e^{\frac{U_{GS}-U_T}{nU_t}} \left(1 - e^{-\frac{U_{DS}}{U_t}} \right) \tag{2}$$

Dabei sind U_{GS} und U_{DS} die Spannungen über Gate und Drain bezüglich Source, U_t die Temperaturspannung ($U_t \approx 25\,\mathrm{mV}$ bei 20°C), I_0 ein von der Technologie und der Temperaturspannung abhängiger Strom, U_T die Schwellspannung und n der Subthreshold Slope-Koeffizient. Je größer die Versorgungsspannung, desto größer kann auch U_{GS} werden und damit hat die Versorgungsspannung exponentiellen Einfluss auf die statischen Verluste.

Der Einfluss von der Schwellspannung U_T soll hervorgehoben werden: Mit sinkendem U_T steigt der Leckstrom exponentiell an. Andererseits ist eine niedrige Schwellspannung notwenig, um kurze Schaltzeiten und somit hohe Taktfrequenzen zu erreichen. Die für kurze Schaltzeiten wesentliche „Propagation Delay Time“ kann nach [3] angegeben werden mit

$$T_{pd} = \frac{\beta C_l U_{DD}}{(U_{DD} - U_T)^{\alpha}} \tag{3}$$

Dabei ist β ein durch die Technologie bestimmter Parameter und α repräsentiert den Geschwindigkeits-Sättingungs-Effekt.

1.3.3. Technologische Entwicklung

Tabelle 2: Skalierung nach Denard um den Faktor k [9]

Parameter		Auswirkungen	
Spannung:	U/k	Packungsdichte:	k^2
Oxyddicke:	t_{ox}/k	Geschwindigkeit:	k
Transistorbreite:	W/k	Leistung:	$1/k^2$
Transistorlänge:	L/k	Leistungsdichte:	const.
Diffusion:	x_d/k		
Dotierung:	kN_a		

Die stetig steigende Integrationsdichte wird für CMOS-Schaltungen in erster Linie durch Skalierung der Bauelemente erreicht (Tab. 2). Allein durch die Skalierung wird die Verlustleistung reduziert. Mit immer kleineren Strukturgrößen erlangen statische Verluste allerdings immer stärkere Bedeutung und steigen aufgrund geringerer Oxyddicken sogar an (Abb. 2).

Steigende statische Verluste mit immer kleineren Strukturgrößen führen sowohl zu thermischen Problemen bei Hochleistungssystemen als auch zu geringeren Laufzeiten bei mobilen Systemen. Insbesondere bei mobilen Applikationen mit langen Ruheperioden müssen statische Verluste auf ein Minimum reduziert werden. Die durch Skalierung

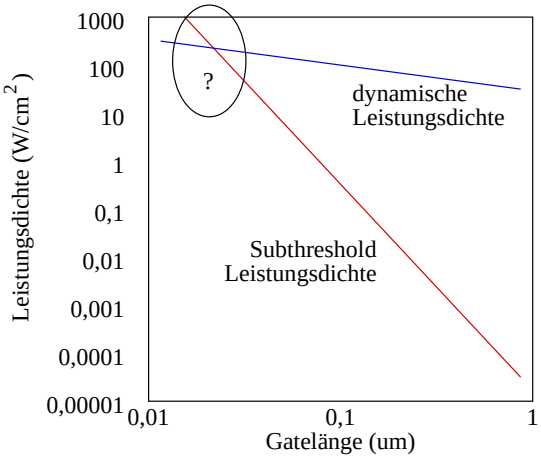

Abbildung 2: Leistungsdichte und Gatelänge [9]

erreichte hohe Integrationsdichte wird in erster Linie für komplexe Hochleistungssysteme benötigt. Bei vielen mobilen Systemen wird auch mit größeren Strukturen eine akzeptable Gatterfläche erreicht, sodass größere, langsamere Transistoren mit geringeren statischen Verlusten eingesetzt werden können.

1.3.4. Prinzipien für die Verlustreduktion auf Hardwareebene

1.3.4.1. Minimierung Statischer Verluste Einen groben Überblick über die Quellen statischer Verlustleistung hat Kap. 1.3.2 gegeben. Die auftretenden Leckströme sind maßgeblich durch die Technologie, die immer kleineren Strukturgrößen und die Temperatur bestimmt [5].

Nicht alle Komponenten einer Schaltung müssen eine hohe Geschwindigkeit erreichen. Hohe Schaltgeschwindigkeiten sind durch hohe Treiberstärken erreichbar, sodass kapazitive Lasten schnell umgeladen werden können. Indem das B/L-Verhältniss von Transistoren durch Vergrößerung der Breite erhöht wird, werden hohe Treiberstärken erreicht aber auch die Transitoren generell vergrößert. Nur im kritischen Pfad lohnt es sich überhaupt, schnelle Transistoren einzusetzen. Der Einsatz angepasster, kleiner Transistoren (Transistor Sizing) sowie höherer Schwellspannungen für Zellen in nicht-kritischen Pfaden verspricht eine starke Reduktion von unnötiger statischer Verlustleistung [11, 12, 13].

Werden Komponenten des Systems zeitweilig nicht benötigt, so werden Leckströme durch Abschalten der Betriebsspannung (Power Gating) wirksam reduziert [5]. Abb. 3 veranschaulicht die Idee. Problematisch ist die sorgfältige Dimensionierung der Sleep-Transistoren, um einen fehlerfreien Betrieb der Schaltung zu gewährleisten und den Anstieg der Schaltzeiten gering zu halten. Auch müssen die Sleep-Transistoren vergleichsweise groß sein, um die oben genannten Anforderungen zu erfüllen. Dies wiederum hat zur Folge, dass beim Übergang zum Sleep-Modus oder zurück in den aktiven Modus die dynamische Verlustleistung in diesen Sleep-Transistoren in Betracht gezogen werden muss. Es existiert somit eine minimal nötige Zeit, die die Schaltung im Sleep-Modus

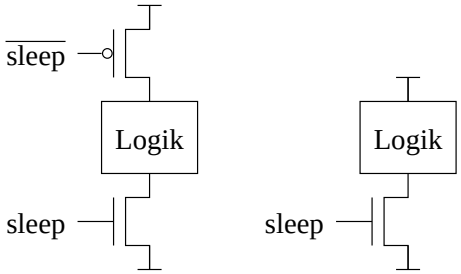

Abbildung 3: Power Gating [5]

verbringen muss, um Energie zu sparen. Über die Reduktion der Leckströme um den Faktor 86 beim Abschalten einer Multiply- & Accumulare Unit mittels Sleep-Transistoren wird in [16] berichtet. Caches können nicht abgeschaltet werden, wenn die Daten gespeichert bleiben sollen, aber U_{SS} kann angehoben werden. 27% geringere statische Verlusten durch die Anhebung von U_{SS} wurden in [17] erreicht.

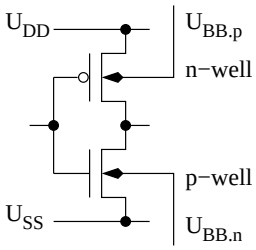

Abbildung 4: Variable Threshold-Voltage CMOS (VTCMOS) [3]

Umso größer die Schwellspannung U_T ist, desto geringer sind die Leckströme. Eine Variation der Schwellspannung ist möglich durch Anlegen einer Offset-Spannung an das Substrat (Abb. 4). Nach [5] kann U_T wie folgt beeinflusst werden:

$$U_T = U_{T_0} + \gamma \left(\sqrt{|-2\Phi_F + U_{SB}|} - \sqrt{|2\Phi_F|} \right) \tag{4}$$

Dabei ist U_{T_0} die Schwellspannung bei Substrat-Offset $U_{SB} = 0$, Φ_F das Substrat-Fermipotential und γ der Body-Effekt Koeffizient. Da die Schaltzeiten bei erhöhter Schwellspannung ebenfalls ansteigen (3), erweist sich der Einsatz von variablen Schwellspannungen als sinnvoll: hohe Schwellspannungen für inaktive Komponenten, niedrige für aktive [3]. Um dies zu ermöglichen, müssen die Schwellspannungen für alle Komponenten individuell regelbar sein.

1.3.4.2. Minimierung dynamischer Verluste Ausgehend von (1) bieten sich mehrere Möglichkeiten, die dynamische Verlustleistung zu minimieren. Aufgrund des quadratischen Einflusses der Betriebsspannung erscheint eine Absenkung der Spannung als aussichtsreichste Methode. Die erste Grenze dieser Methode kann ein inakzeptabler Anstieg der Schaltzeiten sein (3), die zweite Grenze ist die Störfestigkeit der Schaltung.

(Transistoren müssen bei high-Pegel sicher aufgesteuert werden.) Die Absenkung der Schwellspannung U_T kann die vergrößerten Schaltzeiten teilweise kompensieren und die Störfestigkeit sicherstellen, bewirkt aber andererseits einen exponentiellen Anstieg der Leckströme. Neben der allgemeinen Lösung, stets eine Schaltung nur mit der minimal nötigen Versorgungsspannung zu betreiben, erscheint es daher sinnvoll, sowohl Betriebsspannung als auch Schwellspannung im Betrieb zu variieren, um auf unterschiedliche Lastsituationen zu reagieren. Dieser Ansatz ist als „Dynamic Voltage Scaling“ (DVS) bekannt [3, 10, 14, 15, 21].

Innerhalb technologischer Grenzen kann die Versorgungsspannung variiert werden. Wie Abb. 5 beispielhaft illustriert, sind mit höherer Versorgungsspannung höhere Taktfrequenzen erreichbar (3). Analog zum DVS kann auch DFS („Dynamic Frequency Scaling“) eingesetzt werden. Die Generierung der Steuersignale ist für beide Prinzipien gleichartig und der gleichzeitige Einsatz beider Prinzipien wird als DVFS („Dynamic Voltage and Frequency Scaling“) bezeichnet. Die Steuerung von DVFS erfolgt entweder durch den Programmierer manuell oder durch ein Betriebssystem automatisch, abhängig von den Vorgaben eines auszuführenden Tasks. Vorteile von DVFS gegenüber einer vollständigen Abschaltung der Betriebsspannung sind die kürzere Umschaltzeit zwischen den Zuständen und die Erhaltung aller Informationen in Speicherelementen.

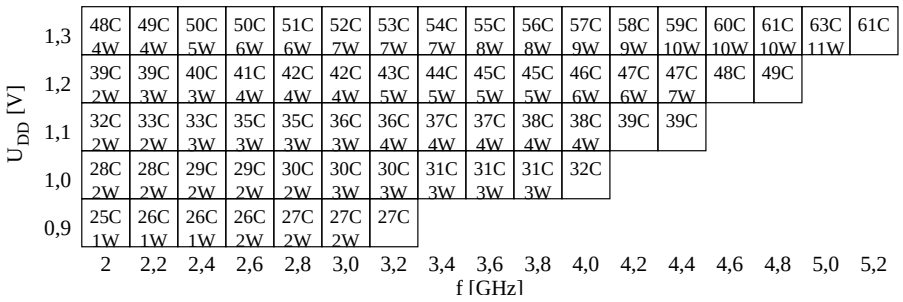

Abbildung 5: Shmoo-Plot einer SPE des CELL-Prozessors [18]

Unter der Voraussetzung, dass die statische Verlustleistung sehr klein ist und vernachlässigt werden kann, hat eine Variation der Taktfrequenz ohne Variation der Spannung lediglich Auswirkungen auf die dynamische Verlustleistung - nicht aber auf die dynamische Verlustenergie, da auch bei langsamerer Abarbeitung dennoch alle Operationen ausgeführt werden müssen, also keine Umschaltvorgänge eingespart werden können.

Die Größe der Eingangskapazität von logischen Gattern ist abhängig von der verwendeten Fertigungstechnologie. Eine Möglichkeit zur Reduktion der parasitären Kapazitäten von Transistoren ist der Einsatz von Silicon-On-Insulator (SOI), also dem Aufbau der Transistorstrukturen auf einem vom Rest der Schaltung isolierten Gebiet. Eine weitere Möglichkeit zur Reduktion der Eingangskapazität von Gattern ist der Einsatz von Transfergates statt der klassischen CMOS-Schaltungstechnik.

Die von einem Gatter zu treibende Lastkapazität hängt vom Fanout ab. Ganz besonders sticht hier das Taktnetzwerk („clock distribution network“) hervor, was maßgeblich an den Gesamtverlusten von digitalen Schaltungen beteiligt ist [4]. Die Lastkapazität

setzt sich aus den Eingangskapazitäten der folgenden Gatter und den Leitungskapazitäten der Verbindungsleitungen zusammen.

Die mittlere Anzahl der Umladevorgänge pro Taktschritt α in (1) hat zwar nur einen linearen Einfluss auf die Verlustleistung, aber der Wertebereich von α ist vergleichsweise groß. Einerseits existieren in digitalen Schaltungen Baugruppen, an deren Eingängen über längere Zeit keine Änderung eintritt, was $0 < \alpha \ll 1$ entspricht und andererseits existieren auch Baugruppen, deren Eingänge sich mehrfach ändern ($\alpha > 1$), an denen also Hazards zusätzlich zu häufigen Umladungen auftreten. Es existieren also Komponenten, die aufgrund ihrer Schalthäufigkeit maßgeblich für die Verlustleistung verantwortlich sind. Als Beispiel zur Illustration wird an dieser Stelle auf [19] verwiesen, wo die verschiedenen Quellen der Verlustleistung zweier ARM-Kerne aufgeschlüsselt werden.

Die Ansteuerung aller Flipflops in einem Schaltkreis ausschließlich mit dem globalen Taktsignal (synchrones Design) ermöglicht hohe Designsicherheit und automatische Verifikation, führt aber auch zu hohen Verlusten. Die Aufteilung des Schaltkreises in verschiedene „Clock Domains“, mit individuell geteiltem oder temporär deaktiviertem lokalem Takt (Abb. 6) führt zu einer Reduzierung der Anzahl der Umladevorgänge. Das Prinzip der Abschaltung kann auch auf andere Signale mit hoher Last angewendet werden.

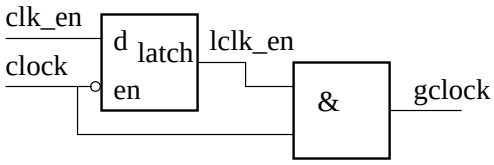

Abbildung 6: Clock-Gating mit Latch [5]

1.3.5. Microcontroller

1.3.5.1. Überblick Microcontroller sind Systeme, die eine CPU, Programm- und Arbeitsspeicher, sowie mehrere verschiedene Baugruppen zur Kommunikation, Messung und Steuerung auf einem einzigen Schaltkreis vereinen (Abb. 7). Sie sind in der vergangenen Jahren ausreichend leistungsfähig geworden, auch komplexere numerische Probleme in der Signalverarbeitung zu lösen, wobei natürlich der numerische Durchsatz ebenso wie die Größe des adressierbaren Speichers begrenzt bleiben und auch genügend Zeit für die Steuerung der Komponenten zur Verfügung stehen muss.

Die Kommunikation der CPU mit den Komponenten wird bei Microcontrollern meist über Adress- und Datenbus realisiert (Abb. 8), wobei Peripheriekomponenten Adressbereiche zugewiesen bekommen, in die sie spezifische Informationen (Bits für Konfiguration, Status und Daten) einblenden. Der Zugriff auf eine Komponente unterscheidet sich damit nicht vom Zugriff auf den Speicher. Ereignisse in Peripheriekomponenten können mit einer Unterbrechungsanforderung (Interrupt) signalisiert werden. Im Gegensatz zur kontinuierlichen Abfrage (Polling) werden viele unnötige Statusabfragen vermieden.

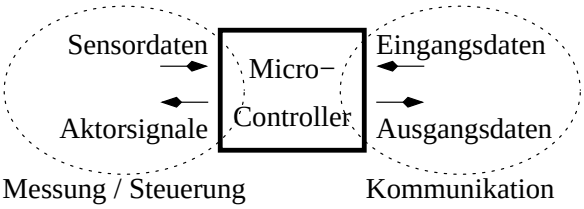

Abbildung 7: Allgemeines Microcontrollersystem

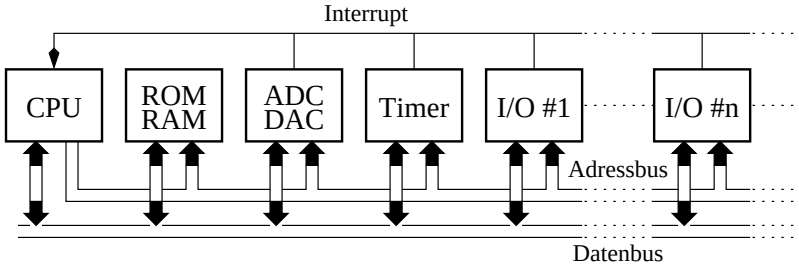

Abbildung 8: Typischer Aufbau von Microcontrollern

Microcontroller werden häufig für mobile Applikationen verwendet, sodass Verlustenergie und Verlustleistung entscheidend sind. Verlustreduktion muss in diesen Systemen sowohl beim Hardware- als auch beim Software-Entwurf betrachtet werden, wobei eine gute Abstimmung von Hard- und Software aufeinander wesentlich für die Effizienz ist. Dazu gehören der unkomplizierte Zugriff auf Daten innerhalb von Hardwarebaugruppen durch Software sowie die Eigenintelligenz und Ereignissignalisierung von Baugruppen.

Hohe Effizienz ist bei jeder Problemlösung gewünscht, nur ist dies durch die begrenzten Ressourcen mobiler Systeme auf Microcontrollerbasis in besonderem Maße erforderlich. Einfache und schnelle Analysemethoden sind nötig, um eine Vielzahl von Möglichkeiten zur Lösung eines Problems zu evaluieren und feinkörnige Analysen unterstützen die Optimierung durch detaillierte Aussagen über Kostenschwerpunkte.

1.3.5.2. Möglichkeiten zur Reduktion der Verlustleistung

Spezialisierte Hardware Für ein spezielles Einsatzgebiet lohnt es sich meist, auch spezialisierte Hardware einzusetzen. Oft kann damit ein Problem mit geringerem Energieverbrauch und höherer Geschwindigkeit gelöst werden als allein mit flexibel programmierbaren CPUs (These 1). Die klassischen Beispiele für solche Anwendungsfälle sind Hardwaremultiplizierer und Floating-Point Units. Je höher die Eigenintelligenz einer Komponente ist, d. h. je umfangreicher die Fähigkeiten der Komponente sind, desto seltener muss die CPU unterstützend und steuernd eingreifen.

Es ist vorteilhaft, ein umfangreiches, komplexes Problem zu partitionieren und die Problemlösung auf spezialisierte Prozessoren bzw. Recheneinheiten aufzuteilen. Beispielhaft für ein derartiges System aus Coprozessoren ist das Pleiades System [20]. Hierbei

handelt es sich um ein heterogenes System aus spezialisierten Prozessoren, welche über ein flexibles Netzwerk miteinander verbunden werden können. Als Hauptprozessor wird ein ARM8-Kern eingesetzt, der das System konfiguriert, den Datenverkehr koordiniert und Rechenaufgaben übernimmt, die auf keinem der Coprozessoren ausgeführt werden können. Jeder Coprozessor kann individuell abgeschaltet werden, sodass nur notwendige Komponenten arbeiten.

Neben der Verteilung von Rechenaufgaben bieten dedizierte I/O-Baugruppen, welche Busprotokolle wie I^2C, SPI, LIN, CAN, IrDA und andere in Hardware umsetzen, ein hohes Potential zur Reduzierung von Verlustleistung. Sie ermöglichen die Nutzung hoher Busdatenraten bei niedrigen CPU-Geschwindigkeiten, setzen weniger Energie um, als eine Softwarerealisierung, in der eine CPU das Busprotokoll durch bitweises Umschalten von allgemeinen I/O-Pins abarbeitet und ermöglichen die parallele Nutzung der CPU für andere Aufgaben während des Datentransfers.

Ist die Entscheidung zum Entwurf einer Hardwarekomponente gefallen, so bietet sich ein weites Spektrum an Möglichkeiten zur Realisierung. Wie viele interne Busse sollen implementiert werden, welche Möglichkeit für einen Zustandsautomaten ist vorteilhaft und setzt ein zusätzliches Pufferregister zu viel Energie um? Um diese und ähnliche Fragen beantworten zu können, ist eine Abschätzung von Verlusten von Hardwarebaugruppen nötig. Liegt eine vollständige synthesefähige Beschreibung der Hardwarekomponente vor, so können mit deren Hilfe die auftretenden Verluste bei Interaktion mit Software abgeschätzt werden. Für eine vorhandene Baugruppe kann zwar ein abstraktes Verlustmodell erstellt und zur Abschätzung eingesetzt werden, aber die Erstellung eines solchen Modells ist sehr aufwändig, fehleranfällig und verbietet Änderungen an der Baugruppe. Verlustabschätzung bei Interaktion von Software mit Hardware ist insbesondere dann von Interesse, wenn aufgrund temporärer Aktivität große Schwankungen der Verlustleistung auftreten, was die Versorgungsquelle überfordern kann, oder wenn alternative Wege zur Lösung eines Problems evaluiert werden sollen. In Kap. 2 wird das Problem der Verlustabschätzung auf Hardwareebene näher untersucht.

Software Ganz gleich, wie vielschichtig ein Problem ist, so existieren doch zumindest einige Verzweigungen im Programmablauf durch verschieden komplexe Entscheidungen und auch einige kleinere mathematische Berechnungen. Für ein extrem verlustarmes System, welches unter Umständen mehrere Jahre mit einer Batterie arbeiten muss, sollte auch bei simplen Problemen nicht unnütz Energie durch eine ineffiziente Lösung verschwendet werden, denn schnell kann dadurch die Betriebsdauer um mehrere Wochen reduziert werden.

Der Programmablauf bei Microcontrollern ist meist ereignisgesteuert. Ein Ereignis kann z. B. das Eintreffen neuer Daten über ein Bussystem zur Kommunikation oder das Abschließen der Aufnahme eines Messwertes durch einen Sensor sein. Mit dem Eintreten eines Ereignisses müssen Rechen- und Steuerungsaufgaben ausgelöst werden. Nach Abarbeitung aller Operationen wartet die CPU auf das nächste Ereignis. Kann ein Ereignis mittels Interrupt signalisiert werden, so kann die CPU wartend in einem Ruhezustand verharren, aus dem sie durch den Interrupt in eine Interrupt Service Routine (ISR) wech-

selt. Beim Beenden der ISR wechselt sie anschließend wieder in den Ruhezustand zurück. Bei Bedarf kann auch ein weiterer Interrupt eine laufende ISR unterbrechen. Interrupts und deren Behandlung sind demzufolge der Kern von Programmen für Microcontroller. Ein Betriebssystem existiert meist nicht und ist aus Sicht der Verlustleistung auch nicht gewünscht, da interruptgesteuerte Programme sehr effizient unnötige Verlustleistung vermeiden können.

Auch wenn Microcontroller flexibel programmierbar sind, so wird doch durch ihre Komponenten und deren spezifische Eigenschaften maßgeblich der Programmfluss beeinflusst. Der CPU fällt zwar immer die Aufgabe der zentralen Koordination zu, aber insbesondere, wenn Komponenten effizient genutzt werden sollen, muss sich diese Koordination den Anforderungen der Komponenten mehr und mehr unterordnen. Ein zu lösendes Problem kann daher selten in einer linear abzuarbeitenden Funktion umgesetzt werden, sondern ist zu partitionieren und auf diverse ISRs zu verteilen. Eine zentrale Steuerinstanz wird durch die ereignisgesteuerte dezentrale Steuerung zum großen Teil ersetzt.

Neben der Interaktion mit Peripheriekomponenten spielt auch die an die spezifischen Einzelheiten einer Microcontroller-CPU angepasste Programmierung eine Rolle. Volle Kontrolle bietet natürlich die Programmierung in Assembler, aber eine Hochsprache, wie ANSI C, bietet immer mehr Abstraktion und Übersicht, sodass algorithmische Optimierungen meist leichter zu entdecken sind. Gerade algorithmische Optimierungen haben meist viel größeren Einfluss auf die Effizienz einer Lösung als Anpassungen an den Instruktionssatz einer CPU, sodass die Nutzung einer Hochsprache oft auch unter dem Aspekt der Effizienzsteigerung zu bevorzugen ist. Durch die Abstraktion und natürlich auch die Qualität der Compiler kann allerdings schnell ein ineffizienter Code entstehen. Dem Problem, dies zu vermeiden und durch gezielte Analyse von Programmen algorithmische Optimierungen zu erleichtern, widmet sich Kap. 3.

2. Abschätzung umgesetzter Energie

Während des Entwurfs von Systemkomponenten müssen die zu erwartenden Verluste abgeschätzt werden können. Mit einer Abschätzung können verlustintensive Baugruppen identifiziert und mögliche alternative Realisierungen evaluiert werden. In erster Linie wird dadurch der Hardwareentwurf unterstützt, aber auch für den Softwareentwurf, insbesondere bei Interaktion mit Hardwarebaugruppen, ist dies von Interesse (These 2). Idealerweise sollte eine solche Abschätzung sehr präzise und schnell sein. Beide Kriterien zu erfüllen, ist im Allgemeinen nicht möglich, sodass Kompromisse nötig werden.

Es gibt verschiedene Ausgangspunkte für Abschätzungen, angefangen mit der System-/ Architekturebene über die Verhaltens- und die Netzlistenebene und schließlich zu Transistor- und Layoutebene. Dementsprechend unterschiedlich sind die Anforderungen bezüglich Genauigkeit und Geschwindigkeit, aber auch die verfügbaren Informationen. Häufig sind nur auf der Layoutebene möglichst genaue absolute Abschätzungen notwendig, während auf den anderen Ebenen relative Vergleiche zwischen verschiedenen Designalternativen im Vordergrund stehen.

Beim Entwurf von Systemen auf Microcontrollerbasis liegt der Schwerpunkt beim Hardwareentwurf auf der Modellierung synthesefähiger Verhaltensbeschreibungen. Mittels Synthese kann schnell eine Netzliste erstellt werden, welche direkt als Basis für Verlustabschätzungen dienen kann. Die erreichbare Genauigkeit von Verlustabschätzungen auf Layoutebene ist zwar durchaus von großem Interesse, aber der Zeitaufwand, für jede Designalternative den Entwurf bis hin zum Layout fortzusetzen, ist oft zu groß. Abschätzungen auf Systemebene werden dagegen beim Entwurf einer Architektur und beim Entwurf von Systemen mit vielen Teilkomponenten für numerische Berechnungen benötigt.

Neben der reinen Rechenzeit für eine Verlustabschätzung ist insbesondere für den Vergleich von einer großen Anzahl von Designalternativen eine möglichst kurze Zeit zum Aufbau der Testumgebung (das Setup) nötig. Ist beispielsweise die Einbeziehung von Leitungskapazitäten und Verzögerungszeiten in die Abschätzung gefordert und soll daher ein Layout erstellt werden, so muss die Layouterstellung zum Setup hinzu gezählt werden, was einen großen Aufwand darstellt. Weniger offensichtlich, aber nicht zu unterschätzen, ist das Setup beim Einsatz von externen Werkzeugen. Die Konvertierung der Daten in ein geeignetes Format, das Sammeln und Einlesen von Stimuli oder statistischen Informationen und der abschließende Vergleich von Ergebnissen der Abschätzung mit der Funktionalsimulation können zu einem erheblichen Aufwand führen, der die Bewertung einer großen Anzahl von Realisierungen unpraktikabel machen kann. Daher ist eine Verlustabschätzung innerhalb der normalen Hardwareentwurfsumgebung von großem Interesse (These 2a). In Kap. 2.2 wird eine solche Methode vorgestellt.

Software spielt beim Entwurf von Systemen eine entscheidende Rolle. Abschätzungen von Verlusten auf Netzlistenebene sind durchaus praktikabel, insbesondere da sie die Simulation aller im System beteiligten Komponenten ermöglichen. Der mögliche Umfang bleibt aber aufgrund der Simulationsgeschwindigkeit begrenzt. Zusätzlich existiert der Medienbruch zwischen Softwareentwicklungsumgebung und Hardwaresimulationsumgebung. (Der Softwareprogrammablauf ist in der Hardwaresimulationsumgebung nur

schwer nachzuvollziehen.) Da häufig Informationen über die Qualität von numerischen Algorithmen benötigt werden, welche nahezu ausschließlich die CPU nutzen, ist eine Verlustabschätzung auf CPU-Instruktionsniveau als Sonderfall der Abschätzung auf Verhaltensebene von Interesse. In Kap. 2.3 wird dieses Thema näher beleuchtet. Einen Schritt weiter gehen die Untersuchungen in Kap. 3, wo nicht nur die reine Bewertung von CPU-Instruktionen, sondern auch Ansätze zur Optimierung innerhalb einer Hochsprache diskutiert werden.

2.1. Bekannte Prinzipien zur Abschätzung von Verlusten

Im Folgenden sollen einige ausgewählte, bekannte Prinzipien zur Abschätzung von Verlusten vorgestellt werden. Sie werden ähnlich der Zusammenstellung aus [35] nach Abstraktionsebenen unterteilt.

Ein Problem bei einem solchen Vergleich stellen Aussagen über Geschwindigkeit und Genauigkeit der Methoden dar. Diese schwanken von Beispiel zu Beispiel sehr stark und es existieren nur wenige Veröffentlichungen, die solche Vergleiche herstellen. Meist wird zudem ein „typischer Fehler“ angegeben. Da allerdings kaum im Voraus eine Aussage getroffen werden kann, ob eine zu untersuchende Komponente „typisch“ ist, sind diese Aussagen nur von begrenztem Wert. Ebenso wenig kann ein maximaler Fehler angegeben werden, da die Komplexität der Abschätzungsmethoden dies kaum ermöglicht. Daher bleiben ausgewählte Beispielschaltungen die einzig praktikablen Vergleichsobjekte.

2.1.1. Transistor- und Layoutebene

Die höchste Genauigkeit für Verlustabschätzungen ist auf der Layoutebene möglich. Mit einem Layout ist ein Schaltkreis vollständig beschrieben und abhängig von Prozessschwankungen und der Qualität der verwendeten Bauelementemodelle können präzise absolute Verlustabschätzungen gemacht werden.

Auf Transistor- und Layoutebene kommen üblicherweise analoge Schaltkreissimulatoren, wie SPICE [27] und dessen Derivate, zum Einsatz. Diese bieten höchste Präzision, sind aber numerisch äußerst anspruchsvoll, sodass es praktisch unmöglich ist, größere Schaltungen über längere Simulationszeiträume zu analysieren. Schaltkreissimulatoren werden aufgrund ihrer Genauigkeit oft als Vergleichsbasis oder zum Charakterisieren von Baublöcken für andere Abschätzungsmethoden verwendet.

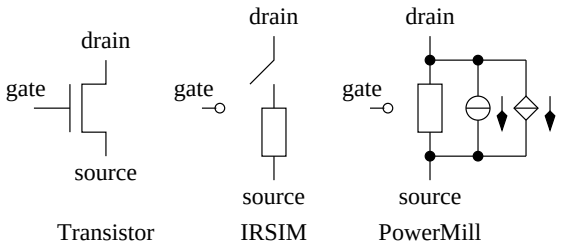

Abbildung 9: Transistormodelle von IRSIM [29] und PowerMill [30]

Müssen nur digitale Schaltungen simuliert werden, sind komplexe Bauelementemodelle oft unnötig. IRSIM [29] nutzt ein radikal vereinfachtes Transistormodell: ein Widerstand in Serie mit einem spannungsgesteuerten Schalter (Abb. 9). Zusätzlich wird jedem Netz eine Kapazität zugeordnet. Damit ist eine wesentliche Beschleunigung gegenüber komplexen Bauelementemodellen möglich, wobei die Genauigkeit der Simulation in akzeptablen Grenzen bleibt. Aus IRSIM wurde PowerMill [30] entwickelt, welches als Transistormodell die Parallelschaltung aus Widerstand, Stromquelle und einer spannungsgesteuerten Stromquelle nutzt. PowerMill nutzt zusätzlich statt eines Systems von Differentialgleichungen, im Voraus berechnete, in einer Tabelle abgelegte Werte und realisiert damit eine stückweise lineare Approximation, was zu einer enormen Reduzierung der Rechenzeit führt. Gegenüber HSPICE [28] erreicht PowerMill nach [31] Geschwindigkeistzuwächse bis zum Faktor 300, wenn für die Transistormodelle die vorausberechneten Werte schon existieren. Anderenfalls ist Faktor 80 möglich. Es existieren aber auch Schaltkreise, bei denen nur weit geringere Geschwindigkeitsgewinne (sogar unter Faktor 10) zu verzeichnen sind. Der relative Fehler gegenüber den SPICE-Derivaten bleibt oft unter 10%. In [31] wird ein Fehler von unter 13%, in [32] von 15-20% angegeben. NanoSim [48] nutzt Techniken von PowerMill. Daher ist von ähnlicher Genauigkeit auszugehen.

Komplexe wie auch vereinfachte Schaltkreissimulatoren haben als Ausgangsbasis eine Netzliste in einem SPICE-ähnlichem Format, welche aus einer HDL-Netzliste erstellt oder aus einem Layout extrahiert werden kann. Aufwändiger ist das Setup einer geeigneten Simulationsumgebung. Diese umfasst Quellen für Stimuli für die Eingänge und kapazitive Lasten für die Ausgänge. Stimuli können während einer Verhaltenssimulation generiert werden, z. B. durch die Mechanismen einer HDL zur Textausgabe, um jede Signaländerung eines Eingangssignales in eine Textdatei zu schreiben. Diese Textdaten können dann, konvertiert in ein geeignetes Format, innerhalb der Schaltkreissimulation eingelesen werden. Das Prinzip wird in Abb. 10 illustriert. Das Hinzufügen oder Entfernen von Ein- oder Ausgängen zieht umfangreiche Änderungen nach sich, sodass die Bewertung vieler verschiedener Designalternativen sehr zeitraubend sein kann, auch wenn die reine Simulationszeit akzeptabel ist. Da Verhaltenssimulation und Verlustabschätzung getrennt voneinander sind, ist die Zuordnung von Verlusten zur logischen Aktion schwierig.

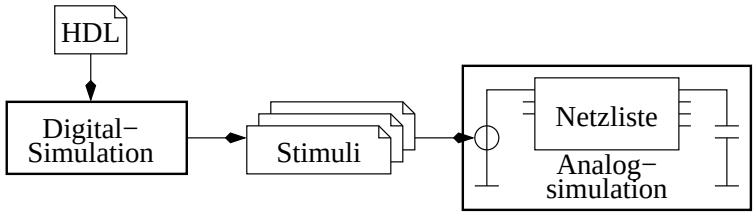

Abbildung 10: Simulationsumgebung für die Analogsimulation

2.1.2. Netzlistenebene

Eine Netzliste beschreibt logische Verbindungen zwischen primitiven Zellen. Als solche werden die mit einer Funktionsbibliothek bereitgestellten kombinatorischen wie sequentiellen Funktionsgatter (AND, XOR, Flipflops uvm.) bezeichnet. Durch Synthese einer Verhaltensbeschreibung wird eine Netzliste generiert. Aus dem bekannten Layout der primitiven Zellen leiten sich deren elektrische Eigenschaften ab. Unbestimmt ist die räumliche Anordnung der Zellen, sodass damit verbundene Verzögerungszeiten und Lastkapazitäten der Verbindungsleitungen abgeschätzt werden müssen. Unter der Annahme, dass logisch zusammengehörende Zellen auch räumlich nahe platziert werden, kann aber für relative Abschätzungen und Vergleiche der Einfluss der Verbindungsleitungen vernachlässigt werden. Für absolute Abschätzungen gilt dies nicht [42]. Die Beschreibung einer Schaltung auf Netzlistenebene ermöglicht wahrscheinlichkeits-/ statistik- oder simulationsbasierende Abschätzungen der Verluste.

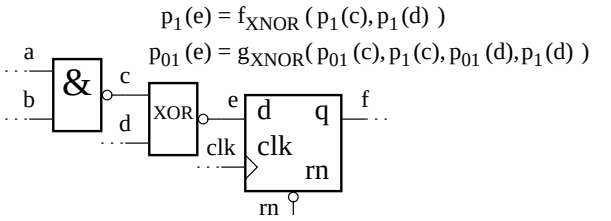

Abbildung 11: Wahrscheinlicheitsbasierende Abschätzung

Wahrscheinlicheitsbasierende Methoden ermitteln für ein Signal x die Wahrscheinlichkeit für eine Umladung während eines Taktschrittes $p_{01}(x) = p_{10}(x)$ anhand der Wahrscheinlichkeit für eine Umladung der Eingangssignale des treibenden Gatters, dessen logischer Funktion und der Wahrscheinlichkeit, dass das Signal gleich Eins ist $p_1(x) = (1 - p_0(x))$, wie in Abb. 11 illustriert

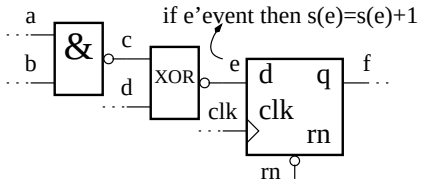

Abbildung 12: Statistikbasierende Abschätzung

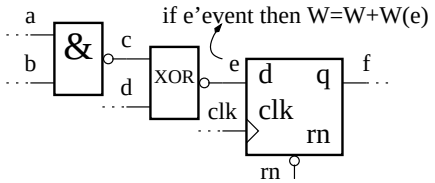

Abbildung 13: Simulationsbasierende Abschätzung

Bei statistikbasierenden Abschätzungen wird die Anzahl der Umschaltungen $s(x)$ während der Simulation ermittelt (Abb. 12). Abhängig von der Wahrscheinlichkeit für die Umladung, bzw. der Anzahl der Umschaltungen wird eine umgesetzte Energiemenge abgeschätzt. Bei simulationsbasierenden Verfahren wird dagegen mit jedem Ereignis direkt die entsprechende Energiemenge $W(x)$ bestimmt und akkumuliert (Abb. 13).

Wahrscheinlichkeits- / statistikbasierende Abschätzungen haben den Vorteil, dank der Abstraktion von Umladevorgängen durch Wahrscheinlichkeiten zur Umladung, eine äußerst schnelle Abschätzung der Verluste durchführen zu können. Eine Reihe solcher Methoden wird in [33] vorgestellt. Problematisch sind sequentielle Schaltungselemente. Cadence PKS [34] und Synopsys PowerCompiler [49] lösen dieses Problem durch statistische Aufnahme von Schalthäufigkeiten der Ausgänge sequentieller Zellen während der Simulation. PowerCompiler liefert Ergebnisse, die gegenüber denen von NanoSim um maximal 20% (meist unter 10%) abweichen [50]. Die Ergebnisse von NanoSim selber weichen wie in Kap. 2.1.1 ausgeführt von denen von SPICE ab.

Die Methode, mit Schalthäufigkeiten Verluste abzuschätzen, hat die prinzipbedingten Nachteile, nicht zu berücksichtigen, dass der Zustand aller Eingänge einer Zelle bei der Umladung eines Einganges für die umgesetzte Energie relevant ist und kaum eine Zuordnung von Verlusten zu logischen Aktionen zuzulassen. Bei statistikbasierende Methoden werden erst nach dem Anlegen von vielen Testvektoren repräsentative Ergebnisse von Schalthäufigkeiten erreicht. Dann lassen sich aber kaum Rückschlüsse auf Verluste durch spezielle Testvektoren ziehen. Bei wahrscheinlichkeitsbasierenden Abschätzungen werden die Schalthäufigkeiten der Eingänge nur durch Modelle, wie Pseudo-Random, beschrieben. Meist werden externe Werkzeuge zur wahrscheinlichkeits-/ statistikbasierenden Abschätzung eingesetzt, sodass ein signifikanter Zeitaufwand für das Setup entstehen kann. Zum Setup muss auch die Datengewinnung durch Simulation oder Modellbildung gezählt werden.

Die Verkürzung der Simulationszeit bei statistikbasierenden Abschätzungen hat das in [36] vorgestellte System zum Ziel. Es wird ein zyklenakkurater Simulator aus einer Netzliste heraus generiert. Pro Taktzyklus wird eine begrenzte Anzahl von Evaluationsschritten durchgeführt, sodass keine ereignisgesteuerte Simulation möglich ist und Hazards nicht simulierbar sind. Die Vereinfachung der Simulationsschritte ermöglicht aber eine sehr hohe Simulationsgeschwindigkeit. Die Zeit für das Setup des Simulators kann bei diesem System problematisch sein.

Bei simulationsbasierenden Methoden werden alle Schaltvorgänge einzeln betrachtet. Dabei werden an den Ausgängen von Zellen anliegende Lastkapazitäten umgeladen. Diese setzen sich aus der Eingangskapazität der folgenden Gatter und der kapazitiven Last durch die Verbindungsleitungen zusammen. Wie in Kap. 1.3.2 ausgeführt, können damit die dynamischen Verluste bestimmt werden. Genutzt wird dieses Prinzip beispielsweise in [37] und [40]. Ein fundamentales Problem ist das Ignorieren von zellinternen Umladevorgängen, wie sie insbesondere bei Flipflops auftreten. Die Eingangskapazität des Takteinganges von Flipflops repräsentiert in keiner Weise die effektiv umzuladende Summe der Kapazitäten. Der relative Fehler der Methode aus [37] liegt daher bei bis zu 60%. In [39] wird dieses Problem teilweise durch Berücksichtigung des Kurzschlussstromes im Zeitpunkt der Umladung kompensiert. Eine indirekte Erfassung von zellinternen Kapazitäten ist möglich. Explizit beachtet wurden sie aber in [39] nicht, sodass relative Fehler von bis zu 40% im Vergleich zu HSPICE auftreten - in Ausnahmefällen sogar deutlich mehr. Bei dem in [41] vorgestellten Ansatz wurden zellinterne Kapazitäten berücksichtigt und es wurde ein relativer Fehler von lediglich 2% bezüglich PowerMill angegeben. Es wurde allerdings nur ein Schaltkreis zur Bestimmung der Genauigkeit herangezogen,

sodass die Angabe kritisch betrachtet werden muss. Der Fehler von PowerMill muss ebenfalls berücksichtigt werden.

Der klassische Ansatz bei simulationsbasierenden Methoden ist das Aufzeichnen aller Umladevorgänge während einer Simulation und die Abschätzung mit einem externen Werkzeug. Nachteile eines externen Werkzeugs sind wieder die problematische Zuordnung von Abschätzung und logischer Aktion sowie die Zeit für das Setup. Die Integration der Abschätzung direkt in die Verhaltenssimulation löst dieses Problem. Eine solche Methode wird in Kap. 2.2 vorgestellt.

2.1.3. Verhaltens- und Systemebene

Es ist schwierig, ausgehend von der Verhaltensbeschreibung eine Abschätzung von Verlusten durchzuführen. In [38] und [43] wird daher als Spezialfall eine RTL-Verhaltensbeschreibung, also die Beschreibung von Verbindungen von Macroblöcken genutzt. Dies reduziert die Verhaltensbeschreibung auf eine Netzliste, welche Macroblöcke (statt Zellen) verbindet. Eine Verlustabschätzung wird durch parametrisierbare Modelle der Blöcke, wie dem sehr einfachen Modell in (5), ermöglicht. Dabei bezeichnen a_{in} und a_{out} die mittleren Umschaltwahrscheinlichkeiten der Eingänge und Ausgänge und die Koeffizienten P_x zugehörige Leistungsanteile.

$$P = P_0 + P_i a_{in} + P_o a_{out} \tag{5}$$

Zur Bestimmung der Parameter werden Simulationen mit einem präziseren Simulationswerkzeug durchgeführt und die Parameter entsprechend angepasst. Auch wenn für unbekannte Macroblöcke eine automatische Charakterisierung existiert, so kann dennoch das verwendete Modell unpassend sein. Die größte Limitierung stellt aber die Beschränkung auf eine Schaltkreisbeschreibung auf die rein strukturale Form (RTL) dar.

Auch das in [36] vorgestellte Werkzeug, welches primär auf Verlustabschätzungen auf Netzlistenebene zielt, ist auf der Verhaltensebene einsetzbar, sofern geeignete Modelle für die Verluste von Baugruppen gefunden werden können.

Die Beschränkung auf eine RTL Beschreibung ermöglicht insbesondere Vergleiche zwischen verschiedenen Architekturen und kann damit auch der Systemebene zugeordnet werden. Speziell für die Systemebene ist die Methode aus [44] entwickelt worden. Im Gegensatz zu RTL-Methoden werden dabei manuell angepasste parametrisierbare Modelle für Baugruppen verwendet, welche durch deren prinzipiellen Aufbau motiviert sind. Registerbänke werden beispielsweise in die Gruppe der Array-Strukturen eingeordnet. Diese Abstraktion ermöglicht Abschätzungen ohne vorhandene synthetisierbare Macroblöcke.

2.2. Energieabschätzung mit im Voraus berechneten Werten mit VHDL

Im Folgenden soll eine Methode zur Abschätzung dynamischer Verlustenergie vorgestellt werden. Um die gewünschte hohe Geschwindigkeit bei der Verlustabschätzung zu erreichen, ist nicht nur eine schnelle Abschätzungsmethode, sondern als ganz wesentliches Element auch die Reduzierung des Aufwandes für das Aufsetzen der Abschätzung nötig (These 2a). Im Voraus berechnete Energiewerte für jeden möglichen Umladevorgang

aller Zellen ermöglichen eine hohe Simulationsgeschwindigkeit und die Einbettung in eine normale VHDL-Simulation bietet einen geringen Aufwand für das Setup. Weiterhin ist auch eine relativ einfache Zuordnung von logischer Aktion zu den abgeschätzten Verlusten möglich. Die Umsetzung dieser Methode wird `energy_estim` genannt. Es werden dynamische Verluste betrachtet. Auf die Modellierung statischer Verluste wird in Kap. 2.2.4.2 kurz eingegangen.

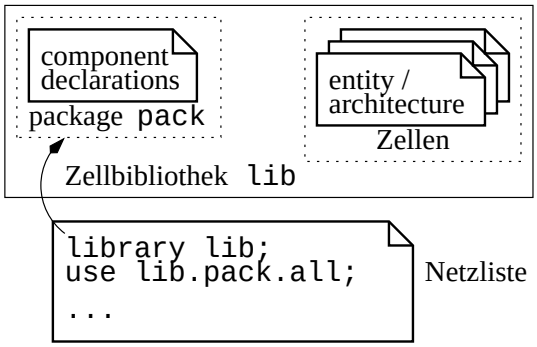

Abbildung 14: Zellbibliothek und Referenzierung aus einer Netzliste

Ist im Folgenden von einer Zellbibliothek die Rede, handelt es sich um eine VHDL-Bibliothek, in welche die Verhaltensmodelle der Zellen compiliert wurden. Die Deklaration der Komponenten wird in einer VHDL `package` abgelegt. Innerhalb einer Netzliste kann auf diese VHDL `package` referenziert werden, wie es in Abb. 14 illustriert ist.

2.2.1. Energieabschätzung während der VHDL-Simulation

Eine Möglichkeit zur Abschätzung von Verlusten innerhalb einer VHDL-Simulation auf Netzlistenebene ist der Einsatz einer modifizierten Zellbibliothek, in welcher neben dem normalen Verhaltens- und Verzögerungsmodell zusätzliche Mechanismen zur Verlustabschätzung enthalten sind. Eine einfache Realisierung dieses Konzepts ist durch den Einsatz von Containerzellen möglich, welche die originalen Zellen in sich instantiieren und in der Bibliothek ersetzen. Dieses Konzept ist unabhängig von einem speziellen Simulator und ist daher plattformübergreifend anwendbar. Die Container bieten die selben I/O-Signale (die selbe `entity`) wie die originalen Zellen an. Der entstehende Namenskonflikt wird durch Umbenennung der originalen Zellen gelöst. In Abb. 15 wird das Prinzip illustriert.

Die instantiierte originale Zelle stellt das Verhaltens- und Verzögerungsmodell bereit, während innerhalb des Containers mit jedem Ereignis eines Eingangssignales die dazugehörenden Verluste abgeschätzt werden. Verluste können in einer `shared variable` (ab VHDL'93) akkumuliert werden. Notwendig ist die Kenntnis über die kapazitive Last am Ausgang der Zelle. Diese wird mittles generischem Parameter (`fanout_q`) übergeben.

Da jede Containerzelle die gleiche `entity`, wie die jeweilige originale Zelle hat, kann die originale Bibliothek einfach durch die Containerbibliothek ausgetauscht werden. Listing 1 illustriert die dafür notwendige Änderung in der Netzliste.

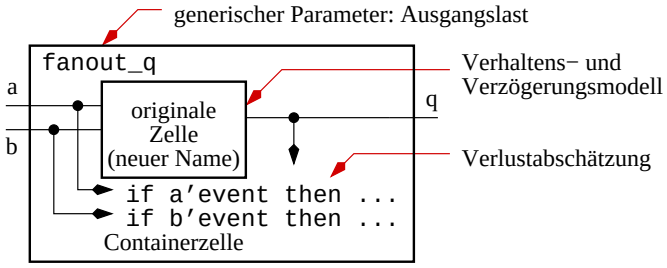

Abbildung 15: Containerzelle zur Verlustabschätzung

```
--library original_lib;
--use original_lib.vcomponents.all;
library energy_estim;
use energy_estim.vcomponents.all;
```

Listing 1: Ersetzen der originalen durch die Containerbibliothek

In der Netzliste muss weiterhin eine Information zur tatsächlichen kapazitiven Last (dem Fanout) übergeben werden. Dazu wird ein `generic map` bei jeder Instantiierung einer Zelle eingefügt. Listing 2 zeigt dies an einem Beispiel für die Last von 63 fF.

```
I1:cell_a generic map(fanout_q=>63) port map(A=>net1,B=>net2,Q=>net3);
```

Listing 2: Modifizierte Instanz in einer Netzliste (Fragment)

Eine derartige Modifikation der Netzliste mit dem `generic map` kann durch ein Werkzeug automatisch geschehen, welches für jede Zelle die Eingangskapazitäten aller jeweils getriebenen Zellen summiert. Ein solches Werkzeug wurde im Zuge dieser Arbeit erstellt und `cap_to_netlist` genannt. Abb. 16 illustriert den Arbeitsablauf.

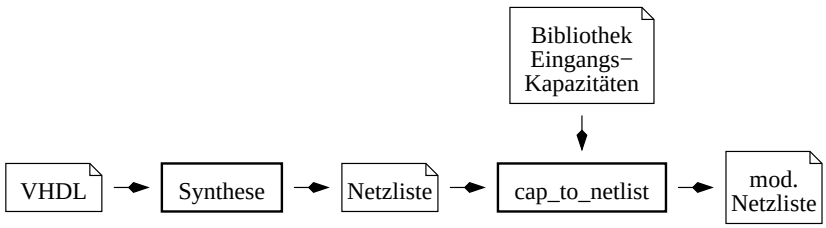

Abbildung 16: Modifikation der Netzliste mittels `cap_to_netlist`

Die Eingangskapazitäten aller Eingänge sämtlicher Zellen können bei der Charakterisierung von Zellbibliotheken mit einem Werkzeug, wie der Master Toolbox (MTB [45]), bestimmt werden. Eine Abschätzung der Last der Verbindungsleitungen (Wire Load Model) kann ebenfalls berücksichtigt werden. Mit dem `generic map` wird dann die Summe aus Eingangs- und Leitungskapazität übergeben.

Jedes Wire Load Model kann Leitungskapazitäten nur abschätzen und besitzt daher einen gewissen Fehler. Für die Bewertung einer Methode zur Abschätzung der Verluste in Zellen ist es deswegen sinnvoll, die Leitungskapazitäten zu ignorieren. (Auf das Problem der Leitungsverluste wird in Kap. 2.2.4.1 eingegangen.) Als Referenz für die Bewertung einer Verlustabschätzung kann ein Schaltkreissimulator, wie Spectre [51], eingesetzt werden. Innerhalb einer Spectre-Simulation werden Leitungskapazitäten ignoriert, wenn zur Simulation nur die synthetisierte Netzliste und nicht das extrahierte Modell nach einem Layout verwendet wird.

Die vorgestellte Methode ermöglicht Verlustabschätzungen direkt in der VHDL-Simulationsumgebung und bietet damit das Potential für schnelle Abschätzungen, wobei sie unabhängig von der spezifischen Abschätzungsmethode bleibt. Es könnten z. B. lastkapazitätsbestimmte Abschätzungen, wie die schon erwähnten Methoden aus [39, 40], verwendet werden. Im Folgenden wird aber eine Energieabschätzung mittels vorausberechneten Werten eingesetzt, die in Kap. 2.2.2 näher erläutert wird.

Ein weiterer wesentlicher Vorteil gegenüber externen Werkzeugen ist die einfache Zuordnung von funktionalem Verhalten und Verlusten. Sowohl die Momentanleistung als auch die nötige Energiemenge für das Ausführen einer Berechnung können mit VHDL Signalen dargestellt werden. Somit kann der zeitliche Verlauf von Verlusten zusammen mit dem funktionalem Verhalten visualisiert werden, wie es in einem Beispiel in Abb. 17 illustriert ist. Es handelt sich dabei um eine Microcontrollerkomponente, die das I^2C

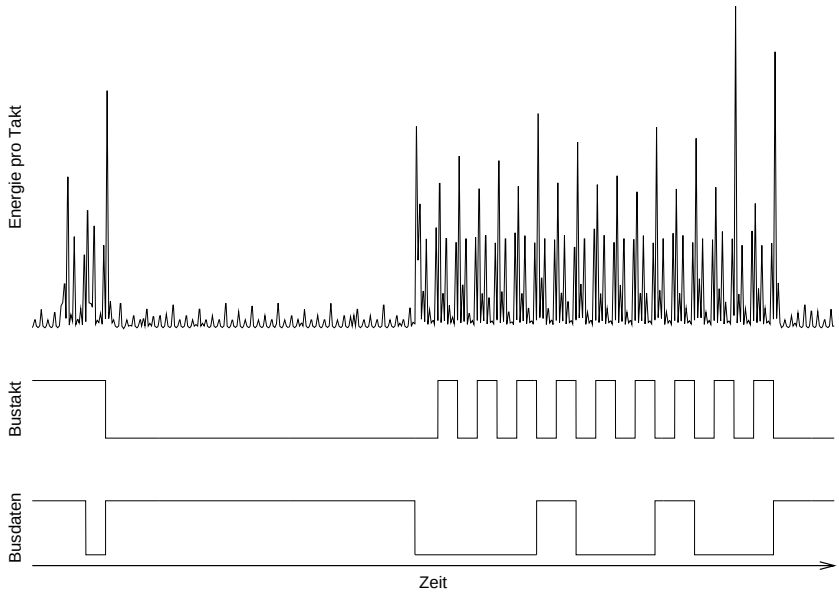

Abbildung 17: Visualisierung der Verluste bei der VHDL-Simulation

Busprotokoll als Master abarbeitet. Dargestellt ist der Start einer Übertragung, eine kurze Pause, während der der Microcontroller einige andere Tests durchführt und das Senden einer Adresse. Die dabei von der Komponente umgesetzte Energie pro Taktschritt (also eine Verlustleistung) wurde mit der hier vorgestellten Methode bestimmt.

Die gemeinsame Darstellung von funktionalem Verhalten mit abgeschätzten Verlusten wird in Abb. 17 illustriert. Die genauen Werte für die Verlustleistung sowie konkrete Zeitpunkte sollen hier nicht von Interesse sein und sind daher nicht dargestellt.

2.2.2. Energieabschätzung mit vorberechneten Werten

2.2.2.1. Das Prinzip Jede Umladung eines Eingangssignales führt zur Umsetzung von Energie (dynamische Verluste). Die Menge der umgesetzten Energie hängt vom Zustand aller Eingangssignale vor der Umladung und der Umladung selbst ab. Für eine kombinatorische Zelle mit n Eingängen existieren 2^n mögliche Zustände und für jeden Zustand n mögliche Einzelsignalumladungen, also $n \cdot 2^n$ Möglichkeiten. Für jede Möglichkeit kann die umgesetzte Energie mit hoher Genauigkeit mit Hilfe eines Schaltkreissimulators, wie Spectre, bestimmt werden. Sind alle Werte in einer Datenbasis abgelegt, so ist eine spätere Wiederverwendung bei Energieabschätzungen möglich. Eine Zelle wird stets die gleiche Energiemenge umsetzen, wenn die Bedingungen gleich sind. Die Tabellen 3 und 4 zeigen ein Beispiel für eine solche Datenbasis für ein AND Gatter mit 2 Eingängen aus einer willkürlich gewählten Zellbibliothek für eine gewählte Lastkapazität. Für sequentielle Zellen ist nicht nur der Zustand aller Eingänge, sondern auch der gespeicherte Wert (der Ausgang) für die umgesetzte Energiemenge relevant.

Tabelle 3: Datenbasis AND2: a

a	b	W [pJ]
$0 \rightarrow 1$	0	−0,074
$0 \rightarrow 1$	1	2,501
$1 \rightarrow 0$	0	0,074
$1 \rightarrow 0$	1	0,668

Tabelle 4: Datenbasis AND2: b

a	b	W [pJ]
0	$0 \rightarrow 1$	−0,062
0	$1 \rightarrow 0$	0,082
1	$0 \rightarrow 1$	2,501
1	$1 \rightarrow 0$	0,575

In den Tabellen 3 und 4 existieren Einträge mit negativen Werten für die umgesetzte Energiemenge. Die Verlustenergie wird ermittelt, indem das Integral über den während des Umladevorganges von der Versorgungsleitung in die Zelle hinein fließenden Strom $i(t)$ bestimmt und mit der Versorgungsspannung multipliziert wird. Das Vorzeichen gibt die Richtung des Energieflusses an. Bei negativem Vorzeichen ist in Summe ein Strom aus der Zelle hinaus in die Versorgungsleitung geflossen. Dies ist möglich, wenn durch das Entladen von zellinternen Kapazitäten mehr Ladung abfließt, als für das Aufladen von anderen Kapazitäten benötigt wird. Im konkreten Fall wird die Kapazität über Gate und Drain der pMOS Transistoren (Miller-Kapazität) entladen, wenn die Eingänge von Null auf Eins wechseln (Abb. 18). Der Effekt ist in der Energiebilanz sichtbar, wenn nach dem Ereignis $a \wedge b = 0$ ist. Ist nach dem Ereignis $a \wedge b = 1$, wird dies durch das Aufladen der viel größeren Ausgangskapazität maskiert. Für das vorgeschaltete Gatter erscheint die Miller-Kapazität als Lastkapazität und wird bei der Umladung einen entsprechenden Strom $i(t)$ nach sich ziehen. Wenn, wie hier im Beispiel, beim AND Gatter ein Energiefluss aus dem Gatter in die Versorgungsleitung auftritt, so muss dafür im vorgeschalteten Gatter eine entsprechende Energiemenge von der Versorgungsleitung entnommen werden. Analog dazu kann für die Charakterisierung aller Zellen zum Erstellen der Da-

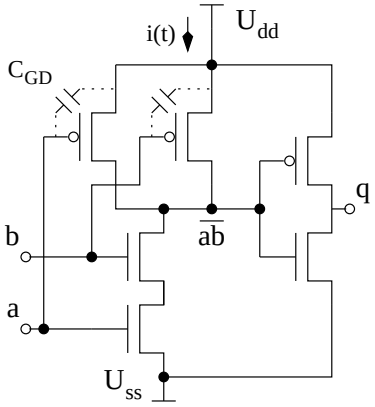

Abbildung 18: AND Gatter

tenbasis generalisiert werden, dass es ausreichend ist, nur $i(t)$ zu berücksichtigen, nicht aber die Ströme, die in die Eingänge der Zellen fließen.

Im Voraus berechnete Werte erlauben eine hohe Geschwindigkeit bei der Abschätzung, da im Wesentlichen aus einer Datenbasis Werte extrahiert werden müssen. Aufwändige Simulationsmodelle sind nicht nötig, da durch die Charakterisierung mittels Analogsimulator viele Einflussgrößen, wie der Effekt der Miller-Kapazität, automatisch berücksichtigt werden (These 2b).

Die Bestimmung der Energiemengen für alle Umladevorgänge, also die Charakterisierung einer kompletten Zellbibliothek, ist ein aufwändiger Prozess. Die Automatisierung desselben ist aufgrund der Menge an möglichen Einzelsignalumladungen notwendig. Es wurde dazu die Software Master Toolbox (MTB [45]) eingesetzt, welche den Schaltkreissimulator Spectre zur Charakterisierung nutzt. Die Laufzeit einer solchen Charakterisierung kann mehrere Tage betragen, muss aber pro Bibliothek nur einmalig ausgeführt werden. Das schon erwähnte Werkzeug Cadence PKS [34] nutzt ein ebensolches Prinzip zur Charakterisierung, setzt aber eine andere, statistische Abschätzungsmethode ein.

Die Nutzung vorausberechneter Werte ermöglicht eine ereignisgesteuerte Abschätzung der Verluste während einer normalen VHDL-Simulation. Im Gegensatz zu statistischen oder wahrscheinlichkeitstheoretischen Methoden wird einem konkreten Ereignis eine konkrete Verlustenergie zugeordnet. Der Zustand sämtlicher Eingänge sowie evtl. der eines gespeicherten Wertes ist bei der Zuordnung ausschlaggebend.

2.2.2.2. Einflussgrößen auf die umgesetzte Energie Die umgesetzte Energie hängt von der Geschwindigkeit der Signaländerung ab. Für eine Digitalsimulation bleibt dies verborgen. Da Synthesewerkzeuge ausreichend starke Treiber einsetzen, um einen schnellen Pegelwechsel zu ermöglichen, muss angenommen werden, dass die Geschwindigkeit der Signaländerung nur in geringem Maße variiert. Sie wird daher im Folgenden vernachlässigt und es wird die Charakterisierung der Zellbibliothek mit einer (möglichst typischen) Signaländerungsgeschwindigkeit durchgeführt.

Nicht allein die zellinternen Kapazitäten sind entscheidend für die umgesetzten Ener-

giemengen, sondern auch in ganz wesentlichem Maße die Last am Ausgang - der Fanout. Da nicht für jede beliebige Last eine Energiemenge im Voraus bestimmt werden kann, muss eine Datenbasis die Umladungsenergien für verschieden große Lasten enthalten und die Werte für die dazwischenliegenden Lasten sind zu interpolieren. Da nach Gleichung (1), S. 5 die Verlustleistung direkt proportional zur kapazitiven Last ist, ist eine lineare Interpolation ausreichend, was auch später mit den Ergebnissen aus Abb. 20 bestätigt wird. Besitzt eine Zelle mehrere Ausgänge, so hat folglich der individuelle Fanout jedes Ausganges einen Einfluss auf die Verluste. Diesem Problem widmet sich Kap. 2.2.2.3.

Wie schon in [42] erwähnt, haben partielle Umladungen (Spikes) einen Einfluss auf die umgesetzte Energie. Partielle Umladungen treten bei Hazards und bei schnell aufeinander folgenden Änderungen von zwei verschiedenen Eingangssignalen auf. Führt die Änderung eines Eingangssignales nicht zur Umladung eines Ausganges, werden oft nur die Eingangskapazitäten der Zelle umgeladen, was eine vergleichsweise geringe Energieumsetzung nach sich zieht und sehr schnell abgeschlossen ist. Für eine Energieabschätzung kann man diesen Fall vernachlässigen und jede Änderung als nicht-unterbrochene Einzelsignaländerung auffassen. Die Umladung eines Ausganges dauert dagegen wesentlich länger und führt zu einer deutlich größeren Energieumsetzung (vgl. Tab. 3 und 4). Ist die Umladung nicht abgeschlossen, wenn eine weitere Änderung eines Eingangssignales auftritt, die wiederum eine Umladung zur Folge hat, so wird weniger Energie durch die zweimalige teilweise Umladung umgesetzt als durch zwei volle Umladungen. Die Spectre-Simulation in Abb. 19, welche den Vergleich von kompletten mit partiellen Umladungen zeigt, untermauert dies. Dabei ist $i(t)$ wieder der Strom von der Versorgungsleitung in die Zelle hinein. In Kap. 2.2.2.4 wird das Problem partieller Umladungen im Detail diskutiert.

2.2.2.3. Zellen mit mehr als einem Ausgang Bei Zellen mit mehreren Ausgängen ist im Allgemeinen die individuelle Last jedes Ausganges für die umgesetzte Energie entscheidend, was für m Ausgänge zu einer m-fach größeren Datenbasis führen würde. Aber das Problem kann vereinfacht werden:

1. Beim Laden der Ausgangskapazität wird Energie von der Versorgungsquelle entnommen, zum Teil abgegeben und zum Teil gespeichert. Dagegen führt das Entladen der Ausgangskapazität zu keiner weiteren Energieentnahme. Unabhängig davon ist das Laden zellinterner Kapazitäten.

2. Meist enthalten Zellbibliotheken nur Zellen, deren Ausgänge logisch nicht unabhängig sind. Bei Flipflops ist meist neben dem normalen Ausgang der invertierte Ausgang vorhanden und bei Addiergliedern existieren Summe und Carry als Ausgang. Ausgelöst durch eine *einzelne* Eingangssignaländerung wird bei diesen Zellen nur *genau ein* Ausgang geladen, während der andere entladen wird oder unverändert bleibt. Das Laden von zwei Ausgängen kann also nur nacheinander, ausgelöst durch zwei Eingangssignaländerungen, geschehen.

Unter diesen Bedingungen wird also für die Umladung eines Einganges nur der kapazitive Fanout genau eines Ausganges entscheidend für die umgesetzte Energie sein. Für einen

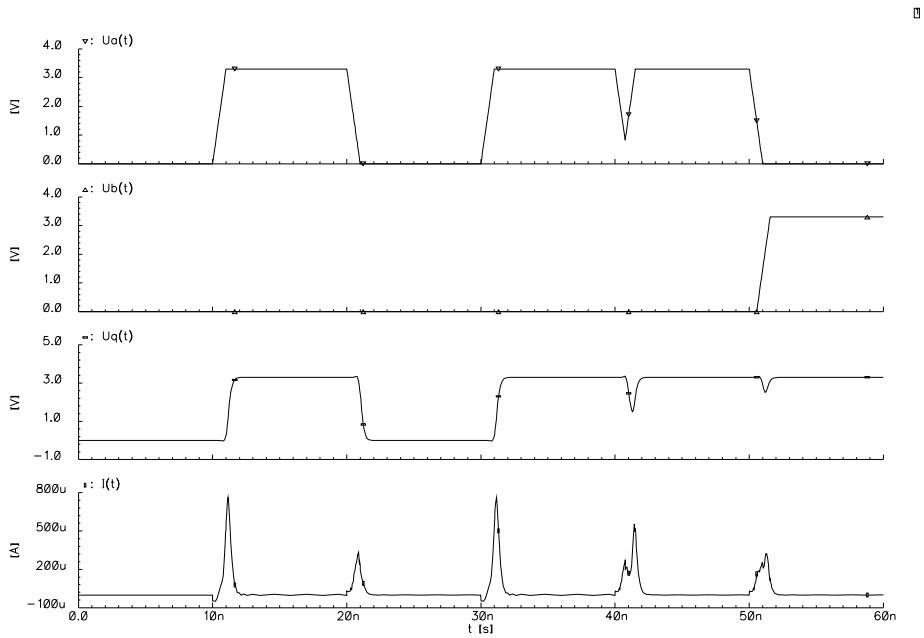

Abbildung 19: Partielle Umladungen am OR-Gatter (Spannungsverläufe für Eingänge `a`, `b` und Ausgang `q` sowie Stromaufnahme aus der Versorgungsleitung)

beispielhaft ausgewählten Volladder einer Standardzellbibliothek mit den Eingängen `A`, `B` und `CI` wird in Abb. 20 exemplarisch die Bestätigung dafür gegeben. Dargestellt ist die umgesetzte Energie für `A`=1, `B`=0 und den Übergang `CI`=1 → 0 in Abhängigkeit von der kapazitiven Last an den Ausgängen `S` und `CO`. Die Ergebnisse entstanden durch Simulation mit Spectre. Im angegebenen Fall wird der Summenausgang `S` geladen und Carry-Out `CO` entladen. Deutlich zu erkennen ist, dass die kapazitive Last an `S` maßgebend ist, während die von `CO` vernachlässigt werden kann. (Die Linien äquivalenter Energien sind nahezu parallel zur `CO`-Achse.) Das analoge Verhalten kann für das Laden des anderen Ausganges und auch für andere Zellen gezeigt werden.

Somit ist es ausreichend, in die Datenbasis nur die vorausberechneten Energien für mehrere Fälle mit stets gleicher kapazitiver Last an allen Ausgängen aufzunehmen. Während der Energieabschätzung kann der maßgebende Ausgang bestimmt werden und abhängig von dessen kapazitiver Last wird ein Wert aus der Datenbasis verwendet.

Aus Abb. 20 ist weiterhin ersichtlich, dass die umgesetzte Energie linear mit der Ausgangslast (des aufgeladenen Ausganges) zunimmt. Damit ist gezeigt, dass lineare Interpolation für Zwischenwerte völlig ausreichend ist.

2.2.2.4. Approximation von Teilumladungen Eine Energieabschätzung, die lediglich bei jeder Signaländerung die vorausberechneten Energiewerte akkumuliert, wird aufgrund von möglichen Teilumladungen zu große Werte liefern. Für ein akzeptable Genau-

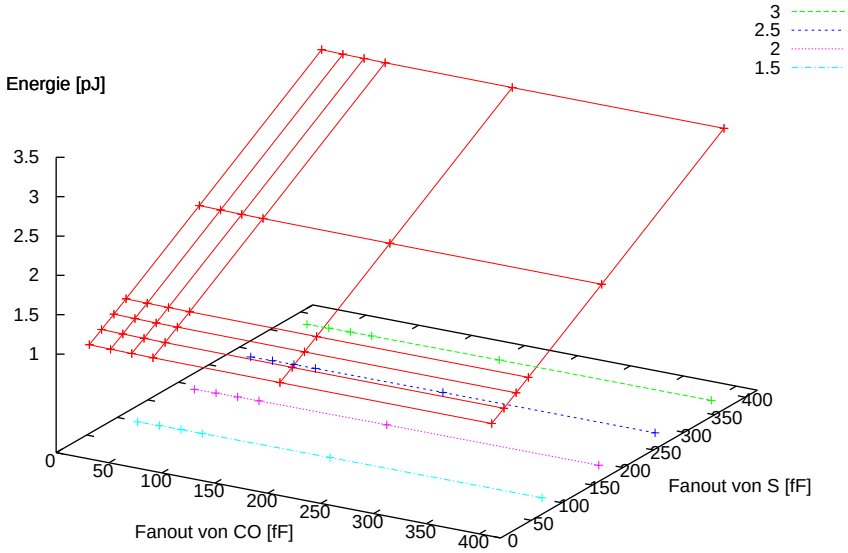

Abbildung 20: Umgesetzte Energie am Volladder für A=1, B=0 und CI=1 → 0

igkeit bei der Abschätzung müssen partielle Umladungen berücksichtigt werden (These 2c). Um die Abschätzung mit VHDL nicht zu stark zu verlangsamen, muss eine numerisch einfache Möglichkeit gefunden werden, das Problem der Teilumladungen zu approximieren. Es gilt für eine Propagation einer einzelnen Signaländerung:

$$W_{p_a} = \int_{t_a}^{t_a+t_{p_a}} u \cdot i(t)\, dt \tag{6}$$

Dabei sind u die Versorgungsspannung, $i(t)$ der Strom, der von der Versorgungsleitung in die Zelle hinein fließt, t_a die Zeit, zu der der Eingang a umgeladen wird, t_{p_a} die Propagationszeit ausgehend von Eingang a und W_{p_a} die dabei umgesetzte Energiemenge.

Teilumladungen entstehen, wenn zu einem Zeitpunkt t_b mit $t_a < t_b < t_a + t_{p_a}$ eine weitere Umladung eines Einganges auftritt, die wiederum zu einer Signalpropagation und Rückumladung führt. Die bei der Teilumladung umgesetzte Energie wird mit einem einfachen Modell approximiert, wie in Abb. 21 ($i(t)$) illustriert. In dem Modell wird von zwei einzelnen virtuellen Signaländerungen und dem dabei fließenden Strömen $i_a(t)$ und $i_b(t)$ ausgegangen. Während t_a und t_b die konkreten, festen Startzeitpunkte der Umladungen sind, ist t_x eine Variable und bezeichnet den Startzeitpunkt einer virtuellen Einzelsignalumladung. Der Wert der Variablen t_x kann u. U. bestimmt werden, ist aber nicht weiter von Interesse.

Mit $i_a(t)$ und $i_b(t)$ werden also die Stromverläufe bezeichnet, die geflossen wären, wenn nur einzelne, nicht unterbrochene Signalpropagationen aufgetreten wären. Es sei darauf hingewiesen, dass in Abb. 21 nur ein schematischer Stromverlauf zur Illustration

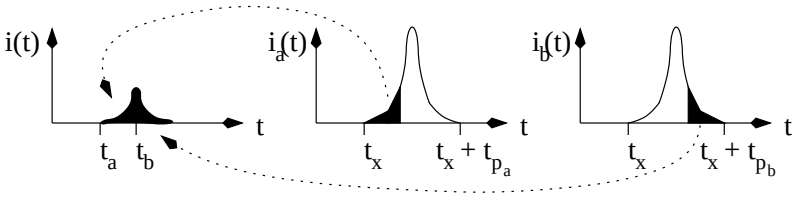

Abbildung 21: Approximation partieller Umladungen

angegeben ist, es sich dabei aber nicht um reale Messkurven wie in in Abb. 19 handelt.

Bis zum Zeitpunkt t_b wird durch die Propagation des Signals a die Energie

$$W_a = \int_0^{t_b - t_a} u \cdot i_a(t)\, dt \tag{7}$$

umgesetzt und der Ausgang ist partiell umgeladen. (Ohne Beschränkung der Allgemeinheit wurde $t_x = 0$ angenommen.) In Abb. 21 ist diese Teilenergie im Diagramm $i_a(t)$ dunkel hervorgehoben. Zum Zeitpunkt t_b setzt die Rück-Umladung ein und es wird eine Teilenergie umgesetzt, wie in Abb. 21 im Diagramm $i_b(t)$ illustriert. Die untere Grenze des Integrales über $i_b(t)$ ist unbekannt. Für eine Abschätzung wird der Wert r eingeführt:

$$r(t_b) = r_{ab} = \frac{t_b - t_a}{t_p(t_a)} = \frac{t_b - t_a}{t_{p_a}} \qquad | \, t_a \leq t_b \leq t_a + t_{p_a} \rightarrow r \in [0, 1] \tag{8}$$

Dies ist der relative zeitliche Anteil, der für die Propagation des Signals a bis zur Unterbrechung zur Verfügung stand, wobei $t_p(t_a)$ die Propagationszeit zum Zeitpunkt t_a darstellt, die hier gleich der vollen Propagationszeit t_{p_a} des Signals a ist. Als Näherung wird angenommen, dass Auf- und Entladung gleichartig verlaufen. Der Energieanteil durch die Signaländerung von b kann dann abgeschätzt werden mit

$$W_b = \int_{(1-r_{ab})t_{p_b}}^{t_{p_b}} u \cdot i_b(t)\, dt \tag{9}$$

Die Signalverläufe $i_a(t)$ und $i_b(t)$ können zwar durch Simulation bestimmt werden, doch die Masse an Daten und der Aufwand einer Integration lassen es nicht zu, diese in der Verlustabschätzung mit VHDL zu verwenden. Bestimmt werden kann nur die Gesamtenergie W_p. Für eine einzelne Umladung und für die Signalverläufe müssen daher einfache Modelle gefunden werden.

Modell des konstanten Umladestroms Das einfachste Modell für einen Umladestrom ist ein konstanter Strom $i_a(t) = I_a$, wie in Abb. 22 illustriert und es folgt für $t_x = 0$ und $u(t) = U$:

$$\int_0^{r \cdot t_{p_a}} u(t) \cdot i_a(t)\, dt = \underbrace{U \cdot I_a \cdot t_{p_a}}_{W_{p_a}} \cdot r_{ab} = r_{ab} \cdot W_{p_a} \tag{10}$$

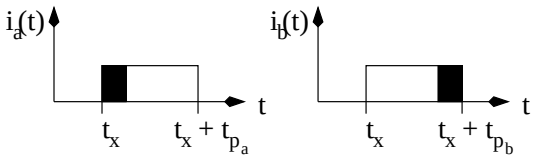

Abbildung 22: Modell des konstanten Umladestroms

Für W_{p_b} gilt aufgrund der Symmetrie das Analoge und somit ergibt sich für die Gesamtenergie $W_{p_{ab}}$ bei einer Teilumladung die folgende Abschätzung:

$$W_{p_{ab}} \approx (W_{p_a} + W_{p_b}) \cdot r_{ab} = \underbrace{W_{p_a}}_{W^+(t_a)} \underbrace{-(1 - r_{ab})W_{p_a}}_{W^-(t_b)} + \underbrace{r_{ab}W_{p_b}}_{W^+(t_b)} \tag{11}$$

Für eine sequentielle Abschätzung während einer ereignisgesteuerten Simulation kann der zweite Teil von (11) verwendet werden. Zum Zeitpunkt t_a wird W_{p_a} addiert, während zum Zeitpunkt t_b die Werte $(1 - r_{ab})W_{p_a}$ subtrahiert und $r_{ab}W_{p_b}$ addiert werden. Bei jedem Ereignis wird also die volle Energiemenge akkumuliert, die umgesetzt werden würde, wenn keine Unterbrechung des Umladevorganges eintritt. Tritt doch eine Unterbrechung ein, muss ein Teil dieser Energiemenge wieder abgezogen werden.

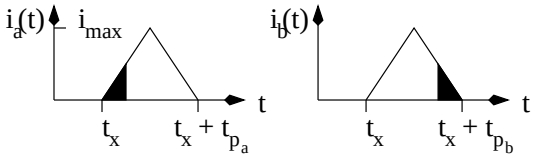

Abbildung 23: Modell des linear steigenden und fallenden Umladestroms

Modell des linear steigenden und fallenden Umladestroms Besser als mit einem konstanten Umladestrom wird der Umladestrom mit dem Modell aus Abb. 23 nachgebildet. Es gilt mit $t_x = 0$:

$$i_a(t) = \begin{cases} \frac{2i_{max}}{t_{p_a}}\, t & , 0 \le t \le \frac{t_{p_a}}{2} \\ -\frac{2i_{max}}{t_{p_a}}\, t + 2i_{max} & , \frac{t_{p_a}}{2} \le t \le t_{p_a} \end{cases} \tag{12}$$

Die Bestimmung von i_{max} kann durch die bekannte Energie W_{p_a} geschehen. Durch die Symmetrie um $\frac{t_{p_a}}{2}$ vereinfacht sich das ein wenig.

$$W_{p_a} = 2 \cdot \int_0^{\frac{t_{p_a}}{2}} U \frac{2i_{max}}{t_{p_a}}\, t\, dt = \frac{1}{2} U i_{max} t_{p_a} \tag{13}$$

Damit ergibt sich

$$i_a(t) = \begin{cases} \frac{4W_{p_a}}{U\, t_{p_a}^2}\, t & , 0 \le t \le \frac{t_{p_a}}{2} \\ -\frac{4W_{p_a}}{U\, t_{p_a}^2}\, t + \frac{4W_{p_a}}{U\, t_{p_a}} & , \frac{t_{p_a}}{2} \le t \le t_{p_a} \end{cases} \tag{14}$$

Um abhängig von r wieder eine geschlossene Lösung analog zu (11) zu erhalten, ist zu unterscheiden, ob $0 \leq r_{ab} \leq \frac{1}{2}$ oder $\frac{1}{2} < r_{ab} \leq 1$.

$$\begin{aligned}
\int_0^{r_{ab}t_{pa}} U \cdot i_a(t)\,dt\,|_{0\leq r_{ab}\leq\frac{1}{2}} &= \int_0^{r_{ab}t_{pa}} U \cdot \frac{4W_{p_a}}{U\,t_{p_a}^2}\,t\,dt \\
&= W_{p_a} \cdot 2r_{ab}^2 \\
\int_0^{r_{ab}t_{pa}} U \cdot i_a(t)\,dt\,|_{\frac{1}{2}<r_{ab}\leq 1} &= \int_0^{\frac{1}{2}t_{pa}} U \cdot \frac{4W_{p_a}}{U\,t_{p_a}^2}\,t\,dt + U\int_{\frac{1}{2}t_{pa}}^{r_{ab}t_{pa}} -\frac{4W_{p_a}}{U\,t_{p_a}^2}\,t + \frac{4W_{p_a}}{U\,t_{p_a}}\,dt \\
&= \frac{1}{2}W_{p_a} + 4W_{p_a}\left(-\frac{r_{ab}^2}{2} + r_{ab} - \frac{3}{8}\right) \\
&= W_{p_a}\left(-2r_{ab}^2 + 4r_{ab} - 1\right) \qquad (15)
\end{aligned}$$

Für W_{p_b} gilt aufgrund der Symmetrie das Analoge und somit ergibt sich für die Gesamtenergie $W_{p_{ab}}$ bei einer Teilumladung die folgende Abschätzung:

$$W_{p_{ab}} \approx \begin{cases} (W_{p_a} + W_{p_b}) \cdot 2r_{ab}^2 & , 0 \leq r_{ab} \leq \frac{1}{2} \\ (W_{p_a} + W_{p_b}) \cdot (-2r_{ab}^2 + 4r_{ab} - 1) & , \frac{1}{2} < r_{ab} \leq 1 \end{cases} \qquad (16)$$

Mehrfache partielle Umladungen Es können mehrfache Teilumladungen auftreten, wie in Abb. 24 illustriert. Wie schon erwähnt, muss mit jeder Unterbrechung einer Umladung zum Zeitpunkt t_y ein Teil der zuvor zu t_x akkumulierten Energie abgezogen ($W^-(t_y)$) und ein reduzierter Energieanteil akkumuliert ($W^+(t_y)$) werden.

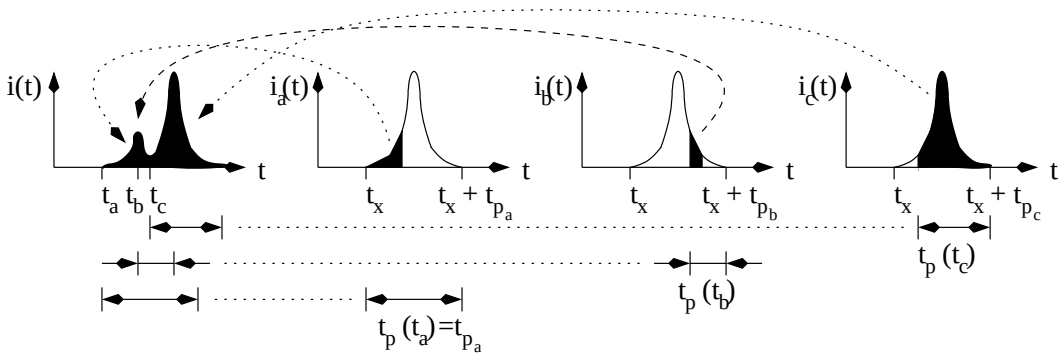

Abbildung 24: Approximation von drei Teilumladungen

Um die Menge der umgesetzten Energie bei partiellen Umladungen abzuschätzen, wurde der relative zeitliche Anteil bis zur Unterbrechung der Umladung r eingeführt (8). In Abb. 25 ist r_{bc} innerhalb mehrfacher partieller Umladungen dargestellt. Dabei ist zu erkennen, dass r_{bc} sich nicht auf den theoretischen Startzeitpunkt t_x des virtuellen Stromverlaufes $i_b(t)$ bezieht. Ein solcher Bezug zu dem theoretischen Startzeitpunkt ist aber zur Abschätzung von der Energiemenge für die folgende Umladung nötig. Weiterhin ist aus Abb. 24 und 25 ersichtlich, dass bei mehrfachen partiellen Umladungen

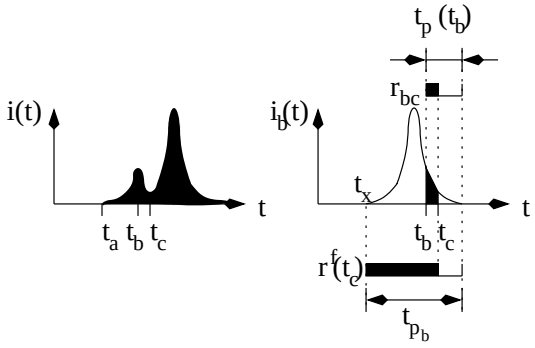

Abbildung 25: Approximation von reduzierten Propagationszeiten und Energiemengen

die Propagationszeit verkürzt werden muss (im Beispiel ist $t_p(t_b) < t_{p_b}$). Auch für die Abschätzung dieser Verkürzung ist ein Bezug zum theoretischen Startzeitpunkt nötig. Darum wird ausgehend von Abb. 25 ein Relativierungsfaktor r^f eingeführt. Dieser kann in Relation zu r gesetzt werden:

$$1 - r^f(t_c) = (1 - r_{bc}) \frac{t_p(t_b)}{t_{p_b}} = (1 - r_{bc}) \frac{r^f(t_b) t_{p_b}}{t_{p_b}} = (1 - r_{bc})\, r^f(t_b) \tag{17}$$

Die verkürzte Propagationszeit wird durch $t_p(t_b) = r^f(t_b) \cdot t_{p_b}$ analog zu den Annahmen bei einer einzelnen partiellen Umladung approximiert. Diese Approximation ist sofort in (17) eingeflossen.

Gleichung (17) kann wie folgt interpretiert werden: Mit Hilfe des Relativierungsfaktors des vorherigen Schrittes ($r^f(t_b)$) und des relativen zeitlichen Anteils (r_{bc}) kann der neue Relativierungsfaktor ($r^f(t_c)$) iterativ berechnet werden. Für eine Unterbrechung einer Signalpropagation lässt sich mit r^f die zu einer globalen Energiesumme zu addierende Energie W^+ abschätzen.

$$W^+(t_c) = g(r^f(t_c)) \cdot W_{p_c} \tag{18}$$

Dabei ist W_{p_c} wieder die volle Energie für eine einzelne Signalpropagation ausgehend von Signal c und g eine Funktion, die das Modell für den Umladestrom beschreibt. Ein Teil der beim vorherigen Ereignis t_b addierten Energie muss aufgrund der aktuellen Unterbrechung wieder abgezogen werden:

$$W^-(t_c) = g(1 - r_{bc}) \cdot W^+(t_b) \tag{19}$$

Es soll hervorgehoben werden, dass in (18) r^f und in (19) r verwendet wird. Für g gilt

$$g(r) = \begin{cases} r & ; \text{konstanter Umladestrom} \\ \begin{cases} 2r^2 & , 0 \le r \le \frac{1}{2} \\ -2r^2 + 4r - 1 & , \frac{1}{2} < r \le 1 \end{cases} & ; \text{linear steigend / fallend} \end{cases} \tag{20}$$

Das Beispiel aus Abb. 24 ist in Tab. 5 im Detail erläutert. Wie in (8) definiert, ist dabei $r(t_b) = r_{ab}$. Zum Zeitpunkt t_a wird keine vorausgehende Signalpropagation unterbrochen

Tabelle 5: Energieakkumulation und Propagationszeit nach Beispiel aus Abb. 24

t	$r(t)$	$t_p(t)$	$W^-(t)$	$W^+(t)$
t_a	1	t_{p_a}	0	W_{p_a}
t_b	$\frac{t_b - t_a}{t_p(t_a)}$	$r^f(t_b)$	$g(1 - r(t_b))W^+(t_a)$	$g(r^f(t_b))W_{p_b}$
t_c	$\frac{t_c - t_b}{t_p(t_b)}$	$r^f(t_c)$	$g(1 - r(t_c))W^+(t_b)$	$g(r^f(t_c))W_{p_c}$

und es wird $r^f(t_a) = 1$ initialisiert. Für alle weiteren Zeitpunkte berechnet sich $r^f(t)$ ausgehend von (17) analog zu:

$$r^f(t_c) = 1 - (1 - r_{bc})\, r^f(t_b) \tag{21}$$

Das iterative Schema aus Tab. 5 wird so lange fortgesetzt, bis kein neues Ereignis innerhalb der (verkürzten) Propagationszeit der letzten Umladung auftritt.

Zusammenfassend kann gesagt werden, dass mit der Kenntnis von Energiemenge (W_{p_x}) und Propagationszeit (t_{p_x}) für eine vollständige Umladung iterativ mit jedem weiteren Ereignis die umgesetzte Energie bei partiellen Umladungen abgeschätzt werden kann und diese Methode daher in einem ereignisgesteuerten Simulator implementierbar ist. Sind vom vorherigen Ereignis Zeitpunkt, abgeschätzte verkürzte Propagationszeit und abgeschätzte reduzierte Energiemenge bekannt (gespeichert), kann zum aktuellen Ereignis ein Teil der beim letzten Ereignis akkumulierten Energiemenge abgezogen und die zugehörige Energiemenge zum aktuellen Ereignis hinzu addiert werden.

Die Methode ist hochgradig flexibel durch die Wahl eines geeigneten Modells g für den Umladestrom. Die Annahme der Gleichartigkeit von Auf- und Entladung bei partiellen Umladungen stellt eine Schwachstelle dar. Eine weitere entsteht durch die Nichtbeachtung des Kurzschlussstroms bei partiellen Umladungen, welcher abhängig von der Dimensionierung der Transistoren und Zellen in Standardzellbibliotheken auftreten kann. Sollte eine Zelle etwa bis zum Kipppunkt umgeladen worden sein und wird die Umladung dann unterbrochen, kann über eine außergewöhnlich lange Zeit ein Kurzschlussstrom fließen.

2.2.2.5. Charakterisierung der Zellbibliothek Die vorgestellte Methode zur Verlustenergieabschätzung macht eine umfangreiche Charakterisierung der verwendeten Zellbibliothek nötig. Dafür wurde die Master Toolbox (MTB [45]) eingesetzt, welche Spectre als Analogsimulator nutzt. Folgende Größen müssen bestimmt werden:

- umgesetzte Energiemengen für alle Eingangssignaländerungen (W_p, Kap. 2.2.2.1)
- Signalpropagationszeiten (t_p, Kap. 2.2.2.4)
- die Information, ob eine Signaländerung am Eingang bis zum Ausgang propagiert wird (Kap. 2.2.2.4) und für Zellen mit zwei Ausgängen, ob ein Ausgang aufgeladen wird und wenn ja, welcher (Kap. 2.2.2.3)
- Eingangskapazitäten der Zellen (für `cap_to_netlist`, Kap. 2.2.1)

Die Energiemengen und die Propagationszeiten wurden für c ausgewählte Lastkapazitäten bestimmt. Wie in Abb. 20 zu erkennen ist, ist eine lineare Inter- bzw. Extrapolation für die Bestimmung von Zwischenwerten sinnvoll.

Der Ausgangszustand vor der Umladung eines Einganges ist abhängig von der Anzahl der Eingänge i bei kombinatorischen Zellen. Für sequentielle Zellen ist die Anzahl der gespeicherten Zustände s zusätzlich relevant. Eine Datenbasis muss demnach $2^{i+s} \cdot c \cdot i$ Messwerte für die Energiemengen und Propagationszeiten und $(i+s) \cdot i$ Informationen zur Umladung des Ausganges bzw. der Ausgänge aufnehmen. Unmögliche Zustände (Reset eines Flipflops aktiv, aber Ausgang ist high) und verbotene Zustände (Reset und Set gleichzeitig aktiv) führen zu ungenutzten Einträgen in der Datenbasis.

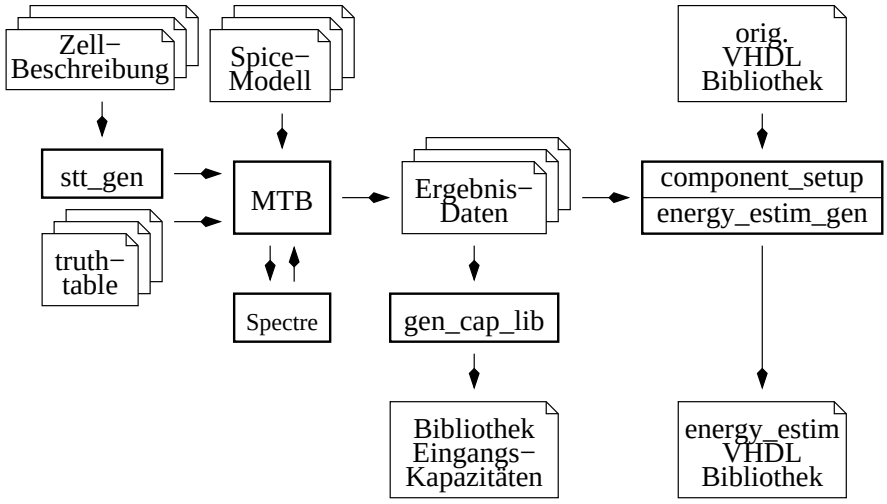

Abbildung 26: Charakterisierung der Zellbibliothek

In Abb. 26 sind die notwendigen Schritte zur Charakterisierung und zum Aufbau der Containerbibliothek illustriert. Die MTB generiert aus den Spice-Modellen mit Hilfe von Spectre die Charakterisierungsergebnisse. Um wirklich alle möglichen Eingangssignaländerungen korrekt mit der MTB zu berücksichtigen, sind Stimuli und für einige Zellen Wahrheitstabellen (truth tables) nötig. Die Stimuli werden mit einem dafür erstellten Werkzeug (`stt_gen`) generiert. Aus den Ergebnissen werden mit dafür erstellten Werkzeugen automatisch die Containerbibliothek `energy_estim` und auch die Bibliothek der Eingangskapazitäten erstellt, welche für die Modifikation der Netzlisten mittels `cap_to_netlist` (Abb. 16 auf S. 21) benötigt wird.

Die Containerbibliothek `energy_estim` enthält die Datenbasis und die Mechanismen für die Abschätzungen, einschließlich der Behandlung von partiellen Umladungen. Lediglich dafür notwendige Funktionen (z. B. für die Interpolation und das Modell des Umladestroms) sind in einer zusätzlichen VHDL `package` ausgelagert.

2.2.3. Ergebnisse

Die Genauigkeit der Energieabschätzung mit VHDL `energy_estim` wurde im Vergleich mit Spectre-Simulationen von Netzlisten ausgewählter Schaltungen bestimmt. Durch den Einsatz von Netzlisten fließen keine Leitungskapazitäten ein und daher wurde für `energy_estim` ebenfalls kein Wire Load Model eingesetzt. Für die Bewertung wurde eine 0,5μm Standardzellbibliothek verwendet. Statische Verluste sind in dieser Bibliothek zu vernachlässigen. Die Vergleichbarkeit ist gegeben, da `energy_estim` ebenfalls in der betrachteten Realisierung nur dynamische Verluste berücksichtigt.

Tabelle 6: Verwendete Bezeichner

Bezeichner	Kurzbeschreibung
W_s	Energie, Spectre
W_{ee}^0	Energie, `energy_estim`, keine Berücksichtigung partieller Umladungen
W_{ee}^R	Energie, `energy_estim`, „Rechteck“-Strommodell
W_{ee}^D	Energie, `energy_estim`, „Dreieck“-Strommodell
ε_{ee}^0	relativer Fehler $\frac{W_{ee}^0 - W_s}{W_s}$
ε_{ee}^R	relativer Fehler $\frac{W_{ee}^R - W_s}{W_s}$
ε_{ee}^D	relativer Fehler $\frac{W_{ee}^D - W_s}{W_s}$

In Tab. 6 sind die verwendeten Bezeichner der in Tab. 7 angegebenen Simulationsergebnisse aufgeführt. Die Behandlung partieller Umladungen wurde in Kap. 2.2.2.4 diskutiert. Erst mit deren Berücksichtigung werden akzeptable relative Fehler erreicht (These 2c). Die Ergebnisse der beiden Modelle für partielle Umladungen sind sehr ähnlich, sodass eine weitere Verfeinerung des Modells nicht erfolgversprechend erscheint. Über die verwendeten Testschaltkreise und die Tests gibt Tab. 8 einen Überblick. Die Angabe der Zellfläche erfolgt in AND2-Äquivalenten.

Tabelle 7: Spectre vs. `energy_estim`

Test	W_s [nJ]	W_{ee}^0 [nJ]	ε_{ee}^0	W_{ee}^R [nJ]	ε_{ee}^R	W_{ee}^D [nJ]	ε_{ee}^D
cpu_inst	936,4	863,497	**−7,79**%	839,942	−10,30%	840,067	−10,29%
cpu_mul	193,0	185,443	−3,92%	178,740	−7,39%	178,847	−7,33%
cpu_swap	260,5	257,385	−1,20%	249,041	−4,40%	249,114	−4,37%
cpu_ladd	213,9	198,610	−7,15%	191,353	**−10,54**%	191,362	**−10,54**%
mssp	104,5	128,261	22,74%	126,770	**21,31**%	126,723	**21,27**%
mul_csa	136,2	190,637	**39,97**%	154,244	13,25%	152,567	12,02%
mul_bcpt	111,1	136,396	22,77%	126,606	13,96%	126,425	13,79%
mul_blfcsa	109,5	114,109	4,21%	103,760	−5,24%	103,630	−5,36%

2.2.3.1. Laufzeit Die Spectre-Simulation des Instruktionstests der CPU des Microcontrollers (cpu_inst) dauert mehr als 4,5 Tage, während die Laufzeit der Abschätzung mit `energy_estim` 23 Sekunden und die der Simulation mit der ursprünglichen Netzliste nur 8 Sekunden beträgt. Damit ist `energy_estim` in diesem Fall lediglich um den Faktor

Tabelle 8: Verwendete Testschaltkreise

Test	Beschreibung	Zellfläche (AND2)
cpu_inst, cpu_mul cpu_swap, cpu_ladd	CPU eines MSP430 Microcontrollers (diverse Softwaretests)	2698
mssp	Peripheriekomponente für SPI/I^{2}C	834
mul_csa	Hardwaremultiplizierer [26]	1289
mul_bcpt	(200 Multiplikationen einer	1258
mul_blfcsa	diskreten Wavelet-Transformation)	1284

3 langsamer als die VHDL-Simulation der originalen Netzliste und rund um den Faktor 17000 schneller als die Spectre-Simulation. Die anderen Tests haben noch kürzere Laufzeiten, sodass der Start des VHDL-Simulators die Relationen stark verzerrt. (Die Laufzeiten von Spectre für die anderen Tests lagen bei einem halben bis einem Tag.) Dieser Geschwindigkeitsvorteil wird durch einen Abschätzungsfehler erkauft, welcher bei den simulierten Schaltkreisen bei Berücksichtigung partieller Umladungen weniger als 22% beträgt.

Nicht nur der Start des VHDL-Simulators verzerrt die Relationen für die Geschwindigkeit, sondern auch der notwendige Simulationsaufwand für die gesamte Testbench. Bei den durchgeführten Tests handelt es sich um einen vollständigen Simulationsaufbau eines Microcontrollers mit diversen Peripheriebaugruppen, die zwar nur als Verhaltensmodell mitsimuliert werden und auch funktional von den Testfolgen unberührt bleiben, aber dennoch den Simulator vor einen gewissen Verwaltungsaufwand stellen. Um die Geschwindigkeit von `energy_estim` in Relation zur Simulation mit der ursprünglichen Netzliste zu setzen, wurde daher eine willkürlich gewählte Simulation durchgeführt, welche nahezu komplett aus Netzlisten besteht. Die Stimuli werden durch einen Pseudo-Random Generator bereitgestellt, welcher ebenfalls synthetisiert wurde. Mit ursprünglichen Netzlisten beträgt die Simulationszeit 115 Sekunden und mit `energy_estim` 458 Sekunden für einen willkürlich gewählten Simulationszeitraum. Damit ist `energy_estim` in diesem Fall um den Faktor 4 langsamer, als die Simulation mit ursprünglichen Netzlisten.

Die Simulationszeit von `energy_estim` hängt davon ab, wie häufig ein Prozess zur Energieabschätzung (innerhalb eines VHDL `process`) durch ein Ereignis in seiner Sensitivitätsliste ausgelöst wird und natürlich wie komplex dieser Prozess ist. Gatter mit mehreren Ausgängen haben beispielsweise geringfügig aufwändigere Energieabschätzungen als solche mit nur einem Ausgang. Treten partielle Umladungen auf, müssen zusätzlich zur normalen Abschätzung die entsprechenden Korrekturen durchgeführt werden. Andererseits ist häufig die zu untersuchende Teilkomponente in Relation zum simulierten Gesamtobjekt (inklusive Testumgebung) klein, sodass die reine Simulationszeit für `energy_estim` durch den Gesamtaufwand überlagert wird. Vorsichtige Schätzungen für vollständig synthetisierte Schaltungen müssen Faktor 4 bis 5, realistische für synthetisierte Teilkomponenten bis Faktor 3 für den Mehraufwand für `energy_estim` im Vergleich zur Simulation mit ursprünglichen Netzlisten annehmen.

Natürlich existieren auch Anwendungsfälle, in denen die Simulationsgeschwindigkeit mit `energy_estim` zu einem kritischen Faktor wird. Bei sehr vielen Fällen ist aber die reine Simulationszeit mit `energy_estim` so kurz, dass sie vernachlässigt werden kann. Wesentlich ist dann das einfache Setup, das lediglich aus dem Austausch der Bibliotheksreferenzierung in der Netzliste (Listing 1) und ihrer automatischen Modifikation mittels `cap_to_netlist` (Abb. 16) besteht. Nichtsdestotrotz ermöglicht die hohe Geschwindigkeit von `energy_estim` erst Abschätzungen, die mit einem Schaltkreissimulator praktisch unmöglich wären. Dies gilt auch im Vergleich mit einem vereinfachten Schaltkreissimulator wie PowerMill.

2.2.3.2. Genauigkeit Der Vergleich der erreichten Genauigkeit und Geschwindigkeit von `energy_estim` mit den in Kap. 2.1 vorgestellten Methoden zeigt, dass `energy_estim` ungenauer als vereinfachte Schaltkreissimulatoren, aber auch deutlich schneller ist. Die Geschwindigkeit statistikbasierender Abschätzungsmethoden auf Netzlistenebene wird nur durch die Anzahl der Zellen einer Komponente bestimmt. Solche Abschätzungen sind damit schneller als simulationsbasierende Methoden und damit auch `energy_estim`. Dies setzt allerdings vorher die Aufnahme der statistischen Daten voraus, welche auch wieder eine konventionelle Netzlistensimulation mit zusätzlichem Abspeichern der Schalthäufigkeiten darstellt. Die Genauigkeit dieser statistikbasierenden Methoden wird in der Literatur nur selten diskutiert. Es muss aber angenommen werden, dass sie in vergleichbaren Regionen liegt wie die von `energy_estim`.

Die Methode der Abschätzung von Verlusten mit vorausberechneten Werten berücksichtigt die Geschwindigkeit einer Eingangssignaländerung, also die Steilheit der Flanken nicht. Die Charakterisierung der Zellbibliothek wurde mit einer „typischen“ Flankensteilheit durchgeführt. Zwar werden durch das Synthesewerkzeug ausreichend starke Treiber bzw. zusätzliche Puffer eingesetzt, um zu langsame Signaländerungen zu vermeiden, aber im konkreten Einzelfall bestimmt die jeweils vorhandene kapazitive Last die tatsächliche Flankensteilheit. Die Wahl einer „typischen“ Flankensteilheit, also einem Mittelwert bei der Charakterisierung, beeinflusst demnach die Qualität der Abschätzung.

Für die Behandlung von Teilumladungen wurde die Gleichartigkeit von Auf- und Entladung angenommen, um relative Zeitanteile abzuschätzen. Auch dies stellt nur ein einfaches, handhabbares Modell für die komplexere Realität dar.

2.2.3.3. Vergleich zu bekannten Methoden Ein direkter Vergleich zu bekannten Methoden ist nicht einfach zu ziehen, da nicht immer ein Zugriff auf diese Methoden oder die in Veröffentlichungen verwendeten Schaltkreise zum Vergleich möglich ist. Es bleibt daher nur der Vergleich veröffentlichter Werte. Diese haben allerdings sehr unterschiedliche Bezugspunkte. Somit kann im Endeffekt nur ein grober allgemeiner Vergleich gezogen werden. Ein solcher ist in Tab. 9 angegeben.

Dabei bezeichnet F den Faktor, um den die angegebene Methode schneller als die Referenz ist. Weiterhin bezeichnen F_{norm} den normierten Geschwindigkeitsfaktor, ε die relative Genauigkeit gegenüber einer Referenz und ε_{norm} die normierte Genauigkeit. Normierungsbezug sind Analogsimulatoren. In Näherung wird die Genauigkeit und die

Tabelle 9: Vergleich verschiedener Verlustabschätzungsmethoden

Methode	Referenz	F	ε [%]	F_{norm}	ε_{norm} [%]
Analogsimulation	-	1	0	1	0
`energy_estim`	Spectre	17000	22	17000	22
PowerMill [30]	HSPICE [31]	< 300	13	< 300	13
NanoSim [48]	PowerMill	1	1	< 300	13
PowerCompiler [49]	NanoSim [50]		10		13+10
[37]	Analogsimulation		60		60
TPS [39]	HSPICE	< 34000	40	< 34000	40
Pythia [41]	PowerMill	23	2	$23 \cdot 300 \approx 7000$	13+2

Geschwindigkeit von HSPICE, Spectre und anderen Analogsimulatoren als gleich angenommen. Bei der Normierung relativer Fehler muss für eine worst-case Abschätzung die Addition der relativen Fehler angenommen werden. Da sich diese aber auch auslöschen können, ist in Tab. 9 die Summe angegeben.

`Energy_estim` erreicht vergleichsweise hohe Simulationsgeschwindigkeiten, wobei der relative Fehler in meist akzeptablen Bereichen bleibt. Genauere Methoden sind allerdings auch meist deutlich langsamer.

PowerCompiler setzt ebenfalls auf eine im Voraus charakterisierte Bibliothek, schätzt aber die Verluste anhand statistischer Daten ab. Die Geschwindigkeit von PowerCompiler ist nicht bekannt. Mindestens eine Simulation mit einer HDL ist allerdings nötig, um statistische Daten zu Schalthäufigkeiten zu sammeln. Die Verlustabschätzung erfolgt danach mit den gesammelten Daten. Somit ist davon auszugehen, dass PowerCompiler mehr als die Simulationszeit für eine Netzlistensimulation benötigt, während es bei `energy_estim` etwa die dreifache Simulationszeit der Netzlistensimulation nötig ist. Wesentlicher Vorteil von `energy_estim` ist aber die Verschmelzung von Abschätzung und Verhaltenssimulation.

2.2.4. Wertung und Ausblick

Die vorgestellten Methoden, die in `energy_estim` realisiert wurden, sind die Berechnung von Verlustwerten im Voraus und die Einbettung von darauf basierenden Abschätzungen in VHDL-Containerzellen. Diese Methoden ermöglichen eine schnelle Abschätzung von Verlusten bei geringem Aufwand für das Setup. Sie gestatten weiterhin die einfache Zuordnung von Verlusten zu funktionalem Verhalten (vgl. Abb. 17, S. 22). `Energy_estim` eignet sich damit besonders zum schnellen Vergleich verschiedener Realisierungsmöglichkeiten für Komponenten einer Schaltung und natürlich auch zur Simulation der gesamten Schaltung inklusive einer (aufwändigen) Testumgebung. Der Nachteil ist der relative Fehler, der bei den betrachteten Schaltkreisen unter 22% für die Abschätzung dynamischer Verluste in den Zellen geblieben ist.

Das Prinzip von `energy_estim` ist völlig unabhängig von der konkreten VHDL-Simulationsumgebung, da ausschließlich VHDL-Routinen (VHDL'93) zum Einsatz kommen und keine Interaktion mit einem speziellen Simulator vorhanden ist. Lediglich die realisierten Werkzeuge zur automatisierten Erstellung der Containerbibliothek (Kap. 2.2.2.5) sind

abhängig von der Charakterisierung der Zellbibliothek und der Umsetzung der originalen VHDL-Zellbibliothek. Diese Werkzeuge extrahieren aber nur Daten aus Textdateien und schreiben diese in geeigneter Form in andere Textdateien, sodass einer Modifikation keine prinzipiellen Probleme entgegenstehen.

Die Erweiterung der Abschätzung durch Betrachtung von Leitungsverlusten und statischen Verlusten ist nötig für genaue absolute Abschätzungen. Insbesondere beim Entwurf von Hardwarebaugruppen sind aber hauptsächlich relative Vergleiche zwischen verschiedenen Alternativen von Interesse. Hier kann die vorgestellte Abschätzung der dynamischen Zellverluste schon ausreichend sein. Die Methoden der Berechnung von Verlustwerten im Voraus und die Einbettung von darauf basierenden Abschätzungen in VHDL-Containerzellen sind auch auf Leitungsverluste und statische Verluste erweiterbar, wie im Folgenden noch als Ausblick gezeigt wird. Ebenso soll als Ausblick die Berücksichtigung variabler Versorgungsspannung skizziert werden (These 2d).

Die nachfolgend erwähnten Modelle zur Abschätzung von Leitungsverlusten und statischen Verlusten sowie die Berücksichtigung variabler Versorgungsspannung besitzen ihrerseits Modellfehler. Die Verbesserung von Modellen für diese Problemstellungen soll nicht Gegenstand in dieser Arbeit sein. Um eine Bewertung der Abschätzungsqualität von `energy_estim` für dynamische Verluste in Zellen unabhängig von diesen Modellfehlern durchzuführen, blieben solche Einflüsse bis hierhin unberücksichtigt.

2.2.4.1. Leitungsverluste Die Betrachtung von Leitungsverlusten ist einfach durch die Einbettung von einem Wire Load Model in das Werkzeug `cap_to_netlist` möglich. Die Qualität der Abschätzung wird dann durch das verwendete Modell bestimmt.

In [46] ist die Funktionsweise eines Wire Load Models beschrieben: Ausgehend vom Fanout wird eine Lastabschätzung für die Verbindungsleitungen durchgeführt. Der Fanout eines Treibers ist die Anzahl von Verbindungen zu dem getriebenen Netz minus eins. Dem Fanout wird in einem statischen Wire Load Model eine Einheitslänge zugeordnet (Beispiel in Tab. 10). Bei der Bestimmung dieser Relation wird u. a. davon ausgegangen, dass Netze mit vielen Verbindungen zu Gattern häufig auch besonders lange Verbindungswege überbrücken. Die Leitungskapazität ist das Produkt aus Einheitslänge und einem Kapazitätsfaktor. Die Relation zwischen Fanout und Einheitslänge sowie der Kapazitätsfaktor sind technologieabhängige Größen.

Tabelle 10: Paare für Fanout und Einheitslänge (Beispiel aus [46])

Fanout	Einheitslänge
1	1,0
2	2,2
3	3,3
4	4,4

Mit diesem Wire Load Model ist es möglich, lediglich aus dem Fanout die Leitungskapazität abzuschätzen. `Cap_to_netlist` ordnet jeder Zelle die Eingangskapazitäten der getrieben Gatter zu. Da dabei auch der Fanout bekannt ist, kann direkt die abgeschätz-

te Leitungskapazität zu der Summe der Eingangskapazitäten hinzu addiert werden. Die Erweiterung von `cap_to_netlist` um ein solches Wire Load Model ist also sehr simpel.

Neben einem Wire Load Model sind auch Verzögerungszeiten durch die Leitungen zu berücksichtigen. Der Mechanismus dafür ist die bekannte „SDF Backannotation“. In der `entity` von Zellverhaltensbeschreibungen existieren generische Parameter, die entsprechende Verzögerungszeiten repräsentieren. Die generischen Parameter werden bei der SDF Backannotation mit den jeweiligen Verzögerungszeiten aus den Standard Delay Files (SDF) überschrieben. Um SDF Backannotation auch mit `energy_estim` möglich zu machen, müssen in der `entity` der Container-Zelle (Abb. 15, S. 21) die gleichen generischen Parameter vorhanden sein wie auch in der jeweiligen instanziierten Verhaltensbeschreibung. Diese können dann mittels `generic map` weitergereicht werden. Diese Herangehensweise kann allerdings nicht zum Erfolg führen, wenn der VHDL-Simulator die durch die SDF Backannotation überschrieben generischen Parameter nicht in die instanziierte Zelle propagiert. Als Lösung bleibt dann nur das Flachmachen der Hierarchie, also die Verschmelzung von Verhaltensbeschreibung und Verlustabschätzung. Eine Verschmelzung führt dazu, dass die dadurch entstandene Zelle nicht mehr konform zum VHDL VITAL-Standard [47] ist. Eine Simulation ist dennoch problemlos möglich.

Tabelle 11: Spectre (Layout) vs. `energy_estim` (mit WLM und SDF Backannotation)

Test	$W_s\,[nJ]$ Layout	W_{ee}^D [nJ] WLM	ε_{ee}^D WLM	W_{ee}^D [nJ] WLM, SDF	ε_{ee}^D WLM, SDF
mssp	87,54	107,16	22,4%	104,80	19,72%
cra	27,78	29,72	7,0%	27,52	−0,94%
prg	82,50	91,80	11,3%	92,00	11,50%

In Tab. 11 wird der Einfluss von Wire Load Model (WLM) und SDF Backannotation auf die Abschätzung mittels `energy_estim` dargestellt. Die bei der Spectre-Simulation bestimmte Energiemenge W_s (hier nun für Layouts) für die dargestellten Testszenarien dient wieder als Maßstab für die Genauigkeit. Aufgrund der enormen Laufzeit der Spectre-Simulationen von Layouts konnten nicht die gleichen Testschaltkreise wie in Tab. 7 verwendet werden. Neben der Peripheriekomponente für SPI/I^2C (mssp) wurden ein 32 Bit Carry-Ribble Adder (cra) und ein Pseudo-Random Generator (prg) zu den Tests herangezogen. Die Abschätzung allein mit Berücksichtigung eines Wire Load Models besitzt nur einen geringfügig größeren relativen Fehler als die Abschätzung von nur zellinternen Verlusten (vgl. Tab. 7). Wird SDF Backannotation durchgeführt, verringerte sich der maximal beobachtete relative Fehler sogar, sodass 20% relativer Fehler für absolute Abschätzungen im Vergleich zu Spectre-Simulationen von Layouts erreicht werden. Während die Simulationszeit von Spectre bei Layouts ein Vielfaches der Zeit bei Simulation von Netzlisten beträgt, bleibt die Zeit für `energy_estim` gleich.

Notwendigkeit der Berücksichtigung von Leitungsverlusten Es bleibt die Frage: Ist es nötig, Leitungsverluste abzuschätzen oder sind Zellverluste ausreichend beim Vergleich der Verluste verschiedener Realisierungen?

Verlustreduktion geschieht durch Verringerung der Anzahl der Zellen (logische Vereinfachung) und der Aktivität von Zellen (z. B. durch clock gating). Reduktion von Zellanzahl und Zellaktivität führt direkt auch zu Reduktion von Leitungsanzahl und Leitungsaktivität und umgekehrt. Die Berücksichtigung von ausschließlich Zellverlusten ist demnach für diese Methoden ausreichend, um durch Vergleich zu entscheiden, welche von mehreren Realisierung optimaler in Bezug auf Verluste ist.

Werden Leitungsverluste berücksichtigt, so werden höhere Verluste abgeschätzt, je höher die Lasten durch Leitungen und je höher die Aktivität von Leitungen mit hohen Lasten sind. Eine gezielte Beeinflussung auf der Ebene einer HDL ist nur in einigen seltenen Fällen möglich, z. B. wenn für eine logische Entscheidung mehrere Signale als Bedingung zur Verfügung stehen. Die Aktivität schlägt sich auch in Zellverlusten nieder und die Last einer Leitung, die in einem Wire Load Model nur vom Fanout abhängt, ist durch die logische Funktion festgelegt, sodass eine Berücksichtigung von Leitungsverlusten die Entscheidung beim Vergleich von Realisierungen nur bei nahe beieinander liegenden Abschätzungsergebnissen beeinflussen würde.

Aufgrund der Bijektivität der oben genannten Implikation und der meist nicht vorhandenen Möglichkeit zur gezielten Beeinflussung von Leitungsverlusten kann somit gefolgert werden, dass die Berücksichtigung von Zellverlusten und das Ignorieren von Leitungsverlusten beim relativen Vergleich von verschiedenen Realisierungen in den meisten Fällen ausreichend ist.

2.2.4.2. Statische Verluste Statische Verluste sind insbesondere bei geringen Strukturgrößen von entscheidender Bedeutung für die Gesamtverluste. Das einfachste Modell für statische Verluste ist die Annahme einer konstanten spezifischen Verlustleistung pro Gatter. Die Summe der statischen Verlustleistungen aller Gatter multipliziert mit der Laufzeit ergibt dann die gesamte statische Verlustenergie. Ein solches Modell wird eine starke Abhängigkeit von der Gatterfläche besitzen.

Ein detaillierteres Modell wird abhängig vom Zustand der Eingangssignale und des gespeicherten Zustandes bei sequentiellen Zellen den Gattern entsprechend statische Verluste zuordnen. Die statischen Verlustleistung pro Gatterzustand kann ebenso wie die dynamische Verlustenergie vorausberechnet und in Tabellen für alle Gatter abgespeichert werden. In `energy_estim` kann ein solches Modell mit gewissen Nachteilen auf die folgende Weise implementiert werden:

1. `Energy_estim` arbeitet ereignisgesteuert und der Zeitpunkt des letzten Ereignisses sowie der Wert aller Eingangssignale (und evtl. des gespeicherten Zustandes) sind bekannt. Mit dem Eintreten eines neuen Ereignisses (t_y) kann somit die seit dem letzten Ereignis (t_x) umgesetzte statische Verlustenergie ($W_s(t_x, t_y)$) durch Tabellenabfrage der Verlustleistung (P_s(Gatterzustand)) und Multiplikation mit der seit dem letzten Ereignis vergangenen Zeit bestimmt und zur Gesamtverlustenergie hinzu addiert werden.

$$W_s(t_x, t_y)|_{\text{Gatter}} = P_s(\text{Gatterzustand}) \cdot (t_y - t_x) \tag{22}$$

   Ein Nachteil dieser Methode ist der Fehler, der beim Simulationsende auftritt. Da

dann kein Ereignis mehr eintritt, bleibt die statische Verlustenergie seit dem letzten Ereignis unbeachtet (Simulationsendefehler). Ein weiterer Fehler entsteht aufgrund der Tatsache, dass nur mit jedem Ereignis die statischen Verluste bestimmt werden, zwischen den Ereignissen die Verluste aber unberücksichtigt bleiben (Schrittfehler).

2. Die Fehler der ersten Methode können kompensiert werden, wenn zusätzlich zu jedem Ereignis noch die Bestimmung der statischen Verlustenergie bei Bedarf erzwungen wird. Das Erzwingen kann durch ein zusätzliches Signal geschehen, welches an alle Containerzellen geleitet wird. Um die `entity` der Containerzellen nicht durch ein zusätzliches Signal zu verändern, kann ein globales Signal verwendet werden. Ein solches wird innerhalb einer VHDL `package` deklariert und der Zugriff ist von allen Zellen möglich, die diese `package` einbinden. Der Testbench fällt die Aufgabe zu, das globale Signal zu steuern (Abb. 27).

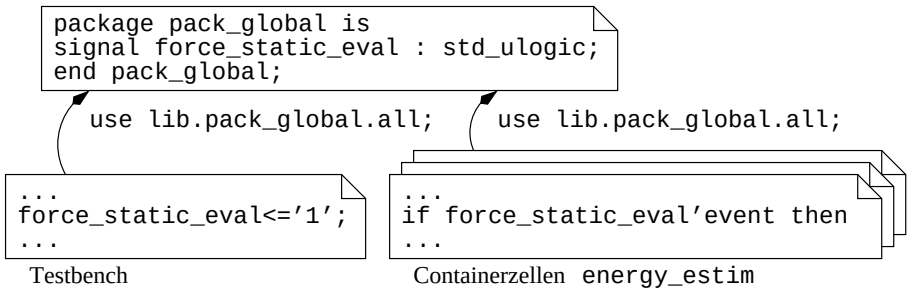

Abbildung 27: Einsatz eines globalen Signals

Je häufiger durch das globale Signal eine Bestimmung der statischen Verluste ausgelöst wird, desto geringer werden Simulationsende- und Schrittfehler, aber desto länger wird auch die Simulationszeit. Kann der Schrittfehler ignoriert werden, so ist es ausreichend, nur einmalig vor Simulationsende die Bestimmung der statischen Verluste zu erzwingen.

Damit ist gezeigt, dass eine Abschätzung von statischen Verlusten mittels der Methoden von `energy_estim` durchaus möglich ist.

2.2.4.3. Variable Versorgungsspannung Die von `energy_estim` verwendeten vorausberechneten Werte beziehen sich auf eine konkrete Versorgungsspannung. Ist in einem System die Versorgungsspannung variabel, muss dies berücksichtigt werden. Möglich ist die Charakterisierung der Zellbibliothek für verschiedene Versorgungsspannungen und das Hinzufügen dieser Werte in die Datenbasis von `energy_estim`. Während der Abschätzung können mittels Interpolation auch Zwischenwerte bestimmt werden. Die Nachteile dieser Methode sind die Vervielfachung des Aufwandes zur Charakterisierung und die Vergrößerung der Containerzellen, und damit die Verringerung der Abschätzungsgeschwindigkeit.

Da aber die Relationen von Spannung und dynamischer sowie statischer Verlustleistung bekannt sind, können auch Skalierungsfaktoren eingesetzt werden. Somit müssen

diese Skalierungsfaktoren nur für verschiedene Versorgungsspannungen bestimmt werden, während die Verlustabschätzung nur unwesentlich aufwändiger wird. Die Einbettung in die VHDL-Umgebung ist sehr einfach. Benötigt wird eine Komponente, die die Spannungsregulierung abstrakt simuliert und die tatsächlich anliegende Spannung über Signale an `energy_estim` meldet, sodass für die Abschätzung von dynamischen sowie statischen Verlusten die entsprechenden Skalierungsfaktoren ausgewählt werden können (Abb. 28).

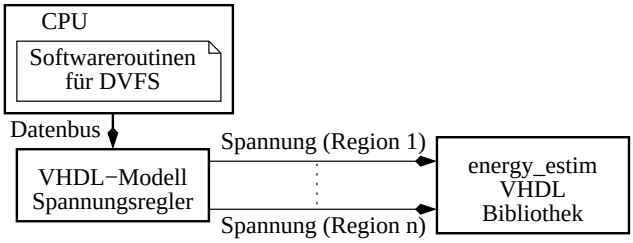

Abbildung 28: Berücksichtigung variabler Versorgungsspannung in `energy_estim`

```
I1 : cell_a  generic map( fanout_q=>63, voltage_region=>2)
             port map(A=>net1 ,B=>net2 ,Q=>net3 );
```

Listing 3: Modifizierte Instanz in einer Netzliste (Fragment)

Die Methode stößt an ihre Grenzen, wenn in einem Schaltkreis für mehrere Regionen mit unterschiedlicher Versorgungsspannung gleichzeitig die Verluste abgeschätzt werden sollen. Auch das ist möglich, aber etwas komplizierter: Während der Modifikation einer Netzliste mittels `cap_to_netlist`, also bei der Zuweisung der kapazitiven Lasten zu den instantiierten Zellen, kann ein weiterer generischer Parameter die Zugehörigkeit zu einer Versorgungsspannungsregion festlegen. Im Beispiel in Listing 3 wird die Zelle der Region mit der Nummer 2 zugeordnet. Für jede dieser Regionen muss durch die abstrakte Komponente zur Spannungsregulierung die jeweils anliegende Spannung signalisiert werden. Bei der Abschätzung mit jedem Ereignis eines Signals kann somit abhängig von der zugeordneten Spannungsregion die entsprechende Versorgungsspannung ausgewählt und der zugehörige Skalierungsfaktor verwendet werden.

2.2.5. Anwendungsbeispiele beim Hardwareentwurf

Verlustabschätzung mit `energy_estim` kann für die Charakterisierung kompletter Baugruppen während der Interaktion von Software mit Hardware eingesetzt werden, wie während der Präsentation der Ergebnisse (Kap. 2.2.3) gezeigt wurde. Die zum Vergleich der Genauigkeit der Methode herangezogenen Beispiele stellen nichts anderes dar, als die Simulation konkreter durch Software vorgegebener Testszenarien.

Neben der Charakterisierung kompletter Baugruppen eignet sich `energy_estim` natürlich ganz besonders für die Bewertung von alternativen Hardwarerealisierungen und im

Speziellen ermöglicht der geringe Aufwand für das Setup die Betrachtung vieler Alternativen. Im Folgenden sollen beispielhaft Designalternativen für einige Problemstellungen mit Hilfe von `energy_estim` beleuchtet werden. Es wurde die schon erwähnte 0,5μm Standardzellbibliothek zugrunde gelegt. Alle Angaben beziehen sich auf die synthetisierte Netzliste.

Der Einfluss der in den folgenden Beispielen dargestellten Einsparungen hängt direkt vom Anteil im Gesamtsystem ab. Aufwand und Nutzen müssen gegeneinander aufgewogen werden. Sind beispielsweise signifikante Einsparungen gefordert, lohnt es wenig, nur einen einzigen kleinen Addierer innerhalb einer großen Schaltung zu optimieren. Für ein System, in dem keine unnötigen Verluste toleriert werden können (wie z. B. in funkversorgten Transpondern), kann dagegen die Summe aus vielen kleinen Optimierungen über die Realisierbarkeit entscheiden.

2.2.5.1. Taktteiler und Zähler In Microcontrollern werden Baugruppen mit nur wenigen unabhängigen, meist sogar nur einem einzigen Takt versorgt. Taktteiler ermöglichen das Arbeiten aller Baugruppen mit möglichst niedrigem Takt. Muss bei Buskommunikation die Sendedatenrate an die Empfangsdatenrate angepasst werden und steht kein Taktsignal von der Gegenstelle sondern nur ein interner schneller Takt zur Verfügung, ist ebenfalls ein Taktteiler nötig.

Ein allgemeiner Taktteiler besteht aus einem Inkrementer, einem Vergleicher und einem Ausgabe-Flipflop, welches beim Erreichen einer Zählgrenze invertiert wird und den geteilten Takt repräsentiert. Als Alternative kann ein Dekrementer verwendet werden, der mit der Zählgrenze vorgeladen wird und bis zu Null zurück zählt. Das Carry-Out zeigt das Erreichen von Null an. Bei beiden Taktteilern ist es möglich, zur Laufzeit beliebige Zählgrenzen zu verwenden.

Das Problem der Taktteilung vereinfacht sich, wenn nur Divisoren 2^n mit $n \in \mathbf{N}$ zu realisieren sind. Ein synchroner Taktteiler, wie in Abb. 29 illustriert, besteht aus einer Gruppe von Flipflops, die alle mit dem ungeteilten Takt angesteuert werden. Das Taktsignal für den Divisor 2^n muss invertiert werden, wenn die Taktsignale für alle Divisoren 2^m mit $m \in [1, n-1]$ Null sind. Gegenüber einem Taktteiler mit Zähler vereinfacht sich damit die notwendige kombinatorische Logik und es stehen alle kleineren Divisoren 2^n mit $n \in \mathbf{N}$ parallel zur Verfügung. Alle geteilten Takte haben den gleichen Versatz zum ungeteilten Takt von nur einer Signalpropagationszeit von einem Flipflop.

Nachteilig in synchronen Realisierungen ist das Ansteuern aller Flipflops mit dem ungeteilten Takt. Der asynchronen Taktteiler vermeidet dies (Abb. 30). Es entsteht allerdings mit jeder Teilerstufe ein zusätzlicher Taktversatz. Ist dieser Versatz inakzeptabel und ist nur der geteilte Takt mit dem größten Divisor von Interesse, kann dieser Takt noch einmal mit dem ungeteilten Takt synchronisiert werden. Da das Synchronisationsflipflop mit dem ungeteilten Takt arbeitet, kann ein solcher Taktteiler erst bei höheren Divisoren effizient sein. Die Phasenverschiebung durch die Resynchronisation ist meist uninteressant.

Ausgehend von der Idee des asynchronen Taktteilers kann auch ein asynchroner Inkrementer entworfen werden (Abb. 31). Nötig ist eine zusätzliche synchrone Prüfung, ob

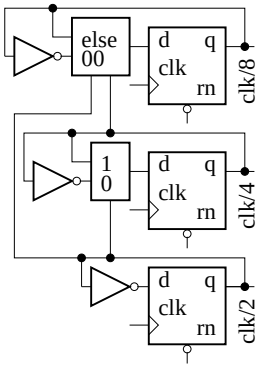

Abbildung 29: Synchroner Taktteiler

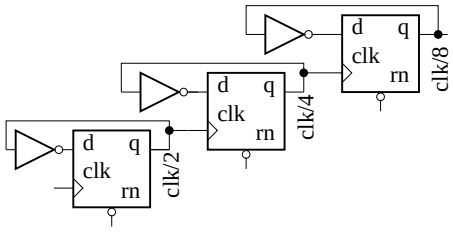

Abbildung 30: Asynchroner Taktteiler

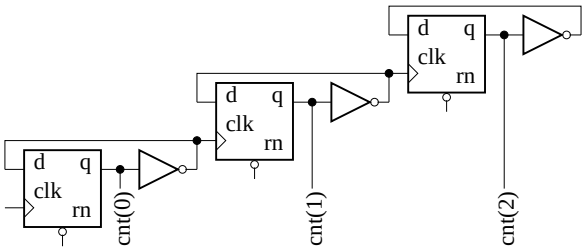

Abbildung 31: Asynchroner Inkrementer

die Zählgrenze erreicht wurde. Ist dies der Fall, muss der Zähler zurückgesetzt werden. Für Bit 0 kann ein synchrones Rücksetzen erfolgen, während für alle anderen Bits ein asynchrones Rücksetzsignal für eine Taktperiode aktiv sein muss.

Abb. 32 zeigt einen Vergleich der verschiedenen Taktteiler bezüglich ihrer Verlustenergie pro ungeteiltem Taktschritt. Auffällig ist beim asynchronen Inkrementer die sinkende Verlustenergie mit der Größe des Divisors, was auf das notwendige asynchrone Reset zurückzuführen ist, welches bei niedrigeren Divisoren häufiger aktiv wird als bei höheren und bestimmend für die Verlustenergie ist.

Es sei nochmals darauf hingewiesen, dass Inkrementer und Dekrementer beliebige (geradzahlige) Divisoren realisieren können, während die reinen Taktteiler auf Divisoren 2^n mit $n \in \mathbf{N}$ festgelegt sind, welche beim Vergleich in Abb. 32 ausschliesslich berücksichtigt wurden. Asynchrone Realisierungen setzen deutlich weniger Verlustenergie um.

2.2.5.2. Multiplexer Serialisierung und Parallelisierung tritt insbesondere bei Kommunikation über ein Bussystem auf und wird durch Multiplexer und Demultiplexer behandelt. (De)Multiplexer können abhängig von der Anzahl der Auswahlmöglichkeiten sehr groß werden, sodass in ihnen große Energiemengen umgesetzt werden. Funktionale Vereinfachungen sind kaum möglich, aber unnötige Umladungen können durch kaskadierte (De)Multiplexer vermieden werden.

Soll beispielsweise aus einer Registerbank ein Datenwort und aus dem Datenwort ein

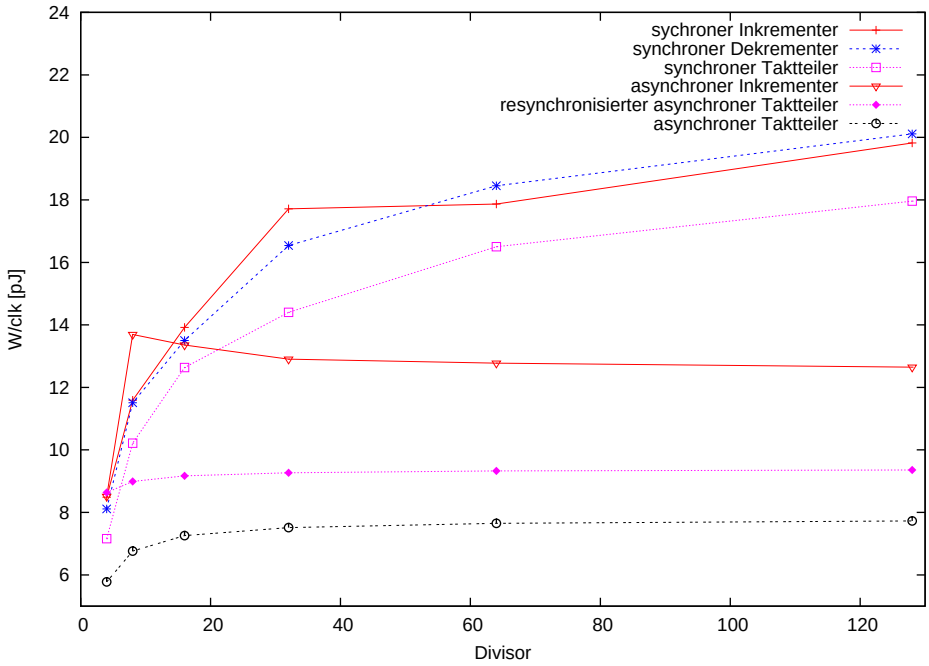

Abbildung 32: Vergleich der Energieumsetzung pro Takt W/clk von Taktteilern

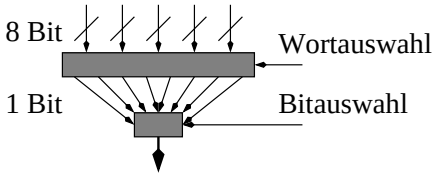

Abbildung 33: Kaskadierter Multiplexer

Bit ausgewählt werden, so kann eine Struktur, wie in Abb. 33 dargestellt, vorteilhaft sein. Sollen die Datenworte aus der Registerbank über ein serielles Bussystem ausgegeben werden, so kann der erste Multiplexer in einem stabilen Zustand verharren, bis alle Bits eines Datenwortes ausgegeben wurden. Somit arbeitet nur der zweite Multiplexer in jedem Taktschritt, während bei einer konventionellen Realisierung dagegen der gesamte Multiplexer arbeiten muss. Der im Beispiel aus Abb. 33 gezeigte kaskadierte Multiplexer ist gegenüber einem konventionellen Multiplexer um 1% größer, aber um 32% schneller und setzt 55% weniger Verlustenergie bei der Serialisierung von 5 willkürlich gewählten Wörtern mit 8 Bit Wortbreite um.

Der Einsatz eines Schieberegisters als Ersatz für den zweiten Multiplexer für das genannte Serialisierungsproblem ist nicht zu empfehlen. Ein solches Register (im Beispiel mit 8 Bit Breite) würde über den ersten Multiplexer mit Daten geladen werden und die auszugebenden Daten automatisch mit jedem Taktschritt serialisieren. Ein 8 Bit Schieberegister setzt aber das 7,3-fache der Energie des entsprechenden Multiplexers um. (Die

Steuerlogik, die für beide Möglichkeiten ähnlich ist und ein evtl. notwendiges Synchronisationsflipflop sind nicht im Vergleich enthalten. Ein Synchronisationsflipflop kann den energetischen Vorteil aber nicht aufwiegen.)

2.2.5.3. Busstrukturen Bei Microcontrollern werden die Komponenten meist in den Adressbereich der CPU eingeblendet und über den normalen Datenbus angebunden. Bei solchen Busstrukturen handelt es sich im Prinzip nur um Multiplexer (vgl. Kap. 2.2.5.2). Anhand eines Beispielsystems soll der Einfluss verschiedener ausgewählter Möglichkeiten zur Realisierung für eine Busstruktur veranschaulicht werden. Im Beispiel sind 8 Komponenten vorhanden, die jeweils 8 Registerworte zum Zugriff darauf bereitstellen. 3 Adressbits für die Register- und 3 für die Komponentenselektion sind also erforderlich. Die Registerwortbreite sei 8 Bit. Dieses System ist in Abb. 34 abstrakt illustriert.

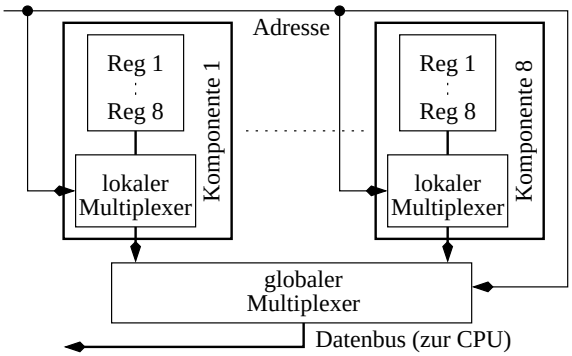

Abbildung 34: Beispiel eines allgemeinen Bussystems

Je nach Realisierung erfolgt ein unterschiedlicher Zugriff auf den Bus. Die abstrakten Multiplexer in Abb. 34 stellen diesen Zugriff her und bilden den Kern der verschiedenen Realisierungsmöglichkeiten. Folgende Möglichkeiten sollen betrachtet werden:

globale Auswahl: Die Komponenten stellen die Registerinhalte direkt zur Verfügung und nur ein globaler Multiplexer wählt anhand der Adresse das selektierte Register aus. Lokale Multiplexer existieren nicht.

tri-state Bus: Ein tri-state Treiber treibt den Datenbus, wenn ein Register der Komponente selektiert wurde. Globale Multiplexer existieren nicht.

OR Bus: Der Inhalt eines Registers wird auf den Ausgang geschrieben, wenn ein Register der Komponente selektiert wurde, anderenfalls werden Nullen geschrieben. Der globale Multiplexer ist eine logische Oder-Verknüpfung (OR) aller Komponentenausgänge.

AND Bus: Gleichartig zum OR Bus mit dem Unterschied, dass Einsen statt Nullen bei Nicht-Auswahl auf den Bus geschrieben werden.

Für tri-state, OR und AND Bus werden je zwei Varianten betrachtet:

1. Ein klassischer Multiplexer trifft eine Vorauswahl und leitet den Wert an den entsprechenden lokalen Multiplexer weiter.

2. Jedes Register besitzt einen eigenen, dem Bussystem entsprechenden Multiplexer.

In Tab. 12 sind die Ergebnisse eines Vergleichs für willkürlich gewählte Daten und Zugriffsmuster angegeben. Alle Werte sind normiert auf die Realisierungsmöglichkeit der globalen Auswahl. Während der Synthese wurde die Hierarchie beibehalten.

Tabelle 12: Vergleich verschiedener Busrealisierungen

Bustyp	Variante	Gatterfläche AND2	Geschwindigkeit [ps]	Verluste [nJ]
globale Auswahl	-	487 (100%)	6,4 (100%)	584 (100%)
tri-state Bus	1	523 (108%)	4,4 (69%)	2248 (385%)
tri-state Bus	2	955 (205%)	7,4 (115%)	759 (130%)
OR Bus	1	499 (103%)	4,7 (74%)	2014 (345%)
OR Bus	2	507 (104%)	5,6 (88%)	501 (**86%**)
AND Bus	1	543 (112%)	4,6 (73%)	1730 (296%)
AND Bus	2	491 (101%)	5,0 (78%)	507 (**87%**)

Tri-state Bussysteme können einfach beim Schaltkreisentwurf mitskalieren, da keine Anpassung eines globalen Multiplexers beim Hinzufügen oder Entfernen einer Komponente im Gegensatz zu den anderen Realisierungsmöglichkeiten nötig ist. Mit steigender Anzahl von tri-state Treibern und damit erhöhter Buslast müsste auch deren Treiberleistung steigen, sodass ein Skalieren nicht endlos möglich ist. In Tab. 12 ist zu erkennen, dass beim tri-state Bus Variante 2 vom Synthesewerkzeug stärkere und damit größere und langsamere Treiber eingesetzt wurden, als in Variante 1, wo durch den Vorauswahlmultiplexer die Anzahl der tri-state Treiber reduziert wurde.

Ob ein OR oder ein AND Bus zu bevorzugen ist, hängt von der Zellbibliothek ab - günstiger als eine globale Auswahl oder tri-state Busse sind beide Alternativen. Als ungünstig bezüglich der Verluste erweist sich ein Multiplexer zur Vorauswahl (Variante 1), da dieser ständig arbeitet (auch wenn die Komponente nicht selektiert wurde) und unnötig Energie umsetzt. Durch die Vorauswahl gleicht das System einem kaskadierten Multiplexer, aber im Gegensatz zum Serialisierungsproblem aus Kap. 2.2.5.2 ist hier die erste Multiplexerstufe hoch aktiv, was sich als ungünstig bei der Kaskadierung erweist.

2.2.5.4. Zustandsautomaten In vielen Zustandsgraphen existieren lineare Abfolgen ohne Verzweigungen. Ein Beispiel dafür ist die serielle Kommunikation, wo meist Informationsblöcke zu 8 Bit gruppiert sind (Adresse, Daten, Checksumme ...). Senden oder Empfangen eines Bits entspricht streng genommen einem Zustand im Zustandsautomaten, anhand dessen (De)Multiplexer gesteuert werden. Dennoch ist die betrachtete Gruppe von Bits ist einem gemeinsamen Teilproblem zugeordnet. Werden in Prozessoren Befehle linear in mehreren Schritten abgearbeitet, treten ebenfalls solche Graphen auf.

Ein Zustandsautomat kann demnach in Haupt- und Unter-Automat kaskadiert werden. Der Hauptautomat muss nur in den folgenden Zustand wechseln, wenn alle Unter-

zustände durchlaufen sind. Die Unterzustände können durch einen Zähler repräsentiert werden. Für den Hauptautomaten lässt sich somit Clock-Gating einsetzen. Abb. 35 illustriert das Funktionsprinzip.

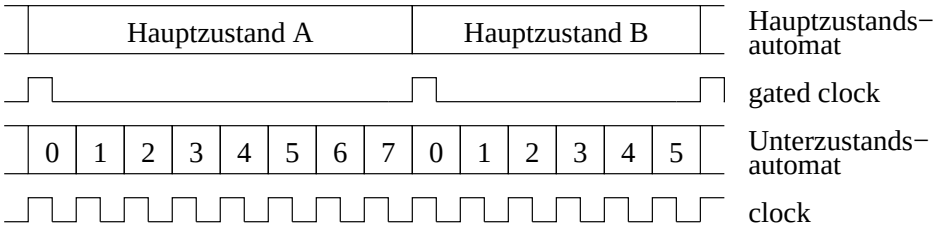

Abbildung 35: Einsatz von kaskadierten Zustandsautomaten

Kaskadierte Zustandsautomaten besitzen die folgenden Vorteile:

- Hauptzustände bleiben über längere Zeit stabil. Kombinatorische Logik, die daraus abgeleitet wird, arbeitet nicht. (vgl. kaskadierte Multiplexer in Kap. 2.2.5.2)
- Clock-Gating versetzt die Flipflops des Hauptautomaten temporär in Ruhe.
- Klassische Zustandswechsellogik beschränkt sich auf den Hauptautomaten. Der Unterautomat arbeitet mit einem einfachen Zähler und kann in verschiedenen Hauptzuständen wiederverwendet werden.

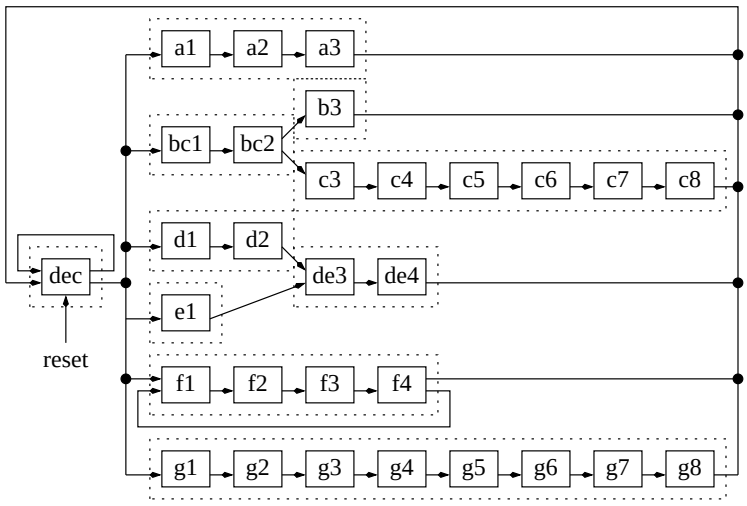

Abbildung 36: Beispiel: Zustandsautomat

In Abb. 36 ist ein willkürlich gewähltes Beispiel für den Vergleich zwischen klassischen und kaskadierten Zustandsautomaten gegeben. Die mittels Punktlinien umrahmten Zustände repräsentieren einen Hauptzustand. In diesem Beispiel werden also 30 konventionelle Zustände, bzw. 10 Haupt- und 8 Unterzustände benötigt. Durch den simplen

Zähler des Unterautomaten und dessen Wiederverwendung ist der kaskadierte Automat allerdings kleiner als der klassische Automat und durch das Clock-Gating des Hauptautomaten setzt er eine geringere Energiemenge um.

Tabelle 13: Vergleich konventioneller vs. kaskadierter Zustandsautomat

Automat	Flipflops	Gatterfläche AND2	Verluste [nJ]
konventionell	5	85 (100%)	495 (100%)
kaskadiert	7	75 (88%)	390 (79%)

In Tab. 13 werden die Ergebnisse präsentiert. Zusätzliche Energie kann in den durch den Automaten gesteuerten Baugruppen aufgrund des stabilen Hauptzustandssignals gespart werden.

2.3. Softwareinstruktionen und Verluste

Zur Bewertung der Qualität von Software ist es wünschenswert, die daraus resultierenden Verluste abzuschätzen. Die meisten Methoden zu Verlustabschätzung für Hardwarebaugruppen (vgl. Kap. 2.1) sind aber nicht auf Abschätzungen für Software übertragbar. Viele Methoden sind zu zeitaufwändig, um damit die Verluste einer großen Schaltung, wie einer CPU, über eine längere Zeit abzuschätzen. Statistische und wahrscheinlichkeitsbasierende Methoden haben keinen direkten Bezug zu den tatsächlichen Instruktionen. Lediglich Methoden, wie `energy_estim` (Kap. 2.2), können praktikabel sein, da sämtliche Datenabhängigkeiten berücksichtigt werden und der Zeitaufwand in akzeptablen Grenzen bleiben kann, sofern nicht übermäßig lange Programme untersucht werden. Nachteilig erweist sich allerdings der Medienbruch zwischen Softwareentwicklungsumgebung und VHDL-Simulation, sodass zwar Verluste zu logischen Schaltoperationen zugeordnet werden können, aber die Zuordnung zu Instruktionen und insbesondere zu Konstrukten in einer Hochsprache wenngleich zwar nicht unmöglich ist, aber schwer fällt.

Als Lösung dieses Problems erscheint die Zuordnung von Verlusten zu CPU-Instruktionen auf den ersten Blick günstig. Damit wäre nicht nur eine Bewertung von Software möglich, sondern daraus könnten auch Ansätze zur Softwareoptimierung erwachsen. Unter gewissen Umständen ergeben sich aber prinzipielle Probleme durch den Ansatz selbst und es kann gezeigt werden, dass nach einer Verallgemeinerung der Kosten für Softwareinstruktionen ein solcher Ansatz für die qualitative Bewertung von Software nicht nötig ist. Dies soll im folgenden diskutiert werden.

2.3.1. Bekannte Methoden

Tiwari [52] beschreibt den wohl ersten Versuch, Verluste zu Instruktionen zuzuordnen. Dazu wurde die Stromaufnahme einer Intel 486DX2-CPU gemessen. Da es mit einem Amperemeter unmöglich ist, die Stromaufnahme während einer einzelnen Instruktion zu messen, wurden sehr viele gleichartige Instruktionen nacheinander ausgeführt. Eine Endlosschleife ermöglichte einen kontinuierlichen Betrieb. Dabei wurde der Schleifenkörper

ausreichend groß gewählt, um den Einfluss der Schleifenkontrolle gering zu halten, aber auch ausreichend klein gehalten, um die Schleife im Cache der CPU zu halten, sodass zeit- und energieaufwändige Cache-Misses vermieden wurden. Allen CPU-Instruktion konnten damit sogenannte Basiskosten (base costs) zugeordnet werden. Einen Einfluss auf die umgesetzte Energie haben natürlich die Daten und Adressen, sowie die jeweils vorherige Instruktion (Inter-Instruction Effect). In [52] wird der Einfluss der Daten mit weniger als 5% angegeben. Der Inter-Instruction Effect führt zu nahezu konstanten zusätzlichen Verlusten und ist nur schwach abhängig von den Instruktionen. Der Einfluss liegt in Regionen von bis zu 10% - meist allerdings bei weniger als 5%. Begründet werden diese geringen Einflüsse damit, dass in einer komplexen CPU, wie dem 486DX2, eine große Menge an Energie in der Verwaltung der Pipeline, dem Taktnetzwerk und beim Laden und Decodieren der Instruktion umgesetzt wird, sodass andere Einflüsse vergleichsweise gering sind. In DSPs ist der Inter-Instruction-Effect etwas größer [54], während es zwischen RISC- und CISC-Architekturen keine wesentlichen Unterschiede gibt. Unter Berücksichtigung von Cache-Misses, Speicherzugriffen und Pipeline-Stalls wird mit dieser Abschätzungsmethode insgesamt eine Genauigkeit von 3% erreicht. Dies ist insofern bemerkenswert, da die genannten Einflüsse eine größere Schwankung aufweisen. Folglich müssen bei der Charakterisierung der Instruktionen Werte ermittelt worden sein, die auch bei den Testprogrammen wieder aufgetreten sind.

In [52] werden Compiler vorgeschlagen, welche, basierend auf den Verlusten für die einzelnen Instruktionen, die Optimierungen durchführen. Tiefgreifender untersucht wird dies in [53]. Da allerdings, wie schon ausgeführt, der Einfluss von instruktionsspezifischen Energiemengen auf die Gesamtenergie gering ist, musste geschlussfolgert werden, dass kurze Programme auch geringe Verlustenergie umsetzen. Die Energieunterschiede zwischen verschiedenen Instruktionen sind nie groß genug, um eine größere Anzahl von Taktzyklen zu kompensieren. Wesentlich für geringe Verluste sind demnach das Vermeiden von Speicherzugriffen und Cache-Misses und die möglichst effiziente Nutzung von Registern der CPU.

In [55] wird am Beispiel von StrongARM SA-1100 und Hitachi SH-4 detailliert aufgeschlüsselt, dass die Unterschiede zwischen Instruktionen bzgl. der Verluste sehr gering sind. Weiterhin wird die experimentelle Bestätigung gegeben, dass die dynamische Verlustenergie bei konstanter Spannung unabhängig von der Taktfrequenz ist und die statische Verlustenergie linear mit der Zeit ansteigt bzw. die statische Verlustleistung konstant ist. Kann also die Spannung nicht im Betrieb variiert werden, ist die möglichst schnelle Abarbeitung eines Problems sinnvoll.

Da bei den untersuchten, sehr komplexen Prozessoren die Verluste für die Instruktion nicht sehr stark voneinander abweichen, kann schon die Angabe einer mittlere Verlustleistung ausreichend sein, um abhängig von der Anzahl der notwendigen Taktschritte eine nicht so genaue, aber dennoch brauchbare Aussage über die Verluste zu machen [56].

Nur unter den Bedingungen, dass der Inter-Instruction Effect signifikant und relativ genau zu quantisieren ist und dass die funktionale Datenabhängigkeit es zulässt, die Instruktionsreihenfolge zu ändern, ist durch gezielte Reduktion des Inter-Instruction Effects durch Instruktionsumordnung eine Verlustreduktion möglich [57]. Die Datenabhängigkeit der Verluste blieb in [57] allerdings unberücksichtigt und die funktionale

Datenabhängigkeit wurde nur auf einem sehr theoretischen Niveau betrachtet.

Zusammenfassend muss also gesagt werden, dass für die meisten Prozessoren auf Verlustleistung optimierte Softwareroutinen gleichzusetzen sind mit Routinen, welche eine geringe Anzahl von Taktzyklen und Speicherzugriffen nach sich ziehen. Numerisch einfache, also schnelle Routinen setzen die geringste Verlustenergie bei den allermeisten Prozessoren und Problemstellungen um (These 2e).

2.3.2. Der Microcontroller MSP430

Signalverarbeitung wird häufig nicht mit komplexen Prozessoren durchgeführt, sondern es werden Microcontroller eingesetzt, welche meist eine sehr einfache CPU besitzen. Es stellt sich daher die Frage: Sind die Ergebnisse aus Kap. 2.3.1 auf Microcontroller übertragbar? Am Beispiel des MSP430 [24] soll dies näher betrachtet werden.

Der MSP430 ist ein 16 Bit RISC-Microcontroller, der keine Pipeline besitzt. Aufgrund seiner sehr einfachen Architektur ist zu erwarten, dass der Einfluss der unterschiedlichen Basiskosten, der Datenabhängigkeit und des Inter-Instruction Effects groß ist. Spectre-Simulationen können dies bestätigen. Beispielsweise für `MOV Register,Register` schwanken die ermittelten Werte im allgemeinen zwischen rund 0,7 und 1,7 pJ. Bei praxisfernen Instruktionsfolgen konnten auch 0,4 pJ ermittelt werden.

Die bekannte Aufteilung in Teilprobleme (Basiskosten plus Summe der Kosten aller zusätzlichen Einflüsse) ist ungeeignet, wenn für die Teilprobleme nur statische Mittelwerte bestimmt werden. Dies gilt, da große Schwankungen der gesamten umgesetzten Energie durch große Schwankungen von Teilkosten bedingt sind. Genauer ausgedrückt addieren sich die Fehler $\triangle x_i$ der einzelnen Teileinflüsse x_i, wenn diese additiv in das Gesamtergebnis y einfließen:

$$\begin{aligned} y &= \sum_i x_i \\ \triangle y &\approx \sum_i \frac{\delta y}{\delta x_i} \triangle x_i = \sum_i \triangle x_i \end{aligned} \tag{23}$$

Eine Verbesserung der Abschätzung nach der Aufteilung in Teilprobleme ist also nur möglich, wenn Informationen hinzugefügt werden. Die detaillierte Behandlung des Inter-Instruction Effects, abhängig von der Folge der Instruktionen, stellt solch eine Vergrößerung der Informationsmenge dar. Sind beispielsweise die Energiemengen für alle Möglichkeiten von zwei aufeinander folgenden Instruktionen bekannt, so kann eine genauere Abschätzung gegeben werden, als sie bei Einsatz des gleichen Wertes für alle Instruktionsfolgen möglich ist.

Die Quantisierung der Einflüsse von Teilproblemen stellt allerdings im Allgemeinen ein Problem dar. Einerseits ist diese sehr aufwändig, was allein z. B. durch die Anzahl von möglichen Folgen aus zwei Instruktionen begründet wird. Andererseits müssen die Teileinflüsse auch zu quantisieren sein. Dazu sind Stimuli nötig, die die anderen Teileinflüsse minimieren bzw. mit denen es möglich ist, die anderen Einflüsse rechnerisch zu eliminieren. Die Lösung dieses Problems ist nicht trivial, da häufig mehrere Teileinflüsse zusammen zur Wirkung kommen. Schwanken die Kosten für ein Teilproblem zudem

stark, müsste dieses weiter in kleinere Teilprobleme zerlegt werden. Nicht immer ist dies möglich. Dann bleibt nur die Möglichkeit, einen „guten“ Mittelwert anzugeben.

Damit stellt sich die Frage: Was ist ein „guter“ Mittelwert? Unpräzise ausgedrückt ist es ein „typischer“ Wert, der auch in typischen realen Programmen auftreten wird. Selbst wenn man typische Programme charakterisieren könnte, bleibt das Problem, typische Bedingungen auch bei der Charakterisierung von Teilproblemen herzustellen. Da aber gerade dafür spezielle, oft sehr synthetische Stimuli nötig sind, ist dies nicht möglich.

Viel einfacher dagegen ist es, ein typisches Programm zu erstellen und die mittleren Gesamtverluste (mittlere Verlustleistung bzw. Verlustenergie pro Taktschritt) zu bestimmen. Zwar bleibt auch dabei ein großes Maß an Unklarheit bei der Frage, was typisch ist, aber der Einsatz realer Programme ist eine sinnvolle Ausgangsbasis.

Die erreichte Genauigkeit der in Kap. 2.3.1 diskutieren Methoden basiert also auf zwei Fakten:

1. Die Schwankungen der Verluste aufgrund der genannten Einflüsse bleiben bei komplexen Prozessoren gering. Abschätzungsmethoden, welche die Verluste pro Instruktion charakterisieren, können somit ohne aufwändige Teilproblemzerlegung brauchbare Abschätzungsergebnisse liefern.

2. Die Charakterisierung und die Überprüfung ist bei ähnlichen (typischen) Bedingungen erfolgt. Dies erleichtert nicht nur instruktionsbasierende Methoden, sondern es ist auch möglich, mit einem Gesamtmittelwert für die Verluste brauchbare Abschätzungen zu geben.

Für die CPU des MSP430 wurde die mittlere Gesamtverlustenergie pro Taktschritt durch Simulation mit Spectre bestimmt. Dazu ist die gesamte umgesetzte Energie für ein willkürlich gewähltes Programm ermittelt und durch die Anzahl der Taktschritte geteilt worden. Mit diesem Ergebnis konnte die Verlustenergie von einigen willkürlich ausgewählten Programmen abgeschätzt werden. Dabei blieb der Fehler unter 10% gegenüber Spectre-Simulationen. Für relative Vergleiche zwischen zwei Programmen ist also schon die Anzahl der notwendigen Taktschritte ausreichend.

Bei einer instruktionsbasierenden Abschätzung für die CPU des MSP430, bei der jeder Instruktion eine mittlere Verlustenergie zugeordnet wird, betrug der beobachtete Fehler bei Abschätzungen 25% gegenüber Spectre-Simulationen. Selbst wenn durch detaillierte Zerlegung in Teilprobleme eine deutlich höhere Genauigkeit erreicht werden würde, so erscheint dennoch der Aufwand dafür angesichts der relativ guten Ergebnisse bei der Abschätzung mittels Gesamtverlustenergie zu hoch.

Schlussendlich muss also festgestellt werden, dass die Anzahl der notwendigen Taktschritte und der Speicherzugriffe geeignete Metriken zur Optimierung von Softwareprogrammen sind. Dies gilt sowohl für komplexe Prozessoren, als auch für einfache. Eine Reduzierung der Verluste entspricht einer Optimierung auf Geschwindigkeit. Geeignete Mittel dafür sind softwarebasierende Methoden zur Programmanalyse und -bewertung (Profiling). Diesem Themengebiet widmet sich Kap. 3.

Es ist davon auszugehen, dass dieses Ergebnis auf CPUs von anderen Microcontrollern übertragbar ist. Besitzt eine CPU allerdings besonders aufwändige Baugruppen wie einen

internen Multiplizierer, wie es beim PIC [22] der Fall ist, so ist zu erwarten, dass Verlustabschätzungen wesentlich dadurch beeinflusst werden. Da eine Hardwaremultiplikation allerdings wesentlich effizienter als eine Softwaremultiplikation ist, bleibt Optimierung auf Geschwindigkeit weiterhin gleichbedeutend mit Verlustreduktion und Profiling ein geeignetes Mittel dazu.

3. Methoden zur Optimierung von Software

3.1. Einleitungsbeispiel: Software- und Hardwaremultiplikation

3.1.1. Motivation

Ist die Lösung eines Problems in Hardware sinnvoll? Welchen Einfluss haben verschiedene Algorithmen auf Softwarerealisierungen? Allgemein können diese Fragen nicht beantwortet werden. Zu unterschiedlich sind Problemstellungen und Möglichkeiten. Im Einzelfall muss und kann aber auch meist eine klare Aussage getroffen werden.

Der Vergleich der Verluste von Software- und Hardwarerealisierungen ist mit einer schnellen, datenabhängigen Abschätzungsmethode, wie `energy_estim` (Kap. 2.2), in vielen Fällen möglich. Grobe Vergleiche können auch gezogen werden, wenn die mittleren Verluste für Hardwarebaugruppen bekannt sind. Ist die Charakterisierung der Baugruppen aufwändig, bietet sich `energy_estim` als Alternative zu einem normalen oder auch vereinfachten Schaltkreissimulator an. Sind Probleme allein mit Software zu lösen, ist dagegen schon die Angabe von Anzahl der notwendigen Taktschritte ein geeignetes Maß für Vergleiche (siehe Kap. 2.3).

Die Multiplikation ist eine Operation, die in vielen Algorithmen benötigt wird. Daher soll am Beispiel der 16-Bit-Multiplikation für den Microcontroller MSP430 die eingangs erwähnte Problemstellung, der Vergleich zwischen Soft- und Hardwarerealisierung und zwischen verschiedenen Softwarealgorithmen, untersucht werden.

3.1.2. Das Beispiel 16-Bit-Multiplikation

Für die Ganzzahlmultiplikation stehen mehrere Möglichkeiten zur Realisierung bereit. Multiplikation in Hardware ist die schnellsten Möglichkeit, kostet aber auch zusätzliche Gatterfläche. Bei einer softwarebasierenden Realisierung wird zwar stets das gleiche Grundprinzip angewendet, welches aus der Schulmathematik bekannt ist, aber die Details der Realisierung lassen Spielraum für Optimierungen.

Das Grundprinzip der Multiplikation wird in Software durch schrittweise Akkumulation realisiert. Bei der vorzeichenlosen Multiplikation von Werten mit n Bit Wortbreite sind somit maximal n und bei der vorzeichenbehafteten Multiplikation $2n$ Akkumulationsschritte nötig (vgl. Abb. 37). Die Rechnung kann abgebrochen werden, wenn nur noch jeweils Null zum Ergebnis akkumuliert werden würde.

Wie schon bei der Hardwaremultiplikation ist auch in Software die Vorzeichenexpansion für vorzeichenbehaftete Multiplikation ein entscheidender Punkt, dessen Realisierung stark die Effizienz beeinflusst. Für die Softwarerealisierung auf Microcontrollern wird aufgrund des begrenzten Programmspeicherplatzes oft eine Lösung mit geringem Programmspeicheraufwand gesucht. Das Ausführen der Vorzeichenexpansion führt zu einer solchen Lösung, welche aber viele unnötige Rechenschritte nach sich zieht.

Zwei Beispiele für die Vorzeichenexpansion bei der vorzeichenbehafteten 4-Bit-Multiplikation sind in Abb. 37 illustriert. Während bei positiven Operanden (linkes Beispiel) nur Nullen ergänzt werden und damit ein Softwarealgorithmus frühzeitigtzung die schrittweise Akkumulation abbrechen kann, muss bei negativem zweiten Operand

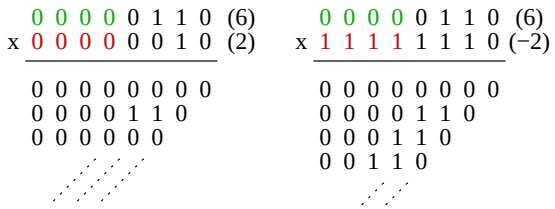

Abbildung 37: Beispiele für die vorzeichenbehaftete 4-Bit-Multiplikation

aufgrund der Vorzeichenexpansion die maximale Anzahl von Akkumulationsschritten ausgeführt werden, was äußerst ineffizient ist. Beim Microcontroller MSP430 wenden der Compiler von IAR [58] und der MSP430-GCC [59] dieses direkte Prinzip an.

Der Baugh-Wooley-Algorithmus (Anhang A) löst das Problem der Vorzeichenexpansion durch gezieltes Invertieren einiger Bits und Einfügen einiger Werte in das Array der partiellen Produkte. Damit werden stets nur genau n Akkumulationsschritte benötigt. Die Manipulation von einzelnen Bits ist in Hardware sehr einfach, in Software aber nur umständlich zu realisieren.

Eine viel einfachere Lösung des Problems der Vorzeichenexpansion soll hier vorgestellt werden: das Rechnen mit absoluten Werten (Beträgen) und die anschließende Korrektur des Vorzeichens des Ergebnisses. Maximal n Akkumulationsschritte sind nötig, je nach Daten kann aber auch früher abgebrochen werden. In Abb. 37 ist weiterhin zu erkennen, dass nur die Vorzeichenexpansion von Operand 2, dessen Bits entscheiden, ob der verschobene Wert von Operand 1 zum Ergebnis hinzu addiert wird oder nicht, der Grund für die Ineffizienz der direkten Vorzeichenexpansion ist. Die Vorzeichenexpansion von Operand 1 ist dagegen völlig unkritisch, da stets Additionen mit Wortbreite $2n$ ausgeführt werden müssen. Die effizienteste Lösung ist es somit, eine normale Vorzeichenexpansion von Operand 1 durchzuführen, aber den Betrag von Operand 2 zu bilden und beim Ergebnis, abhängig vom Vorzeichen von Operand 2, eine Vorzeichenkorrektur durchzuführen.

Für den MSP430 sollen die Realisierung des Compilers von IAR, des MSP430-GCC, der Baugh-Wooley-Algorithmus in Software (BW) und die vorgestellte Multiplikation mit absoluten Werten (ABS) im folgenden für die $16 \cdot 16 \rightarrow 32$-Bit-Multiplikation verglichen werden. Als Alternative wird die Realisierung eines Hardwaremultiplizierers (HW) entgegengestellt. Die Ergebnisse sind in Tab. 14 dargestellt. Es wurden 200 Multiplikationen mit Pseudo-Random Daten (PR16) und 8 Bit Pseudo-Random Daten (PR8), welche vorzeichenrichtig auf 16 Bit erweitert wurden, durchgeführt. Vorzeichenlose oder vorzeichenbehaftete Multiplikation ist zufällig anhand von Pseudo-Random Daten ausgewählt worden. Die angegebenen, mit `energy_estim` abgeschätzten Energiewerte, beziehen sich nur auf die Multiplikation. Die durch den Rest des Programms umgesetzte Energie wurde abgezogen.

Generell wurden für die vorzeichenbehaftete und die vorzeichenlose Multiplikation separate Funktionen verwendet. Wenn wie beim IAR- oder GCC-Algorithmus der einzige Unterschied zwischen beiden Varianten in der Vorzeichenexpansion besteht, könnte der

Tabelle 14: Vergleich der Realisierung der Multiplikation

Variante	ROM [Bytes]	PR16 Takte	PR16 W [nJ]	PR8 Takte	PR8 W [nJ]
HW	50 (44%)	4200 (7%)	4870 (11%)	4200 (9%)	4546 (12%)
IAR	114 (100%)	56133 (100%)	44694 (100%)	47023 (100%)	37448 (100%)
GCC	106 (93%)	49013 (87%)	38556 (86%)	40054 (85%)	31067 (83%)
BW	138 (121%)	49171 (88%)	39759 (89%)	45068 (96%)	35288 (94%)
ABS	98 (86%)	33097 (59%)	27215 (61%)	21273 (45%)	17422 (47%)

Kern der Routine aber auch für beide Varianten verwendet werden, was Programmspeicherplatz (ROM) einsparen würde. Alle Angaben beziehen sich aber auf die gewählte Realisierung mit separaten Funktionen.

Bei Variante BW wurde der Baugh-Wooley Algorithmus nur für die vorzeichenbehaftete Multiplikation eingesetzt. Für die vorzeichenlose Multiplikation wurde stattdessen der Algorithmus von IAR verwendet. Wie zu erwarten war, kostet beim Baugh-Wooley Algorithmus die Bitmanipulation viel Zeit, sodass der einfachere Algorithmus des MSP430-GCC überlegen ist. In der Realisierung des Compilers von IAR wird unnötigerweise geprüft, ob auch Operand 1 Null ist und damit abgebrochen werden kann. Das ist äußerst selten der Fall und daher ineffizient. Der MSP430-GCC Compiler führt diese Prüfung nicht durch. An allen anderen Stellen sind beide Realisierungen identisch. Deutlich überlegen gegenüber den anderen Softwarevarianten ist die vorgestellte Möglichkeit, mit absoluten Werten zu rechnen. Insbesondere bei negativen Operanden mit kleinem Betrag wird dies deutlich.

3.1.3. Zusammenfassung

Multiplikation in Hardware ist wie erwartet viel schneller als eine Softwarerealisierung, aber auch von der Energiebilanz zeigt sich deutlich, dass trotz zusätzlicher Hardware die Hardwaremultiplikation zu bevorzugen ist. Somit ist nicht nur jede Multiplikation in Hardware energieeffizienter als in Software, sondern die erhöhte Geschwindigkeit ermöglicht auch entweder höheren Durchsatz (ohne einen schnelleren und komplexeren Microcontroller zu verwenden) oder längere Ruheperioden (These 1).

Algorithmische Veränderungen können enormen Einfluss auf die Effizienz einer Realisierung haben. Bei der Multiplikation gibt es bei den diskutierten Varianten nur relativ geringe algorithmische Unterschiede, welche aber dennoch zu bemerkenswerten Unterschieden bezüglich der Effizienz führen. Ein großes Problem stellt allerdings die Entwicklung von Algorithmen dar. Eine generelle oder gar automatisierbare Lösung dieses Problems ist nicht in Sicht. Die strukturale Optimierung vorhandener Algorithmen, also die effiziente Umsetzung sowie die Anpassung an den Prozessor bietet nichtsdestotrotz weitreichende Möglichkeiten und ist ebenso wichtig. Methoden zur Realisierung dieser Aufgabe und zur automatisierten Unterstützung dabei sollen im Folgenden Kernthema sein.

3.2. Analyse der Hochsprache ANSI C

3.2.1. Einleitung

Traditionell wurde Software für Microcontroller in Assembler verfasst. Dies ermöglicht zwar volle Flexibilität und einfache Sprachkonstrukte beim Zugriff auf in den Speicherbereich eingeblendete Komponenten, aber der oft schwer zu erkennende Zusammenhang zwischen Sprachkonstrukten und Algorithmen erschwert nicht nur den Entwurf und die Fehlersuche, sondern auch das Erkennen von Möglichkeiten zur Optimierung.

Der Einsatz einer Hochsprache stellt eine wesentliche Erleichterung beim Entwurf von Software dar. Der höhere Überblick und die Abstraktion ermöglichen weiträumigere Optimierungen auf konzeptioneller, algorithmischer und strukturaler Ebene als bei der Verwendung von Assembler und die höhere Entwurfsgeschwindigkeit ermöglicht die Betrachtung von mehr Alternativen sowie die intensivere Suche nach Optimierungsmöglichkeiten. Werkzeuge zur Analyse von Programmen müssen demnach die Entwicklung auf Hochsprachenebene fördern (These 3a). Im Gegensatz zu den Vereinfachungen auf höheren Ebenen der Programmierung kann insbesondere auf Instruktionsebene durch den Einsatz einer Hochsprache aber auch schnell eine ineffiziente Lösung entstehen, da die Details der Umsetzung verborgen bleiben. Dies gilt es zu vermeiden.

Es ist auch mit einer Hochsprache möglich, hardwarenahe und effiziente Konstrukte zu beschreiben. ANSI C ist eine Hochsprache, welche dies aufgrund ihrer hohen Hardwarenähe auf vergleichsweise einfache Weise erlaubt. Daher hat sich ANSI C weiträumig beim Entwurf von Software für Microcontroller etabliert.

Doch welche Konstrukte gilt es zu vermeiden und welche zu nutzen? Es ist nicht das Ziel, mit einer Hochsprache einzelne Assemblerbefehle darzustellen, solange es nicht unbedingt nötig ist. Die errungene Abstraktion gegenüber Assemblercode gilt es zu wahren, ohne aber dabei unnötig Ressourcen zu verschenken. Es ist also eine Brücke zwischen Assembler und Hochsprache zu finden. Ein erster Schritt wird durch die Analyse der Konstrukte einer Hochsprache (Kontrollstrukturen und Funktionen) mit Bezug auf ihre Realisierung mit Maschinenbefehlen und die Analyse der Art der Speicherung von Variablen erreicht (These 3b). Dieser Schritt setzt allerdings einige grundlegende Kenntnisse auch in Assemblerprogrammierung voraus. Je weiter diese Analyse geht, desto mehr soll aber abstrahiert werden, sodass schlussendlich mit nur minimalen Kenntnissen über die Architektur eines Prozessors, einfache Optimierungen innerhalb der Hochsprache zur Verbesserung der Effizienz möglich werden. Der Analyse von ANSI C widmet sich dieses Kapitel. Die daraus abgeleiteten Prinzipien werden in den folgenden Kapiteln genutzt.

3.2.2. Betrachtete Microcontroller

Eine Analyse einer Hochsprache gänzlich losgelöst von einem Prozessor ist zwar möglich, kann aber aufgrund der mannigfaltigen Unterschiede zwischen verschiedenen Prozessoren nur auf recht abstraktem Niveau bleiben. Um das Verständnis zu erleichtern, sollen 3 Microcontroller bzw. deren CPUs als konkrete Beispiele betrachtet werden:

Microchip PIC [22]: Der PIC besitzt eine 8 Bit Havard-Architektur, also getrennten

Programm- und Datenspeicher. Es steht ein separater Stack für Subfunktionen und Interrupts bereit, der allerdings nur eine Tiefe von 31 besitzt. Er verfügt nur über ein einziges allgemeines Register und bis zu 4 Hilfsregister für diverse Adressierungsmodi. Je nach Version steht ein erweiterter Datenadressraum bis 12 Bit und ein Programmadressraum bis 21 Bit Adressbreite zur Verfügung. Die Programmwortbreite beträgt bis zu 16 Bit. Es handelt sich um eine RISC-Architektur, die je nach Version gewisse Erweiterungen aufweist. Für die Folgenden betrachteten Beispiele wurde der PIC18 verwendet.

Texas Instruments MSP430 [24]: Der MSP430 ist eine 16 Bit von-Neumann-Architektur, besitzt also einen Speicher für Programm, Daten und Stack. Es stehen 12 Register zur freien Verwendung und 4 Spezialregister (PC, SP, Statusregister, Konstantengeneratoren) bereit. Als Besonderheit sind Adressierungsmodi und Instruktionen orthogonal, d. h. beliebig miteinander kombinierbar. Es handelt sich um eine RISC-Architektur.

Zilog Z80 [23]: Der Z80 ist eine 8 Bit von-Neumann-Architektur. Er besitzt 12 mal 8 Bit (gruppierbar zu 6 mal 16 Bit) Register in zwei umschaltbaren Bänken, 2 mal 16 Bit Register und 4 Spezialregister (PC, SP, Interruptvektor, Memory Refresh). Als Besonderheit sind nicht für alle Register alle Operationen erlaubt. Der Z80 besitzt eine erweiterte 16 Bit Funktionalität, sodass der 16 Bit Adressraum direkt adressierbar ist und es werden spezielle 16 Bit Instruktionen bereitgestellt. Es handelt sich um eine CISC-Architektur.

Unabhängig vom tatsächlich eingesetzten Prozessor stellen die bereitgestellten Register die Grundlage für die meisten strukturalen Optimierungen dar. Einerseits werden durch das Zwischenspeichern von Operanden in Registern verlustintensive Speicherzugriffe gespart und andererseits sind Registerzugriffe auch einfacher, sodass Instruktionen mit Registerzugriffen auch weniger Takte benötigen können als Instruktionen mit Speicherzugriffen. Der PIC macht eine Ausnahme, da alle Befehle stets in zwei Schritten abgearbeitet werden und auch nur ein einziges Register zur Verfügung steht, sodass Speicherzugriffe unvermeidlich sind. Beim MSP430 dagegen hängt die notwendige Taktanzahl ganz wesentlich davon ab, ob Registerzugriffe genutzt werden können. Minimal ist nur ein Taktschritt für ein simples `MOV RegX,RegY` nötig, komplexere Befehle benötigen bis zu 6 Takte.

Sowohl PIC als auch Z80 müssen einen Adressierungsbereich nutzen, der durch die interne Wortbreite der Datenverarbeitung nicht oder nur zum Teil überdeckt werden kann. Adressarithmetik kann also schnell sehr aufwändig werden.

Je komplizierter eine Architektur ist, desto schwieriger ist es auch für einen Compiler, möglichst effizienten Code zu erzeugen. Zusätzlich stehen verschiedene Compilereinstellungen zur Optimierung zur Verfügung, von denen der resultierende Maschinencode abhängt. Werden daher im folgenden in Beispielen z. B. Laufzeiten angegeben, hängen diese natürlich vom Compiler und dessen Einstellung ab. Die Ergebnisse sind also als Richtwerte zu verstehen.

Moderne Compiler sind allerdings durchweg in der Lage, eine Beschreibung in einer Hochsprache in die dafür optimale Kette von Maschineninstruktionen zu übersetzen. Wenn dem Compiler Optimierungen erlaubt sind, wird nur selten durch den Compiler zusätzlicher unnötiger Overhead produziert. Somit bestimmt die Qualität der Beschreibung auf Hochsprachenebene auch direkt über die Qualität des Programms.

3.2.3. Begriffsdefinitionen

Die im Folgenden verwendeten Begriffe lehnen sich an [60] an.

Deklaration: Bekanntmachung der Eigenschaften von Namen; es findet keine Reservierung von Speicherplatz statt

Definition: Deklaration und Bereitstellung von Speicherplatz

Vereinbarung: Oberbegriff für Deklaration und Definition

externe / interne Variable: Variablen können außerhalb jeder Funktion oder auch innerhalb einer Funktion definiert sein.

Instruktion: direkt von einer CPU ausgeführte Anweisung („Assemblerbefehl“)

Operator: Zeichen, dass eine Abbildung repräsentiert (z. B. `+`, `*`, `<<`)

Befehl: Gruppe von einer bis mehreren Instruktionen, beschrieben durch Nutzung von einem bis mehreren Operatoren in einer Hochsprache (z. B. `c = a + *b;`).

Register: Speicherplatz für ein Datum innerhalb der CPU; Register sind i. A. nicht adressierbar, sondern werden direkt durch die Instruktion angesprochen. Der Begriff ist mehrdeutig und wird durch den Kontext unterschieden. Innerhalb einer Hardwarebeschreibungssprache und im Kontext von Microcontrollerkomponenten, welche in den Adressbereich einer CPU eingeblendet sind, bezeichnet der Begriff Register etwas anderes. Im Folgenden ist stets das „CPU-Register“ gemeint.

Registervariable: Variable, die auf Register abgebildet werden kann. Wohin die Variable tatsächlich abgebildet wird, ist damit nicht bestimmt (siehe Kap. 3.2.4.3).

3.2.4. Variablen

3.2.4.1. Elementare Datentypen ANSI C bietet nur eine geringe Anzahl an elementaren Datentypen. Deren Repräsentation (siehe Tab. 15) ist abhängig vom verwendeten Compiler. Das Compiler Reference Manual listet die tatsächlich verwendeten Größen auf und auch mittels `sizeof(type)` kann der Platzbedarf bestimmt werden. Für portablen Code kann es sinnvoll sein, mittels `typedef` eigene Typen bereitzustellen und diese angepasst an die Zielarchitektur zu definieren. In `stdint.h` werden solche Typen auch zur Verfügung gestellt. Aus den Datentypen und deren Repräsentation ergeben sich einige generalisierende Folgerungen:

Tabelle 15: Elementare und abgeleitete Datentypen und Repräsentation

Typ	typische Größe	MSP430; Compiler: [58], [59]
`char`	1 byte	1 byte
`short`	2 byte	2 byte (≡ `int`)
`int`	Wortbreite der Maschine	2 byte
`long`	4 byte	4 byte
`float`	32 bit	4 byte
`double`	64 bit	4 byte (≡ `float`)

- Arithmetische Operationen können nur auf Variablen des gleichen Typs angewendet werden. Nahezu jede Typkonvertierung kostet Rechenzeit. Sind Konvertierungen unausweichlich, so ist deren Anzahl zu minimieren.
- Der native Datentyp der Maschine kann am effektivsten verarbeitet werden. Bei klassischen Integer-basierenden Prozessoren ist dies typischerweise `int`. Häufig gibt es bei Compilern für 8-Bit-Prozessoren Ausnahmen, die `int` dennoch als 16 Bit Datentyp auffassen. Oft, aber nicht immer, können auch Subtypen des nativen Datentyps ohne Nachteile verarbeitet werden. So kann `char` auf Integer-Prozessoren ebenso einfach zu verarbeiten sein wie `int`. Der Instruktionssatz gibt ebenso wie eine Laufzeitmessung Auskunft über eine solche Funktionalität.
- Der Einsatz von Gleitpunktdatentypen sollte auf Prozessoren, welche nur Fixpunktarithmetik beherrschen, weitgehend vermieden werden, da Gleitpunktarithmetik dann sehr aufwändig in Software realisiert werden muss.
- Beim Speichern von Daten erscheint der Datentyp mit dem kleinsten Platzbedarf vorteilhaft. Dies führt aber zu zusätzlichen Konvertierungen, wenn nicht auch für die Berechnungen dieser Datentyp verwendet werden kann.

Konvertierungen kosten zwar meist Rechenzeit, doch auch das Vermeiden kann ineffizient sein, wenn dadurch aufwändigere Berechnungen nötig sind. Es muss daher im Einzelfall geprüft werden, was vorteilhaft ist.

Bei Konvertierungen von Integer-Datentypen (Beispiele in Tab. 16) von hoher zu niedriger Wortbreite werden die MSBs abgeschnitten. Dies kann durch explizites Löschen der MSBs oder aber auch sehr effizient beim Kopieren der Daten auf eine andere Variable nebenbei erledigt werden. Bei automatischer Konvertierung wird die Konvertierung hin zu größerer Wortbreite durchgeführt. Ist dies unerwünscht, muss explizit ein Type Cast angegeben werden.

Die Konvertierung in Richtung größerer Wortbreiten ist aufwändiger. Zusätzlich muss `unsigned` und `signed` unterschieden werden. Bei vorzeichenloser Konvertierung werden die neu hinzukommenden MSBs mit Null initialisiert, während bei vorzeichenbehafteter Konvertierung eine aufwändigere Vorzeichenexpansion durchgeführt werden muss. Wenn möglich, ist das Erzwingen der vorzeichenlosen Konvertierung mittels Type Cast oder Vereinbarung als `unsigned` vorteilhaft.

Tabelle 16: Integer-Konvertierungen - Beispiele (MSP430)

`unsigned long a;`
`unsigned int b;`
`unsigned char c;`	

`a = 0x01234567;`	`mov #0x4567, R10` `mov #0x0123, R11`
`b = (unsigned int)a;`	`mov R10, R8`
`c = (unsigned char)b;`	`mov.b R8, R9`
`if (a == b)` `if ((unsigned int)a == b)`	automatische Konvertierung nach `long` möglich, da `unsigned`
`b = (unsigned int)c;`	`mov.b R9, R8`
`a = (unsigned int)b;`	`mov R8, R10` `mov #0, R11`

3.2.4.2. Zeiger, Vektoren, Strukturen, Unionen Klassischen CPUs stehen Register und externer Speicher zum Verarbeiten von Daten zur Verfügung. Jeder Speicherzelle ist eine Adresse zugeordnet, über die auf den Inhalt der Speicherzelle zugegriffen werden kann. Register befinden sich direkt in der CPU und sind für eine Hochsprache wie C verborgen. Sie besitzen keine Adresse, sondern nur eine Nummer bzw. einen Namen, mit dem sie durch eine Instruktion direkt angesprochen werden. Der PIC macht eine Ausnahme: Das einzige CPU-Register WREG ist auch adressierbar.

Der Compiler nutzt automatisch die Register zum Zwischenspeichern von Werten und als Zeiger (Pointer) auf Daten im Speicher. Typischerweise kann die CPU Instruktionen, die auf Register zugreifen, schneller ausführen als Instruktionen, die auf den externen Speicher zugreifen. Somit werden Verluste durch die kürzere Ausführungszeit reduziert und es werden verlustintensive Speicherzugriffe vermieden. Wesentlich für gute Software ist es, die vorhandenen Register möglichst optimal zu nutzen. Moderne Compiler unterstützen dies, aber ineffiziente Software kann auch Register ungenutzt lassen oder unnötig blockieren.

Datenspeicher Kann eine Variable nicht auf Register abgebildet werden, muss sie im Datenspeicher (RAM) abgelegt werden. Der Datenspeicher kann sowohl mit als auch ohne Hilfe eines Datenstacks verwaltet werden. Ein Datenstack ist ein Stapelspeicher, dessen Füllstand (Adresse des obersten Elements) durch den Stackpointer (SP) angezeigt wird. Als Alternative kann eine Variable auch auf einer festen Adresse abgelegt werden. Dies geschieht auf jeden Fall, wenn sie extern oder `static` deklariert wird. Variablen, die `const` deklariert sind, liegen im Programmspeicher (ROM).

Datenspeicherung ohne Stack hat die Einschränkung, dass die Adressen aller Variablen bekannt, also statisch sein müssen. Ohne Einsatz eines Datenstacks verhält sich jede Deklaration einer Variablen wie eine `static` Deklaration. Es wird schon bei der Deklaration für jede Variable Speicher reserviert und die Adressen aller Variablen sind zur Compile-Zeit bekannt, sodass Berechnungen von Adressoffsets zur Laufzeit entfallen, was zu schnellerer Programmabarbeitung führen kann. Der Speicherbedarf ist aber gleich der Summe des Datenvolumens aller Variablen. Ein guter Compiler, der keinen Stack

nutzt, wird Subfunktionsverzweigungen erkennen und versuchen, Speicherplatz mehrfach zu verwenden. Sollen beispielsweise aus `main` die Funktionen `A` und `B` aufgerufen werden und rufen sich diese Funktionen niemals gegenseitig auf, so kann für beide Funktionen Speicherplatz an der selben Stelle reserviert werden. Durch die Limitierung auf statische Adressen sind solche Optimierungen nicht immer möglich.

Ein Compiler, der einen Datenstack zur Verfügung hat, kann interne, nicht `static` deklarierte Variablen zur Laufzeit auf den Stack erzeugen und verschwinden lassen, indem einfach der SP entsprechend dem Platzbedarf der Variable modifiziert wird. Der Wert des aktuellen SP plus ein dem Compiler bekannter Offset ergibt die Position (Adresse) einer Variablen. Die Modifikation des SP geschieht am Anfang und am Ende einer Funktion. Die Position von Variablen auf dem Stack ist damit nur innerhalb der die Variable erzeugenden Funktion bekannt. Beim Zugriff muss aus der relativen Position (SP plus bekannter Offset) die absolute Position berechnet werden. Dies geschieht zur Laufzeit und der Mechanismus wird automatisch durch den Compiler bereitgestellt.

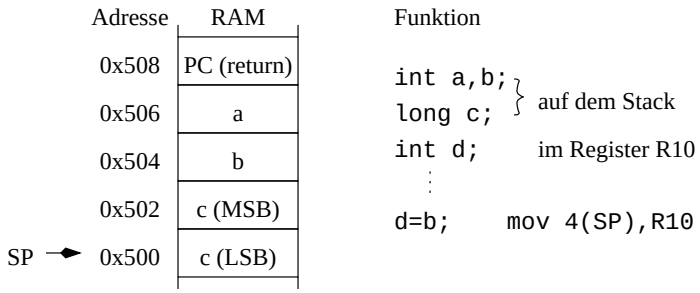

Abbildung 38: Stackverwaltung beim MSP430

Der MSP430 nutzt den RAM als Speicher für den Programmstack. Beim Eintritt in Funktionen oder bei Interrupts wird der alte Programmcounter (PC) als Rücksprungadresse für das Beenden von Funktion oder Interrupt auf dem Programmstack abgelegt. Weiterhin wird im RAM ein Datenstack realisiert. Die Mechanismen zur Verwaltung des Stacks werden von der Architektur des MSP430 bereitgestellt, sodass Programm- und Datenstack gemeinsam verwaltet werden. Das Stackprinzip für den MSP430 wird Abb. 38 illustriert. Als Besonderheit wächst der Stack in Richtung kleinerer Adressen und jedes Byte ist adressierbar, wobei aber dennoch 16 Bit Zugriffe auf dem RAM möglich sind.

Der PIC besitzt dagegen nur einen separaten Stack für den PC mit maximaler Tiefe 31. Ein Datenstack kann aber erzeugt werden, indem ein Datenspeicherplatz die Rolle des Stackpointers übernimmt (FSR-Register). Ein Compiler für den PIC kann optional also einen Datenstack bereitstellen und somit durch dynamische Speichernutzung den Gesamtbedarf an Datenspeicher drastisch reduzieren. Die Verwaltung eines solchen Datenstacks muss in Software realisiert werden und wird insbesondere bei PIC-Modellen mit erweitertem Datenadressraum (> 8 Bit) sehr aufwändig.

Im Folgenden wird davon ausgegangen, dass ein Datenstack verwendet wird, da heute meist die durch einen Stack gebotene Flexibilität gewünscht oder nötig ist und Speicherplatz knapp bleibt. Notwendig ist ein Datenstack für Rekursion. Die Verwaltung

eines Datenstacks ist aber immer mit einem Overhead an Instruktionen verbunden. Wie schwerwiegend der Overhead ist, hängt von der Architektur des Microcontrollers ab. Sämtliche Aussagen, die im Folgenden die Berechnung einer Position einer Variablen bezüglich SP betreffen, gelten somit selbstverständlich nur bei der Nutzung eines Datenstacks.

Zeiger Ein Zeiger (Pointer) ist eine Variable, deren Wert eine Adresse ist. Idealerweise ist die Wortbreite eines Zeigers die native Wortbreite der Maschine. Ein 16-Bit-Prozessor kann somit meist 2^{16} verschiedene Speicherplätze adressieren. Arithmetische Operationen mit Zeigern sind dann ebenso aufwändig wie Operationen mit dem nativen Datentyp der Maschine (meist `int`). Bei Prozessoren mit erweitertem Datenadressraum wie dem PIC ist Zeigerarithmetik dagegen aufwändiger.

Soll ein Zeiger `dp` auf ein Datum `d` zeigen (`int d, *dp=&d;`), so muss `d` zwangsläufig im Speicher abgelegt werden, da Register keine Adresse besitzen. Es wird also für `d` Speicher auf dem Stack reserviert und die Adresse (der Wert von `dp`) wird, wenn möglich, in einem Register oder anderenfalls auch auf dem Stack abgelegt. Es entsteht somit ein Overhead durch den zusätzlichen Zeiger und es wird verhindert, dass das Datum `d` direkt in einem Register abgelegt werden kann, sofern es sich um eine Registervariable handelt. Zeiger auf Registervariablen (wie im Beispiel `int`) sollten demzufolge nur eingesetzt werden, wenn dies unbedingt nötig ist, wie z. B. bei der Übergabe an Funktionen, wenn die Daten durch die Funktion modifiziert werden müssen. (Mehr dazu in Kapitel 3.2.7.)

Bei einer Havard-Architektur kann es vorkommen, dass die Adressräume für Daten- und Programmspeicher unterschiedlich groß sind. Ein Zeiger muss demnach nicht nur Informationen darüber enthalten, welcher Speicher ausgewählt werden soll, sondern muss im allgemeinen Fall auch groß genug sein, um beide Adressräume vollständig referenzieren zu können. (Zeiger in den Programmspeicher sind z. B. für die Initialisierung von Variablen, `const` deklarierten Variablen und für Zeiger auf Funktionen nötig.) Beim PIC mit 12 Bit Daten- und 21 Bit Programmadressraum muss somit ein allgemeiner Zeiger 3 Byte groß sein, was natürlich für den Zugriff auf den Datenspeicher einen riesigen Overhead mit sich führt. Ein Compiler sollte demnach auch kleinere Zeiger anbieten. Wenn der Compiler Zeiger mit sinnvoll angepasster Wortbreite nicht automatisch verwenden kann, muss dies manuell erzwungen werden. Da ANSI C aber keinerlei derartige Mechanismen definiert, müssen Compiler Erweiterungen anbieten, z. B. ein Schlüsselwort für die Vereinbarung, wie `int __keyword *a;`. Dieses Konstrukt wird „Typed Data Pointer“ genannt und ist nicht standardisiert. Das jeweilige Compiler Reference Manual gibt weitere Auskunft.

Einige Microcontroller besitzen einen erweiterten Speicherbereich, der aus mehreren Speicherbänken besteht. Häufig finden sich in solchen Architekturen dann Befehle, mit denen auf die erste Bank besonders effizient durch verkürzte Zeiger zugegriffen werden kann, während beim Zugriff auf die anderen Bänke die volle Adresswortbreite nötig ist. Auch hier kann der Einsatz von Typed Data Pointern dem Compiler die Optimierungen erleichtern.

Vektoren Ein Vektor ist ein Block aus mehreren gleichartigen Elementen. Ein Element kann ein komplexes Gebilde, aber auch einfach eine Variable mit einem elementaren Typ sein. Die Definition `int vec[10];` definiert einen Vektor mit 10 Elementen (0 bis 9), die alle vom Typ `int` sind. Mit `vec[n]` greift man auf das n-te Element des Vektors zu. Vektoren werden immer im Speicher abgelegt und somit besitzt jeder Vektor und auch jedes Element eine Adresse. Mit `int *vecp=vec;` wird die Adresse des Vektors bestimmt, die gleichzeitig die Adresse des ersten Elementes (`&vec[0]`) ist.

Tabelle 17: Zugriff auf das Vektorelement `vec[1]` (`vecp=&vec[0];`)

`vec[1]`	Die Adresse des Vektors liegt an einer dem Compiler bekannten Stelle relativ zum SP (statischer Offset). Zur Vektoradresse wird der Offset zum Element 1 (dynamischer Offset) hinzu addiert. Auf die so bestimmte Adresse kann nun zugegriffen werden. Zwei Additionen sind i. A. zur Bestimmung der Adresse eines Elementes .
`*(vecp+1)`	`vecp=&vec[0]` hält die Vektoradresse und es muss nur noch der Offset der Elementadresse hinzu addiert werden, bevor auf das Element zugegriffen werden kann. Im Gegenzug zur vereinfachten Berechnung mit nur einer Addition muss für `vecp` ein Speicherplatz (Register oder RAM) bereitgestellt werden.
`*(++vecp)`	Insbesondere wenn ein Vektor z. B. in einer Schleife schrittweise durchlaufen wird, bietet es sich an, den Zeiger vor oder nach jedem Zugriff zu modifizieren, um beim nächsten Zugriff eine Offsetberechnung einzusparen. Microcontroller unterstützen dies durch Post- oder Preinkrement/-dekrement-Operationen (zusammengefasst „Autoinkrement“).

```
int vec1[5]={1,2,3,4,5};
int vec2[5];
int i;

/*Takte (Schleife) = 66*/
for(i=0; i<5; i++) {
   vec2[i]=vec1[i];
}
```

```
int vec1[5]={1,2,3,4,5};
int vec2[5];
int *vecp1=vec1;
int *vecp2=vec2;
int i;

/*Takte (Schleife) = 76*/
for(i=0; i<5; i++) {
   *(vecp2+i)=*(vecp1+i);
}
```

```
int vec1[5]={1,2,3,4,5};
int vec2[5];
int *vecp1=vec1;
int *vecp2=vec2;
int i;

/*Takte (Schleife) = 56*/
for(i=0; i<5; i++) {
   *(vecp2++)=*(vecp1++);
}
```

Listing 4: Einige Möglichkeiten zum Kopieren eines Vektors (MSP430)

Um auf den Wert `vec[1]` zuzugreifen stehen mehrere Möglichkeiten zur Verfügung. Einige sind in Tab. 17 aufgeführt. Listing 4 zeigt als Beispiel mögliche Realisierungen der Kopie eines Vektors. Die Variante, die Postinkrement einsetzt, ist aufgrund des Instruktionssatzes des MSP430, der Autoinkrement unterstützt, am effizientesten. Siehe dazu auch Kap. 3.2.5.2. Die mittlere Variante ist der linken unterlegen, da in der linken Variante das feste Verhältnis zwischen Ziel- und Quelladresse vom Compiler gewinnbringend ausgenutzt werden konnte. Unter anderen Umständen kann der Zugriff über Zeiger günstiger sein.

```
struct mystructtype {
    float c;
    char b;
    int a;
};
void myfunction(struct mystructtype *mystructp);

int main(void) {
    struct mystructtype mystruct;
    struct mystructtype *mystructp = &mystruct;

    mystruct.a = 0xABCD;    /* Zugriff direkt */
    mystructp->a = 0x5678; /* Zugriff über Zeiger */
    /* Beide Zugriffe sind gleich effektiv bei MSP430. */

    myfunction(mystructp); /* beide Funktionasaufrufe gleichwertig */
    myfunction(&mystruct);
    /* Zeiger auf Struktur ist hier überflüssig, da Zugriff direkt möglich. */
    return 0;
}

void myfunction(struct mystructtype *mystructp) {
    mystructp->a = 0x1234;
    /* Zeiger wegen Übergabe an Funktion vorhanden. */
}
```

Listing 5: Strukturen und Zeiger auf Strukturen (MSP430)

Dem Compiler ist der Speicherbedarf für eine Variable aufgrund der Typbezogenheit eines Zeigers bekannt. Somit kann Zeigerarithmetik verborgen vor der Hochsprache unabhängig vom Typ bereitgestellt werden.

Strukturen Vom Standpunkt der Zeigerarithmetik sind Strukturen (`struct`) vergleichbar mit Vektoren. Der Unterschied besteht darin, dass in Strukturen statt gleichartiger Elemente Komponenten mit unterschiedlichem Typ zusammengefasst werden. Demzufolge kann auch auf jede Struktur bzw. auf jeden Komponente einer Struktur ein Pointer zeigen. Durch die Deklaration einer Struktur sind deren Aufbau und somit gleichzeitig die Adressoffsets der Komponenten bekannt.

Beim Zugriff auf die Komponenten einer Struktur besteht gegenüber dem Zugriff auf Elemente von Vektoren ein prinzipieller Unterschied bezüglich Zeigerarithmetik. Auf ein Vektorelement `vec[i]` wird zugegriffen, indem die Software den Offset zur Laufzeit bestimmt (Stackpointer plus statischer Offset des Vektors bezüglich Stackpointer plus dynamischer Offset `i`). Auf eine Strukturkomponente wird dagegen explizit mit `mystruct.a` zugegriffen. Der Adressoffset bezüglich des Stackpointers ist somit vollständig statisch und muss nicht erst zur Laufzeit ermittelt werden. (Wenn eine Komponente dynamisch ausgewählt werden soll, sind Verzweigungen nötig.)

Die Behandlung statischer Adressoffsets ist meist im Instruktionssatz eines Microcontrollers verankert (Indexed Addressing - Kap. 3.2.5.1), aber je nach Architektur kann dies unterschiedlich gut genutzt werden. Beim MSP430 sind im Gegensatz zum Z80 statische Offsets bezüglich des Stackpointers erlaubt. Somit gilt für MSP430, nicht aber für Z80: Der Zugriff mittels eines Zeigers auf eine Struktur (`mystructp=&mystruct;`) ist mit `mystructp->a` gleich effektiv wie der direkte Zugriff mit `mystruct.a`. Im ersten Fall wird der Offset einer Komponente bezüglich des Zeigers auf die Struktur und im zweiten bezüglich des Stackpointers berechnet. Listing 5 zeigt ein Beispiel mit speziellem Blick

auf den MSP430. Da statische Adressoffsets bezüglich SP beim Z80 nicht möglich sind und zusätzlich auf den SP nur schwer zugegriffen werden kann, sind Zeiger auf Strukturen beim Z80 dagegen meist sinnvoll, benötigen aber ein zusätzliches 16-Bit-Register. Auch beim PIC können Zugriffe über Zeiger effektiver sein, da Indexed Addressing nur für Move-Instruktionen zur Verfügung steht.

Unterschiedliche Komponenten von Strukturen haben unterschiedlichen Bedarf an Speicher. Manche Typen müssen auf einigen Architekturen auf bestimmte Adressen ausgerichtet sein. Dies kann zu unbenannten Speicherlöchern in Strukturen führen. Beim MSP430 muss beispielsweise `int` immer auf eine gerade Adresse ausgerichtet sein. Liegt vor einer Variablen vom Typ `int` eine Variable vom Typ `char` und ist diese auf eine gerade Adresse ausgerichtet, wird die ungerade Adresse, die der `char` Variablen folgt, ungenutzt bleiben.

```
struct badstructtype {
    char ac;
    int bi;
    char cc; };
/* Größe: 6 bytes */

struct goodstructtype_a {
    char ac;
    char cc;
    int bi; };
/* Größe: 4 bytes */

struct goodstructtype_b {
    int bi;
    char ac;
    char cc; };
/* Größe: 4 bytes */
```

Listing 6: Löcher in Strukturen (MSP430)

Es ist in der C-Sprachbeschreibung definiert, dass die Adressen von Strukturkomponenten in der Reihenfolge ihrer Vereinbarung größer werden. Listing 6 zeigt ein Beispiel für entsprechend angepasste Strukturdefinitionen. Löcher in Strukturen sind nur Verschwendung von RAM. Es treten keine Auswirkungen auf Verluste auf.

Unionen Muss eine Variable Daten von verschiedenen Typen aufnehmen, so sind Unionen (`union`) ein sinnvolles Mittel, um nicht n verschiedene Variablen für n Datentypen bereitzustellen. Die Definition einer Union reserviert so viel Speicher, dass eine Variable mit dem größten Datentyp, den eine Union aufnehmen kann, Platz findet. Sind alle Elemente einer Union Registervariablen (Kapitel 3.2.4.3), so kann die Union selbst auch auf Register abgebildet werden, wenn der Compiler dies ermöglicht.

Ein praktisches Beispiel für den Einsatz von Unionen illustriert Listing 7. Solch eine Union wird benötigt, um Vorzeichen, Exponent und Mantisse einer `float` Variablen zu extrahieren oder zu bearbeiten. Dies ist durch Bitsetzen, Bitlöschen und Verschieben möglich. Diese Operationen sind nur für Integervariablen, nicht aber für Floating Point Variablen definiert, sodass eine Union als Container zur Konvertierung nötig wird.

```
union floatlongtype {
    float f;    /* beide Elemente 32 Bit */
    long int l; /* ermöglicht Bitzugriff auf die float-Variable */
};
```

Listing 7: Beispiel für den Einsatz von Unionen

Komplizierte Gebilde Prinzipiell kann man Vektoren, Strukturen und Unionen miteinander kombinieren und damit beliebige komplexe Gebilde erzeugen. Während auf eine

Komponente innerhalb einer Substruktur innerhalb einer Struktur (`s.subs.a`) noch direkt und ohne zusätzlichen Aufwand zugegriffen werden kann, da bei Strukturen sämtliche Adressoffsets voll statisch sind, muss beim Zugriff auf eine Strukturkomponente innerhalb eines Vektors aus Strukturen (`svec[i].b`) schon eine aufwändige Berechnung der Adresse der referenzierten Struktur erfolgen (Größe der Struktur multipliziert mit `i`). Die dynamischen Offsets von Vektorelementen lassen bei Kombination aus Vektoren und Strukturen zu komplexen Gebilden den Aufwand beim Zugriff auf ein Datum schnell stark ansteigen.

Zeiger als Komponenten von Strukturen (z. B. bei rekursiven Strukturen / verketteten Listen aber auch als Zeiger auf Subkomponenten (`s->subs->a`)) machen es nötig, ihnen schrittweise durch einzelne Referenzierungen zu folgen. Hinzu kommt der Aufwand, die im RAM abgelegten Zeiger in ein Zeiger-Register zu laden, um eine Form von Indirect Addressing (Kapitel 3.2.5.2) zu realisieren. Daher sollten Zeiger als Komponenten von Strukturen nur verwendet werden, wenn dies funktional unbedingt nötig ist (rekursive Strukturen). Lokale Zeiger (Kap. 3.2.4.3) können dann den Zugriff beschleunigen (vgl. [61]).

3.2.4.3. Unterstützung der Nutzung von Registern Jede Variable, auf die ein Zeiger zeigt (deren Adresse einmal bestimmt wird), alle extern oder intern `static` deklarierten Variablen und Variablen mit komplexem Datentyp werden im Datenspeicher abgelegt. Beim Zugriff auf solche Daten sind somit verlustintensive Speicherzugriffe nötig. Dabei sind neben dem reinen Zugriff auf die Daten eventuell auch zusätzliche Zugriffe auf den Programmspeicher nötig, um Adressoffsets, die der Compiler bestimmt hat, nachzuladen.

Die Wertzuweisung einer Variablen mit elementarem Datentyp (auf die niemals mittels Zeiger zugegriffen wird / deren Adresse niemals bestimmt wird) oder die Wertzuweisung eines Zeigers selbst kann zur Nutzung von Registern führen, sofern die nötige Anzahl frei ist, die Variable oder der Zeiger nicht extern oder intern `static` deklariert sind und der Compiler dies zulässt. Solche Variablen, die in Register abgebildet werden können, sollen Registervariablen genannt werden. (Das hat nichts mit der Vereinbarung `register` in ANSI C zu tun.) Das jeweilige Compiler Reference Manual gibt Auskunft darüber, welche Typen auf Register abgebildet werden.

Wird ein Datum, was im Speicher abgelegt ist, aber eine Registervariable ist, in einer Routine wiederholt genutzt, so lohnt es sich, eine lokale Kopie auf ein Register anzulegen, dieses in der Routine zu nutzen und eventuell am Ende ein Ergebnis auf die ursprüngliche Variable im RAM zurückzuschreiben.

Listing 8 zeigt ein Beispiel für die Suche des kleinsten Wertes in einem Vektor und die Bestimmung von dessen Position. Der Vorteil der zweiten Variante ist es, nicht für jeden Vergleich das kleinste Element aus dem Speicher zu lesen, sondern es in der Variablen `smallest` lokal zu speichern.

Manchmal sind Compiler in der Lage, selbstständig das Prinzip der lokalen Kopie zu nutzen. Der MSP430-GCC kann beispielsweise eine solche anlegen, wenn auf ein Strukturelement häufig zugegriffen wird. Ist kein Register mehr frei, muss eine lokale Kopie vom Compiler ebenfalls im RAM abgelegt werden. Auch dies kann dennoch vorteilhaft

```
int vector_test(void) {
    int vec[8]={5,6,7,8,1,2,3,4};
    int res_1, res_2;
    res_1 = smallest_element_1(vec, 8);
    res_2 = smallest_element_2(vec, 8);
    return !(res_1 == res_2);
}

int smallest_element_1(int *vec, int elements) {
    int i;
    int pos=0;
    for(i=1; i<elements; i++) {
        if (vec[i] < vec[pos]) {
            pos = i;
        }
    }
    return pos;
} /*141 Takte mit dem Testvektor*/

int smallest_element_2(int *vec, int elements) {
    int i;
    int pos = 0;
    int smallest=vec[0];
    for(i=1; i<elements; i++) {
        if (vec[i] < smallest) {
            smallest = vec[i];
            pos = i;
        }
    }
    return pos;
} /*108 Takte mit dem Testvektor*/
```

Listing 8: Einsatz einer lokalen Kopie (Taktangaben für MSP430)

sein, wenn dadurch aufwändige Adressberechnungen entfallen. Beim PIC, der nur ein einziges Arbeitsregister besitzt, sodass stets lokale Kopien auf den RAM abgebildet werden müssen, ist dies häufig der Fall. Diese Art lokaler Kopie kann von Compilern meist nicht automatisch angewendet werden.

Jedes Schreiben (jede Zuweisung) führt zur Belegung einer Variablen. Das letzte Lesen vor einem erneuten Schreiben beendet die Belegung. Wird eine Variable in Register abgebildet, so werden entsprechend Register belegt oder freigegeben. Allein die Anzahl der deklarierten Registervariablen lässt somit keine Aussage über die tatsächliche Registernutzung zu. Unnötige Belegung von Variablen sollte daher vermieden werden.

Ist kein Register frei, so wird der Compiler auch eine Registervariable auf dem Stack ablegen. Jeder Compiler hat eigene Optimierungsstrategien, um zu erkennen, welche der Registervariablen auf dem Stack und welche in Register abgebildet werden. Idealerweise sollten häufig genutzte Registervariablen, wie die Zählvariable einer Schleife, bevorzugt in Registern abgelegt werden. Ein Hilfsmittel dafür kann die Vereinbarung einer Variable analog zu `register int i;` sein. Der Compiler kann dies aber ignorieren.

Ist eine Variable keine Registervariable (ist also eine lokale Kopie auf ein Register unmöglich), so kann ein lokaler Zeiger, der direkt auf die Variable zeigt, die Berechnung von Adressoffsets einsparen. Die Zeigervariable selbst ist eine Registervariable - ihr Wert zeigt nur auf ein Datum von beliebig komplexem Typ. Ein Anwendungsfall ist der (häufige) Zugriff auf ein einzelnes Element aus einem Vektor oder einem Teil eines komplexen Gebildes. Ein lokaler Zeiger auf eine Strukturkomponente ist nur in wenigen Fällen vorteilhaft (siehe dazu Kap. 3.2.5.1).

Die tatsächliche Registernutzung zu bestimmen, ist kompliziert. Klassische Möglich-

keiten sind Debugger und das Studium des disassemblierten Maschinencodes. Beides ist sehr aufwändig. Die in Kap. 3.4 vorgestellte Methode vereinfacht die Bestimmung stark, kann aber nicht die Belegung einer Variablen im Detail aufschlüsseln. Letztendlich bleibt aber der Laufzeitvergleich als hinreichendes Kriterium, um die Effizienz von lokaler Kopie und lokalem Zeiger am konkreten Beispiel zu untersuchen.

Müssen z. B. Zustände des Systems dauerhaft zwischengespeichert werden, so dürfen die dazu verwendeten Variablen nicht beim Verlassen der Funktion verschwinden. Externe oder intern `static` deklarierte Variablen können dafür verwendet werden. Insbesondere in Interrupt Service Routinen (ISR) kann dies sehr nützlich sein. Da solche Variablen dauerhaft im RAM (an festen Adressen) liegen, kann eine lokale Kopie vorteilhaft sein.

3.2.4.4. Bit-Felder Zum Abspeichern einer Statusinformation wird häufig nur ein einziges Bit benötigt. Sind mehrere Statusinformationen nötig, wäre es Verschwendung, für jede Information eine eigene Variable zu nutzen. Stattdessen können die einzelnen Bits von Wörtern genutzt werden. ANSI C bietet mit Bit-Feldern dafür einen abstrakten Mechanismus auf Hochsprachenebene. Alternativ ist auch die manuelle Bitmanipulation möglich.

```
struct {
   unsigned int flag_A : 1;
   unsigned int flag_B : 1;
   unsigned int flag_C : 1;
} myflags;
...
myflags.flag_A = 1;
if (myflags.flag_A == 1) { ... }
```

```
#define flag_A 1
#define flag_B 2
#define flag_C 4
...
int myflags;
...
myflags |= flag_A;
if ((myflags & flag_A) != 0) { ... }
```

Listing 9: Bit-Felder und manuelle Bitmanipulation

Listing 9 zeigt den Einsatz von Bit-Feldern im Vergleich zu manueller Bit-Manipulation. Die Repräsentation eines Bit-Feldes ist abhängig vom Compiler. Manche Compiler legen Bit-Felder prinzipiell im RAM ab. Ein guter Compiler würde im Beispiel aus Listing 9 erkennen, dass das Bit-Feld in ein Register passt und es darauf abbilden, wenn möglich. Bei manueller Bit-Manipulation existiert durch die Deklaration des Statusspeichers (`int myflags;`) die Möglichkeit dies besser zu beeinflussen, wenn das Verhalten des Compilers unbekannt ist oder Bit-Felder prinzipiell nicht auf Register abgebildet werden.

3.2.4.5. Initialisierung von Variablen und Wertzuweisung Wird eine interne Variable deklariert, die auf den Stack abgebildet wird, so wird nach dem Eintritt in die die Variable deklarierende Funktion entsprechend Stackspeicher durch Modifikation des SP reserviert. Ein Stackspeicherplatz steht einer Variablen exklusiv zur Verfügung, wird also nicht für andere Variablen wiederverwendet. Damit ist es egal, wann die erste Zuweisung eines Wertes erfolgt. Für Variablen, die auf Register abgebildet werden, gilt dies nicht, da Register wiederverwendet werden. Die Belegung von Registern erfolgt durch Zuweisungen, die Freigabe durch das letzte Lesen.

Kann eine Variable auf ein Register abgebildet werden, so kann je nach Compiler die Definition einer Variablen (also eine Initialisierung wie `int i=0;`) sofort zur Belegung eines Registers führen. Ein guter Compiler wird solch eine Zuweisung erst ausführen, wenn die Variable das erste Mal gelesen wird und eine unsinnige Zuweisung (ein neuer Wert wird später zugewiesen, ohne dass die Variable vorher gelesen wurde) ignorieren. Steht aber kein Compiler zur Verfügung, der solche Mechanismen bereitstellt, so sollten zu zeitige und unsinnige Initialisierungen unbedingt vermieden werden, da dadurch unnötigerweise Register belegt werden. Die Definition einer Variablen innerhalb eines Blocks (`{int i=1; ...}`) zu dem Zeitpunkt, wo sie wirklich benötigt wird, ist dann eine Möglichkeit, einen solchen Compiler zu unterstützen.

Wird eine Variable mit einem konstanten Wert initialisiert, so kann ein guter Compiler sie auch als Konstante ansehen, bis ihr ein neuer Wert zugewiesen wird. So führt die Befehlsfolge `int i=4; long j=1; j += 2*i;` dann zur direkten Zuweisung von `9` zu `j`, ohne dass eine arithmetische Operation ausgeführt wird. Unterstützt ein Compiler einen solchen Mechanismus nicht, sollten derartige Konstrukte weitestgehend vermieden werden, indem manuell der entsprechende Wert zugewiesen wird. Dies gilt im besonderen für Gleitkommavariablen, da solche Optimierungen für den Compiler schwieriger als bei Integervariablen sind [61].

```
long val_assign_1(long x, long y) {
   long a,b;
   a = x+y; /*in Register R6+R7*/
   b = x-y; /*in Register R8+R9*/
   if (b > 0) {
      return x;
   } else if (b < 0) {
      return y;
   } else {
      return a;
   }
} /*56 Takte*/
```

```
long val_assign_2(long x, long y) {
   long a,b;
   b = x-y;    /*in Register R8+R9*/
   if (b > 0) {
      return x;
   } else if (b < 0) {
      return y;
   } else {
      a = x+y; /*in Resultatregistern*/
      return a;
   }
} /*40 Takte*/
```

Listing 10: Zeitige und verzögerte Wertzuweisung für `a` (MSP430)

Nicht nur für die Initialisierung von Variablen, die auf Register abgebildet werden, sondern auch für deren Wertzuweisung ist es vorteilhaft, die Zuweisung möglichst spät auszuführen, da der Compiler nicht immer in der Lage ist, Zuweisungen zu verschieben. Unnötig zeitige Zuweisungen führen dann zu unnötiger Belegung von Variablen, wie in Listing 10 an einem Beispiel illustriert ist.

Soll ein konstanter Wert definiert werden, hilft das Attribut `const` bei der Variablendefinition dem Compiler dies zu erkennen. Allerdings wird dadurch die Variable lediglich als „nur lesbar“ (read-only) definiert. Es bleibt dem Compiler überlassen, die entsprechenden Optimierungen durchzuführen, also weder Register noch Stackspeicher zu reservieren. Als bessere Alternative können Konstanten durch `#define` Anweisungen durch den Präprozessor eingesetzt werden.

Die Deklaration eines Vektors oder einer Struktur lokal in einer Funktion reserviert zwar Speicher (der Stackpointer wird modifiziert), aber ansonsten werden keine weiteren Aktionen ausgeführt. Die Initialisierung nur eines Vektorelements oder einer Strukturkomponente führt aber zur kompletten Initialisierung des gesamten Gebildes. Alle In-

itialisierungswerte für das gesamte Gebilde werden im Programmspeicher (z. B. hinter allen Funktionen) abgelegt und bei der Initialisierung auf die Variable kopiert. Somit wird Programmspeicherplatz und Ausführungszeit für die Speicherkopie verschwendet. Eine Deklaration und nachfolgende Zuweisung der Werte nur für die Elemente / Komponenten des Gebildes, die tatsächlich initialisiert werden müssen, ist somit weit effektiver. (Mit `int i[10]; i[0]=1;` wird tatsächlich nur `i[0]` initialisiert, während mit `int i[10]={1};` der gesamte Vektor initialisiert wird. Den Elementen `i[1]` bis `i[9]` wird Null zugewiesen.)

Extern oder intern `static` deklarierte Variablen werden immer komplett initialisiert. Werden keine Initialisierungswerte angegeben, werden sie mit Null initialisiert. Dies schränkt die Effizienz solcher Variablen ein (vgl. dazu Kapitel 3.2.7.3). Da die Initialisierung dieser Variablen auf jeden Fall durchgeführt wird, sollte sie auch genutzt werden, wenn dadurch funktionale Vereinfachungen in Routinen erreicht werden.

3.2.5. Adressierungsmodi

Zwei Adressierungsmodi zum Zugriff auf Daten im RAM sind weit verbreitet: Indexed und Indirect Adressing. Weitere Modi können bereitstehen, sind aber meist Sonderfälle oder erweiterte Versionen eines dieser Modi. Als Beispiel sei Immediate Addressing (das Laden eines festen, beliebigen, konstanten Wertes) genannt, was eine erweiterte Sonderform von Indirect Adressing ist.

3.2.5.1. Indexed Addressing Eine Möglichkeit zur Adressierung von Daten im Speicher ist die Addition einer Adresse (abgelegt in einem Zeiger-Register `Rn` bzw. dem Stackpointer SP) mit einem Wert `X` und der anschließende Zugriff auf das Datum an der resultierenden Adresse `X+Rn`. Üblicherweise wird dabei das Zeiger-Register selbst nicht verändert. Ein solcher Zugriff (inklusive Adressberechnung) kann bei vielen CPUs innerhalb einer Instruktion effizient ausgeführt werden.

Mittels Indexed Adressing kann in ANSI C auf alle Variablen mit voll statischem Adressoffset direkt zugegriffen werden. Dies umfasst externe wie interne Variablen elementaren Typs und Strukturen sowie deren Elemente. Ein lokaler Zeiger auf Elemente von Strukturen kann demzufolge nur in wenigen Fällen vorteilhaft sein.

Beim MSP430 und beim Z80 muss der Wert `X` eine Konstante sein, die dem Compiler bekannt ist. Dies ist bei voll statischen Adressoffsets der Fall. Variabel kann der Wert `X` beim PIC sein. Beim Z80 ist Indirect Addressing bezüglich SP nicht erlaubt, sodass ein Datenstack nur umständlich verwaltet werden kann.

Da Vektoren durch den Vektorindex `i` einen teilweise dynamischen Offset haben, ist der Zugriff prinzipiell etwas komplizierter und kann nicht in nur einer Instruktion mit Indexed Addressing ablaufen. Die Adresse muss erst aus SP, statischem und dynamischem Teil des Offsets vor dem Zugriff gebildet werden. Daher kann es im Einzelfall effizienter sein, auf Vektoren mittels Zeiger zuzugreifen. (Wenn `p=&vec;`, kann im Einzelfall `*(p+i)` günstiger sein als `vec[i]`.)

3.2.5.2. Indirect Addressing Eine weitere Möglichkeit zum Zugriff auf eine Adresse besteht darin, die Zieladresse selbst in einem Zeiger-Register abzulegen und dann direkt über diesen Zeiger auf das gewünschte Datum zuzugreifen. Bei einem Zugriff in dieser Form entfallen Offsetberechnungen (sie wurden vorher einmalig durchgeführt, wenn nötig), was abhängig von der Architektur der CPU zu einer kürzeren Instruktionslänge gegenüber Indexed Adressing führen kann. Diese Form der Adressierung entspricht dem Zugriff über einen Zeiger (`*p`) auf ein Datum (`p=&d`).

Um Operationen in Datenfeldern zu beschleunigen, bieten einige Microcontroller eine Modifikation der Adresse beim Zugriff an. Der MSP430 bietet Postinkrement (vergleichbar mit `*(p++)`), der PIC Postinkrement, Postdekrement (vergleichbar mit `*(p--)`) und sogar Preinkrement (vergleichbar mit `*(++p)`). Beim Z80 stehen Postinkrement und Postdekrement nur bei einigen sehr komplexen Instruktionen zur Verfügung, die durch den Compiler meist nicht genutzt werden. Zusätzlich zu den automatischen Adressmodifikationen bieten der PIC und der Z80 spezielle Inkrement- und Dekrement-Instruktionen und der MSP430 sehr effiziente Operationen mit Hilfe von bereitgestellten Konstanten, um bei Adressoperationen eine Beschleunigung des Programmablaufes zu erreichen. Ein Beispiel für den Einsatz von Postinkrement befindet sich in Listing 4 auf S. 64 (Kopieren eines Vektors). Autoinkrement-Operatoren können es dem Compiler vereinfachen, derartige Funktionen zu nutzen. Bietet eine CPU keine solchen Funktionen an, so werden Autoinkrement-Operatoren durch konventionelle Additionen ersetzt. Nachteilig sind diese Operatoren daher nie, aber nicht immer können sie auch mit einem Vorteil gegenüber konventionellen Additionen umgesetzt werden.

3.2.6. Operatoren und Kontrollstrukturen

3.2.6.1. Operatoren Addition, Subtraktion, die logischen Operationen {`&`, `|`, `^`}, Vergleichs- und Äquivalenzoperatoren {`>`, `>=`, `<`, `<=`, `==`, `!=`} (kurz „Vergleiche“) sind üblicherweise nativ in einem Instruktionssatz vorhanden und können damit effizient verarbeitet werden. Vergleiche werden durch eine Compare-Instruktion realisiert. Sprünge können auf Pipeline-Architekturen zu Pipeline-Flushs führen.

Die Zuweisungsoperatoren {`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`} sowie Inkrement und Dekrement Operatoren (z. B. `i++`) werden nur zum Teil direkt durch den Instruktionssatz repräsentiert, erleichtern es aber dem Compiler, effizienten Code zu generieren.

Nicht alle Microcontroller besitzen einen dedizierten Multiplizierer. Auch wenn sie einen solchen besitzen, ist die Multiplikation nicht immer ebenso schnell wie andere Befehle. In [60] existieren viele Beispiele, bei denen ein Datum `var` über mehrere Rechenschritte modifiziert wird und dessen Vorzeichen am Ende der Routine beim Eintreten einer Bedingung invertiert werden soll. Die Lösung aus [60] sieht so aus, dass durch eine Folge von Befehlen in einer Hilfsvariable `j` der Wert 1 oder -1 erzeugt und schlussendlich das Vorzeichen des Datums mittels `var*=j;` entsprechend angepasst wird. Dies kann aber sehr ineffizient sein. Günstiger ist meist `if (Bedingung) {var = -var;}`. Oft vereinfacht dies auch das Formulieren der Bedingung zur Invertierung. So könnte z. B. jede beliebige negative Zahl anzeigen, dass invertiert werden soll.

Verschiebeoperationen (`<<`, `>>`) (Shifts) sind unbedingt Ganzzahldivisionen mit Divi-

soren 2^n ($n \in \mathbf{N}$) vorzuziehen, da die Division weitaus komplexer als das Shiften ist. Ein Compiler wird diese Divisionen aber nur durch Shiftoperationen ersetzen, wenn klar ist, dass der Dividend vorzeichenlos (`unsigned`) ist. Bei negativen ungeraden Dividenden entspricht das einfache Shiften nicht einer Division [62], da die normale Integerdivision nicht rundungsrichtig durchgeführt wird (`a=-3/4;` ergibt ebenso wie `a=3/4;` Null). Betragsbildung, Shiften und abschließende Vorzeichenkorrektur können bei vorzeichenbehaftetem Dividenden vorteilhaft sein. Die Multiplikation mit 2^n ($n \in \mathbf{N}$) sollte nur auf einen Multiplizierer abgebildet werden, wenn dieser schneller ist als eine Folge von Shifts. Der PIC bietet Multiplikation in einem Instruktionsschritt.

Es gilt für Modulo

$$\begin{aligned} a \,\%\, b &= a - \left[\frac{a}{b}\right] \cdot b \\ a \,\%\, 2^n &= a \texttt{ AND } (2^n - 1) \qquad (a \geq 0, n \in \mathbf{N}, n \neq 0) \end{aligned} \tag{24}$$

Dabei bezeichnet $[x/y]$ Integerdivision (nicht-vorzeichenrichtig). Wie bei der Division muss dem Compiler bekannt sein, dass der Dividend vorzeichenlos ist, um die Vereinfachung aus (24) zu nutzen.

Multiplikation, Division und insbesondere Modulo sind in Software sehr aufwändig und sollten auf algorithmischer Ebene vermieden werden. Wenn nötig, sollte alles auf die Multiplikation zurückgeführt werden, da diese numerisch am einfachsten ist. (Beispielsweise ist die Division durch eine Konstante gleich der Multiplikation mit der Inversen der Konstanten.) Ein Hardwaremultiplizierer sollte, wenn vorhanden, auch bis auf die oben genannte Ausnahme genutzt werden (vgl. dazu auch Kap. 3.1.2).

3.2.6.2. Minimale Instruktionssätze Minimale Instruktionssätze (RISC) sollen im Folgenden kurz beleuchtet werden. Diese können als Untermenge für komplexere Instruktionssätze dienen. Sonderinstruktionen, wie sie in CISC-Architekturen auftreten, und spezielle Varianten von Instruktionen können zusätzlich bereitstehen, sollen aber hier ausgeklammert bleiben. Kann ein Operator in C nicht durch eine einzelne Instruktion repräsentiert werden, sind Instruktionsfolgen nötig.

Üblicherweise enthalten Instruktionssätze Move, Addition / Subtraktion (optional mit Carry-In), Compare, AND, OR, XOR, Bit Test / Set / Clear, Shift und Rotate, bedingte und unbedingte Sprünge, Subfunktionsverzweigung (CALL / RET) und Return from Interrupt (RETI). Die Multiplikation dagegen ist nicht immer vorhanden. Der Z80 besitzt keinen Hardwaremultiplizierer, der Multiplizierer des MSP430 (wenn vorhanden) ist eine externe Komponente und es existiert damit keine direkte Instruktion und nur der PIC hat tatsächlich eine Multiplikationsinstruktion.

Viele Instruktionen modifizieren bei ihrer Ausführung Flags, welche Eigenschaften des Ergebnisses charakterisieren. Üblicherweise sind drei Flags vorhanden: Negative, Zero und Carry-Out. Ob bei bedingten Sprüngen der Sprung durchgeführt wird oder nicht, wird anhand von Flags entschieden. (Beispielsweise bei Jump on Zero (JZ) wird der Sprung ausgeführt, wenn das Zero-Flag gesetzt ist.) Compare ist eine Subtraktion, bei der kein Ergebnis auf eine Variable geschrieben wird, aber bei deren Ausführung Flags

gesetzt werden, um Sprünge zu kontrollieren. Die Instruktion Bit Test verhält sich analog und entspricht einem AND. Programmverzweigungen beruhen auf dem Prinzip des Setzens und Evaluierens von Flags. Flags werden üblicherweise nicht bei Move, CALL, RET, RETI, Sprüngen sowie Bit Set und Bit Clear gesetzt.

Das Setzen von Flags durch viele Instruktionen lässt sich direkt ausnutzen, wie z. B. mit `if (--i == 0)`. Die Variable `i` wird dekrementiert und danach wird das Zero-Flag, welches durch die Subtraktion modifiziert wurde, beim folgenden bedingten Sprung evaluiert. Bei `if (++i == 123)` muss dagegen nach dem Inkrementieren noch durch Compare mit `123` das Zero-Flag modifiziert werden, um eine geeignete Sprungbedingung zu erhalten. Es ist also lohnenswert, Prüfbedingungen so zu formulieren, dass Flags als Nebenprodukte normaler Instruktionen genutzt werden können, wenn dies funktional möglich ist. Dazu muss eine Programmroutine umstrukturiert werden (z. B. andere Zählvariable einer Schleife oder andere Zählrichtung). Sinnvoll sind solche Modifikationen, wenn sich dadurch nicht andere Teile verkomplizieren.

3.2.6.3. Effiziente Multiplikation Multipliziert man zwei Variablen mit n Bit Wortbreite, so entsteht kein Overflow, wenn das Ergebnis eine Wortbreite von $2n$ Bit besitzt. Hardwaremultiplizierer führen daher immer eine $n\,\text{Bit} \cdot n\,\text{Bit} \Rightarrow 2n\,\text{Bit}$ Multiplikation durch. Auch in C muss diesem Umstand Rechnung getragen und die Wortbreite dem Compiler angegeben werden. Es gelten im Folgenden die Deklarationen `int a,b; long res;`. Der Typ `int` sei 16 Bit und `long` sei 32 Bit breit.

Falsch ist `res = a * b;`, da hier 2 `int` Variablen miteinander multipliziert werden und für den C Compiler ein `int` Ergebnis entsteht. Erst durch die Zuweisung wird implizit die Konvertierung auf `long` ausgeführt, sodass sich dieses Konstrukt wie `res = (long)((int)(a * b));` verhält. Um ein korrektes Ergebnis zu erhalten, muss also mindestens ein Operand vom Typ `long` sein. Führt man aber `res = (long)a * (long)b;` aus, so entsteht eine $32 \cdot 32 \Rightarrow$ 64-Bit-Multiplikation, von deren Ergebnis die unteren 32 Bits `res` zugewiesen werden. Das Ergebnis ist korrekt, aber unnötigerweise entsteht der doppelte Aufwand. Um ein korrektes Ergebnis mit minimalem Aufwand zu erhalten, muss daher genau ein Operand vom Typ `long` sein: `res = (long)a * b;`.

Nicht alle Compiler sind allerdings in der Lage, damit eine $16 \cdot 16 \Rightarrow$ 32-Bit-Multiplikation zu erkennen und führen unnötigerweise eine $32 \cdot 32 \Rightarrow$ 64-Bit-Multiplikation durch, wie z. B. der Compiler von IAR [58]. Beim MSP430 gibt es aber eine Alternative, falls der Compiler ineffizienten Code produziert: Der Multiplizierer beim MSP430 ist als externe Komponente in den Adressbereich des Microcontrollers eingeblendet. Beide Operanden und das Ergebnis liegen an bekannten Adressen. Für Operand 1 stehen 4 Adressen zur Verfügung. Je nachdem, auf welche das Datum geschrieben wird, wird vorzeichenlos (`MPY`), vorzeichenbehaftet (`MPYS`), vorzeichenlos akkumuliert (`MAC`) oder vorzeichenbehaftet akkumuliert (`MACS`) gerechnet. Der Schreibzugriff auf Operand 2 (`OP2`) startet die Multiplikation. Das Ergebnis befindet sich in den 16 Bit Speicherplätzen `RESHI` und `RESLO`, auf welche indirekt über den symbolischen Bezeichner `RESULT` zugegriffen wird. Listing 11 zeigt ein Beispielmakro. Die Variablen `a`, `b` und `res` sind wie oben genannt definiert. Ist auf anderen Architekturen der Zugriff auf die Operanden- und Ergebnisre-

```
#define mul(type,op1,op2,res) { \
        type = op1; \
        OP2 = op2; \
        res = RESULT; }
...
mul(MPY,a,b,res);   /* unsigned */
mul(MPYS,a,b,res);  /* signed */
mul(MAC,a,b,res);   /* unsigned + accumulate */
mul(MACS,a,b,res);  /* signed + accumulate */
```

Listing 11: MSP430 - manuell ausgelöste Multiplikation

gister des Multiplizierers möglich, so ist das gezeigte Prinzip übertragbar.

Der MSP430 besitzt im Multiplizierer einen Akkumulator, der das aktuelle Produkt zum vorherigen dazu addieren kann. Viele Filteralgorithmen besitzen als Kern der Berechnungen dieses „Multiply & Accumulate“. Selten ist ein Compiler in der Lange, diese Funktionalität zu nutzen, sodass das manuelle Auslösen wie in Abb. 11 dann die einzige Möglichkeit zur Verwendung dieser effizienten Eigenschaft ist.

3.2.6.4. Kontrollstrukturen Das Vereinfachen logischer Ausdrücke wie $a\overline{b}+\overline{a}b = a \oplus b$ ist Grundvoraussetzung für effiziente Verwzeigungen in Entscheidungsbäumen. Während beim Hardwareentwurf Synthesewerkzeuge dies automatisch durchführen, ist dies bei Softwarecompilern im allgemeinen nicht der Fall [64].

If - else if In einer `if - else if` Kette müssen zum Erreichen von tiefer stehenden Alternativen alle vorherigen Tests durchgeführt werden. Daher sollten in der Folge von Tests häufig eintretende Bedingungen oder, falls die Wahrscheinlichkeiten nicht bekannt sind, funktional einfache Bedingungen zuerst angeordnet werden [63], wenn sich dadurch keine Bedingungen verkomplizieren.

Logische Verknüpfungen Die logischen Verknüpfungen `||` und `&&` werden strikt von links nach rechts bewertet. Dies kann ausgenutzt werden, wenn die Wahrscheinlichkeiten für das Eintreten von Bedingungen grob abgeschätzt werden können. Als Beispiel sei `if ((bed1 && bed2) || bed3) ...` gegeben. Tritt die Bedingung `bed3` sehr häufig ein (true) und ist `bed2` häufiger unwahr als `bed1`, so ist sinnvoller `if (bed3 || (bed2 && bed1)) ...` zu formulieren, da dadurch häufiger die Kette der Tests eher abgebrochen und zum `else`-Zweig gesprungen werden kann.

Bei Übertragen dieser Aussage auf arithmetische Operationen ergibt sich die Richtlinie, nur wirklich nötige Rechnungen auszuführen und Abkürzungen zu nutzen („Lazy Evaluation“ und „Short-Circuit Monotone Functions“ [63]).

Switch Eine Verzweigung mit `switch` kann häufig etwas effektiver realisiert werden als mit `if`, hat aber den Nachteil, dass sie auf mehrere einfache Alternativen eines Ausdruckes beschränkt ist.

Listing 12 vergleicht beide Kontrollstrukturen. Im Beispiel soll `c` ein Zeichen enthalten und je nachdem, um welches es sich handelt, soll ein Zähler für Ziffern, Zwischenräume

```
/* MSP430: 1084 clocks
   Z80:    7894 clocks
   PIC:    2032 clocks */
switch(c) {
case '0': case '1':
case '2': case '3':
case '4': case '5':
case '6': case '7':
case '8': case '9':
   ndigit[c-'0']++;
   break;
case ' ': case '\n':
case '\t':
   nwhite++;
   break;
default:
   nother++;
   /* break; */
}
```

```
/* MSP430: 1309 clocks
   Z80:    9460 clocks
   PIC:    2065 clocks */
if (c=='0' || c=='1'
    || c=='2' || c=='3'
    || c=='4' || c=='5'
    || c=='6' || c=='7'
    || c=='8' || c=='9')
   ndigit[c-'0']++;
else if (c==' '
         || c=='\n'
         || c=='\t')
   nwhite++;
else
   nother++;
```

```
/* MSP430:  857 clocks
   Z80:    7694 clocks
   PIC:    1883 clocks */
if (c>='0' && c<='9')
   ndigit[c-'0']++;
else if (c==' '
         || c=='\n'
         || c=='\t')
   nwhite++;
else
   nother++;
```

Listing 12: `if` und `switch`

oder andere Zeichen inkrementiert werden. Das Beispiel lehnt sich an eines aus [60] an. In Listing 12 sind Ausführungszeiten für ein komplettes Programm auf MSP430, Z80 und PIC angegeben, welches einen willkürlich gewählten String durchsucht und die oben genannten Häufigkeiten zählt. Zur besseren Übersichtlichkeit sind in Listing 12 allein die Kerne der Routinen dargestellt. Eine `if`-Verzweigung ist nur effizienter als eine `switch` Verzweigung, wenn funktionale Vereinfachungen möglich sind, wie hier im Beispiel `if (c>='0' && c<='9')`. Eine `break` Anweisung in dem dargestellten `default`-Zweig ist überflüssig, aber ein guter Compiler kann dies erkennen und setzt dafür keinen zusätzlichen Code ein. Ist unsicher, wie der Compiler sich verhält, sollte diese Anweisung entfernt werden.

Aufzählungskonstanten (enumeration constants: `enum`) zusammen mit der `switch`-Anweisung bieten eine effiziente Möglichkeit zur Beschreibung von State Machines in C. Ein Beispiel wird in [65] vorgestellt. Wird die State Variable intern `static` deklariert, bleibt ihr Wert erhalten, auch wenn die Funktion verlassen wird, was insbesondere bei Interrupt Service Routinen hilfreich sein kann.

Für `switch` gilt die gleiche Aussage bezüglich der Reihenfolge der `case`-Statements, wie für `if - else if`.

Bedinger Ausdruck Der bedingte Ausdruck `expr1 ? expr2 : expr3` wird häufig als funktional ähnliche Alternative für eine `if`-Verzweigung verwendet. Dabei ist aber zu beachten, dass es sich tatsächlich um einen Ausdruck handelt, also in einer Zwischenvariable entweder `expr2` oder `expr3` abgelegt wird. Listing 13 illustriert dies. Der bedingte Ausdruck ist also ähnlich zu Variante 2, der Beschreibung mit der Variable `temp`. Es hängt vom verwendeten Compiler ab, wie gut die gezeigten Alternativen umgesetzt werden können. Oft kann ein Compiler die rechte Variante am einfachsten optimieren.

Der Vorteil des bedingten Ausdruckes liegt in seinem Einsatz innerhalb komplexer größerer Ausdrücke, wo auch genau das Erzeugen einer Zwischenvariable erwünscht ist. Dies kann beispielsweise bei einem Funktionsaufruf günstig sein, wo ein Parameter mittels bedingtem Ausdruck ausgewählt wird.

```
result += (a > b) ? a : b;
/* add max(a,b) to result*/
```

```
if (a > b)
   temp = a;
else
   temp = b;
result += temp;
```

```
if (a > b)
   result += a;
else
   result += b;
```

Listing 13: `if` und bedingter Ausdruck „`?:`“

Schleifen Die Schleifen `for` und `while` sind äquivalent. Die Prüfbedingung der Schleife kann im Assemblercode vor oder nach das Ausführen des Schleifenkörpers vom Compiler gesetzt werden. Liegt die Prüfbedingung vor dem Schleifenkörper, wird stets nach dem Schleifenkörper zur Prüfbedingung zurückgesprungen, was nach dem letzten Durchlauf theoretisch überflüssig wäre. Liegt die Prüfbedingung am Ende, muss vor Eintritt in die Schleife zuerst zur Prüfbedingung gesprungen werden, da beide Schleifen abblockend sind. Eine `do-while` Schleife prüft stets am Schleifenende und springt nur zum Einsprungpunkt des Schleifenkörpers zurück, wenn die Abbruchbedingung noch nicht erreicht wurde, da sie nicht abblockend ist. Falls funktional möglich, ist also `do-while` zu bevorzugen.

Schleifen besitzen prinzipiell zusätzlichen Aufwand durch die Schleifenbehandlung. Dieser wird durch Loop Unrolling zu Lasten des Programmspeicherplatzes eliminiert. Partial Loop Unrolling (Listing 14), also die teilweise Elimination der Schleife, stellt einen Kompromiss zwischen Aufwand für Schleifenbehandlung und Programmspeicherplatz dar.

```
for (i=0; i<16; i++) {
   result += func(i);
}
```

```
for (i=0; i<16; i+=4) {
   result += func(i);
   result += func(i+1);
   result += func(i+2);
   result += func(i+3);
}
```

Listing 14: Partial Loop Unrolling (synthetisches Beispiel)

Abbrüche und Sprünge In Listing 12 wurde schon eine Einsatzmöglichkeit von `break` gezeigt. Mit `break` lassen sich nicht nur `switch` Verzweigungen, sondern auch Schleifen verlassen. Weiterhin existieren `continue` für die unmittelbare Wiederholung einer Schleife, `goto` zum direkten Sprung und `return` zum gleichzeitigen Verlassen einer Funktion. Tief verschachtelte Schleifen und Verzweigungen kann man sehr einfach mit `goto` verlassen (ohne gleich wie bei `return` die Funktion zu verlassen), sodass sehr viel Prüflogik für Abbruchbedingungen eingespart werden kann. All diese Mechanismen gelten als unsauberer Programmierstil, gleich dem `GOTO` in BASIC, da Sprünge im Programm nur schwer nachzuvollziehen und häufig Quellen für Programmfehler sind. Das Überspringen zahlreicher funktional ungewollter Prüfungen mit den genannten Mechanismen, die meist durch einfache Sprünge (ohne Bedingung) realisiert werden, ist aber äußerst effektiv.

```
#define swap(type,vx,vy) {type temp=vx; vx=vy; vy=temp;}

void swap_test(int vec[], int i, int j) { /* Aufruf mit i=0, j=1 */
    swap_f(vec,i,j);                      /*  5 Takte /  4 Bytes plus swap_f */
    swap_f(vec,0,1);                      /*  8 Takte / 10 Bytes plus swap_f */
    swap(int,vec[i],vec[j]);              /* 17 Takte / 22 Bytes */
    swap(int,vec[0],vec[1]);              /* 12 Takte / 12 Bytes */
}

void swap_f(int v[],int nx, int ny) {     /* 21 Takte / 26 Bytes */
    int temp=v[nx];
    v[nx]=v[ny];
    v[ny]=temp;
}
```

Listing 15: Makro und Funktion (MSP430)

3.2.7. Funktionen

3.2.7.1. Vorbetrachtungen Eine Funktion ist eine eingekapselte, wiederverwendbare Subroutine. Funktionen ermöglichen eine übersichtlichere Programmierung (es genügt zu wissen, was getan wird und nicht wie) und sparen Programmspeicherplatz, sofern die Subroutine mehrfach und evtl. mit unterschiedlichen Parametern aufgerufen wird.

Jeder Funktionsaufruf kostet allerdings Rechenzeit. Mindestens ein `CALL` (call subroutine) und `RET` (return from subroutine) müssen ausgeführt werden. Werden bei der Parameterübergabe und dem Funktionsresultat Werte oder Zeiger übergeben, so sind dafür ebenfalls Kopieraktionen erforderlich. Für intern deklarierte Variablen muss zudem Speicherplatz geschaffen und evtl. freigeräumt werden (Stackspeicher oder Register).

Mit Bedacht eingesetzt, können Makros eine Alternative zu Funktionen sein. Makros sind schlichte Textersetzungen durch den Präprozessor. Somit wird jedes Mal, wenn ein Makro aufgerufen wird, der entsprechende Ersatztext in den Code eingefügt und damit auch jedes Mal Programmspeicherplatz belegt. Dafür entfällt der Overhead des Funktionsaufrufs. Das Vorgehen ist vergleichbar mit Loop Unrolling.

Listing 15 stellt Makro und Funktion gegenüber. Laufzeiten und Codegrößen sind für MSP430 angegeben und verhalten sich bei PIC und Z80 ähnlich. Als Aufgabe im Beispiel sollen zwei Elemente vom Typ `int` eines Vektors vertauscht werden. Der Funktion wird die Adresse des Vektors und die Nummern der zu vertauschenden Elemente übergeben, während das Makro direkt die Elemente vertauscht. Das Makro muss mit der Angabe des Typs der Elemente aufgerufen werden und ist damit unabhängig vom Typ. Signifikante Vorteile ergeben sich beim Aufruf des Makros mit festen Parametern, da dann Offsetberechnungen entfallen.

Makros setzen aufgrund des geringeren Overheads weniger Energie als Funktionen um. Insbesondere bei kleinen Subroutinen und ausreichend Programmspeicherplatz sind sie Funktionen vorzuziehen. Als Alternative zu Makros bieten einige Compiler Function Inlining an. Dabei wird der Code der Funktion wie beim Makro direkt an die Stelle des Aufrufs gesetzt und der Overhead des Funktionsaufrufs entfällt ebenfalls.

Zusatz: Es existiert eine weitere Möglichkeit zum Austausch von Integer-Variablen `a` und `b` [62]. Die Befehlsfolge `a ^= b; b ^= a; a ^= b;` ermöglicht den Austausch ohne temporäre Variable. Mit dieser Variablen können aber durch Registernutzung Speicherzugriffe eingespart werden, wie schon in [62] erwähnt. Beim Z80, wo bei Move-Be-

fehlen mehr Adressierungsmodi bereitstehen als bei XOR-Befehlen, kann dies ebenfalls ungünstig sein. Es ist im Einzelfall zu prüfen, welche Variante effizienter ist.

3.2.7.2. Parameterübergabe und Funktionsresultat Eine Funktionsdeklaration (der Funktionsprototyp) hat die Form

```
Resultattyp Funktionsname( evtl. Parameterdeklarationen );
```

Parameter und Resultat werden abhängig von der Architektur des Prozessors und vom Compiler übergeben. Dazu werden die Variablen auf einen Speicherplatz kopiert, worauf die Funktion dann zugreift. Da Funktionen daher stets mit Kopien von Daten arbeiten, können sie auch keine Daten direkt modifizieren. Dazu sind Zeiger notwendig.

Bei der Parameterübergabe werden Registervariablen auf Register kopiert, wenn geeignete Register frei sind. (Der Compiler definiert in der „Calling Convention“, welche Register zur Parameterübergabe geeignet sind.) Sind keine geeigneten Register frei oder müssen nicht-Registervariablen übergeben werden, so werden sie in einer durch die Calling Convention festgelegten Reihenfolge auf den Stack kopiert. Je mehr Parameter und je komplexer der Datentyp der Parameter ist, desto umfangreicher sind die notwendigen Kopieraktionen. Das Funktionsresultat wird analog zu den Parametern übergeben.

Als Beispiel soll folgender Prototyp beim Einsatz des MSP430-GCC-Compilers [59] erläutert werden:

```
int myfunction(int a, int b, long c, int d);
```

Die Werte werden für `a` auf Register `R15`, für `b` auf `R14`, für `c` auf `R12`+`R13` und für `d` auf den Stack zur Übergabe kopiert. Das Resultat wird über `R15` übergeben. Der MSP430-GCC reserviert also 4 Register (`R12` bis `R15`) für die Parameterübergabe. Für Variable `d` sind die reservierten Register erschöpft.

Kopieraktionen kosten Rechenzeit. Die Kosten sind besonders hoch, wenn die Register zur Parameterübergabe erschöpft oder nicht geeignet sind, sodass Parameter über den Stack übergeben werden müssen. Daher sollte die Anzahl der Parameter klein bleiben und Parameter wie Funktionsresultat Registervariablen sein. Das klassische Negativbeispiel ist die Wertübergabe einer Struktur, da dann die gesamte Struktur kopiert wird. Es ist wesentlich sinnvoller, nur einen Zeiger mit der Adresse der Struktur zu übergeben.

Oft wird daher generalisierend die Wertübergabe als schlecht und die Übergabe von Zeigern als erstrebenswert angesehen. Rein funktional betrachtet hat der Einsatz von Zeigern nahezu nur Vorteile, wie z. B. die effiziente Datenübergabe auch großer Datenblöcke und die Möglichkeit zur Modifikation der Daten auch innerhalb einer aufgerufenen Funktion. Der Einsatz von Zeigern verhindert aber die Nutzung von Registern, da eine Variable, von der eine Adresse bestimmt wird, nicht in Registern liegen kann.

Eine Alternative zur Parameterübergabe ist die Nutzung extern deklarierter Variablen. Dies gilt aber als unsauberer Programmierstil, da schwer nachzuvollziehen ist, welche Funktionen über solche Variablen Daten austauschen. Auch liegen extern deklarierte Variablen stets im RAM, sodass Register ungenutzt bleiben. Insbesondere der Zugriff auf große Datenblöcke, auf die von unterschiedlichen Funktionen zugegriffen werden muss

(Filteralgorithmen und Signalkonditionierung in Messsystemen), die also prinzipiell im RAM liegen, kann aber effizient mit extern deklarierten Variablen realisiert werden. Auch ist kein Zeiger nötig, um extern deklarierte Variablen zu referenzieren.

3.2.7.3. Klassifikation von Funktionen Funktionen können anhand der übergebenen Parameter und des Resultates klassifiziert werden. Wird an eine Funktion ein Wert übergeben, so wird diese Übergabe im Folgenden als Input bezeichnet, da keine Modifikation des ursprünglichen Wertes möglich ist. Die Übergabe eines Zeigers ermöglicht dagegen die Modifikation der ursprünglichen Daten und soll daher als Inout bezeichnet werden. Das Funktionsresultat kann stets als Output angesehen werden.

Drei verschiedene Möglichkeiten für Resultate von Funktionen (Output) gibt es: kein Resultat (`void`), ein Wert von einem bestimmten Typ (`type`) und ein Zeiger auf einen bestimmten Typ (`type *`). Analog lassen sich die Möglichkeiten für die Parameterübergabe auflisten: kein Parameter (`void`), 1..n Inputs (`type`), 1..m Inouts (`type *`) und die Kombination aus 1..n Inputs und 1..m Inouts. Anhand der Klassifikation ergeben sich typische Anwendungsszenarien für bestimmte Funktionsarten sowie prinzipielle Vor- und Nachteile. Im Folgenden stehen `<parameter>` und `<output>` für beliebige Werte einschließlich `void`.

`void function(void);` Eine autonome Funktion ohne direkten Input und Output ist sinnvoll für feste Routinen wie z. B. Konfiguration und Start einer Microcontrollerkomponente, beim Arbeiten mit extern deklarierten Variablen (z. B. Filteralgorithmus über ein Vektor aus Messwerten) oder für Interrupt Service Routinen (ISR).

`type function(<parameter>);` Gibt eine Funktion Daten aus, ist die Nutzung des Funktionsresultates dafür die nahe liegendste Möglichkeit. Sehr effizient ist dies für die Ausgabe von Registervariablen während anderenfalls aufwändiges Kopieren über den Stack notwendig wird.

`type *function(<parameter>);` Muss eine Funktion Daten von beim Funktionsaufruf unbekannter Menge übergeben, ist die Reservierung der benötigten Menge an Speicher im Heap mittels `malloc` innerhalb der Funktion und die Übergabe des Zeigers auf den Datenblock eine geeignete Lösung. Dabei muss natürlich zusätzlich eine Information übergeben werden, wie groß der Datenblock tatsächlich ist. Diese Information kann u. a. im Datenblock selbst (z. B. als erstes Element) enthalten sein. Weitere Betrachtungen zur Nutzung von Heap folgen in Kap. 3.2.8.

Soll das Ergebnis einer Funktion z. B. ein Zeiger auf ein gesuchtes Element sein, so kann die Nutzung des Funktionsresultates günstiger als die von Inout sein, da dadurch die Parameteranzahl reduziert wird. Nur eine begrenzte Anzahl von Parametern kann mittels Register übergeben werden.

`<output> function(type, type, ...);` Die Wertübergabe von Registervariablen als Input ist sehr effektiv, wenn deren Anzahl bzw. deren gesamte Größe den Platz in den zur Verfügung stehenden Registern nicht übersteigt. Anderenfalls werden

Daten umständlich über den Stack kopiert und es kann sinnvoller sein, die Daten in einem Vektor oder einer Struktur zusammenzufassen und nur einen Zeiger darauf zu übergeben. Die Nutzung von Input erleichtert generell die Nutzung von Registern außerhalb und innerhalb der Funktion sowie bei deren Aufruf.

`<output> function(type *, type *, ...);` Mit der Übergabe von Zeigern wird die Realisierung von Inout ermöglicht. Die Komplexität der Übergabe eines Zeigers ist meist gleich der einer Variable mit dem nativen Datentyp der Maschine (üblicherweise `int`). Zeiger ermöglichen die Übergabe auch riesiger Datenmengen in sehr effizienter Weise. Eine sehr große Anzahl von zu übergebenden Zeigern kann sinnvoll in einem Vektor oder einer Struktur zusammengefasst werden. In allen Fällen sind innerhalb der Funktion lokale Kopien und lokale Zeiger zum Zugriff auf die Daten oft sinnvoll.

Erzeugt die Funktion eine große Menge an Daten und ist deren Menge bekannt, kann die Reservierung von ausreichend Speicherplatz vor dem Funktionsaufruf durch Deklaration einer entsprechenden Variablen, welche dann auf den Stack abgebildet wird, und die Übergabe eines Zeigers darauf sinnvoll sein. Damit wird die Nutzung von Heap vermieden.

Für Registervariablen ist häufig die Art der Übergabe am effizientesten, welche den Charakter der Übergabe am besten repräsentiert. Die Nutzung von Inout sollte vermieden werden, wenn die Übergabe mittels Input bzw. Output geschehen kann. Für die Übergabe großer Mengen an Daten und Variablen, die auf den Stack abgebildet werden, ist die Nutzung von Zeigern (also Inout) dagegen vorteilhaft. Zeiger selbst sind Registervariablen. Lokale Kopien und lokale Zeiger sorgen bei Nutzung von Zeigern als Parameter für einen effizienten Zugriff, müssen aber meist explizit beschrieben werden, während Wertübergabe häufig automatisch eine Registernutzung nach sich zieht.

Generell stellt die Datenübergabe bei Funktionen einen Flaschenhals dar. Völlig ohne direkte Übergabe von Daten ermöglichen extern deklarierte Variablen den effektiven Zugriff auf Datenfelder. Solche Variablen liegen allerdings permanent im RAM. Vollständig zu eliminieren ist der Funktionsaufruf durch Makros oder Function Inlining. Mit jedem Aufruf wird dann aber auch Programmspeicherplatz dafür benötigt. Insbesondere bei kleinen oder selten ausgeführten Routinen sind diese Mechanismen sinnvoll. Partielle Elimination gelingt, wenn mehrere Funktionen verschmolzen werden können.

Wesentlich zur Auswahl einer effizienten Lösung ist die detaillierte Analyse des Funktionsaufrufs. Die reine Ausführungszeit eignet sich schon gut als Metrik, aber vorteilhaft ist es, auch den tatsächlichen Speicherort der Parameter zu ermitteln. In Kap. 3.4 wird darauf näher eingegangen.

3.2.8. Speicherverwaltung

Werden Daten erzeugt, so müssen diese an geeigneter Stelle abgespeichert werden. Ist bekannt, welchen Umfang die Daten haben, so stehen intern und extern deklarierte Variablen zur Verfügung. Externe und intern static deklarierte Variablen werden an

zur Compile-Zeit bekannten festen Positionen unterhalb des Stacks und alle anderen Variablen auf dem Stack abgelegt (sofern sie nicht in Register abgebildet werden können).

Ist der Umfang der abzuspeichernden Daten aber variabel, so kann es verschwenderisch sein, Variablen zu deklarieren, die groß genug sind, in jedem Fall die Daten aufzunehmen. Manchmal existiert eine theoretische Maximalgröße auch nicht, sodass theoretisch unbegrenzte Datenmengen auftreten können (nur technisch limitiert durch die Größe des RAM). Hier bietet sich die dynamische Allozierung von Speicher (`malloc`), also die Nutzung des Heaps an (Abb. 39).

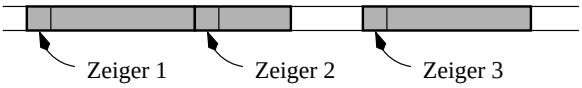

Abbildung 39: Speicherblöcke im Heap und Zeiger darauf

RAM ist üblicherweise in einem Microcontroller als ein adressierbarer Block realisiert. Von der einen Adressgrenze wächst der Stack, von der anderen der Heap (Abb. 40). Gefährlich ist das Wachsen von Heap und Stack ineinander, da dadurch der Program-

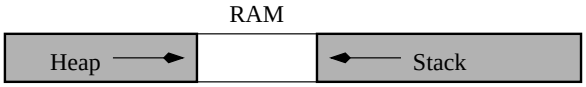

Abbildung 40: Heap und Stack im RAM

mablauf zerstört werden kann (Return from Subroutine springt zu unsinnigen Adressen) oder Daten im Heap oder Stack überschrieben werden. Dies führt zu einem gefährlichen Absturz des Programms oder zum ebenso gefährlichen Weiterarbeiten mit verfälschten Daten. Der Einsatz von Heap ist daher besonders sorgfältig zu prüfen.

Die Verwaltung des Heaps kostet zudem Rechenzeit und die Routinen dafür Programmspeicherplatz. Das Setup des Heaps wird automatisch durch den Compiler in den Startup-Code eingefügt und wird vor dem Einsprung in `main` ausgeführt. Ein minimales Setup generiert einen Zeiger auf die Spitze des Heaps und auf die Adresse der maximalen Ausdehnung, was nur wenige Takte kostet. Jede Allozierung (`malloc`) oder Freigabe (`free`) kostet Rechenzeit (beim MSP430 jeweils rund 200 Takte). Dieser Aufwand kann minimiert werden, indem nicht ständig neue Speicherbereiche alloziert und freigegeben werden, sondern möglichst einmalig der benötigte Speicher alloziert wird.

Neben der Rechenzeit ist auch Speicher für die Verwaltung von Heap notwendig. So muss für jeden allozierten Speicherblock die Größe und ein Flag, welches anzeigt, ob der Speicherbereich noch alloziert oder schon freigegeben ist, abgespeichert werden. Zwei Speicherplätze pro Speicherblock plus zwei Speicherplätze für den gesamten Heap sind somit nötig.

Werden mehrere Speicherblöcke nacheinander alloziert und wird nicht der letzte Block zuerst freigegeben, entstehen Löcher im Heap. Da das Einfügen von neu allozierten Speicherblöcken in diese Löcher aufwändige Tests nach sich zieht, wird nicht jede Heap-Verwaltung dies durchführen. Stattdessen bleiben Löcher ungenutzt und neuer Spei-

cher wird auf der Spitze des Heaps alloziert. Dies kann schnell zur Verschwendung von großen Teilen des RAMs führen und auch diese einfache Verwaltung der Löcher kostet Rechenzeit. Speicherblöcke, die lange benötigt werden, sollten daher vor Speicherblöcken alloziert werden, die eine kürzere Lebensdauer haben.

Das Vergrößern des allozierten Speichers mittels `realloc` führt zu aufwändigen Kopieraktionen, wenn sich der Speicherbereich nicht an der Spitze des Heaps befindet, oder hinter dem Bereich kein ausreichend großes Loch existiert. In solchen Fällen sind die Daten des bereits allozierten Teils an die Spitze des Heaps in den Bereich des neu allozierten größeren Blocks zu kopieren. Wird der allozierte Speicher verkleinert, so sind nur Größeninformationen anzupassen. Ungünstige Nutzung von `realloc` kann somit zu großen Löchern im Heap und damit Verschwendung von Speicherplatz führen. Häufig muss `malloc` von `realloc` aufgerufen werden, sodass die Kosten für `realloc` meist über denen von `malloc` (aber nicht über denen der Summe aus `malloc` und `free`) liegen.

Werden die genannten Richtlinien beim Einsatz von Heap berücksichtigt und ist der verbleibende Aufwand im Vergleich zum restlichen Programm gering, so kann die Nutzung von Heap eine sinnvolle Alternative sein. In vielen Fällen wird aber der Einsatz von Variablen fester Größe effizienter als die Nutzung von Heap bleiben.

3.3. Methoden zur Optimierung auf Hochsprachenebene

Softwareoptimierung auf Hochsprachenebene ist meist ein top-down-Prozess. Die Ausgangsbasis bildet ein Algorithmus. Für viele numerische Probleme existieren bereits hoch optimierte Routinen, wie sie z. B. in [66] zu finden sind. Neben der Tatsache, dass es nicht für jedes Problem eine fertige Lösung geben kann, existiert auch stets das Problem, einen effizienten Datenfluss zwischen den Routinen zu erreichen, also alle Teile zu einem sinnvollen Ganzen zusammenzuführen. Wesentliche Punkte bei der Optimierung auf strukturaler Ebene sind angepasste Datenstrukturen, Programmflussoptimierung, Reduktion von Speicherzugriffen und auch die effiziente Nutzung des Instruktionssatzes (These 3c).

3.3.1. Angepasste Datenstrukturen

Wesentlich für eine Gesamtlösung ist die Auswahl geeigneter Datenstrukturen. Dies umfasst Variablen elementaren Typs, Vektoren, Strukturen und komplexe Gebilde sowie davon abgeleitete Formen wie Listen und Bäume. Existiert eine alles bestimmende Kernroutine, so wird diese meist die Datenstruktur vorgeben. Häufige Datenübergaben zwischen Funktionen können allerdings ebenfalls relevant sein, sodass eine effiziente Datenstruktur für diese Übergaben vorteilhaft sein kann (vgl. Kap. 3.2.7).

Stehen verschiedene Daten miteinander in Relation, so erscheint eine Gruppierung in Strukturen vorteilhaft für eine inhaltliche Verkettung. Dies fördert das Verständnis des Quellcodes. Eine Gruppierung in Strukturen kann im Einzelfall vorteilhaft sein, wenn nur ein einziger Zeiger zu Referenzierung großer Datenmengen nötig ist. Sie kann aber auch Nachteile beim Datenzugriff durch umfangreiche Adressberechnungen und unnötige Speicherzugriffe haben. Das Verteilen des Datenbestandes auf elementare Datentypen

oder Vektoren aus elementaren Datentypen sollte als Alternative untersucht werden.

Es kann vorteilhaft sein, Sentinels in die Datenstruktur einzubringen. Sentinels sind Markierungen, die in Vektoren oder verkettete Listen eingebracht werden, um z. B. die Suche in diesen Gebilden zu beschleunigen [63]. Das klassische Beispiel für einen Sentinel ist in einer Zeichenkette das Zeichen `'\0'`. Wird in einer verketteten unsortierten Liste ein Datum gesucht, so kann es sinnvoll sein, das gesuchte Datum zusätzlich an das Ende der Liste zu hängen und dann die Liste zu durchlaufen. Dadurch kann der Test, ob das Listenende erreicht wurde, eingespart werden und es müssen nur die Listenelemente mit dem gesuchten Datum verglichen werden. Zusätzliche Prüfungen sind nötig, um zu erkennen, ob nur der Sentinel oder tatsächlich das gesuchte Element gefunden wurde.

3.3.2. Arithmetische Vereinfachung und Programmflussoptimierung

Sowohl komplexe Routinen wie, z. B. eine FFT, als auch kleinere Berechnungen und der Programmfluss sollten an das spezifische Problem angepasst und optimiert werden.

Das Vermeiden doppelter Berechnungen kann zu signifikanten Einsparungen führen. Das Zwischenspeichern von (Teil-)Ergebnissen ist die offensichtlichste Möglichkeit dafür. Etwas abstrakter ist die Kopplung ähnlicher Berechnungen, z. B. die gleichzeitige Minimum- und Maximum-Suche in einem Vektor [63] oder die Verschmelzung von gleichartigen Schleifen. Besonders wirkungsvoll kann auch das Abspeichern von vorausberechneten Werten in „Lookup Tables“ sein.

Durch die gewählte Struktur der Programmverzweigungen können unsinnige Berechnungen vermieden werden. Es ist sinnvoll, einfache Tests vor komplizierten zu durchlaufen. Besonders häufige oder seltene Möglichkeiten sollten direkt ausgeführt oder mit geringem Aufwand übersprungen werden. Verzweigungen mittels `switch` sind effektiver als gleichartige if-else Verzweigungen (Kap. 3.2.6). Nicht nur bei Verzweigungen können unsinnige Berechnungen vermieden werden. Bei Schleifen eliminiert oder reduziert (Partial) Loop Unrolling die Schleifenbehandlung.

Multiplikation (ohne Hardwaremultiplizierer), Division und Modulo sind aufwändige Operationen. Sind stattdessen Shiftoperationen möglich, führt dies zu großen Einsparungen (Kap. 3.2.6.1). Die Division durch eine Konstante sollte durch die Multiplikation mit der Inversen ersetzt werden, da die Division viel aufwändiger ist.

Manuell ausgelöste Typkonvertierungen mittels Type Cast können unnötig aufwändige Konvertierungen vermeiden und es kann die Suche nach überflüssigen Konvertierungen erleichtert werden, da sie dann im Quellcode ersichtlich sind. Idealerweise sollte stets der native Datentyp der Maschine (üblicherweise `int`) verwendet werden.

Der Einsatz von Gleitpunktdatentypen (`float`, `double`) kann schnell zu hohen Rechenzeiten führen, wenn ohne FPU Gleitpunktarithmetik emuliert werden muss. Eine Alternative kann Fixpunktarithmetik mit einem ausreichend breiten Integer-Datentyp sein, bei dem das Komma mittels Shift-Operation virtuell an eine geeignete Position verschoben wird. Ein Beispiel mit Angabe der Ausführungszeit für den MSP430 wird in Listing 16 gezeigt. Die Ausführungszeiten für den PIC haben ähnliche Relationen. Die konkreten Werte sind stark abhängig von der Qualität der eingesetzten Floating-Point

```
int var = 20;
float af = 1.7;
float bf = 0.4;
float cf = 15.629;
long al = 0x1b3;
/* 1.7 = 2^0 + 2^(-1) + 2^(-3) + 2^(-4) + 2^(-7) + 2^(-8) = 1,10110011b */
long bl = 0x66;
/* 0.4 = 2^(-2) + 2^(-3) + 2^(-6) + 2^(-7) = 0,01100110b */
long cl = 0xFA1;
/* 15.629 = 2^3 + 2^2 + 2^1 + 2^0 + 2^(-1) + 2^(-3) + 2^(-8) = 1111,10100001b */
float resf;
int resi,resi_round;

resf = (float)var*af +
       (float)var*bf +
       cf; /* resf=57,629; 1201 clocks */

resi = (int)((
              ((((signed long)var<<8)*al)>>8) +
              ((((signed long)var<<8)*bl)>>8) +
              cl
             )>>8); /* resi=57; 146 clocks */

resi_round = (int)((
                    ((((signed long)var<<8)*al + (1<<7))>>8) +
                    ((((signed long)var<<8)*bl + (1<<7))>>8) +
                    cl + (1<<7)
                   )>>8); /* resi_round=58; 155 clocks */
```

Listing 16: Gleitpunkt- versus Fixpunktarithmetik (MSP430 mit Multiplizierer)

Bibliothek. Die Fixpunktverschiebung wurde im Beispiel mit 8 festgelegt - siehe (25).

$$\begin{aligned} product &= a \cdot b = \frac{2^{16}}{2^{16}} a \cdot b = \frac{1}{2^{16}} \left(a\ 2^8 \cdot b\ 2^8 \right) \\ product \cdot 2^8 &= \frac{1}{2^8} \left(a\ 2^8 \cdot b\ 2^8 \right) \end{aligned} \tag{25}$$

Das Beispiel aus Abb. 16 zeigt auch das prinzipielle Problem von Rundung bei Fixpunktarithmetik. Für Ergebnis `resi` wurden einfach die Nachkommastellen abgeschnitten. Soll das Ergebnis rundungsrichtig ermittelt werden (`resi_round`), so ist vor dem Shiften um n Bit der Wert 2^{n-1} zu den Daten hinzuzuaddieren.

Im Beispiel sind Multiplikationen enthalten. Steht kein Hardwaremultiplizierer bereit, sind diese eventuell sehr aufwändig in Software zu realisieren. Wird das selbe Beispiel für den Z80 (der keinen Hardwaremultiplizierer besitzt) compiliert, ist die Variante mit dem Datentyp `float` die schnellste von allen, da beim verwendeten Z80-Compiler eine `float` Softwaremultiplikation deutlich schneller realisiert ist als eine `long` Softwaremultiplikation. Beim MSP430 bleiben auch bei Softwaremultiplikation die `long`-basierten Versionen deutlich die schnellsten.

Der Einsatz von Fixpunktarithmetik ist sorgfältig zu prüfen, da die numerische Genauigkeit deutlich schlechter und der Wertebereich deutlich kleiner ist als bei Gleitpunktarithmetik. Overflow kann zu einem Problem werden.

3.3.3. Reduktion von Speicherzugriffen

Die Reduktion von Speicherzugriffen ist der Schlüssel zur Optimierung auf Befehlsebene. Wichtigstes Mittel dafür ist eine effiziente Registernutzung. Da diese der Hochsprache verborgen bleibt, ist eine Analyse des Programms nötig, wie sie mit der in Kap. 3.4

vorgestellten Methode bereitgestellt wird. Eine Steuerung der Registernutzung ist in einer Hochsprache nur indirekt möglich. Das Abspeichern von Daten in Registervariablen statt in komplexen Gebilden und die Übergabe von Funktionsparametern als Registervariablen sind geeignete Mittel.

Die Nutzung von Funktionsparametern im Sinne der Klassifikation von Funktionen (Kap. 3.2.7.3) erleichtert die effiziente Übergabe. Da Funktionsaufrufe ebenfalls Speicherzugriffe nötig machen, kann das Flachmachen der Struktur, also der Einsatz von Makros oder Function Inlining sinnvoll sein.

Eine unnötige Blockierung von Registervariablen durch zu zeitige Zuweisungen von Daten oder Initialisierungen sollte vermieden werden. Es ist vorteilhaft, konstante Werte mittels `const` oder besser `#define` zu definieren, sodass sie nicht als theoretisch veränderbare Variablen Register blockieren, falls der Compiler dies nicht erkennt.

Das wichtigste Prinzip zur Optimierung ist die Nutzung lokaler Kopien und lokaler Zeiger. Idealerweise sollten dafür ausreichend Register frei sein, aber selbst wenn dies nicht der Fall ist (wie beim PIC), kann durch die Elimination von Adressarithmetik beim Zugriff auf diese Daten durch lokale Kopien oder Zeiger ein Vorteil erreicht werden.

3.3.4. Effiziente Nutzung des Instruktionssatzes

Es ist nicht Ziel, in einer Hochsprache auf Instruktionsebene zu programmieren. Dennoch kann durch geeignete Konstrukte der Compiler unterstützt werden, auch ohne den Instruktionssatz eines Prozessors tatsächlich zu kennen.

Die Operatoren für Autoinkrement (`x++`, `++x`, `x--`, `--x`) sollten stets genutzt werden, insbesondere beim Zugriff auf Daten über Zeiger (z. B. `*(p++)`).

Lokale Zeiger und deren direkte Modifikation (im Idealfall mittels Autoinkrement) ermöglichen manchmal zusätzlich eine Entkopplung der Speicherzugriffe von Schleifenzählvariablen. Beispielsweise ist beim Durchlaufen eines Vektors `vec[]` (z. B. zur Maximum-Suche) nicht der Zugriff über eine Schleifenzählvariable mittels `vec[i]` nötig, wenn stattdessen ein lokaler Zeiger mit jedem Zugriff modifiziert wird (`*(pvec++)`). Die Schleifenzählvariable ist somit frei wählbar und Flags für Abbruchbedingungen können direkt genutzt werden (Kap. 3.2.6.2). Decrementieren der Zählvariable bis Null führt beispielsweise zur Nutzung des Zero-Flags. Derartige Feinheiten mögen minimalen Einfluss haben, aber da viele Schleifen sehr häufig durchlaufen werden, kann dies in der Summe zu signifikanten Einsparungen führen. Ähnliches gilt für die Regel, dass do-while allen anderen Schleifen vorzuziehen ist, falls es möglich ist.

Wie in Kap. 3.2.6.4 ausgeführt, werden `break`, `continue` und auch `goto` durch sehr effiziente Sprünge repräsentiert, sodass damit viele unnötige zusätzliche Prüfungen übersprungen werden können. Wird `return` nicht nur am Ende von Funktionen eingesetzt, führt dies ebenfalls zu Abkürzungen.

Erst wenn alle Möglichkeiten auf Hochsprachenebene ausgereizt sind und durch das Studium des disassemblierten Maschinencodes weiteres Optimierugspotential ersichtlich wird, sollten Inline Assembly, also das Einflechten von Assembler-Instruktionen innerhalb der Hochsprache oder Funktionen mit Assemblercode genutzt werden.

```
if (Bedingung1)
{
  /*kurzer Kommentar #1*/
  aktion1a();
  aktion1b();
}
else
{
  if (Bedingung2)
  {
    /*kurzer Kommentar #2*/
    aktion2a();
    aktion2b();
  }
  else
    /*
    Dies ist ein langer Kommentarblock, der
    einen komplizierten Sachverhalt erklärt.
    */
    aktion3();
  if ((Bedingung3 || Bedingung4) && Bedingung5 && (Bedingung6 ^ Bedingung7))
  {
    aktion4a();
    aktion4b();
  }
}
```

Listing 17: Quellcodeformatierung - Variante 1

3.3.5. Quelltextformatierung

Keine direkte Methode zur Optimierung, aber ein sehr wichtiger Punkt ist eine konsistente, effiziente Formatierung des Quelltextes. Ein gut lesbarer Quelltext ermöglicht es, sowohl algorithmische als auch strukturale Optimierungen einfach zu erkennen und ist damit wesentlich für den Erfolg der Suche nach Optimierungsmöglichkeiten.

Es gibt viele verschiedene Meinungen, wie Quelltext formatiert werden sollte, meist wird aber nur der persönliche, konsistente Stil betont. Eine recht verbreitete Möglichkeit ist in Listing 17 illustriert. Für die effiziente Suche nach Optimierungsmöglichkeiten kristallisieren sich aber die folgenden konkreten Richtlinien heraus:

Vertikale Kompression: Komplexe Verzweigungen sind umso schwerer zu erfassen, je weiter sie gestreut sind. Da Quelltexte schnell sehr lang werden können, ist die begrenzte vertikale Auflösung von Computerbildschirmen ein Problem. Dagegen ist die horizontale Auflösung meist mehr als ausreichend.

Trennung von Code und Kommentar: Das menschliche Auge wird durch klare Strukturen geleitet. Kurze Kommentare unterbrechen den Fluss und sollten daher, wenn möglich neben dem Quellcode platziert werden, sodass quasi ein zweispaltiger Text entsteht. Dies ermöglicht auch eine vertikale Kompression.

Klare Strukturierung: Komplizierte Ausdrücke können leichter erfasst werden, wenn Klammern unterstützend eingesetzt und Teile gruppiert werden. Starkes Einrücken von Quelltext ermöglicht eine leichte Zuordnung von Abschnitten.

Eine Möglichkeit zur Umsetzung dieser Richtlinien wird in Listing 18 illustriert. Nicht immer können alle Richtlinien gleichzeitig erfüllt werden.

```
if (Bedingung1) {
        aktion1a();                                  /*kurzer Kommentar #1*/
        aktion1b();
} else {
        if (Bedingung2) {
                aktion2a();                          /*kurzer Kommentar #2*/
                aktion2b();
        } else {
                /* Dies ist ein langer Kommentarblock, der
                 einen komplizierten Sachverhalt erklärt. */
                aktion3();
        }
        if (    (Bedingung3 || Bedingung4)
                && Bedingung5
                && (Bedingung6 ^ Bedingung7)) {
                /**/
                aktion4a();
                aktion4b();
        }
}
```

Listing 18: Quellcodeformatierung - Variante 2

3.4. Erweiterte Programmanalyse

3.4.1. Einführung

Die Extraktion von Informationen über den Programmfluss während der Programmabarbeitung nennt man „Profiling“ - ein Werkzeug, das diese Aufgabe erledigt „Profiler“. Mit Hilfe dieser gewonnenen Informationen, ist eine Analyse möglich und Schritte zur Optimierung können abgeleitet werden.

Die meisten klassischen Profiler bereiten Informationen auf der Hierarchieebene von Funktionen auf und präsentieren deren Kosten. Diese grobe Darstellung ermöglicht es zwar, die Funktionen zu identifizieren, die die größten Kosten verursachen, aber es werden nahezu keine Information präsentiert, warum die ermittelten Kosten entstehen. Feinkörnige Programmanalyse und konkrete Bezugspunkte für die Gründe der Kosten sind für die bei Sensorsystemen geforderte hoch optimierte Software nötig.

Als Kosten können verschiedene Größen angesehen werden. Die wichtigste ist die Laufzeit. Sie ermöglicht zwar eine Analyse und einen Vergleich verschiedener Realisierungen, allein aus der Kenntnis der Laufzeit sind aber kaum die dafür verantwortlichen Gründe abzuleiten. Daher werden die folgenden Methoden zur Verbesserung vorgeschlagen:

1. Zur Optimierung von Software ist nicht nur die Lokalisierung der Kosten auf Funktions-, sondern auf Befehlsebene nötig. Detailliertes, feinkörniges Profiling auf Befehlsebene mit Angabe von mehreren Größen (nicht nur der Laufzeit), ermöglicht die Erkennung der Hauptquellen für die Kosten (These 4a).

2. Die Analyse von Variablen, deren Speicherort und Zugriffe darauf decken Gründe für die Kosten auf. Nach der Identifikation der kostenintensivsten Funktionen und Befehle stellen diese Informationen eine Grundlage dar, aus der konkrete Ansätze zur Optimierung entstehen können (These 4b).

3. Automatisch generierte Hinweise für mögliche Optimierungsschritte mit direktem Bezug auf die konkreten Positionen im Quelltext können insbesondere für weniger erfahrene Softwareentwickler hilfreich sein (These 4c).

Diese Vorschläge zielen auf feinkörniges Profiling für hoch optimierte Software, wie sie insbesondere für Microcontroller benötigt wird, ab. Grobkörniges Profiling klassischer Profiler hat oft nur einen geringen Informationswert für Programme für Microcontroller, da diese üblicherweise nicht extrem komplex und unüberschaubar sind, sodass grobe Aussagen über besonders kostenintensive Funktionen direkt aus der Programmierung heraus bekannt sind. Weiterhin ist die Architektur der CPUs von Microcontrollern meist nicht sehr komplex. Pipelines sind simpel [22] oder existieren nicht [23, 24], Caches werden nicht genutzt und hauptsächlich Single-Task Software wird eingesetzt. Daher werden Befehle stets gleichartig abgearbeitet. Die Informationen von feinkörnigem Profiling sind demzufolge wiederverwendbar bei der Programmoptimierung, da die Kosten nicht durch Pipeline- oder Cache-Effekte überlagert werden.

Für den Microcontroller MSP430 wurde im Zuge dieser Arbeit ein Profiler, genannt `eprof`, realisiert, der die im Folgenden diskutierten Profilinginformationen bereitstellt. Der Profiler `eprof` ist für ANSI C als Hochsprache und den MSP430-GCC [59] als Beweis für die Realisierbarkeit der vorgestellten Konzepte entworfen worden. Da er eine Realisierbarkeitsstudie darstellt, gibt `eprof` alle Informationen als reinen Text aus. Eine grafische Ausgabe ist nicht ausgeschlossen und wäre durchaus vorteilhaft, aber der Schwerpunkt liegt hier in der Extraktion der Informationen und weniger auf deren Präsentation.

3.4.2. Bekannte Ansätze

Hauptsächlich geben heute Profiler die Laufzeit an, die ein Programm für eine Funktion benötigt hat („Flat Profile“) und die Information, welche Funktion, welche Subfunktion aufgerufen hat („Call Graph“). Beispiele dafür sind `gprof` [67], `OProfile` [68] und `VTune` [69]. Das Flat Profile und der Call Graph sind klassische grobkörnige Informationen („Profiles“). Die Anzahl der Ausführungen („Annotated Source“) und die Analyse der Ausführungszeit („Line-by-Line Profile“) jeder einzelnen Zeile Quellcodes sind erste Schritte hin zu feinkörnigen Informationen. Während `gprof` diese noch getrennt behandelt, kombinieren sie `OProfile` und `VTune` und vereinfachen damit die Bewertung seitens des Programmierers. Nichtsdestotrotz werden noch nicht genügend Informationen bereitgestellt. Außerdem gibt `OProfile` ungünstigerweise nur relative Ausführungszeiten an.

Feinkörniges Profiling kann in sehr beschränkter und aufwändiger Weise mittels Disassembler auch manuell durchgeführt werden, wie beispielsweise `objdump` [67], um den Maschinencode mit Hochsprachenbefehlen in Relation zu setzen. `OProfile` kann dies in automatisierter Form realisieren, was eine große Erleichterung darstellt, aber bei `OProfile` existieren Probleme, Instruktionen den Befehlen zuzuordnen, da Gruppen von Instruktionen im Maschinencode auseinander gerissen und weit verteilt vom Compiler platziert werden können.

Die Analyse von Variablen und Zugriffen kann in rudimentärer, sehr begrenzter Form mittels Debugger, wie beispielsweise `ddd` [70], durchgeführt werden. `Kprof` [71] ermöglicht die Analyse von Objekten und gibt die Laufzeit von Methoden an, analysiert aber Variablen nicht. Da objektorientierte Programmierung bei Microcontrollern unüblich ist, ist aber die Analyse von Variablen von größerem Interesse.

`VTune` präsentiert Hinweise für Optimierungen bei Detektion von speziellen Ereignissen, wie Cache Misses, und nutzt Compilertechniken, um algorithmische Verbesserungen vorzuschlagen. Compilertechniken bieten ein weites Feld für Generierung von Hinweisen, aber auch in den Debugging-Informationen des ausführbaren Programms sind genügend Informationen enthalten, um Hinweise auszulösen.

Alle genannten Profiler sind für den Einsatz bei komplexen Prozessoren, wie x86er, entworfen worden. Für die Softwareentwicklung bei Microcontrollern sind Profiler mit feinkörnigen Profiles nicht sehr verbreitet. In der Entwicklungsumgebung von IAR [58] für MSP430 werden beispielsweise nur Flat Profile und Call Graph präsentiert. Gerade aber für Microcontroller sind detaillierte Informationen wünschenswert.

3.4.3. Methoden zur erweiterten Programmanalyse

3.4.3.1. Detaillierte Analyse der Programmkosten Das Annotated Source Profile gibt die Anzahl der Ausführungen eines Befehls an. Befehle können durch mehrere Instruktionen repräsentiert werden, welche zusätzlich abhängig sind von der konkreten Speicherposition der beeinflussten Variablen. Somit beschreibt die Anzahl der Ausführungen nur grob die tatsächlichen Kosten. Präziser ist die Angabe der Laufzeit durch das Line-by-Line Profile, aber häufig ist nicht allein die Laufzeit von Interesse. Die Verschmelzung von Flat, Annotated Source und Line-by-Line Profile führt zu brauchbaren Angaben über die durch die Ausführung verursachten Kosten. `OProfile` stellt eine solche Verschmelzung her, gibt allerdings nur relative Ausführungszeiten an und bezieht diese auf den disassemblierten Maschinencode, nicht aber direkt auf die Befehle in der Hochsprache.

Um eine Abstraktion der Programmanalyse vom Maschinencode zu erhalten, wird daher vorgeschlagen, die Kosten auf die Hochsprachenbefehle zu beziehen und verschiedene Kosten anzugeben, sodass je nach konkretem Optimierungsproblem geeignete Angaben zur Verfügung stehen. Solch eine Ausgabe wird im Folgenden *Cost Profile* genannt.

```
/*     1 clk,     2 bytes i,     2 bytes r,     0 bytes w,     0.975pJ (CPU),     1 exe*/
int cost_example(int in1, int in2, int in3) {
    int ivar;
/*     3 clk,     4 bytes i,     4 bytes r,     0 bytes w,     2.549pJ (CPU),     1 exe*/
    if (in1 == in2) {
```

Listing 19: Cost Profile (Fragment auf Befehlsebene)

Der Profiler `eprof` gibt ein solches Cost Profile aus, wie es Listing 19 illustriert. In dieser Form bezieht sich eine Kommentarzeile mit den Kosten auf die jeweils folgende Quellcodezeile. Im Beispiel ergeben sich also für den Eintritt in die Funktion `cost_example` die Kosten: 1 Taktzyklus Ausführungszeit; 2 Bytes Programmspeicher für die Instruktion; 2 Bytes wurden während der Ausführung aus dem Speicher gelesen und 0 Bytes geschrieben; 0.975 pJ setzte die CPU um und die Befehlszeile wurde einmal ausgeführt. Die Angabe der Kosten vor der entsprechenden Befehlszeile ermöglicht es, Befehlszeilen leichter zu erkennen, denen nicht direkt Programmcode zugeordnet ist (wie im Beispiel `int ivar;`), oder die nicht ausgeführt werden, weil z. B. ein Verzweigungsast nicht ausgeführt wurde.

Kann die umgesetzten Energie pro Instruktion abgeschätzt oder bestimmt werden, so ist es möglich sie wie im Beispiel aus Listing 19 im Cost Profile anzugeben. Die Bestimmung dieser Größe wurde in Kap. 2.3 diskutiert. Wie dort schon ausgeführt, ist eine solche Angabe allerdings nur sinnvoll, wenn die Schwankungsbreite der umgesetzten Energiemenge relativ groß ist. Generell ist die Angabe stark mit der Ausführungszeit korreliert und und hat daher nur geringen Informationswert.

Die Angabe der Anzahl der aus dem Speicher gelesenen und in den Speicher geschrieben Bytes kann als Indikator verwendet werden, wie effizient der Befehl umgesetzt wurde. Da jeder Speicherzugriff Zeit kostet und dadurch große Energiemengen umgesetzt werden, ist die Minimierung der Zugriffe entscheidend für die Effizienz von Optimierungen. Da selbstverständlich auch die Instruktionen selbst aus dem Speicher gelesen werden müssen, ist die Angabe von deren Umfang ebenfalls von Bedeutung. Die damit gleichzeitig angegebene Programmgröße kann bei Microcontrollern mit ihren üblicherweise begrenzten Ressourcen von Interesse sein.

Neben der Angabe der Kosten pro Befehlszeile bleibt die Summe aller Kosten pro Funktion und für das gesamte Programm analog zum Flat Profile natürlich weiterhin von Interesse und wird von `eprof` ebenfalls ausgegeben. Wesentlich für das Cost Profile ist aber die Angabe von möglichst vielen relevanten Größen und der direkte Bezug auf die Befehle in der Hochsprache, ohne dass die Details der Maschineninstruktionen hervortreten müssen. Dies ermöglicht eine genaue Lokalisierung der Hauptquellen für die Programmkosten auf Hochsprachenebene.

3.4.3.2. Analyse von Variablen

Motivation Die Analyse der Programmkosten mittels Cost Profile (Kap. 3.4.3.1) ist eine wichtige Methode zur Analyse von Software, aber es ergeben sich daraus nur wenige direkte Ansätze zur Optimierung. Steht ausschließlich das Cost Profile zur Verfügung, so kann ohne die aufwändige manuelle Analyse des disassemblierten Maschinencodes in den meisten Fällen nur geschätzt werden, warum bestimmte Befehle die ausgegebenen Kosten verursachen. Lediglich die Anzahl der Speicherzugriffe liefert erste Einblicke in die Ursachen, bleibt aber noch recht unkonkret.

Jede CPU-Architektur hat spezifische Vor- und Nachteile und daher existieren spezielle Optimierungsmöglichkeiten, aber mit der Beschränkung des Blickwinkels auf Microcontroller ergeben sich einige allgemeine Fakten:

Speicherposition: Variablen können in CPU-internen Registern oder im RAM abgelegt werden. Normalerweise sind Zugriffe auf den RAM langsamer und setzen mehr Energie um als Registerzugriffe. Welche Variablen im RAM abgelegt werden, wurde in Kap. 3.2.4.3 erläutert.

Adressarithmetik: Die Position einer Variablen im RAM wird durch ihre Adresse angegeben. Die Adresse kann fest und zur Compile-Zeit bekannt sein, bestimmt werden durch einen Zeiger plus eventuell einen Offset oder muss berechnet werden. Das

Laden eines festen Wertes für die Adresse ist normalerweise die schnellste und einfachste Möglichkeit, während eine Berechnung viele Maschineninstruktionen umfassen kann.

Beschleunigte Arithmetik: Häufig stellen CPUs von Microcontrollern Instruktionen für beschleunigte Adressarithmetik bereit. Für tabellenähnliche sequentielle Zugriffe (wie sie bei der Suche in einem ungeordneten Vektor oder beim Kopieren von Speicherbereichen auftreten) können Instruktionen bereitstehen, die einen Zeiger nach oder vor jedem Zugriff mit nur geringem oder gar keinem zusätzlichen Aufwand inkrementieren oder dekrementieren (siehe Autoinkrement in Kap. 3.2.5). Weiterhin stehen oft einige spezielle Konstanten oder zumindest allgemein anwendbare Inkrement- / Dekrement-Befehle zur Verfügung. Alle genannten Mechanismen werden automatisch von Compilern genutzt, wenn sie ausreichend klar in der Hochsprache beschrieben sind und führen zu kompakteren, schnelleren Programmen.

Funktionsaufrufe: Jeder Funktionsaufruf selbst verursacht Kosten. Das umfasst `CALL` und `RETURN`, das Sichern von Registerinhalten (`PUSH` und `POP`) und die Übergabe von Parametern (vgl. Kap. 3.2.7).

Ausgehend von diesen Fakten können einige grundlegende, komprimierte Methoden zur Softwareoptimierung abgeleitet werden: Es ist vorteilhaft, lokale Kopien von oft genutzten Variablen anzulegen, lokale Zeiger zu nutzen, welche direkt auf Elemente innerhalb größerer Gebilde zeigen, komplexe Gebilde wie Vektoren von Strukturen, Listen oder Bäume nur in notwendigen Fällen einzusetzen, beim Speicherzugriff Schemen zu verwenden, die beschleunigt werden können und vorhandene Konstanten zu nutzen. Die Übergabe von Funktionsparametern in Registern ist zu bevorzugen, wenn genügend Register frei sind und ansonsten kann die Übergabe eines einzigen Zeigers auf eine Struktur eine sinnvolle Alternative sein.

Um diese Methoden anwenden zu können, sind Informationen über die Registernutzung, die Größe von Variablen, deren Speicherposition, Zugriffshäufigkeit und die Kosten für die Befehlsausführung nötig. Können solche Informationen verfügbar gemacht werden, so ist nur noch geringe Kenntnis über die CPU-Architektur und Compiler-Mechanismen nötig, um die korrekten Schlussfolgerungen für Optimierungen zu ziehen. In einem solchen Fall wird nur noch die Kenntnis über die Anzahl der Register, mögliche beschleunigte Adressarithmetik, vorhandene Konstanten und die Organisation der Funktionsparameterübergabe benötigt. Kenntnis in Assemblerprogrammierung ist nicht mehr notwendig (These 4d).

Variablen- und Typanalyse Ein wesentlicher Schritt zur Bestimmung der Ursachen für Programmkosten ist die Kenntnis über die Behandlung von Variablen nach Compilation und Assemblierung. Diese Idee hat in geringem Maße etwas mit der *Object and Method Analysis* von `Kprof` gemeinsam, führt aber zu feinkörnigeren Aussagen. Auch werden objektorientierte Hochsprachen kaum bei der Softwareentwicklung für Microcontroller eingesetzt, sodass die Analyse prozeduralen Programmierens, also die Analyse von Funktionen und Variablen von höherem Interesse ist, als die Analyse von Objekten.

```
struct stype {
    int int_component;
    char vec_component[9];
    struct exmpl_stype *next; };
```

Listing 20: Beispiel für eine Struktur

Compiler, wie GCC [72], können Debugging-Informationen liefern, die die Speicherposition von Variablen, also die Registernummer oder deren Speicheradresse und die Definitionen von allen Typen, einschließlich deren Größe, beinhalten. Das ermöglicht eine detaillierte Analyse der verwendeten Variablen und deren Abbildung auf Speicherpositionen. Die meisten der in der Motivation zu diesem Kapitel erläuterten Methoden setzen diese Kenntnis voraus.

Neben der statischen Analyse der Variablen und ihrer Speicherpositionen, werden auch dynamische Informationen über ihre Nutzung benötigt. Mittels eines CPU-Simulators ist es möglich, jeden Zugriff auf ein Register oder eine Speicheradresse zu zählen. Das ermöglicht (mit einigen Einschränkungen) Zugriffszähler, die oft genutzte Variablen identifizieren. Dies kann auch angewendet werden, um oft genutzte Komponenten von Strukturen zu bestimmen. Für Elemente von Vektoren gilt das gleiche, aber Zugriffszähler für jedes Vektorelement würden schnell zu viele Informationen für den Anwender bereitstellen, besonders, wenn Vektoren von Strukturen analysiert werden würden. Somit erscheint dies für Vektorelemente nicht sinnvoll und Vektoren sollten daher besser insgesamt betrachtet werden.

Die Ausgabe der erwähnten Informationen soll im folgenden *Variable / Type Profile* genannt werden. Der Profiler `eprof` gibt ein solches Profile aus, wie es in Listing 21 illustriert ist. Im Beispiel werden zwei Integervariablen `i` und `j` sowie die Struktur mit dem Namen `example_struct`, welche in Listing 20 deklariert ist, analysiert. Alle drei Variablen wurden in einer nicht näher zu betrachtenden Funktion intern definiert. Innerhalb der Funktion wurden die betrachteten Variablen auf verschiedene Art und Weise genutzt, was sich im Variable / Type Profile widerspiegelt. Wie die Nutzung im einzelnen ausgesehen hat, ist hier nicht weiter von Interesse.

Im Variable / Type Profile aus Listing 21 werden die Informationen ausgegeben, dass `i` eine Stackvariable ist, `j` auf Register R14 abgebildet wurde und `example_struct` ebenfalls eine Stackvariable ist. Die Größe der Struktur (14 bytes) ist ebenso wie die Größe der Komponenten angegeben und es ist vermerkt, dass ein Loch von der Größe eines Bytes in der Struktur existiert. Während der Programmausführung wurde `i` 26-mal gelesen und 9-mal geschrieben, `int_component` einmal gelesen und geschrieben und `vec_component` 9-mal gelesen und 8-mal geschrieben. Zugriffe auf Register 14 können zwar gezählt werden und werden auch von `eprof` ausgegeben, aber da Register für mehrere Variablen wiederverwendet werden, kann nur der Compiler bestimmen, wann welche Variable aktiv ist. Daher können nur Zugriffszähler für die Register, aber nicht direkt für die darauf abgebildeten Variablen realisiert werden. Die Werte in eckigen Klammern (z. B. am Ende der ersten Zeile) sind „Fußnoten“ bzw. „Querverweise“, welche zum Hint Profile gehören, das in Kap. 3.4.3.3 vorgestellt wird.

```
/*
 ...
 * i is stack variable of type integer ("int", 2 byte, 26 r, 9 w) [8]
 * j is register variable of type integer ("int", 2 byte in R14)
 * example_struct is stack variable of type struct "stype" (14 byte) [7]
      with element int_component (16 bit) of type integer ("int", 1 r, 1 w)
      with element vec_component (72 bit) of type array [6]
            with 9 elements of type integer ("char", 1 byte)
            with total size of 9 bytes (9 bytes r and 8 bytes w)
      with element next (16 bit) of type pointer to [16] struct "stype"
      with 1 byte unused
 ...
 */
/*****************************************************************************/
/*(user) register usage:                                                     */
/*R4 :      0 times read,          0 times written                           */
            ...                    ...
/*R11:      0 times read,          0 times written                           */
/*R12:     24 times read,         16 times written (function parameter register)   */
/*R13:     33 times read,         25 times written (function parameter register)   */
/*R14:      3 times read,          1 times written (function parameter register)   */
/*R15:     14 times read,          5 times written (function parameter register)   */
/*sum:     74 register reads,     47 register writes                        */
/*****************************************************************************/
```

Listing 21: Beispiel 1 für ein Variable / Type Profile (Fragment; vgl. Listing 20)

```
long vartype_function(int a, long b, float c);
```

Listing 22: Beispiel für eine Funktion

Eine sehr wichtige Information für die Programmoptimierung im Beispiel ist der Fakt, dass `i` auf den Stack abgebildet wurde und darauf vergleichsweise häufig zugegriffen wird. Dies führt zu längeren Programmausführungszeiten und größerer Energieumsetzung als die Nutzung eines Registers. Variablen vom Typ `int` sind Registervariablen, aber der Compiler hat `i` auf den Stack abgebildet, da offensichtlich die Adresse von `i` einmal bestimmt worden sein muss, denn die Register R4 bis R11 sind frei. Das Bestimmen der Adresse von `i` sollte demnach vermieden werden und falls das nicht möglich ist, würde eine lokale Kopie vorteilhaft sein. Eine lokale Kopie für `int_component` ist dagegen unsinnig, da auf diese Komponente nur sehr selten zugegriffen wurde.

Zu Gunsten der Übersichtlichkeit blieb in Listing 21 die Analyse von anderen in der Beispielfunktion deklarierten Variablen und des Funktionsaufrufs, also die Analyse der Funktionsparameter und des Resultates, unberücksichtigt. Die Analyse des Funktionsaufrufs soll daher in einem weiteren Beispiel veranschaulicht werden. Für einen Aufruf der in Listing 22 deklarierten Funktion ist in Listing 23 das entsprechende Fragment des Variable / Type Profiles angegeben.

```
/*
 ...
 * vartype_function is function of type integer ("long int", 4 byte) [1]
 * a is register function-parameter of type integer ("int", 2 byte in R15)
 * b is register function-parameter of type integer ("long int", 4 byte in R10+R11) [1]
 * c is stack function-parameter of type floating point ("float", 4 byte) [3] [9]
 ...
 */
```

Listing 23: Beispiel 2 für ein Variable / Type Profile (Fragment; vgl. Listing 22)

Die Variablen `a` und `b` werden über Register übergeben, während für `c` die verfügbaren Register erschöpft sind. Da beim MSP430 4 Register für Funktionsparameter genutzt werden, wäre es sinnvoll, `b` und `c` über Register zu übergeben, da diese je 2 Register (4 Bytes) beanspruchen, während `a` nur 1 Register (2 Bytes) belegt. Dies ist in einfacher Weise durch das Vertauschen der Reihenfolge der Funktionsparameter möglich. Lokale Kopien auf Register von Stackparametern können je nach Zugriffshäufigkeit ebenfalls sinnvoll sein.

Hinweis: Die Variable `b` wurde im Beispiel auf die Register R10 und R11 kopiert (aber im konkreten Fall über R13 und R14 übergeben, was aus dem Variable / Type Profile nicht mehr ersichtlich ist), um R13 und R14 für das Funktionsresultat frei zu räumen.

Die neue Methode des Variable / Type Profiles stellt einen wesentlichen Schritt zur Lokalisierung der Ursache von Programmkosten dar. Kostenunterschiede zwischen verschiedenen Programmvarianten werden in hohem Maße durch Speicherzugriffe bestimmt. Eine Reduzierung derselben ist möglich, wenn deren Ursachen bekannt sind. Diese Ursachen werden in den meisten Fällen aufgedeckt, wenn die Speicherposition von Variablen und die Anzahl von Zugriffen darauf bekannt ist. Eine feinkörnige statische Analyse der Typen von Variablen kann dann helfen, Alternativen zu finden. Das Variable / Type Profile bietet zusammen mit dem Cost Profile damit eine starke Ausgangsbasis zur Programmanalyse für den prozeduralen Softwareentwurf. Erst durch die neuartige feinkörnige Analyse von sowohl Funktionen als auch Variablen erschließen sich die Quellen für die Verluste.

3.4.3.3. Kontextsensitive Hinweise Profiler können ein Programm nur analysieren. Es wird dem Anwender überlassen, die Informationen zu interpretieren, Rückschlüsse zu ziehen und schließlich Optimierungen durchzuführen. Mit den beim Profiling gesammelten Daten ist es aber auch in einigen Fällen möglich, die Aufmerksamkeit des Programmierers auf spezielle Dinge zu fokussieren, indem konkrete Hinweise in Form eines *Hint Profiles* gegeben werden. `VTune` generiert Hinweise ausgehend von Hardwareereignissen, wie Cache Misses und gibt Ratschläge für algorithmische Verbesserungen durch den Einsatz von Compilertechniken. Hinweise können aber auch auf Instruktionsebene gegeben werden, ausgelöst durch Debugging-Informationen. Zwar vermitteln solche Hinweise keine anderen Informationen als Ratschläge in einem Programmierhandbuch zur Effizienzsteigerung, aber sie haben einen wesentlichen Vorteil: Hinweise können an zutreffender Stelle gegeben und der entsprechende Quellcode in der Hochsprache kann hervorgehoben werden.

Der Profiler `eprof` setzt Fußnoten in Form von Zahlen in eckigen Klammern, wie es in Listing 21 und 23 zu erkennen ist. Die tatsächlichen Hinweise werden gesammelt und am Ende der Ausgabe aufgeführt.

Die folgenden Ereignisse bei der Analyse der Debugging-Informationen können Hinweise auslösen. Der Profiler `eprof` gibt diese aus.

- Einsatz nicht-nativer Datentypen: Der unnötige Einsatz von breiteren Datentypen als `int` oder der Einsatz von Floating Point Arithmetik bei Prozessoren ohne FPU sollte vermieden werden.

- Registervariablen auf dem Stack: Variablen und Funktionsparameter, die Registervariablen sind, wurden auf den Stack abgebildet. Wird ein Zeiger auf Registervariablen entdeckt, wird darauf hingewiesen.
- Einsatz von Vektoren: Lokale Zeiger und direkte Modifikation dieser Zeiger können vorteilhaft sein.
- Einsatz von Vektoren oder Strukturen: Lokale Kopien von einzelnen Elementen bzw. Komponenten können sinnvoll sein.
- Deklaration von `static` Variablen: Der Vorteil `static` deklarierter Variablen ist, dass sie permanent existieren, der Nachteil, dass sie in jedem Fall auf den RAM abgebildet werden.
- Strukturen als Funktionsparameter: Die Übergabe von Strukturen als Funktionsparameter oder Resultat schließt aufwändige Kopieraktionen ein.
- kein Funktionsresultat: Ein ungenutztes Funktionsresultat (`void`) ist meist Verschwendung.
- aufwändige Adressberechnungen: Vektoren von Strukturen ziehen umfangreiche Adressberechnungen nach sich. Durch Zeiger innerhalb von Strukturen muss die tatsächliche Adresse in mehreren Schritten bestimmt werden. Listen und Bäume sollten also mit Bedacht eingesetzt werden.
- Makros statt Funktionen: Bei nichtrekursiven Funktionen kann stattdessen ein Makro oder Function Inlining sinnvoller sein.
- Hervorhebung von Schleifen: Schleifen haben oft großen Anteil an den Gesamtkosten, sodass eine Optimierung sinnvoll erscheint. Schleifen können auch automatisch durch den Compiler entstehen, z. B. beim Kopieren einer Struktur.

In erster Linie richten sich derartige Hinweise an weniger erfahrene Programmierer. Hinweise können aber besonders wertvoll sein, wenn sie kontextsensitiv gegeben werden.

3.4.3.4. Zuordnung von Maschineninstruktionen Nicht alle Befehle können von einem Compiler in optimaler Form in Instruktionen übersetzt werden. Auch existiert in ANSI C keine direkte Methode, Flags, wie das Carry-Flag, zu prüfen oder zu nutzen. In diesen Fällen kann es sinnvoll sein, abschnittsweise direkt Assemblercode einzusetzen. Beim GCC ist dies in parametrisierbarer Form möglich, wobei die Operanden die Parameter sind und somit dem Entwickler deren tatsächliche Speicherposition nicht bekannt sein muss. Mit Hilfe von Funktionen und Makros kann der Assemblercode auch verborgen werden, sodass die Übersichtlichkeit der Hochsprache nicht verloren geht.

Ist die Nutzung von Assemblercode gewünscht, so muss dieser aus den disassemblierten Maschineninstruktionen heraus zu erkennen sein, um eine Funktionskontrolle durchführen zu können. Auch ist es oft von Interesse, welche Instruktionen ein Compiler

für eine Befehlszeile erzeugt, sodass daraus leichter Verbesserungen abzuleiten sind. Für dieses Anwendungsgebiet sind Disassembler nötig.

Herkömmliche Disassembler, wie `objdump` [67], ordnen die Instruktionen linear mit aufsteigender Adresse an und ordnen, wenn möglich, die zugehörigen Befehlszeilen in der Hochsprache zu. Da durch den Compiler Instruktionen aber an beliebiger Stelle platziert und Abschnitte mehrfach verwendet werden können, entspricht die Reihenfolge der Instruktionen nicht immer der der Befehle. Ist wie bei herkömmlichen Disassemblern die Reihenfolge der Instruktionen bestimmend, so ist in den genannten Fällen keine Zuordnung von Befehlszeilen zu Instruktionen möglich. Vorteilhaft ist es daher, stattdessen die Instruktionen den Befehlszeilen zuzuordnen, wenn es wie in dem beschriebenen Anwendungsfällen darum geht, abschnittsweise Assemblercode im Austausch für einzelne Befehlszeilen einzusetzen. Das detaillierte Cost Profile ermöglicht die Lokalisierung kritischer Befehlszeilen und durch die Zuordnung von Instruktionen zu Befehlszeilen sind Verbesserungen auf Assemblerebene leichter zu realisieren (These 4d).

Nachteilig bei der Zuordnung von Instruktionen zu Befehlszeilen ist der Verlust des Überblicks auf Maschinencodeebene über das gesamte Programm, da die Instruktionen nicht mehr mit aufsteigender Adresse angeordnet sind. Ist der Überblick auf Hochsprachenebene gewünscht, so ist dies irrelevant. Anderenfalls ist ein herkömmlicher Disassembler einzusetzen.

Die generelle Ausgabe von disassemblierten Instruktionen ist für einen Profiler allerdings nicht zu empfehlen, da ja gerade das Loslösen der Programmanalyse von den Maschineninstruktionen durch die in den vorangegangenen Kapiteln vorgestellten Methoden das Ziel war. Der Profiler `eprof` gibt daher die disassemblierten Instruktionen nur optional aus.

3.4.4. Informationsextraktion

Compiler können in den übersetzten Maschinencode Informationen für Debugger einbetten. Diese Informationen enthalten die notwendigen Fakten für die Variablenanalyse (Typ, Typdefinition, Speicherposition) und ordnen die Maschineninstruktionen bestimmten Zeilen im Quellcode zu. Der MSP430-GCC Compiler [59] erzeugt als Ausgabe ein Maschinenprogramm im Format ELF [73] und bettet darin die Fakten im „stabs“ Debug Format [74] ein, wenn die Übersetzung mit Option `-g` ausgeführt wird. `Eprof` extrahiert aus der ELF Datei Maschinencode und Debugging-Informationen und generiert daraus die Profiles (Abb. 41).

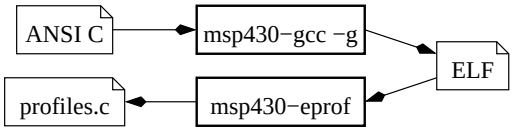

Abbildung 41: das Arbeitsprinzip von `eprof` (für MSP430)

Eine statische Programmanalyse nur aus den Debugging-Informationen und den Maschineninstruktionen heraus, führt nicht zu brauchbaren Resultaten. Eine dynamische

Analyse, also die Einbeziehung von Laufzeitinformationen, ist nötig und der Einsatz eines CPU-Simulators ist dafür eine praktikable Möglichkeit zur Realisierung. CPU-Simulatoren stehen in den meisten Entwicklungsumgebungen zur Verfügung, oder sind mit Kenntnis über die Interna einer CPU relativ schnell implementierbar.

Eine Simulation nicht nur der CPU, sondern auch der Peripheriekomponenten von Microcontrollern (Ports, Timer, ...) ist für die Softwareentwicklung vorteilhaft, setzt aber das Vorhandensein von Simulatoren der Peripheriekomponenten voraus. Diese können sehr komplex werden und setzen häufig auch einen taktgenauen CPU-Simulator voraus, während ein einfacherer, instruktionsgenauer CPU-Simulator für die Simulation ohne Peripheriekomponenten ausreichend ist. Hinzu kommt der Fakt, dass das Hauptinteresse beim Einsatz von Profilern die Evaluation von Umsetzungen von Algorithmen (z. B. Sortieren, Filter, Interpolation, Transformation) ist. Dieses Anwendungsgebiet wird durch einen CPU-Simulator abgedeckt.

`Eprof` enthält einen instruktionsgenauen CPU-Simulator für die MSP430 CPU und den Hardwaremultiplizierer, aber für keine anderen Peripheriebaugruppen. Folgende Informationen müssen während der Simulation aufgezeichnet werden, um genügend Fakten für das Profiling zu sammeln:

- Anzahl der Ausführungen jeder Maschineninstruktion: Aus dieser Anzahl können direkt Ausführungszeit, Programmspeicherbedarf, Anzahl von Register- und Speicherzugriffen sowie Energieumsetzung auf Instruktionsebene abgeleitet werden. Die Zuordnung der Instruktionen zu Befehlen und Funktionen erfolgt später bei der Auswertung.

- Anzahl der Zugriffe auf jede Speicheradresse: Bestimmt wird damit die Gesamtanzahl von Speicherzugriffen und es werden Zugriffszähler für `static` deklarierte Variablen für das Variable / Type Profile ermöglicht.

- Anzahl der RAM-Zugriffe, zugeordnet zu Funktionen: `Eprof` gibt nicht nur alle Speicherzugriffe insgesamt an, sondern liefert auch die Anzahl der Zugriffe in den RAM gesondert und ordnet die Summe einer Funktion zu. Diese Zuordnung könnte auch in späteren Schritten bei der Auswertung gemacht werden.

- Zugriffe in das Stacksegment einer Funktion: Zugriffszähler für Stackvariablen für das Variable / Type Profile erfordern es, diese Zugriffe gesondert zu erfassen.

Mit Hilfe der bei der Simulation gesammelten Daten, den Kenntnissen über den Instruktionssatz der CPU und der Verknüpfung der Daten mit den Debugging-Informationen können die vorgestellten feinkörnigen Programmanalysen durchgeführt werden. Prinzipiell sind diese Mechanismen in jede Entwicklungsumgebung aus Simulator und Debugger integrierbar.

Vorteilhaft wäre eine direkte Zuordnung von Variablen und Registern, sodass stets die aktuelle Registerauslastung bekannt sein würde. Die im stabs Debug Format eingebetteten Informationen allein reichen nicht aus, um diese Zuordnung durchzuführen, da Register von mehreren Variablen verwendet werden. Zwei Ansätze existieren, um diese Zuordnung zu realisieren: Einerseits könnten Compiler-Techniken dies ermöglichen

und andererseits könnten auch die Debugging-Informationen im stabs Format für diese Bedürfnisse erweitert werden, was somit die Zuordnung in den Compiler verlagert.

3.4.5. Präsentation der Ergebnisse

Wie schon erwähnt, ist `eprof` zur Verifikation der Konzepte realisiert worden, sodass die Präsentation der Ergebnisse einen untergeordneten Stellenwert hat und die reine Textausgabe dafür gewählt wurde. Dennoch soll noch einmal kurz in einem Überblick die Ausgabe von `eprof` vorgestellt werden, um einen Eindruck zu vermitteln, welche Informationen in welchem Zusammenhang ausgegeben werden. Gerade diese Zusammenführung von Informationen ist ein wesentlicher Punkt beim Konzept feinkörnigen Profilings.

Die Ausgabe von `eprof` ist in Kommentare eingebettet, welche mit Quellcode zusammen ausgegeben werden. Eine komplette Ausgabe umfasst die folgenden Dinge:

1. einleitende Kommentare, Erklärung von Abkürzungen
2. Kosten für Instruktionen, die keinen Funktionen zugeordnet werden können, wie der Programmstart vor dem Sprung in `main`
3. Analyse aller Funktionen und den darin deklarierten Variablen:
   a) Cost Profile auf Befehlszeilenebene (Listing 19)
   b) Gesamtkosten der Funktion, inklusive Angabe der expliziten Angabe der Zugriffe auf den RAM und der Relation zum Gesamtprogramm
   c) Angabe der Anzahl der verschiedenen Positionen, von denen die Funktion aufgerufen wurde
   d) Variable / Type Profile mit Analyse des Funktionsaufrufs (Listing 23) und allen Variablen (Listing 21) inklusive Zugriffszählern für Stackvariablen
   e) Zugriffszähler für alle Benutzerregister (Listing 21)
4. Analyse externer Variablen inklusive Zugriffszähler
5. Hinweistexte vom Hint Profile (Verweise darauf wurden an geeigneter Stelle gesetzt.)
6. Gesamtkosten für das Programm, inklusive Angabe der expliziten Angabe der Zugriffe auf den RAM
7. optionale Zuordnung von Maschineninstruktionen zu Befehlszeilen in der Hochsprache

Wesentlich ist die gemeinsame Ausgabe aller Informationen bzw. die Möglichkeit dazu. Dies führt zwar einerseits zu einer großen Informationsfülle, ermöglicht aber andererseits die Verknüpfung der Informationen. Wird beispielsweise in einer Befehlszeile einer Variablen ein Wert zugewiesen, so liefert das Cost Profile die damit verbundenen Kosten

```
void quicksort_o(int v[], int left, int right) {
    int i, last;

    if (left >= right)
        return;
    swap_f(v, left, (left+right)>>1); /*keine Division*/
    last = left;
    for(i=left+1; i<=right; i++)
        if (v[i] < v[left]) {
            swap_f(v, ++last, i);
        }
    swap_f(v, left, last);
    quicksort_o(v, left, last-1);
    quicksort_o(v, last+1, right);
}

void swap_f(int v[],int i, int j) {
    int tmp=v[i];
    v[i]=v[j];
    v[j]=tmp;
}
```

Listing 24: Quicksort aus [60] - geringfügig modifiziert

und das Variable / Type Profile am Ende der dazugehörenden Funktion die Analyse der Variablen und damit die Ursachen für die Kosten.

Einer Präsentation der Ergebnisse mit einer modernen graphischen Oberfläche (GUI) steht prinzipiell nichts im Wege. Syntax-Hervorhebung, ausblendbare Detailinformationen, Tooltips, Diagramme zum Vergleich von z. B. Laufzeiten u. v. m. erleichtern das Erfassen der präsentierten Information ungemein. Das Konzept wird dadurch allerdings nicht verändert, sodass für die Verifikation des Konzeptes darauf verzichtet wurde.

3.4.6. Anwendungsbeispiele

3.4.6.1. Quicksort Die Methoden zur erweiterten Programmanalyse sollen an einem Beispiel mit Hilfe des Profilers `eprof` veranschaulicht werden. Dafür wurde die geringfügig modifizierte Quicksort-Funktion aus [60] gewählt. (Listing 24: Die enthaltene Division wurde durch eine Shiftoperation ersetzt.) Es wurde in [60] erwähnt, dass diese Variante nicht die schnellste Implementierung von Quicksort darstellt, aber während der Softwareentwicklung treten unoptimierte Routinen häufig auf.

Listing 25 zeigt die Ausgabe von `eprof` für die Funktion `quicksort_o`. Die anderen Ausgaben für die Subfunktion `swap_f`, die Funktion, aus der die Routine aufgerufen wurde, die Hinweise u. v. m. sind zugunsten der Übersicht nicht dargestellt. Die Routine wurde aufgerufen mit einem Vektor aus 20 ungeordneten, willkürlich gewählten Elementen, welche natürlich den Programmablauf beeinflussen. In Listing 25 sind die Kosten für jede ausgeführte Befehlszeile (gefolgt von der entsprechenden Befehlszeile), die Summe der Kosten (auch in Relation zum Gesamtprogramm), die Anzahl der Zugriffe auf den RAM (von Adresse `0x200` bis `0x9ff` im Beispiel) und das Variable / Type Profile mit Registernutzung angegeben. Mit der Ausgabe von `eprof` können die folgenden Schlussfolgerungen zur Optimierung gezogen werden:

- Am offensichtlichsten ist der Einsatz eines Makros oder Function Inlining für die Funktion `swap_f`, was von `eprof` auch als Hinweis gegeben wird (nicht enthalten in

```
/*****************************************************************************/
/*  600 clk,   20 bytes I,  500 bytes r,  350 bytes w,  551.225pJ (CPU),   25 exe*/
void quicksort_o(int v[], int left, int right) {
   int i, last;

/*   75 clk,    4 bytes I,  100 bytes r,    0 bytes w,   63.725pJ (CPU),   25 exe*/
   if (left >= right)
      return;
/*  120 clk,   14 bytes I,  168 bytes r,   24 bytes w,  103.632pJ (CPU),   12 exe*/
   swap_f(v, left, (left+right)>>1); /*keine Division*/
/*   12 clk,    2 bytes I,   24 bytes r,    0 bytes w,   11.700pJ (CPU),   12 exe*/
   last = left;
/*  388 clk,   22 bytes I,  612 bytes r,    0 bytes w,  328.600pJ (CPU),   70 exe*/ /*[18]*/
   for(i=left+1; i<=right; i++)
/*  700 clk,   12 bytes I,  980 bytes r,  140 bytes w,  616.350pJ (CPU),   70 exe*/ /*[18]*/
      if (v[i] < v[left]) {
/*  576 clk,   16 bytes I,  768 bytes r,   96 bytes w,  493.584pJ (CPU),   48 exe*/ /*[18]*/
         swap_f(v, ++last, i);
      }
/*   96 clk,   10 bytes I,  120 bytes r,   24 bytes w,   82.332pJ (CPU),   12 exe*/
   swap_f(v, left, last);
/*  108 clk,   12 bytes I,  144 bytes r,   24 bytes w,   91.896pJ (CPU),   12 exe*/
   quicksort_o(v, left, last-1);
/*  120 clk,   12 bytes I,  144 bytes r,   24 bytes w,  102.132pJ (CPU),   12 exe*/
   quicksort_o(v, last+1, right);
/*  425 clk,   16 bytes I,  800 bytes r,    0 bytes w,  403.720pJ (CPU),   25 exe*/
}
/*****************************************************************************/
/*Total costs of the function:                                               */
/* 3220 clk,  140 bytes I, 4360 bytes r,  682 bytes w, 2848.896pJ (CPU),   25 exe*/
/*   62.6%      56.0%          60.7%          67.1%          64.4%           */
/*RAM (0x200  to 0x9ff ):   680 bytes r,  682 bytes w                        */
/*This function has been CALLed from     3 different locations.              */
/*****************************************************************************/
/*
 * quicksort_o is function of type void [13]
 * v is register function-parameter of type pointer to integer ("int", 2 byte in R10) [14]
 * left is register function-parameter of type integer ("int", 2 byte in R8)
 * right is register function-parameter of type integer ("int", 2 byte in R6)
 * i is register variable of type integer ("int", 2 byte in R11)
 * last is register variable of type integer ("int", 2 byte in R5)
 */
/*****************************************************************************/
/*(user) register usage:                                                     */
/*R4 :     0 times read,          0 times written                            */
/*R5 :   146 times read,         97 times written                            */
/*R6 :   169 times read,        132 times written                            */
/*R7 :   168 times read,         61 times written                            */
/*R8 :   135 times read,         50 times written                            */
/*R9 :   254 times read,         97 times written                            */
/*R10:   253 times read,         50 times written                            */
/*R11:   320 times read,        107 times written                            */
/*R12:     0 times read,          0 times written (function parameter register)*/
/*R13:   158 times read,         96 times written (function parameter register)*/
/*R14:   146 times read,        109 times written (function parameter register)*/
/*R15:   519 times read,        342 times written (function parameter register)*/
/*sum:  2268 register reads,   1141 register writes                          */
/*****************************************************************************/
```

Listing 25: Ausgabe von `eprof` für `quicksort_o` aus Listing 24 (Fragment)

```
#define swap(type,vx,vy) {type temp=vx; vx=vy; vy=temp;}

void quicksort_m(int v[], int left, int right) {
    int i, last, tmp, v_left, *v_i;

    if (left >= right)
        return;
    tmp = (left+right)>>1;          /*keine Division*/
    swap(int, v[left], v[tmp]);     /*Makro*/
    last = left;
    v_left = v[left];               /*lokale Kopie*/
    v_i = v+left+1;                 /*lokaler Zeiger*/
    i=right-(left+1);
    do {
        if (*v_i < v_left) {
            ++last;
            swap(int, v[last], *v_i); /*Makro*/
        }
        v_i++;
    } while ((--i) >= 0);           /*optimierte Schleife*/
    swap(int, v[left], v[last]);    /*Makro*/
    quicksort_m(v, left, last-1);
    quicksort_m(v, last+1, right);
}
```

Listing 26: Quicksort - optimierte Version

Listing 25). Diese Schlussfolgerung kann aber auch aus dem dargestellten Cost Profile direkt gezogen werden: Der Funktionsaufruf von `swap_f` innerhalb der Schleife und die Kosten der Funktion selbest haben großen Anteil an den Gesamtkosten.

- Mit der Hinweisnummer `[18]` ist die enthaltene Schleife gekennzeichnet, die auch durch ihre Kosten als Ziel für Optimierungen auffällt. Aber wie kann diese Schleife effizienter gestaltet werden? Der Einsatz eines lokalen Zeigers und dessen Modifikation zum Zugriff auf die Vektorelemente kann die Adressberechnung vereinfachen. (Dies wird auch mit einem Hinweis bei der Detektion der Definition eines Vektors vermerkt. Die Definition ist außerhalb von Quicksort und nicht in Listing 25 dargestellt.) Der Vergleich `v[i] < v[left]` ist nun die entscheidende Stelle, dieses Prinzip umzusetzen. Auch ist zu erkennen, dass die Register R4 und R12 ungenutzt bleiben, sodass lokale Zeiger und lokale Kopien erfolgversprechend sind.

- Der Einsatz von lokalen Zeigern entkoppelt die Schleifenzählvariable `i` vom Zugriff auf den Vektor, da der Zeiger direkt auf das Vektorelement zeigt, ohne einen Offset `i` zu benötigen. Somit spielt der Wert von `i` und auch die Zählrichtung keine Rolle mehr. Das Dekrementieren bis Null ist vorteilhaft, da dann das Negative-Flag in der CPU ausgewertet werden kann. Da die Schleife mindestens einmal durchlaufen werden muss, ist `do-while` vorzuziehen.

Listing 26 zeigt eine modifizierte Version von Quicksort. In Tab. 18 wird diese mit der originalen verglichen. Der größte Gewinn wird durch Einsatz eines Makros statt einer Funktion für den Austausch zweier Vektorelemente erreicht, aber auch innerhalb der Schleife summieren sich kleine Verbesserungen. Auffällig ist, dass eine Routine wie Quicksort in der vorliegenden Form noch solch großes Potential zur Optimierung bietet. Weniger optimierte Routinen können ein deutlich höheres Potential besitzen.

Die Lesbarkeit der modifizierten Version ist gegenüber des Originals schlechter. Daher ist es eine gute Praxis, zuerst eine klar strukturierte Implementierung anzustreben,

Tabelle 18: Vergleich von `quicksort_o` mit `quicksort_m`

Metrik	`quicksort_o` + `swap_f`	`quicksort_m`
Anzahl der Takte	3220 + 1512=4832	3262 (68%)
Programmspeicher [Bytes]	140 + 26=166	156 (94%)
Bytes gelesen	4360 + 2304=6664	4724 (71%)
Bytes geschrieben	682 + 288=970	686 (71%)
abgeschätzte Energie [nJ]	2849 + 1231=4080	2822 (69%)

um daraus algorithmische Verbesserungen schnell erkennen zu können und erst danach Feinoptimierungen anzuwenden.

3.4.6.2. Optimierung einer Gleitkommabibliothek

Motivation Generell sollte Gleitpunktarithmetik vermieden werden, doch es existieren Probleme, die einen hohen Dynamikumfang der Variablen voraussetzen. Das Lösen algebraischer Gleichungssysteme ist z. B. ein solches Problem.

Ausgangspunkt für dieses Beispiel ist die Gleitkommabibliothek des MSP430-GCC [59], welche einer allgemein von Prozessoren unabhängigen Bibliothek entspricht und Gleitkommaoperationen in Software realisiert. Für die Effizienz der Bibliothek hat zwar der architekturunabhängige Charakter auch eine Bedeutung, aber wesentlichen Einfluss hat auch die Wahl der zugrunde liegenden Datenstruktur, wie im Folgenden gezeigt wird.

Die Optimierung dieser Bibliothek wird in hohem Maße durch `eprof` vereinfacht. Nach dem Beispiel Quicksort aus Kap. 3.4.6.1 sollen nun weniger die Details des Profilings, sondern die daraus abzuleitenden Methoden im Fokus stehen.

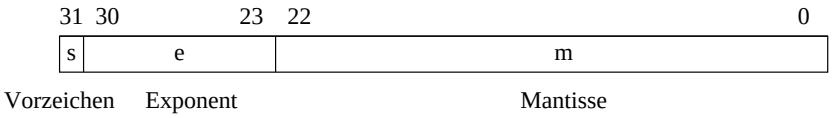

Abbildung 42: 32-Bit-Gleitkommaformat (IEEE-754 single precision [75])

Problemüberblick Gleitkommazahlen werden in einem Container transportiert, wie in Abb. 42 gezeigt. Sofern nicht spezielle Werte wie „Unendlich“ dargestellt werden, ergibt sich der Wert der Zahl f wie folgt:

$$f = (-1)^s \cdot 2^{(e-127)} \cdot (1{,}m) \tag{26}$$

In der Mantisse ist eine führende 1 nicht enthalten. Diese und das Komma werden implizit ergänzt. Der Exponent ist mit einem Offset gespeichert und muss korrigiert werden.

```
typedef struct {
        char fp_class;
        unsigned char sign;
        short normal_exp;
        unsigned long fraction;
} fp_number_type;
```

Listing 27: Struktur für Gleitkommazahlen (vereinfacht)

Mit einem solchen Container sind Transport und Speicherung von Gleitkommawerten in Rechnern effizient möglich. Für Rechnungen müssen allerdings die einzelnen Teile extrahiert werden. Folglich sind drei Hauptschritte für Rechnungen mit Gleitkommazahlen nötig: (1) Auspacken, (2) Ausführen der Rechnung und (3) Einpacken des Ergebnisses.

Aus- und Einpacken sind aufwändige Operationen. Es sind nicht nur viele Bitmaskierungen und Schiebeoperationen nötig, die meist nicht sehr effizient in Software realisierbar sind, sondern es müssen auch auch verschiedene Ausnahmen behandelt werden, die durch die Darstellung von Werten wie „Unendlich“ oder „Not a Number“ (NaN) entstehen. Aufgrund der Komplexität erscheint es auf den ersten Blick sinnvoll, diese beiden Operationen in separaten Funktionen zu realisieren.

Liegen die Daten in ausgepackter Form vor, belegen die einzelnen Komponenten jeweils Speicherplätze, deren Wortbreite durch die üblichen Größen in Rechnern bestimmt ist. Für Gleitkommazahlen, wie aus Abb. 42, sind für die Mantisse 32 Bit nötig, da kein Überlauf bei Berechnungen entstehen darf und Rundungen mit geringem Fehler durchgeführt werden sollten. Für den Exponenten werden 16 Bit benötigt, um ebenfalls Überlauf zu vermeiden. Für das Vorzeichen muss mindestens der kleinste Datentyp verwendet werden und damit sind üblicherweise 8 Bit belegt. Bei den meisten Operationen sind zwei Operanden nötig, was weiteren Speicherplatz kostet. Das Speichern dieser ausgepackten Daten im RAM ist im Allgemeinen kein Problem, wesentlich problematischer ist aber die effiziente Nutzung von Registern, da deren Anzahl sehr begrenzt ist. Da auf die Daten im Laufe der Rechnung häufig zugegriffen wird, ist Registernutzung allerdings vorteilhaft. Zu einem Problem wird auch die Menge der Daten bei der effizienten Datenübergabe zwischen Funktionen.

Problematisch an der Gleitkommaarithmetik ist demzufolge die Verwaltung der anfallenden Daten. Beim MSP430 mit seinen 12 Benutzerregistern zu je 16 Bit Wortbreite, welche damit zwei ausgepackte Gleitkommaoperanden aufnehmen können, ist es demzufolge entscheidend, wie effizient die Register genutzt und Speicherzugriffe möglichst vermieden werden können. Dabei ist zu beachten, dass nicht nur für die Operanden, sondern auch für Zählvariablen und Zwischenergebnisse Register benötigt werden.

Die Datenstruktur Die originale Gleitkommabibliothek des MSP430-GCC setzt zur Aufbewahrung der Datenteile einer Gleitkommazahl konsequent auf die Nutzung von Strukturen. Eine stark vereinfachte Struktur dafür ist in Listing 27 angegeben. Da die originale Bibliothek auch die in Kap. 3.4.6.2 erwähnten drei Hauptschritte in separate Funktionen aufteilt, ist für die Verarbeitung der so ausgepackten Daten noch die Einteilung in Zahlenklassen (normale Zahl, Unendlich, NaN, ...) nötig.

Die Datenübergabe zwischen Funktionen ist damit effizient über Zeiger auf diese Strukturen möglich. Der wesentliche Nachteil ist aber, dass diese Strukturen im RAM liegen und nicht auf Register abgebildet werden. Der Lösungsansatz dafür ist der schon oft vorgeschlagene Einsatz von lokalen Kopien für die Komponenten. Im konkreten Fall führt das einfache Kopieren der Komponenten auf Registervariablen allerdings nur zu begrenztem Erfolg. Die Ursache ist die Datenmenge: Neben den Datenteilen der Gleitkommazahl ist noch deren Klasse und auch ein Zeiger auf die Struktur nötig. Bei zwei Operanden für beispielsweise die Addition reichen dann die Register des MSP430 nicht mehr aus, um alle lokalen Kopien auf Register abzubilden. Weiterhin verursacht das Kopieren der Daten in die Strukturen für die Parameterübergabe zwischen den Funktionen wesentliche Kosten.

Der nächste Lösungsansatz ist die Elimination der Strukturen und damit der Einsatz von verteilten Variablen für alle Strukturkomponenten. Die lokalen Kopien der Daten werden also direkt zum Aufbewahrungsort. Die Programmübersicht kann durch sinnvolle Benennung der Variablen erhalten bleiben. Mit diesem Ansatz werden zwar die Zeiger auf die Strukturen eliminiert und somit Speicherplätze in Registern gewonnen sowie Kopieraktionen vermieden, aber ein weit größeres Problem stellt die Datenübergabe von ausgepackten Daten zwischen den Funktionen dar.

Elimination redundanter Rechnungen Im ersten Ansatz wurden die drei Hauptschritte in separate Funktionen aufgeteilt, da Aus- und Einpacken aufwändig sind und es das Ziel ist, durch Wiederverwendung gleichartiger Routinen Programmspeicherplatz zu sparen. Das Separieren der Hauptschritte macht die Einordnung in eine Zahlenklasse beim Auspacken nötig, welche dann aber bei der Rechnung wieder zu evaluieren ist. Für das Ergebnis gilt das analoge beim Einpacken. Die notwendigen Verzweigungen durch die Behandlung der verschiedenen Zahlenklassen, sind somit redundant in allen drei Hauptschritten enthalten. Weiterhin offenbart die Kostenanalyse der Funktionsaufrufe (mittels `eprof`) für das Ein- und Auspacken, dass Funktionseintritt und -austritt selbst sehr kostenintensiv sind. Der Grund ist das Sichern von Registern (`PUSH` und `POP`). Konsequenterweise ist somit das Zusammenführen der drei Hauptschritte der nächste Optimierungsschritt (Abb. 43). Damit ist in jeder Funktion neben der Rechenoperation, wie Addition, Multiplikation oder Division auch das Aus- und Einpacken enthalten, aber redundante Verzweigungen werden vermieden. Zusätzlich muss die Einteilung in Zahlenklassen nicht mehr in einer Variablen abgespeichert werden, sondern ergibt sich indirekt aus der Programmverzweigung.

Weitere Eliminationen redundanter Rechnungen ergeben sich unter anderem, wenn bei der Behandlung der speziellen Werte frühzeitig Abbruchbedingungen entstehen. Beispielsweise führt die Addition einer normalen Zahl mit „Unendlich“ wieder zu „Unendlich“.

Feinkörnige Optimierungen Bei Gleitkommaoperationen sind im Speziellen beim Aus- und Einpacken Bitmaskierungen und Verschiebungen nötig. Auf Variablen des Types `float` sind solche Operationen nicht definiert. Ein effizienter Zugriff ist durch den Einsatz

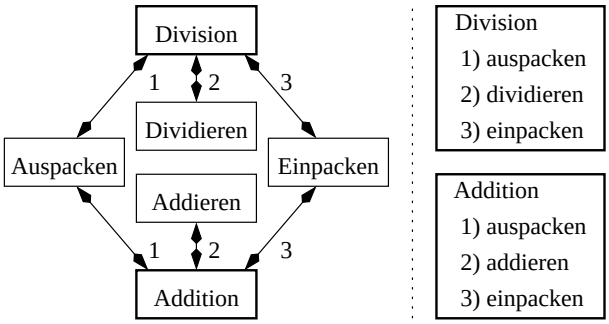

Abbildung 43: Verteilte Funktionen oder Zusammenfassung der Hauptschritte

von einer Union, bestehend aus einer `float` und einer 32-Bit-Intergervariablen (`long`) möglich, wie schon in Listing 7 auf S. 66 gezeigt. Der MSP430-GCC betrachtet diese Union als Registervariable, was sehr vorteilhaft ist. Kompliziertere Unionen können evtl. nicht auf Register abgebildet werden und sind zu vermeiden.

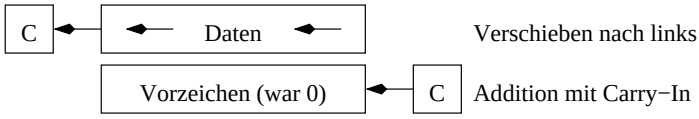

Abbildung 44: Verschieben des Vorzeichens über das Carry-Flag

Nicht alle Verschiebeoperationen sind mit ANSI C effizient beschreibbar. Beim Auspacken des Vorzeichens ist dies am offensichtlichsten: Die gänstigste Lösung ist das Verschieben der Daten des Containers aus Abb. 42 um ein Bit nach links und damit das Verschieben des Vorzeichens in das Carry-Flag. Dessen Inhalt kann dann auf die Variable für das Vorzeichen verschoben werden (Abb. 44). Hierfür ist zwingend Assemblercode notwendig. In ANSI C wäre dagegen eine Verschiebung nach rechts um 31 Bit nötig, welche beim MSP430 sehr aufwändig ist.

Bei der Multiplikation von Gleitkommawerten müssen die Mantissen multipliziert werden, wobei eine $32 \cdot 32 \rightarrow$ 64-Bit-Multiplikation entsteht. Die Mantissen der Operanden werden nach der Multiplikation nicht mehr benötigt. Das kann ein Compiler zwar erkennen, aber die Register, in denen die Mantissen gespeichert waren, können vom Compiler nur für das Ergebnis wiederverwendet werden, wenn sie in einem Block hintereinander liegen (z. B. R4 bis R7). Das ist im Allgemeinen nicht der Fall und nicht immer ist der Compiler in der Lage, die Registernutzung darauf zu optimieren. Mittels Assemblerinstruktionen stellt es aber kein Problem dar, beliebige Register zu nutzen. Innerhalb der Hochsprache können dann die Variablen für die Mantissen für die zwei Teile des Ergebnisses weiterverwendet werden (Abb. 45).

Das Komma der Mantissen und des Ergebnisses muss bei der Multiplikation beachtet werden. Es ist vorteilhaft, die Mantissen vor der Multiplikation so zu verschieben, dass die Ergebnismantisse ohne weitere Verschiebungen in den oberen 32 Bits des Multiplikationsergebnisses entsteht. Die unteren 32 Bits sind dann nur für das Runden relevant,

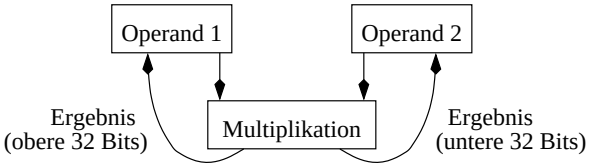

Abbildung 45: Wiederverwenden der Register mittels Assemblercode

was im nächsten Abschnitt diskutiert wird.

Irrelevanzreduktion Irrelevanzreduktion geht stets einher mit Informationsverlust. Für Gleitkommaoperationen bedeutet dies eine (geringe) Reduktion der Genauigkeit zugunsten der einfacheren Verarbeitung. Es hängt vom konkreten Anwendungsfall ab, ob dies akzeptabel ist oder nicht.

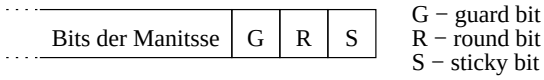

Abbildung 46: Zusätzliche Bits zum Runden [76]

Runden ist bei Gleitkommaoperationen mit mehreren Alternativen standardisiert [75]. Es ist empfohlen, standardmäßig „Round to Nearest“ einzusetzen, bei dem nach dem Runden die jeweils nächste im Gleitkommaformat darstellbare Zahl gewählt werden soll oder wenn zwei Zahlen vom ungerundeten Ergebnis gleich weit entfernt sind, die Zahl, deren niederwertigstes Bit Null ist. Um dies zu realisieren, sind mindestens 3 zusätzliche Bits nötig (Abb. 46). Die Behandlung der zusätzlichen Bits ist zusammen mit der Mantisse möglich, da die effektiv 24 Bits der Mantisse in einem 32 Bit Datentyp gespeichert sind und somit die zusätzlichen Bits mit aufgenommen werden können. Aufwändig sind allerdings notwendige Verschiebungen, um die zusätzlichen Bits einzufügen und zu entfernen, sowie natürlich die Behandlung des Rundens selbst. Der Standard beschreibt auch die einfachste Form „Round to Zero“, was nichts weiter als das Abschneiden überflüssiger Bits ist. Die Originalbibliothek des MSP430-GCC nutzt „Round to Nearest“ und setzt nicht nur ein „Round Bit (R)“, sondern mehrere ein, um den Rundungsfehler zu minimieren. Dies ist allerdings auch kostenintensiv.

Neben normalen Zahlen können auch betragsmäßig sehr kleine Zahlen („Denormalized Numbers“) mit dem Gleitpunktformat dargestellt werden. Deren Behandlung führt zu aufwändigen Schiebeoperationen und Prüfungen. Auch wenn diese Zahlen während der Rechnung nicht auftreten, so muss zumindest ein Test durchgeführt werden, dass es sich nicht um eine solche Zahl handelt, was ebenfalls Rechenzeit kostet. Treten solche Zahlen auf und werden sie dagegen in einer Bibliothek nicht behandelt, so wird deren Wert als Null interpretiert. Die Originalbibliothek des MSP430-GCC unterstützt „Denormalized Numbers“.

Für spezielle Anwendungsfälle kann auch ein 16-Bit-Gleitkommaformat ausreichend sein, wie es mit OpenEXR [77] existiert.

Ergebnisse Zum Vergleich der beschriebenen Optimierungen wurde eine FFT über 1024 reelle ganzzahlige Werte unter Nutzung des FFT-Algorithmus `realft` aus [66] durchgeführt. In Tab. 19 sind die Ergebnisse angegeben.

Tabelle 19: Vergleich von Gleitkommabibliotheken bei einer FFT

Variante	Multiplikation	Denormalized Numbers	Programmgröße (Bytes)	Taktanzahl
Original	Software	ja	6416	41340077 (100%)
optimiert	Software	nein	5562	18925204 (46%)
optimiert	Hardware	ja	5548	8278076 (20%)
optimiert	Hardware	nein	5470	7959089 (19%)

Durch die Originalbibliothek des MSP430-GCC wird ein Hardwaremultiplizierer nicht verwendet, was einen großen Nachteil darstellt, sofern dieser vorhanden ist. Wesentliche Einsparungen konnten aber auch durch die beschriebenen Änderungen der Datenstruktur und des Programmflusses erreicht werden. Dabei ist hervorzuheben, dass die Programmgröße der optimierten Version geringer als die der Originalversion ist, obwohl das Aus- und Einpacken in jeder Operation extra realisiert wurde, was auf die beschriebene Redundanzelimination der Programmverzweigungen zurückzuführen ist.

Assemblercode ist in der optimierten Version zu kleinen Teilen enthalten, wurde aber in parametrisierter Form realisiert. Der Einsatz von Makros dafür führt zu einem übersichtlichem Quellcode. Die große Masse des Quellcodes ist aber in der Hochsprache ANSI C realisiert, was besonders betont werden soll. Damit ist gezeigt, dass auch mittels Hochsprachen eine starke Optimierung möglich ist, sodass Wartung und weitere Anpassung des Quellcodes von der Abstraktion von Hochsprachen profitieren.

Die wesentlichen Werkzeuge zur Lokalisierung der hauptsächlichen Kosten sind das Cost Profile und zur Analyse der Variablen, also dem Aufdecken der Ursachen, das Variable / Type Profile des Profilers `eprof`. Die Optimierungen mit dem größten Einfluss sind mit der Hilfe von `eprof` ohne Assemblerkenntnisse zu erkennen und durchzuführen. Für feinkörnige Optimierungen auf Assemblerniveau ist die Zuordnung von Maschinenintruktionen zu Hochsprachenbefehlen (Kap. 3.4.3.4) von hohem Wert.

Es soll nicht verschwiegen werden, dass Fixpunktarithmetik statt Gleitpunktarithmetik zu noch weit effizienteren Programmen führen kann. Ziel dieses Beispiels war es aber, das Potential von durch erweiterte Programmanalyse gestützter Optimierung zu zeigen.

3.5. Ereignisgesteuerte Software

3.5.1. Ein Sensorsystem

Anhand eines abstrakten Sensorsystems (Abb. 47) soll der Einsatz von ereignisgesteuerter Software diskutiert werden. Das System bestehe aus einem Sensor, der CPU und einem Bussystem zur Kommunikation mit einer übergeordneten Verwaltungsinstanz (einem Steuerrechner). Es soll in regelmäßigen Abständen ein Messwert aufgenommen werden. Mit den Messwerten wird eine Signalkonditionierung und -verarbeitung durch-

geführt. Dem Steuerrechner können Messdaten oder extrahierte Informationen (z. B. „Grenzwert überschritten“) gemeldet werden und der Steuerrechner kann Befehle erteilen (z. B. Änderung des Messintervalls).

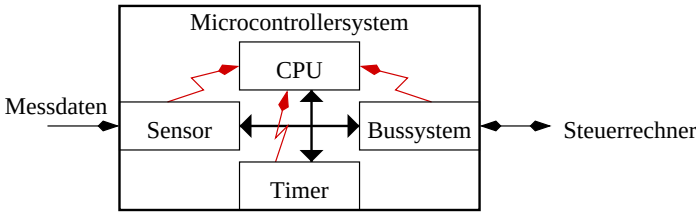

Abbildung 47: Beispiel eines Microcontrollersystems

Durch den skizzierten Aufbau existieren drei verschiedene Ereignisquellen für die CPU: das Messintervall, das Beenden der Messung und die Kommunikation über den Bus. Am effizientesten werden diese Ereignisse mittels Interrupt signalisiert und somit müssen drei Interrupt Service Routinen (ISR) existieren (Abb. 48).

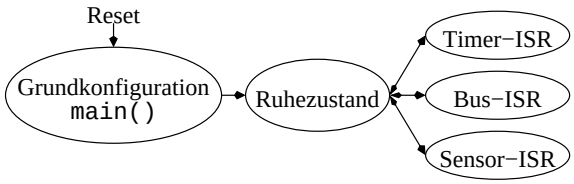

Abbildung 48: Softwareroutinen des Beispielsystems

Die wesentliche Funktionalität wird innerhalb der ISRs realisiert. Damit existiert kein zentraler Programmkörper, der sich in verschiedene Teile verzweigt, sondern es existieren nur die spezialisierten Programmteile in den jeweiligen ISRs. Der Programmablauf wird demzufolge nach dem folgenden Schema organisiert:

1. Konfiguration der Funktion und Start einer Komponente mit einer Aufgabe
2. Versetzen der CPU in einen Ruhezustand oder Verarbeitung anderer Aufgaben
3. Aufwachen oder Unterbrechung anderer Aufgaben durch Interrupt

Die Signalisierung mittels Interrupt nach dem Eintreten eines Ereignisses steuert den Programmfluss. Warteschleifen sind unnötig und zu vermeiden. Als logische Konsequenz verbleibt in ANSI C in der Funktion `main` nur Grundkonfiguration des Systems und Aktivierung des Ruhezustandes, welcher nur durch Interrupts verlassen wird.

Wird als Beispiel vom Steuerrechner der Befehl zum Starten einer kontinuierlichen Messung mit dem Sensor empfangen, so muss zur Ausführung dieses Befehls in der Sensorkomponente eine Messung ausgelöst werden. Für eine kontinuierliche, zeitlich äquidistante Messung ist weiterhin ein Timer nötig, der beim Erreichen des Messintervalls dies über einen Interrupt signalisiert, sodass eine weitere Messung ausgelöst werden kann. Die

Sensorkomponente signalisiert durch einen Interrupt, wann ein neuer Messwert zum Abholen bereitsteht. In dieser Sensor-ISR wird die Kernfunktionalität des Sensorsystems, also die Datenverarbeitung, realisiert. Dies umfasst z. B. Messwertkorrektur, Interpolation, Plausibilitätstest, Frequenzanalyse und das Auslösen einer Sendung von Ergebnissen über das Bussystem.

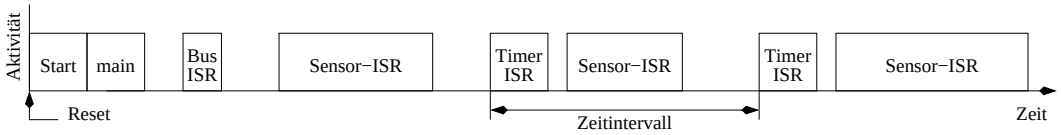

Abbildung 49: Beispielsensorsystem mit Interrupt-Schema #1

Der skizzierte Programmablauf ist in Abb. 49 illustriert. Es wird darauf hingewiesen, dass die Sensor-ISR abhängig von den Daten unterschiedlich lang ist. Wird beispielsweise ein unplausibler Messwert entdeckt, sind geeignete Maßnahmen erforderlich. In Abb. 49 wurde angenommen, dass jede ISR beendet werden kann, bevor ein neuer Interrupt ausgelöst wird. Ist dies nicht der Fall, so muss ein Interrupt eine laufende andere ISR unterbrechen können. Ist beispielsweise die Signalverarbeitung in der Sensor-ISR ausnahmsweise sehr komplex und kann nicht vor dem Erreichen des Zeitintervalls beendet werden, so sollte die Timer-ISR die Sensor-ISR unterbrechen, um damit die nächste Messung zu starten. Die Daten der neuen Messung können abgeholt werden, sobald die alte Rechnung beendet wird und der Sensor die Messung vollzogen hat (Abb. 50). Es muss selbstverständlich vom System sichergestellt werden, dass es nie zu einem überlauf kommt, bzw. es müssen geeignete Schutzmechanismen bereitstehen. Ebenso muss sorgfältig definiert werden, ob und wann eine ISR unterbrochen werden darf.

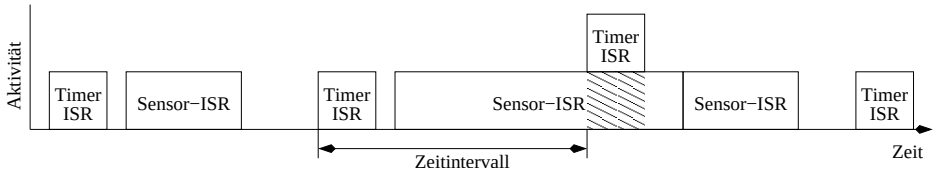

Abbildung 50: Beispielsensorsystem mit Interrupt-Schema #2

3.5.2. Diskussion

In ereignisgesteuerten Programmen ist die Funktionalität eines Systems auf verschiedene ISRs verteilt. Die ISRs sind keine Funktionen, die sich gegenseitig aufrufen, also kann kein direkter Datenaustausch zwischen ihnen stattfinden. Die Lösung des Problems besteht in der Nutzung extern deklarierter Variablen. Muss eine ISR eine Statusinformation speichern, so sind intern `static` deklarierte Variablen sinnvoll. Da beide Arten von Variablen im RAM liegen und somit der Zugriff darauf hohe Kosten verursacht, sollte die Datenmenge bzw. die Anzahl der Zugriffe darauf reduziert werden. Können ISRs

unterbrochen werden, ist sicherzustellen, dass gültige Daten in externen Variablen vorliegen, wenn diese wieder gelesen werden. Eventuell ist dafür ein Statuswort als zusätzliche externe Variable sinnvoll.

Timer stellen ein wesentliches Mittel zur Realisierung ereignisgesteuerter Software dar. Kann eine Komponente selbst keinen Interrupt auslösen, ist der Einsatz von Timern als Alternative meist möglich und deutlich effizienter als Warteschleifen.

Durch das Warten auf Ereignisse und die Reaktion darauf kann eine CPU hochgradig flexibel und datenabhängig zwischen aktivem Zustand und Ruhezustand wechseln, sodass unnötige CPU-Aktivität nahezu vollständig vermieden werden kann. Dennoch existieren Nachteile:

- Teilen sich mehrere Komponenten einen Interrupt, ist (evtl. rechenzeitintensiv) die Quelle zu bestimmen. Eine genau definierte Startzeit für die eigentliche Subroutine kann nicht immer einfach angegeben werden.
- Können ISRs unterbrochen werden, ist genau zu definieren, wann dies geschehen darf und wie Fehlerfälle behandelt werden.
- Durch mehrere Interruptquellen, die zu beliebigen Zeitpunkten einen Interrupt auslösen können, entstehen nicht oder schwer vorhersehbare Programmabläufe.

Die Hauptnachteile liegen also in der Schwierigkeit begründet, alle Eventualitäten auch bei komplizierten Programmabläufen zu erkennen und dagegen robuste Routinen zu entwerfen. Diese Robustheit kostet Rechenzeit.

3.5.3. Die optimale Taktfrequenz

Mit welcher Taktfrequenz sollte ein solches ereignisgesteuertes System betrieben werden? Dies hängt maßgeblich von den Möglichkeiten des Systems, aber auch von der Kenntnis über das System ab.

Jede Baugruppe sollte unabhängig von anderen Baugruppen mit möglichst geringem Takt betrieben werden. Die Bereitstellung unabhängiger Takte ist aber nicht für jede Baugruppe möglich, da der dafür nötige Aufwand zu groß ist. Eine Kopplung schränkt dagegen die Möglichkeiten zur Taktvariation ein. Individuelle Taktteiler schaffen nur bedingt Abhilfe.

3.5.3.1. Die Energie als Hauptkriterium In einem ereignisgesteuerten System setzt sich die Verlustenergie aus Verlusten im aktiven und inaktiven Zustand zusammen.

$$\begin{aligned} W &= W^{aktiv} + W^{inaktiv} \\ &= \left(W^{aktiv}_{stat} + W^{aktiv}_{dyn} + W^{aktiv}_{osc}\right) + \\ &\quad \left(W^{inaktiv}_{stat} + W^{inaktiv}_{dyn} + W^{inaktiv}_{osc} + W^{wechsel}\right) \end{aligned} \tag{27}$$

Die statische Verlustenergie W_{stat} ist abhängig von der Spannung. Die dynamischen Verlustenergie im aktiven Zustand W^{aktiv}_{dyn} ist allein abhängig von der Spannung, da als

Randbedingung für jede Rechenoperation eine bestimmte, unveränderbare Anzahl von Taktschritten benötigt wird. Dagegen kann die Anzahl der Taktschritte im inaktiven Zustand durch die Reduzierung der Frequenz verkleinert werden. Weiterhin sind die Verluste des Oszillators W_{osc} frequenzabhängig. Die Verluste beim Wechsel zwischen verschiedenen Taktfrequenzen und Spannungen $W^{wechsel}$ werden als konstant angenommen.

$$\begin{aligned} W &= \left(W_{stat}^{aktiv}(U) + W_{dyn}^{aktiv}(U) + W_{osc}^{aktiv}(f)\right) + \\ &\quad \left(W_{stat}^{inaktiv}(U) + W_{dyn}^{inaktiv}(U,f) + W_{osc}^{inaktiv}(f) + W^{wechsel}\right) \end{aligned} \tag{28}$$

In der Form von (28) ist das Problem außerordentlich komplex. Dabei ist weniger entscheidend, dass das Problem zweidimensional ist, sondern der Fakt, dass die einzelnen Teilverluste nur sehr schwer zu quantisieren sind. Abschätzungen und Näherungen können das Problem im Einzelfall vereinfachen:

- Typischerweise werden für Microcontroller keine Fertigungsprozesse verwendet, die hohe statische Verluste nach sich ziehen, da Microcontroller oft sehr lange autonom arbeiten müssen: $W_{stat}^{aktiv} \approx W_{stat}^{inaktiv} \approx 0$.
- Um einen wirkungsvollen Ruhezustand für die CPU wie auch für Peripheriekomponenten zu ermöglichen, sollte der Takt für die CPU bzw. die Komponenten abschaltbar sein. Damit wird $W_{dyn}^{inaktiv} \approx 0$. Lediglich durch andere, noch aktive Baugruppen werden die Eingänge von abgeschalteten Baugruppen umgeladen.
- Sind die Verluste im Oszillator sehr schwach von dessen Frequenz abhängig, können sie in Näherung als konstant angenommen werden. Bei Systemen mit sehr langen Ruheperioden kann $W_{osc}^{inaktiv}$ bestimmend werden und eine Oszillatorabschaltung notwendig machen.
- In vielen Fällen ist eine Regelung der Versorgungsspannung nicht möglich, weil dies technisch nicht vorgesehen oder die Umschaltung zu zeitaufwändig ist.
- Im Allgemeinen stellt das Problem eine zweidimensionale Extremwertaufgabe dar. Häufig sind allerdings nur wenige diskrete Werte für Frequenz und Spannung wählbar. Insbesondere wenn noch andere Vereinfachungen gemacht werden können, reduziert sich die Anzahl der Möglichkeiten schnell.

In Tab. 20 werden die Möglichkeiten zur Regelung von Taktfrequenz und Spannung unter verschiedenen Randbedingungen beleuchtet. Ein Stern bezeichnet alle Varianten und Tab. 21 gibt eine übersicht über die darin verwendeten Bezeichner. Zugrunde liegt ein System, welches einen stabilen Oszillator (z. B. einen externen Quarz) und einen internen, instabilen, aber schnellen und regelbaren Oszillator besitzt, welcher über eine FLL an den stabilen Oszillator gekoppelt ist. Es wird $W_{osc}^{inaktiv}$ als gering angenommen (die Aktivität des Systems sei hoch).

Hervorzuheben ist sicher die dritte Zeile in Tab. 20: der Betrieb mit konstanter Betriebsspannung, nicht abschaltbarem Quarzoszillator und die Möglichkeit zur Abschaltung der CPU sowie der Komponenten im Ruhezustand (Glock-Gating). Bei diesem

Tabelle 20: Die optimale Taktfrequenz

U_{dd} aus?	U_{dd} var.?	Quarz aus?	W'_{osc} (int.)	Clock-Gating?	W^w	Regelung
nein	nein	nein	*	nein	gering	min. int. Takt
nein	nein	nein	klein	nein	groß	abhängig von $W_{dyn}^{inaktiv}$ vs. W^w
nein	nein	nein	klein	ja	*	nein
nein	nein	nein	groß	nein	groß	min. int. Takt
nein	nein	nein	groß	ja	gering	min. int. Takt
nein	nein	nein	groß	ja	groß	abhängig von W_{osc} vs. W^w
nein	nein	ja	*	*	*	Quarz aus + Regelung, wie oben
nein	ja	nein	*	*	*	nach (28) - meist min. int. Takt
nein	ja	ja	*	*	*	Quarz aus + Regelung, wie oben
ja	*	*	*	*	*	U_{dd} aus + Regelung wie oben

Tabelle 21: Bezeichner in Tab. 20

Bezeichner	Beschreibung
U_{dd} aus?	Ist die Betriebsspannung abschaltbar?
U_{dd} var.?	Ist die Betriebsspannung variabel? (DVS)
Quarz aus?	Ist der Quarzoszillator abschaltbar? (Ist es technisch und funktional möglich?)
W'_{osc}	$W'_{osc} = W'_{osc}(f) = \frac{\delta W_{osc}(f)}{\delta f}$; Abhängigkeit der Verlustenergie im internen, schnellen Oszillator von der Frequenz
Clock-Gating?	Können Baugruppen (insbesondere die CPU) im Ruhezustand abgeschaltet werden?
W^w	$W^w = W^{wechsel}$; Energie für die Umschaltung - abhängig von deren Komplexität (Taktumschaltung ist einfacher, als Abschaltung von U_{dd})

sehr häufig bei Microcontrollern eintretenden Fall ist eine Taktregelung unsinnig, da der Aufwand für die Regelung die minimalen Vorteile aufwiegt. Nur unscharf charakterisiert sind die letzten vier Zeilen in Tab. 20, da dann viele Randbedingungen zu beachten sind und die einzelnen Teilverluste quantisierbar sein müssen.

3.5.3.2. Die Leistung als Hauptkriterium Neben der Verlustenergie W kann noch die Verlustleistung P eine entscheidende Randbedingung sein. Eine Reduktion von P_{dyn}^{aktiv} ist durch Reduktion der Taktfrequenz möglich. Davon bleibt W_{dyn}^{aktiv} unberührt. Für die Taktumschaltung wird zusätzliche Energie benötigt, aber die Umschaltung kann den Betrieb des Systems erst ermöglichen (Abb. 51). Ist (zusätzlich) eine Spannungsreduktion vorgesehen, so kann sowohl Verlustenergie als auch Verlustleistung reduziert werden.

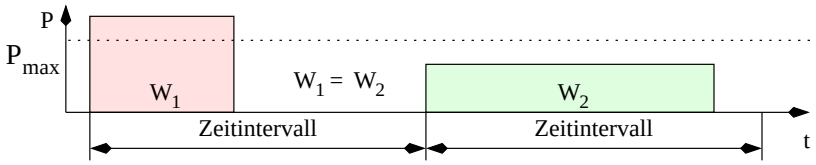

Abbildung 51: Reduktion der Verlustleistung durch Taktfrequenzreduktion

3.5.3.3. Funktionale Anforderungen Wird kurzzeitig eine erhöhte Rechenleistung benötigt, wie z. B. in Abb. 50, und soll aber vermieden werden, den Microcontroller dauerhaft mit der erhöhten Taktfrequenz zu betreiben (weil z. B. dafür eine erhöhte Spannung nötig ist, P_{osc} zu groß wird oder $W_{dyn}^{inaktiv} \gg 0$), so ist eine Regelung der Taktfrequenz unumgänglich. Bei Einsatz eines Betriebssystems kann über ein API durch Funktionen der Bedarf an höherer Taktfrequenz signalisiert werden. Mehrere unabhängige Funktionen signalisieren ihren individuellen Bedarf und das Betriebssystem regelt dann Takt und Versorgungsspannung abhängig von der höchsten Anforderung [78]. Auch für interruptgesteuerte Programme von Microcontrollern, die ohne Betriebssystem arbeiten, kann das Analoge mit Hilfe einer Funktion realisiert werden, welche von verschiedenen Routinen (verschiedenen ISRs) aufgerufen wird. Die angeforderten Werte für die Taktfrequenz können in externen Variablen dauerhaft abgespeichert werden, sodass stets die höchste Anforderung bekannt ist. Schematisch wird das Prinzip in Abb. 52 illustriert.

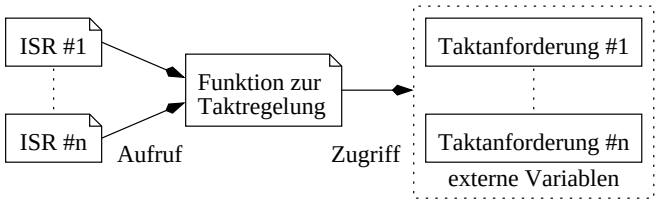

Abbildung 52: Taktanforderung bei ereignisgesteuerter Software

So einfach, wie das Prinzip der Taktumschaltung nach Anforderung erscheint, so problematisch kann die Umsetzung im Einzelfall werden, wenn die Takte verschiedener Kom-

ponenten nicht unabhängig voneinander sind, wie es bei Microcontrollern sehr häufig der Fall ist. Komponenten zur Buskommunikation, ADCs / DACs und Timer erlauben nur eine Taktumschaltung, wenn sie gerade nicht arbeiten. Wenn schon die Umschaltung nicht generell verhindert wird, so kann zumindest eine sofortige Umschaltung unmöglich sein. Außerdem kosten die Regelung und das Abschalten von Oszillatoren oder der Betriebsspannung Zeit für Umschaltung, Reaktivierung bzw. Neustart (Booten). Diese Zeit muss im System zur Verfügung stehen, was bei schnell reagierenden Systemen oder kurzen Deadlines nicht der Fall sein kann.

3.5.3.4. Die Wahl der Taktfrequenz Ist eine Taktfrequenzumschaltung möglich und sinnvoll, so verbleibt die Frage nach der jeweils notwendigen Frequenz. Nimmt ein System x Messwerte auf und führt danach eine aufwändige Berechnung, wie z. B. eine FFT, durch, so sollte eine niedrige Frequenz während der Messwertaufnahme und eine hohe zur Berechnung eingesetzt werden. Die jeweils notwendige Frequenz kann beim Entwurf bestimmt werden. In einem Echtzeitsystem, welches überlauf verbietet, muss die Auswahl natürlich für den Worst Case erfolgen und ist damit für eine Vielzahl von Fällen überdimensioniert, wenn aufgrund der tatsächlich auftretenden Daten eine schnellere Abarbeitung als im Worst Case möglich ist. Das Maß an überdimensionierung kann reduziert werden, indem nicht nur zu Beginn einer Routine eine Frequenzanpassung vorgenommen wird, sondern auch mehrfach innerhalb der Routine, wobei dann der jeweilige Worst Case weniger stark von der tatsächlich benötigten Ausführungszeit abweichen wird. Im Beispiel in Abb. 53 werden zum Start der Routine sowie beim Aufruf zweier Subroutinen die für den jeweiligen Worst Case dimensionierten Frequenzen f_1, f_2 und f_3 eingestellt. Das tatsächliche Ende wird vor der Deadline erreicht.

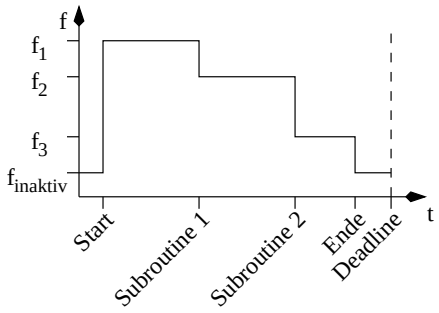

Abbildung 53: Taktumschaltung in einem Echtzeitsystem

Kann ein System auch autonom die gänstigste Taktfrequenz auswählen, ohne dass beim Entwurf konkrete Vorgaben gemacht werden? Nur in einem zeitunkritischen System, in welchem keine Deadline existiert bzw. für jede Aufgabe weit mehr Zeit als tatsächlich benötigt zur Verfügung steht, kann diese Frage in der Form überhaupt gestellt werden. Existiert eine Deadline, muss zumindest eine zum Einhalten der Deadline minimale Taktfrequenz verwendet werden.

Kann es aber sinnvoll sein, eine Berechnung so schnell wie möglich auszuführen, um eine möglichst lange Zeit im Ruhezustand zu verharren oder sollte die Rechnung besser

so langsam wie möglich ausgeführt werden? Seien T^{aktiv} und $T^{inaktiv}$ die Zeitdauer in dem jeweiligen Zustand, dann gelten folgende Zusammenhänge:

$$\begin{aligned} f^{inaktiv} &= const. \\ U_{dd}^{inaktiv} &= const. \\ T^{inaktiv} &\sim W^{inaktiv} \\ f^{aktiv} &\sim 1/T^{aktiv} \sim T^{inaktiv} \\ f &\sim U_{dd} \quad | \text{(mindestens linear [55])} \\ W_{dyn}^{aktiv} &\sim (U_{dd})^2 \\ P_{stat}^{aktiv} &\sim e^{(U_{dd})} \\ P_{stat}^{aktiv} &\sim T^{aktiv} \end{aligned} \tag{29}$$

Mit höherer Taktfrequenz f^{aktiv} und dadurch bedingte höhere Spannung U_{dd}^{aktiv} steigt W_{dyn}^{aktiv} mehr als quadratisch und P_{stat}^{aktiv} exponentiell, während durch die kürzere Dauer des aktiven Zustandes W_{stat}^{aktiv} lediglich linear reduziert wird aber $W^{inaktiv}$ linear steigt. Somit ist eine Problembearbeitung mit geringstmöglicher Taktfrequenz günstiger als eine maximal schnelle Bearbeitung.

Bisher wurde neben dem aktiven Zustand nur von einem Ruhezustand ausgegangen. Besitzt ein System mehrere unterschiedlich tiefe Ruhezustände, werden die Zusammenhänge komplizierter. Ist es z. B. möglich, aufgrund von längerer $T^{inaktiv}$ in einen tieferen, leistungsärmeren Ruhezustand zu wechseln, so könnte eine erhöhte f^{aktiv} sinnvoll sein. Dies ist allerdings selten, da die Länge von $T^{inaktiv}$ gerade im Grenzbereich liegen muss, in dem eine solche Entscheidung überhaupt möglich ist.

Neben einerseits harten Echtzeitanforderungen und andererseits zeitunkritischen Systemen existieren noch Systeme mit „weichen“ Echtzeitanforderungen. Solche Systeme haben keine feste Deadline und können daher ihre Rechenaufgaben beliebig lange ausdehnen. Sollten zu viele neue Aufgaben anfallen, kommt es zu einer Art Stau und die Aufgaben werden verzögert abgearbeitet. Eine solche überlastsituation kann insbesondere bei Interaktion mit einem Benutzer sehr störend sein, sodass eine automatische Erhöhung der Taktfrequenz in diesem Fall zu Lasten der Verlustenergie wünschenswert ist. Dieses Verhalten ist sehr untypisch für Microcontrollersysteme, aber entspricht genau dem Arbeitsprofil eines Personal Computers. In PCs wird daher (sofern möglich) der Takt des Prozessors vom Betriebssystem abhängig von der Last geregelt. Da Verlustenergie dennoch reduziert werden soll, ist eine schrittweise Anhebung der Taktfrequenz sinnvoll. Kleine Rechenaufgaben können somit beendet werden, noch bevor die Taktfrequenz weiter angehoben wird, während bei länger dauernden Berechnungen in mehreren Schritten schlussendlich der Maximaltakt erreicht wird. In interruptgesteuerten Systemen kann ein ähnliches Verhalten mittels Timer realisiert werden. Dies wird in Abb. 54 an einem Beispiel illustriert. Beim Start einer Berechnung wird der Timer gestartet. Er unterbricht die Rechnung und daraufhin wird die nächsthöhere Taktfrequenz eingestellt. Der Timer wird beim Ende der Berechnung oder beim Erreichen der Maximalfrequenz deaktiviert. Beim Beenden der Routine wird die Taktfrequenz auf das Minimum reduziert.

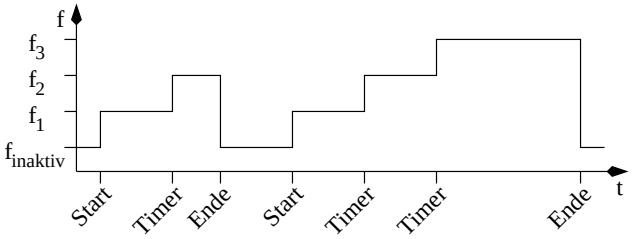

Abbildung 54: Taktumschaltung bei „weichen“ Echtzeitanforderungen

3.5.3.5. Praktische Dimensionierung Anhand von Abschätzungen können grundlegende Aussagen über die Auswahl der geeigneten Taktfrequenz getroffen werden, wie in den vergangenen Kapiteln gezeigt wurde. Der Profiler `eprof` erlaubt die Optimierung von Software und auch die Bestimmung der für den Worst Case nötigen Anzahl von Takten für eine Routine. Damit ist die minimal notwendige Taktfrequenz bekannt. Ohne die Verlustenergie tatsächlich abzuschätzen, kann mit dieser Information die Verlustreduktion für die CPU eines Systems für eine Vielzahl der diskutieren Fälle durchgeführt werden.

Können die Relationen zwischen den verschiedenen Teilverlusten aus (28) grob abgeschätzt werden, so lassen sich auch daraus Schlussfolgerungen ziehen. Sei ein System gegeben, das für eine Berechnung n Taktschritte benötigt und dafür die Zeit T zur Verfügung hat. Die Taktfrequenz f sei in gewissen Grenzen variierbar, sodass zur Ausführung der Berechnung die Zeit T^{aktiv} benötigt wird und es gilt $T = T^{aktiv} + T^{inaktiv}$, sowie $0 < T^{aktiv}(f_{max}) < T^{aktiv}(f_{min}) < T$. Die Versorgungsspannung

$$U_{dd}(f) = m_U \cdot f + U_0 \tag{30}$$

sei linear abhängig von der eingestellten Taktfrequenz. Der Anstieg m_U muss bestimmt werden. Er ist Null, wenn U_{dd} konstant ist. Die Verlustleistung des Oszillators

$$P_{osc}(f) = m_{osc} \cdot f + P_{osc_0} \tag{31}$$

wird ebenfalls als linear von der Taktfrequenz abhängig angenommen. Neben einer Abschätzung für m_{osc} muss eine Relation zu P_{dyn}^{aktiv} hergestellt werden. So könnte in einem System z. B. $P_{osc_0} = P_{dyn}^{aktiv}(f_{min})/10$ betragen. Es gelte weiterhin

$$\begin{aligned} P_{stat}(U) &= m_{stat} \cdot e^U \\ P_{dyn}^{aktiv}(U, f) &= m_{dyn}^{aktiv} \cdot U^2 f \end{aligned} \tag{32}$$

Zwischen P_{stat} und P_{dyn}^{aktiv} muss ebenfalls eine Relation abgeschätzt werden. Damit sind alle Unbekannten entweder durch Abschätzungen bestimmt oder direkt abhängig von P_{dyn}^{aktiv}. Diese letzte Abhängigkeit von P_{dyn}^{aktiv} kann durch Normierungen eliminiert werden.

In Abb. 55 werden mehrere auf solchen Relationen basierenden Zusammenhänge visualisiert. Dargestellt ist in jedem Diagramm die vom System während der Zeitspanne

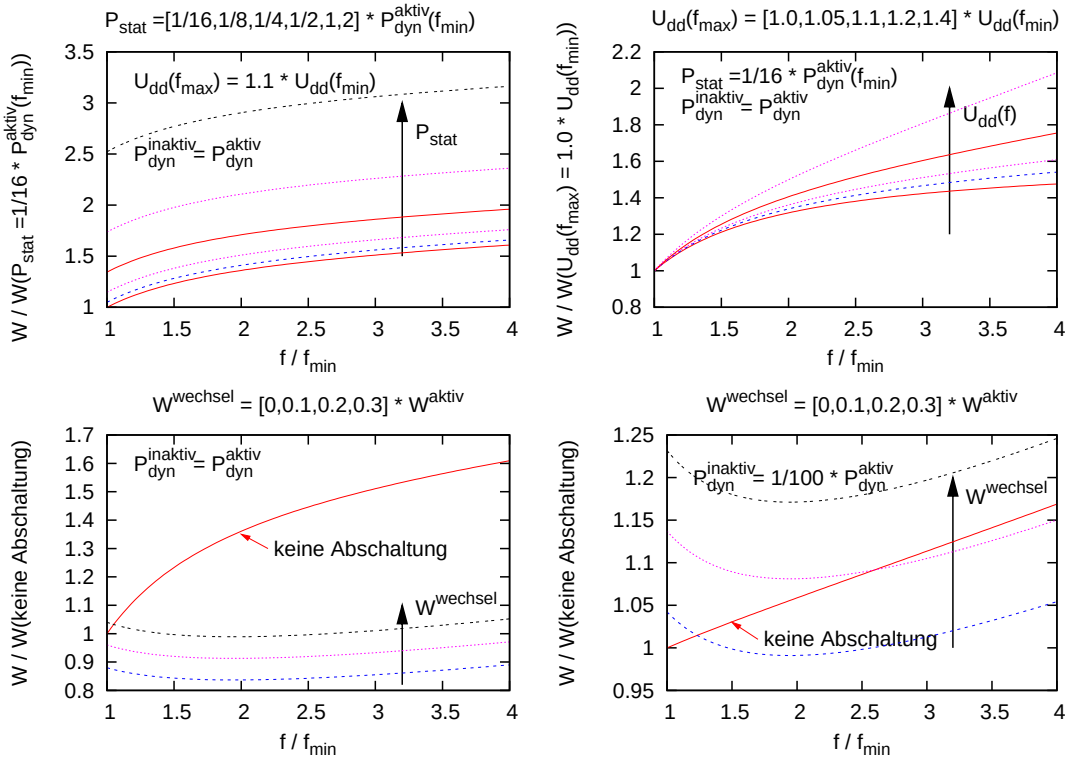

Abbildung 55: Abgeschätzte normierte Verluste (Beispielsysteme)

T umgesetzte Energie W in Abhängigkeit von der Taktfrequenz im aktiven Zustand. Dabei ist W stets normiert auf einen für den jeweiligen Vergleich sinnvollen Fall.

Die beiden oberen Graphen visualisieren ein System mit $W_{dyn}^{inaktiv} > 0$, bei dem Takt und Spannung nicht deaktiviert werden können. Verschieden hohe statische Verluste P_{stat} (linkes Diagramm) und unterschiedlich starke Anstiege der Funktion $U_{dd}(f)$ (rechtes Diagramm) sind dargestellt. Da alle Kurven streng monoton steigend sind, wird die Aussage aus dem vergangenen Abschnitt bestätigt, dass in diesem Fall der Betrieb mit geringstmöglicher Taktfrequenz am effizientesten ist. Das gleiche kann gezeigt werden für konstante Betriebsspannung. Da $W^{wechsel}$ als konstant angenommen wird, hat dieser Wert durch die Normierung der Energie keinen Einfluss auf den Kurvenverlauf.

Die beiden unteren Graphen beziehen sich auf ein System, welches im inaktiven Zustand Takt und Spannung abschalten kann, sodass in Näherung $P^{inaktiv} = 0$ gilt. Normierungsbezug ist die Variante ohne Abschaltung. Für die Abschaltung muss eine Energiemenge aufgewendet werden, die in $W^{wechsel}$ einfließt. In dem linken System mit hohen dynamischen Verlusten im inaktiven Zustand (kein Clock-Gating) ist eine Abschaltung in den meisten Fällen vorteilhaft. Im rechten System, welches Clock-Gating für inaktive Baugruppen verwendet, kann dagegen das Vermeiden der Abschaltung vorteilhaft sein, wenn $W^{wechsel}$ zu groß ist. Ist eine Abschaltung aber vorteilhaft, so zeigen die Graphen ein Minimum für die im aktiven Zustand verwendete Taktfrequenz. Dies ist durch die

Abhängigkeit der Betriebsspannung von der Frequenz $U_{dd}(f)$ bedingt. Im Beispiel wurde angenommen, dass bei maximaler Frequenz 10% mehr Spannung benötigt wird als bei minimaler. Ist U_{dd} dagegen konstant, so kann gezeigt werden, dass die Graphen bei aktivierter Abschaltung streng monoton fallend sind.

Mit solchen Graphen, die auf abgeschätzten Relationen zwischen den verschiedenen Teilverlusten basieren, kann die optimale Taktfrequenz grob abgeschätzt werden. Nicht immer ist für reale Systeme diese Methode praktikabel. Insbesondere beim rechten unteren Beispiel aus Abb. 55 hängt eine Entscheidung ganz erheblich von der Qualität der Abschätzung der Relationen ab. Die Umsetzung von verschiedenen Realisierungsmöglichkeiten und deren Bewertung kann dann unumgänglich werden. Die Integration der dafür nötigen Energieabschätzung in einen Profiler ist aufwändig und aufgrund der Abstraktion der zu erstellenden Verlustmodelle fehleranfällig. Günstiger ist die Verlustabschätzung basierend auf realen Schaltungen, wie sie mit `energy_estim` möglich ist. Steht für eine Komponente keine synthesefähige Beschreibung zur Verfügung, so können auch mit Hilfe eines abstrakten Modells in VHDL die Verluste abgeschätzt werden.

Zusammenfassend kann verallgemeinert werden: Je länger die Ruheperioden eines Systems sind, desto eher wird eine Reduzierung von Takt und Spannung oder eine Abschaltung sinnvoll sein. Die bei Microcontrollern häufig vorhandenen Randbedingungen (vernachlässigbare statische Verlustleistung, keine variable Versorgungsspannung, komplexe Taktabhängigkeiten, vernachlässigbare dynamische Verluste im inaktiven Zustand uvm.) lassen eine Taktregelung oft unsinnig werden.

3.6. Softwareentwurf in der Hardwaresimulationsumgebung

Nicht alle Probleme des Softwareentwurfs lassen sich allein in einer Softwareentwurfsumgebung und deren integriertem Simulator für den Prozessor lösen. Häufig stellt die Entwicklung direkt auf der existierenden Schaltung, in der auch externe Mess- und Steuersignale verfügbar sind, eine sinnvolle Alternative dar (in-circuit development / debugging). Eine andere Alternative ist eine Hardwaresimulationsumgebung und die Verwendung von synthesefähigen oder Verhaltensmodellen in einer HDL. Nachteilig gegenüber der Entwicklung auf der existierenden Schaltung ist die Notwendigkeit, externe Stimuli durch Verhaltensmodelle bereitzustellen. Große Vorteile sind aber die Möglichkeit, alle relevanten internen Signale beobachten zu können, sodass auch ein komplexer interruptgesteuerter Programmablauf einfach nachvollziehbar bleibt und Verlustabschätzungen abhängig vom ausgeführten Programm durchzuführen. Auch ist es möglich, noch vor der Fertigung der ersten Hardware die Softwareentwicklung zu beginnen und so auch Wünsche zur Änderung des Verhaltens der Hardware berücksichtigen zu können.

3.6.1. Zuordnung von Befehlen bei der Hardwaresimulation

Nachteilig erweist sich der Medienbruch zwischen Softwareentwicklungsumgebung und Hardwaresimulation. Während der Hardwaresimulation ist es oft schwierig nachzuvollziehen, welchen Teil eines Programms eine CPU gerade ausführt. Es soll im Folgenden gezeigt werden, wie dieser Medienbruch verkleinert werden kann.

Ein wesentliches Verbindungselement zwischen Hardwaresimulation und der zu simulierenden Software existiert in Form des Programmcounters (PC). Der PC zeigt beim Laden der Instruktion auf deren Adresse. Da ein Eingriff in eine Beschreibung einer CPU zur Bestimmung des PCs oft nicht möglich ist, muss dieser extern bestimmt werden. Dies ist möglich mit Hilfe einer Baugruppe zur Simulation, welche bei jedem Instruktionsladezyklus (Instruction Fetch) den Opcode auf dem Datenbus sowie den PC auf dem Adressbus mitliest (Abb. 56).

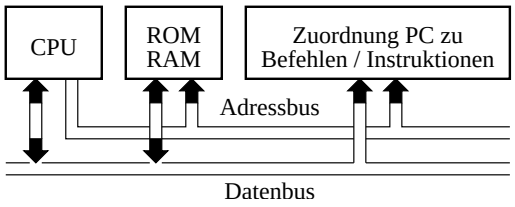

Abbildung 56: Bestimmung des jeweils aktuellen PCs

Der in Kap. 3.4 vorgestellte Profiler `eprof` kann einem Befehl auf Hochsprachenebene die entsprechenden Maschineninstruktionen zuordnen, indem er diese Information aus den stabs Debug Informationen extrahiert. Verknüpft man in der Hardwaresimulationsumgebung den bestimmten PC mit dieser Zuordnung, erhält man die Möglichkeit, jederzeit die aktuell ausgeführte Hochsprachenbefehlszeile während der Hardwaresimulation anzuzeigen.

Tabelle 22: Mögliche Zuordnung von Adresse und Hochspracheninstruktion

`...`	
`0x2418`	Instruktionsadresse
`mov R8,R4`	Maschineninstruktion
`75`	Zeilennummer in der Hochsprache
`b=(unsigned long)a;`	Hochsprachenbefehlszeile
`main`	aktuelle Funktion
`...`	

Es ist sinnvoll, die extrahierte Zuordnung in einem einfachen Format abzuspeichern, sodass der Aufwand beim Einlesen mit einer HDL gering bleibt. Günstig erweist sich das zeilenweise Abspeichern von Tupeln wie in Tab. 22 illustriert. In VHDL können Instruktion, Befehlszeile und Funktionsname Signalen vom Typ `string` zugewiesen und in der Simulation angezeigt werden. Somit kann der Anwender direkt die gerade abgearbeiteten Befehle nachvollziehen und der Medienbruch zwischen Hardwaresimulation und Hochsprache wird verkleinert (These 5).

Zur Umsetzung des beschriebenen Konzepts wurde im Zuge dieser Arbeit ein Softwarewerkzeug erstellt, welches aus einer ELF-Datei mit Debuginformationen die Zuordnungen extrahiert (Abb. 57), sowie eine VHDL Komponente, die diese Zuordnungen einliest und mit dem aktuellen PC verknüpft. Für die Umsetzung für MSP430 wird das damit erreichte Ergebnis mit der Hardwaresimulation in Abb. 58 gezeigt. Das Signal `asm` hält

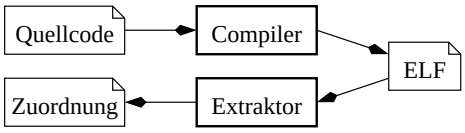

Abbildung 57: Extraktion der Zuordnung

den disassemblierten Maschinencode, das Signal `c_source` die Befehlszeile und `c_line` deren Zeilennummer. Weiterhin dargestellt sind der PC und einige Signale von Adress- und Datenbus (`ab`, `db_o`, `db_i`, `dbcsl_n`, `dbwr_n`).

Zugunsten der Übersicht wurde mit der Funktion aus Listing 28 ein einfaches Beispiel gewählt. Das Produkt wird mit Hilfe eines Hardwaremultiplizierers bestimmt. Die mit `energy_estim` abgeschätzte Energie pro halbem Taktschritt für den Multiplizierer inclusive dessen Interface zum Adress- und Datenbus ist mit dem Signal `energy_lastclk` dargestellt.

```
long myfunction(int a, int b){
        a += b;
        b >>= 1;
        return (long)a * b;
}
```

Listing 28: Beispielfunktion für die Simulation in Abb. 58

Mit den vorgestellten Methoden ist es möglich, innerhalb einer Hardwaresimulationsumgebung das funktionale Verhalten unter Berücksichtigung komplexer Softwareroutinen zu simulieren, die dabei umgesetzte Energie abzuschätzen und die Zuordnung zu Hochsprachenbefehlen in einfacher Form herzustellen. Insbesondere der Entwurf ereignisgesteuerter Software für Microcontroller wird dadurch unterstützt.

3.6.2. Profiling auf Hardwarebasis

Ein Profiler und auch `eprof` ist angewiesen auf einen Simulator für zumindest die CPU. Je mehr weitere Komponenten simuliert werden, desto mehr Möglichkeiten existieren für analysierbare Programme. Das Erstellen von Simulationsmodellen für Microcontrollercomponenten ist aufwändig. Steht dagegen eine Beschreibung einer Komponente in einer Hardwaresimulationsumgebung zur Verfügung, entsteht der Wunsch, diese stattdessen zu nutzen. Offensichtlich ist das auf direktem Wege nicht möglich.

Die Grundidee von Profiling in Form des Flat Profile basiert auf dem Zählen der Anzahl der Ausführungen von Maschineninstruktionen. Genau dies ist eine wesentliche Aufgabe des in `eprof` integrierten Simulators. Die Anzahl der Ausführungen kann aber auch während der Hardwaresimulation einfach bestimmt werden. Eine Möglichkeit ist das simple Abspeichern der Adresse der ausgeführten Intruktion in einer Datei. Diese kann von `eprof` eingelesen und daraus die Anzahl der Ausführungen der Instruktionen ausgezählt werden.

Mit dieser äußerst einfachen Methode sind neben dem vollständigen Cost Profile auch große Teile des Variable / Type Profiles und des Hint Profiles realisierbar. Unbekannt

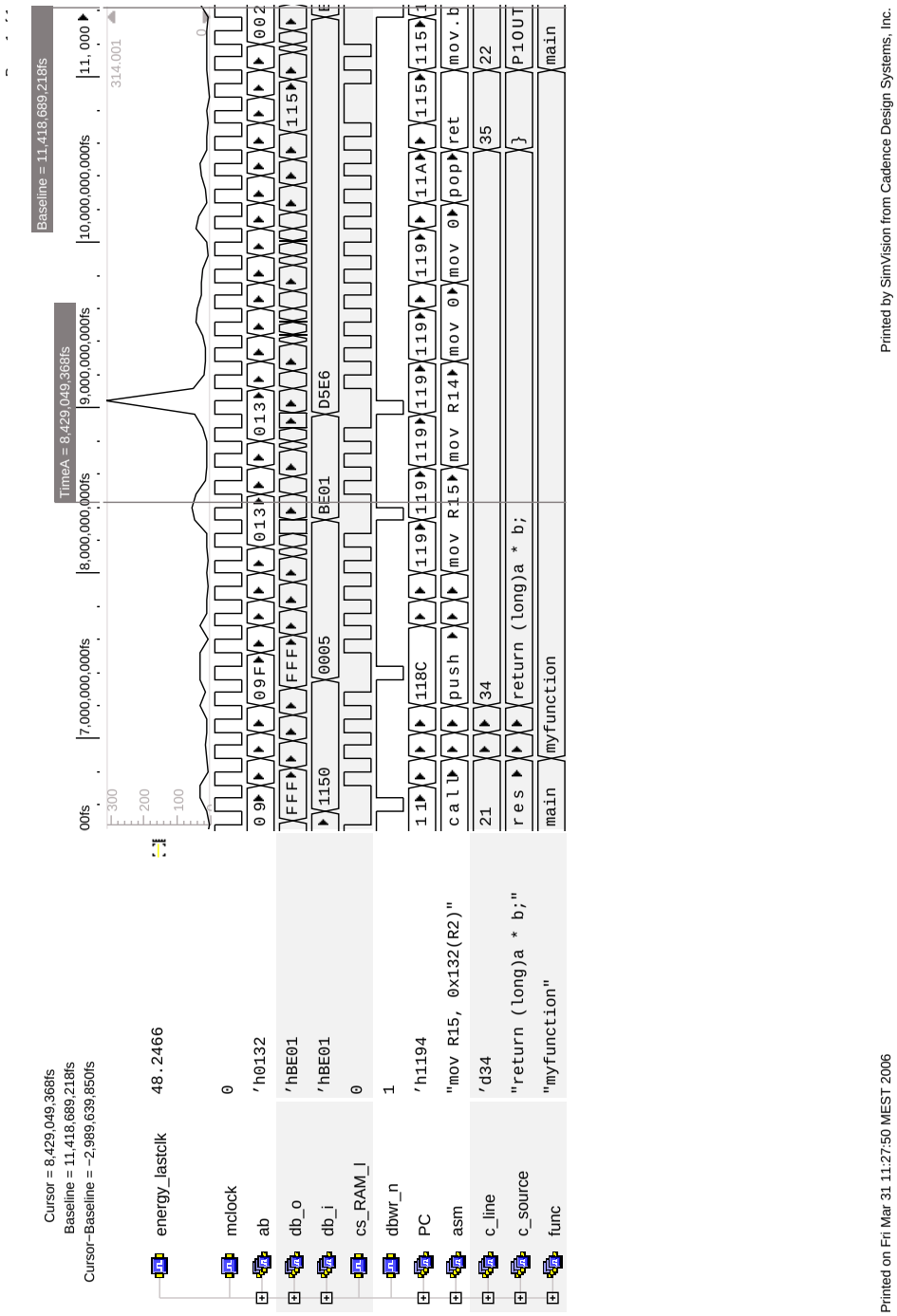

Abbildung 58: Simulation mit Befehlszeilenzuordnung und Energieabschätzung

bleiben die Zugriffe auf externe und Stackvariablen, aber die Anzahl der Registerzugriffe sowie die statische Analyse von Variablen und Typen können angegeben werden. Der Verlust der Zugriffszähler für externe und Stackvariablen ist ein Nachteil, aber die statische Analyse der Variablen deckt zumindest deren Speicherort auf, sodass durch das so reduzierte Variable / Type Profile durchaus noch sehr nützliche Informationen gegeben werden. In Abb. 59 wird der klassischen Weg eines Profilers dem des hardwaregestützten Profilings gegenübergestellt.

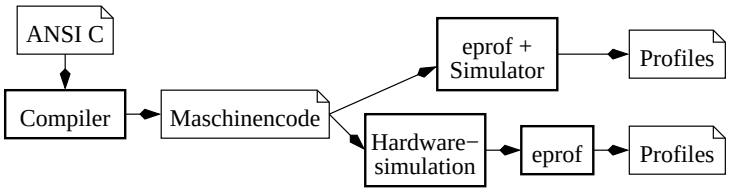

Abbildung 59: `eprof` und Unterstützung durch Hardwaresimulation

Auch innerhalb einer Hardwaresimulation ist es vorstellbar, alle Informationen zusammen zu tragen, die auch der in `eprof` integrierte Simulator extrahiert. (Details zu den gesammelten Informationen befinden sich in Kap. 3.4.4.) Der Aufwand dafür ist allerdings sehr groß. Auch wird die vollständige Funktionalität von `eprof` in erster Linie bei der Optimierung von Teilroutinen benötigt, welche allein CPU und Multiplizierer benötigen, während bei der Betrachtung des ereignisgesteuerten Programmflusses die Hardwarebaugruppen die notwendigen Interaktionen direkt bestimmen, was die Anzahl von alternativen Realisierungen einschränkt. Die Grenze ist fließend. Je stärker der Fokus auf Details gerichtet wird, um höchste Optimierungen zu erreichen, desto mehr wird die volle Funktionalität von `eprof` benötigt.

4. Zusammenfassung

Die Reduktion von Verlusten digitaler Schaltungen auf Hardware- wie auch Softwarebasis ist ein wesentliches Entwurfsziel. Insbesondere für autonome, intelligente, mobile Systeme, welche häufig auf Basis von Microcontrollern realisiert werden, wird eine hohe Effizienz gefordert. Um dies zu erreichen, müssen Werkzeuge zur Analyse bereit stehen, welche in einfacher und schneller Form Vergleiche zwischen verschiedenen Realisierungen ermöglichen.

Mit der in Kap. 2.2 vorgestellten Methode zur Abschätzung von Verlusten während der VHDL-Simulation (`energy_estim`) wird sowohl die Bewertung verschiedener Hardwarerealisierungen als auch die Abschätzung von Verlusten bei der Interaktion von Soft- und Hardware ermöglicht. Wesentlich für die Bewertung von Hardwarerealisierungen ist die geringe Zeit für das Aufsetzen der Simulation (das Setup) der vorgestellten Methode, sodass schnell verschiedene Möglichkeiten auch mit untereinander stark unterschiedlichen Input / Output Signalen untersucht werden können. Während bei vielen herkömmlichen Methoden zusätzlich zur HDL-Simulation externe Werkzeuge benötigt werden, welche zum Teil aufwändige Interaktion vom Benutzer erfordern, ist bei `energy_estim` lediglich die automatisierte Modifikation der Netzliste der zu untersuchenden Schaltung nötig. Weiterhin ermöglicht die Kopplung von funktionaler Simulation und Verlustabschätzung die gegenseitige Zuordnung von Verhalten und den damit einhergehenden Verlusten.

Die hohe Geschwindigkeit von `energy_estim` bei der Abschätzung ermöglicht die Untersuchung von komplexen Schaltungen und damit auch der Interaktion von Soft- und Hardware. Insbesondere zur Abschätzung von Verlusten in Komponenten von Microcontrollern ist dies von Interesse, da selten abstrakte Modelle zur Verlustabschätzung von Peripheriekomponenten zur Verfügung stehen. Auch hängt die Aktivität der Komponenten maßgeblich von der Software ab und häufig wechseln Perioden mit hoher Aktivität mit Ruheperioden. Für solche Szenarien sind kaum pauschale Verlustabschätzungen möglich und es wird auch häufig eine Abschätzung für einen konkreten Anwendungsfall gewünscht. `Energy_estim` erfordert lediglich das Vorhandensein einer synthesefähigen Beschreibung der zu untersuchenden Komponente und kann einfach zusammen mit der meist sowieso nötigen Verhaltenssimulation eingesetzt werden. Die erreichte Genauigkeit von unter 22% relativem Fehler bei Abschätzungen von zellinternen Verlusten und von unter 20% bei Einbeziehung eines Wire Load Models und SDF-Backannotation im Vergleich zu Layoutsimulationen mittels Spectre ist ausreichend, um verschiedene Realisierungsmöglichkeiten zu vergleichen und ermöglicht auch eine brauchbare Abschätzung der zu erwartenden absoluten Verluste.

Die CPU von Microcontrollern hat einen großen Anteil an den Verlusten in einem Microcontrollersystem, da sie einen der aufwändigsten Teile des Systems darstellt und zudem hoch aktiv ist. Ineffiziente Softwarerealisierungen können schnell ein Vielfaches an Rechenzeit, verglichen mit einer effizienten Umsetzung, benötigen. Die Bewertung und Analyse von Software ist daher von besonders hohem Interesse. Bei sehr vielen Prozessoren und insbesondere bei Microcontrollern sind Verluste weniger abhängig von den Instruktionen als von der benötigten Taktanzahl und der Anzahl der Zugriffe auf den Speicher bzw. auf in den Speicherbereich abgebildete Komponenten. Unter diesen

Voraussetzungen ist Softwareoptimierung für geringe Verluste gleich der Optimierung auf numerische Effizienz.

Hoch optimierte Software ist nicht gleichzusetzen mit dem Entwurf in Assembler, sondern es konnte durch die Analyse der Hochsprache ANSI C in Kap. 3.2 gezeigt werden, dass insbesondere mit Hochsprachen auch sehr effiziente Programme realisierbar sind. Der Abstraktionsgrad von Hochsprachen ermöglicht das einfache Erkennen von funktionalen und algorithmischen Vereinfachungen und das Vermeiden ungünstiger Konstrukte verhindert das Entstehen von ineffizientem Maschinencode.

Die in Kap. 3.4 vorgestellten Methoden zur Softwareanalyse (Profiling) ermöglichen die Bewertung und zielgerichtete Optimierung von Software. Wesentliche Merkmale sind das erweiterte Profiling auf Basis von detaillierten Angaben zu Programmkosten auf Befehlsebene der Hochsprache, die neuartige Analyse von Variablen und Typen und die Unterstützung des Anwenders durch gezielte Hinweise. Insbesondere die Analyse von Variablen und Typen ermöglicht es, den Grund für die beobachteten Kosten zu ermitteln, wobei die Analyse der Abbildung von Variablen auf Register oder RAM der wichtigste Punkt ist. Der Anwender ist somit nicht nur auf seine Erfahrung und sein Gespür für mögliche Alternativen bei Optimierungen angewiesen, sondern kann durch die Kenntnis der Gründe gezielte Optimierungsstrategien entwickeln. Für das Verständnis der Mechanismen sind dank der abstrakten Analysemethode keine Kenntnisse in Assemblerprogrammierung und nur geringe Kenntnisse über die Struktur der verwendeten CPU notwendig. Sind Assemblerkenntnisse vorhanden, so können diese zusätzlich für Verbesserungen genutzt werden, wobei durch die vorgestellte Methode stets der Blick auf Hochsprachenebene beibehalten werden kann und nur einzelne, besonders kritische Befehle hervorgehoben werden.

Die Energieabschätzung auf Hardwareebene mit VHDL und die Analyse von Software durch einen Profiler besitzen zwar einige Überschneidungen, ergänzen sich aber dennoch beim Entwurf von kompletten Microcontrollersystemen. Für den Entwurf von Hardwarebaugruppen und kompletten Komponenten ist eine Verlustabschätzung notwendig. Optimierungen von Softwarealgorithmen sind dagegen das Einsatzgebiet eines Profilers. Für die Interaktion von Soft- und Hardware kann zwar ein Softwaremodell der betrachteten Hardwarekomponente für einen Profiler erstellt werden, aber es ist meist einfacher, stattdessen eine bestehende Verlustabschätzung zu verwenden. Insbesondere bei der Bewertung eines interruptgesteuerten Systems wie es in Kap. 3.5 diskutiert wurde, kann eine Verlustabschätzung, wie die von `energy_estim`, von großem Interesse sein.

Bindeglieder zwischen Hardwaresimulation und Softwareentwurf stellen das hardwaregestützte Profiling und die neuartige Präsentation von Hochsprachenbefehlen innerhalb der Hardwaresimulation dar (Kap. 3.6).

Schwerpunkte dieser Arbeit waren die schnelle Verlustabschätzung `energy_estim`, welche ein besonders einfaches Setup der Simulation bietet und die hoch detaillierte Programmanalyse `eprof`, welche für die Effizienz von Programmen wesentliche Parameter besonders untersucht und damit zielgerichtete Softwareoptimierungen allein auf Hochsprachenniveau ermöglicht.

Anhang

A. Der Baugh-Wooley-Algorithmus

Der Baugh-Wooley-Algorithmus [25] ist ein sehr verbreiteter Algorithmus, zur Vorzeichenexpansion in klassischen Multiplizierern. Mittels einer Herleitung soll er erläutert werden:

Eine Zahl im Zweierkomplement wird dargestellt mit

$$a = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \tag{33}$$

Die Multiplikation zweier Zahlen ergibt sich daraus direkt zu:

$$\begin{aligned} a \cdot b &= \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right)\left(-b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j\right) \\ &= a_{n-1}b_{n-1}2^{2(n-1)} - a_{n-1}2^{n-1}\sum_{j=0}^{n-2} b_j 2^j - b_{n-1}2^{n-1}\sum_{i=0}^{n-2} a_i 2^i + \sum_{i=0}^{n-2} a_i 2^i \sum_{j=0}^{n-2} b_j 2^j \end{aligned} \tag{34}$$

Dies kann vereinfacht werden, indem folgender Zusammenhang ausgenutzt wird:

$$\begin{aligned} -a_{n-1}2^{n-1}\sum_{j=0}^{n-2} b_j 2^j &= -2^{n-1}\left(\sum_{j=0}^{n-2} a_{n-1}b_j 2^j\right) \\ &= -2^{n-1}\left(-0\cdot 2^{n-1} + \sum_{j=0}^{n-2} a_{n-1}b_j 2^j\right) \\ &= 2^{n-1}\left(-2^{n-1} + \sum_{j=0}^{n-2} \overline{a_{n-1}b_j} 2^j + 1\right) \\ &= -2^{2(n-1)} + \sum_{j=0}^{n-2} \overline{a_{n-1}b_j} 2^{j+n-1} + 2^{n-1} \end{aligned} \tag{35}$$

Damit ergibt sich:

$$\begin{aligned} a \cdot b = {} & \underbrace{a_{n-1}b_{n-1}2^{2(n-1)}}_{A} + \underbrace{\sum_{j=0}^{n-2} \overline{a_{n-1}b_j} 2^{j+n-1}}_{B} + \underbrace{\sum_{i=0}^{n-2} \overline{b_{n-1}a_i} 2^{i+n-1}}_{C} + \\ & + \underbrace{2(2^{n-1} - 2^{2(n-2)})}_{\underbrace{2^n}_{D}\ \underbrace{-2^{2n-1}}_{E}} + \underbrace{\sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j}}_{F} \end{aligned} \tag{36}$$

Die graphische Interpretation von (36) für die Wortbreite $n = 4$ wird in Abb. 60 illustriert.

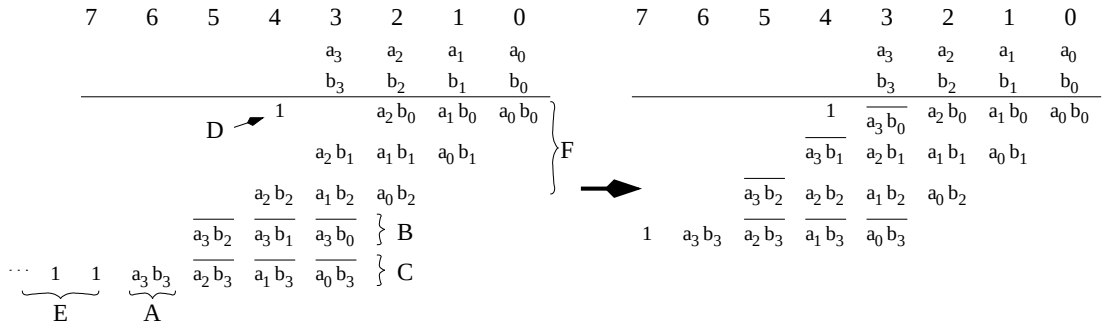

Abbildung 60: Baugh-Wooley-Algorithmus

Der Term E ist negativ. Dies hat allerdings keine problematischen Auswirkungen, da die Vorzeichenbits eine Wertigkeit größer $2^{(2n-1)}$ besitzen und demzufolge ignoriert werden können. übrig bleibt nur der Summand 2^{2n-1}.

Der Baugh-Wooley-Algorithmus beschreibt also eine gezielte Inversion einiger Bits im Array der partiellen Produkte und das Hinzufügen einiger Bits. Mit diesem geringen Aufwand ist vorzeichenbehaftete Multiplikation ohne aufwändige Vorzeichenexpansion möglich.

B. Thesen

1. Spezialisierte Hardware setzt geringere Verlustenergie um als eine Softwarelösung. Dies gilt insbesondere auch dann, wenn ausreichend Rechenzeit zur Verfügung steht, also auch eine Softwarerealisierung ohne zusätzlichen Aufwand realisierbar wäre (Kap. 1.3.5.2, 3.1).

2. Eine Abschätzung von Verlusten ist nötig, um verlustintensive Baugruppen zu identifizieren und mögliche alternative Realisierungen zu evaluieren. In erster Linie wird dadurch der Hardwareentwurf unterstützt, aber auch für den Softwareentwurf (insbesondere bei Interaktion mit Hardwarebaugruppen) ist dies von Interesse. Eine solche Abschätzungsmethode (`energy_estim`) wird in dieser Arbeit vorgestellt.

   a) Um die gewünschte hohe Geschwindigkeit bei der Verlustabschätzung zu erreichen ist nicht nur eine schnelle Abschätzungsmethode, sondern als ganz wesentliches Element auch die Reduzierung des Aufwandes für das Aufsetzen der Abschätzung nötig (Kap. 2).

   b) Verlustabschätzung mittels im voraus berechneter Verlustwerte für alle möglichen Einzelsignaländerungen von Eingangssignalen, welche mit Hilfe eines analogen Schaltkreissimulators bestimmt werden, ermöglicht eine hohe Geschwindigkeit und die Berücksichtigung vieler Einflussgrößen während der Abschätzung von Verlusten (Kap. 2.2.2).

   c) Partielle Umladungen müssen berücksichtigt werden, um eine akzeptable Genauigkeit bei Verlustabschätzungen zu erreichen (Kap. 2.2.2.4 und 2.2.3).

   d) Die Konzepte der Berechnung von Verlustwerten im Voraus und die Einbettung von darauf basierenden Abschätzungen in VHDL-Containerzellen sind auf Leitungsverluste und statische Verluste erweiterbar und nicht nur auf die Abschätzung dynamischer Verluste in Zellen beschränkt. Variable Versorgungsspannungen können berücksichtigt werden (Kap. 2.2.4).

   e) Für die meisten Prozessoren sind auf Verlustleistung optimierte Softwareroutinen gleichzusetzen mit Routinen, welche eine geringe Anzahl von Taktzyklen und Speicherzugriffen nach sich ziehen. Numerisch einfache, also schnelle Routinen setzen die geringsten Verlustenergien um (Kap. 2.3).

3. Die Effizienz von Software wird maßgeblich durch die Ausführungszeit und die Anzahl der nötigen Speicherzugriffe bestimmt. Die Optimierung unter diesen Gesichtspunkten ist der Kernpunkt beim Entwurf effizienter Software.

   a) Die Optimierungen können auf konzeptioneller, algorithmischer und struktureller Ebene erfolgen. In allen Fällen ist Optimierung nicht gleichbedeutend mit dem Einsatz von Assemblercode, sondern gerade die Abstraktion einer Hochsprache lässt das einfache Erkennen von Optimierungsmöglichkeiten auf höheren Ebenen zu. Werkzeuge zur Analyse von Programmen müssen demnach die Entwicklung auf Hochsprachenebene fördern (Kap. 3.2.1).

b) Konzeptionelle und algorithmische Optimierungen werden durch die Abstraktion einer Hochsprache selbst erleichtert. Für die strukturale Optimierung von Software auf Hochsprachenebene ist aber nicht nur die Kenntnis der Funktionalität der Hochsprache, sondern auch der dahinter verborgenen Mechanismen nötig. Das impliziert nicht die Kenntnis über die Umsetzung auf Maschineninstruktionen, sondern umfasst Kenntnisse über die Art der Speicherung von Variablen und die funktionale Umsetzung von Hochsprachenkonstrukten, wie Kontrollstrukturen und Funktionen. Eine Analyse einer Hochsprache unter diesen Gesichtspunkten ist somit nötig als Basis für strukturale Hochsprachenoptimierungen und wird in dieser Arbeit für die Hochsprache ANSI C präsentiert (Kap. 3.2).

c) Wesentliche Punkte bei der Optimierung auf strukturaler Ebene sind angepasste Datenstrukturen, Programmflussoptimierung, Reduktion von Speicherzugriffen und auch die effiziente Nutzung des Instruktionssatzes (Kap. 3.3).

4. Wesentlich für die Softwareoptimierung ist ein Werkzeug zur Analyse von Programmen, ein Profiler. Ein erweiterter Profiler, wie der in dieser Arbeit vorgestellte `eprof` ermöglicht im Gegensatz zu klassischen Profilern eine zielgerichtete Softwareoptimierung und vereinfacht damit den Optimierungsprozess.

   a) Zur Optimierung von Software ist nicht nur die Lokalisierung der Kosten auf Funktions-, sondern auf Befehlsebene nötig. Detailliertes, feinkörniges Profiling auf Befehlsebene mit Angabe von nicht nur der Laufzeit, sondern mehreren Größen, ermöglicht die präzise Erkennung der Hauptquellen für die Kosten (Kap. 3.4).

   b) Die Analyse von Variablen, deren Speicherort und Zugriffe darauf decken Gründe für die Programmkosten auf. Nach Lokalisierung der kostenintensivsten Funktionen und Befehle stellen diese Informationen eine Grundlage dar, aus der konkrete Ansätze zur Optimierung entstehen können (Kap. 3.4).

   c) Automatisch generierte Hinweise für mögliche Optimierungsschritte mit direktem Bezug auf die konkreten Positionen im Quelltext können insbesondere für weniger erfahrene Softwareentwickler hilfreich sein (Kap. 3.4).

   d) Mit Hilfe feinkörniger Programmanalyse, insbesondere durch Variablenanalyse, ist es möglich, konkrete Ansätze zur Softwareoptimierung abzuleiten, sodass nur geringe Kenntnisse über die Architektur einer CPU und keine Kenntnisse in Assemblerprogrammierung vorhanden sein müssen (Kap. 3.4.3.2). Sind genauere Kenntnisse vorhanden, so können diese dank detaillierter Analyse zielgerichtet an kritischen Punkten verwendet werden (Kap. 3.4.3.4).

5. Die Präsentation von Befehlen auf Hochsprachenebene innerhalb der Hardwaresimulation erlaubt das einfache Nachvollziehen des Programmflusses. Insbesondere der Entwurf ereignisgesteuerter Software für Microcontrollersysteme wird dadurch unterstützt. Die Darstellung von ANSI C Quellcode innerhalb der VHDL-Simulation wird in dieser Arbeit präsentiert (Kap. 3.6.1).

Literatur

[1] Massoud Pedram, Jan Rabaey (Editoren): *Power Aware Design Methodologies*; Kluwer Academic Publishers 2002; ISBN: 1-4020-7152-3

[2] David J. Frank: *CMOS Device Technology Trends for Power-Constrained Applications*; Kapitel in [1]

[3] Tadahiro Kuroda: *Low-Power Digital Circuit Design*; Kapitel in [1]

[4] Chulwoo Kim, Sung-Mo (Steve) Kang: *Low Power Flip-Flop and Clock Network Design Methodologies in High-Performance System-on-a-Chip*; Kapitel in [1]

[5] Farzan Fallah, Massoud Pedram: *Circuit and System Level Power Management*; Kapitel in [1]

[6] Liqiong Wei, Kaushik Roy, Vivek K. De: *Low Voltage Low Power CMOS Design Techniques for Deep Submicron ICs*;
http://www.es.lth.se/home/vikt/LowPower/articles/royLVLP.pdf

[7] Qi Wang, Sarma B. K. Vrudhula: *Static Power Optimization of Deep Submicron CMOS Circuits for Dual V_T Technology*; http://www.sigda.org/Archives/ProceedingArchives/Iccad/Last20/Papers/1998/ICCAD98_490.PDF

[8] http://www.dch-faq.de/kap03.html
http://users.erols.com/chare/

[9] Adrian M. Ionescu: *An Outlook of Technology An Outlook of Technology Scaling Beyond Moore's Law*;
http://si.epfl.ch/webdav/site/si/shared/OutlookBeyondMoore_Ionescu.pdf

[10] Pai H. Chou, Jinfeng Liu, Dexin Li, Nader Bagherzadeh: *IMPACCT: Methodology and Tools for Power-Aware Embedded Systems*;
http://www.ece.uci.edu/impacct/d_research/d_pub/impacct.pdf

[11] Mark Horowitz, Thomas Indermaur, Ricardo Gonzales: *Low-Power Digital Design*;
http://mos.stanford.edu/papers/mah_slpe_94.pdf

[12] Manjit Borah, Robert Michael Owens, Mary Jane Irwin: *Transistor Sizing for Minimizing Power Consumption for CMOS Circuits under Delay Constraint*; Proceedings of the 1995 Intl. Symposium on Low Power Design;
http://www.cse.psu.edu/~ugrain/vlsi-cad/LOW-POWER/islpd95r6_3.ps

[13] Stephanie Augsburger, Borivoje Nikoli: *Combining Dual-Supply, Dual-Threshold and Transistor Sizing for Power Reduction*; Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 02); http://www.iccd-conference.org/proceedings/2002/17000316.pdf

[14] Thomas D. Burd, Trevor Pering, Anthony Stratakos, Robert W. Brodersen: *A Dynamic Voltage Scaled Microprocessor System*
http://bwrc.eecs.berkeley.edu/Publications/2000/
presentations/IEEE_ISSCC_2000/ISSCC-00-Slides.pdf

[15] Seongsoo Lee, Takayasu Sakurai: *Run-Time Voltage Hopping for Low-power Real-time Systems*; Design Automation Conference 2000; http://citeseer.ist.psu.edu/
cs?q=Takayasu+Sakurai&submit=Search+Documents&cs=1

[16] Stephan Henzler, Thomas Nirschl, Stylianos Skiathitis, Joerg Berthold, Juergen Fischer, Philip Teichmann, Florian Bauer, Georg Georgakos, Doris Schmitt-Landsiedel: *Sleep Transistor Circuits for Fine-Grained Power Switch-Off with Short Power-Down Times*; IEEE International Solid-State Circuits Conference ISSCC 2005
http://ieeexplore.ieee.org/iel5/9995/32118
/01493989.pdf?tp=&arnumber=1493989&isnumber=32118

[17] Chris H. Kim, Jae-Joon Kim, Ik-Joon Chang, Kaushik Roy: *PVT-Aware Leakage Reduction for On-Die Caches with Improved Read Stability*; IEEE International Solid-State Circuits Conference ISSCC 2005
http://www.cs.pitt.edu/~cho/cs3410/currentsemester/handouts/kim-isscc05.pdf

[18] B. Flachs, S. Asano, S. H. Dhong, P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano:*A Streaming Processing Unit for a CELL Processor*; IEEE International Solid-State Circuits Conference ISSCC 2005
http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/
E815CC047A60914687256FC000734156/$file/ISSCC-07.4-Cell_SPU.PDF

[19] Simon Segars: *Low Power Design Techniques for Microprocessors*; ISSCC 2001;
http://www.arm.com/about.nsf/eedd0f0f16b9df36802569de0057e0ad/
b856e3180691c4d5802569f200596d61/$FILE/SS_ISSCC2001.PDF

[20] Arthur Abnous: *Low-Power Domain-Specific Processors for Digital Signal Processing*; http://bwrc.eecs.berkeley.edu/Publications/2001/
Theses/L-pwr_domain-spec_processors/Abnous/thesis.pdf

[21] Anantha Chandrakasan, Rex Min, Manish Bhardwaj, Seong-Hwan Cho, and Alice Wang: *Power Aware Wireless Microsensor Systems*;
http://www.mit.edu/~rmin/research/chand-esscirc02.pdf

[22] Microchip: PIC; http://www.microchip.com

[23] Zilog: Z80; http://www.zilog.com

[24] Texas Instruments: *MSP430 ultra low power microcontroller*; http://www.ti.com

[25] A. R. Baugh, B. A. Wooley: *A two's complement parallel array multiplication algorithm*, IEEE Trans. Computers, Vol. C-22, No. 1-2, December 1973, pp. 1045-1047
(Dieses Dokument wurde für diese Arbeit nicht direkt verwendet.)

[26] Ralf Hildebrandt: *Power Comparison of Low Bitwidth Multipliers*; International Conference on Microelectronics (ICM) 2004

[27] *SPICE*: http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/

[28] HSPICE: http://www.synopsys.com/products/mixedsignal/hspice/hspice.html

[29] *IRSIM*: http://bwrc.eecs.berkeley.edu/Classes/IcBook/IRSIM/index.html

[30] Huang, Zhang, Deng, Swirski: *The Design and Implementation of Power Mill*; Proceedings of the International Symposium on Low Power Design (ISLPD) 1995, S. 105-110

[31] A. Sangiovanni-Vincentelli: *Techniques for Low Power Design*; http://www-cad.eecs.berkeley.edu/Respep/ Research/classes/ee219a/fall98/lec/power.ps

[32] Gareth Keane, Roger Woods: *Algorithms and Architectures for Low Power IC Design*; http://www.ee.qub.ac.uk/dsp/research/vlsi/lpid/download/d11.pdf

[33] Farid N. Najm: *A Survey of Power Estimation Techniques in VLSI Circuits*; IEEE transactions on very large scale integration (VLSI) systems, vol. 2, no. 4, december 1994; http://ieeexplore.ieee.org/iel4/92/7882/00335013.pdf?arnumber=335013

[34] Cadence *Physically Knowledgeable Synthesis*: http://www.cadence.com

[35] Jerry Frenkil: *Tools and Methodologies for Low Power Design*; http://ieeexplore.ieee.org/iel3/4655/13048/00597120.pdf?arnumber=597120

[36] Ronny Krashinsky, Seongmoo Heo, Michael Zhang, Krste Asanovic: *SyCHOSys: Compiled Energy-Performance Cycle Simulation*; http://www.ece.rochester.edu/~albonesi/wced00/papers/krashinsky.ps

[37] Alessandro Bogliolo, Luca Beniniy, Giovanni De Micheli: *Characterization-Free Behavioral Power Modeling*; http://www.hwswworld.com/downloads/f2/10B_2.PDF

[38] A. Bogliolo, I. Colonescu, R. Corgnati, E. Macii, M. Poncino: *An RTL Power Estimation Tool with On-Line Model Building Capabilities*; http://patmos2001.eivd.ch/program/Repro%5CArt_2_3.pdf

[39] Gerd Jochens, Lars Kruse, Wolfgang Nebel: *Application of Toggle-Based Power Estimation to Module Characterization*; http://www.dice.ucl.ac.be/~anmarie/patmos/papers/S5/5_4.pdf

[40] A.Th. Schwarzbacher, P.A. Comiskey, J.B. Foley: *PowerCount - Measuring the Power at the VHDL Netlist Level*; http://www.electronics.dit.ie/staff/ aschwarzbacher/research/eds98%20powercount.pdf

[41] Thucydides Xanthopoulos, Yoshifumi Yaoi, Anantha Chandrakasan: *Architectural Exploration Using Verilog-Based Power Estimation: A Case Study of the IDCT* (DAC 97);
http://www-mtl.mit.edu/researchgroups/icsystems/
pubs/conferences/1997/xanthopoulos_dac_paper.pdf

[42] Puneet Gupta, Andrew B. Kahng: *Quantifying Error in Dynamic Power Estimation of CMOS Circuits*; http://www.gigascale.org/pubs/466/c152.pdf

[43] Renu Mehra, Jan Rabaey: *Behavioral Level Power Estimation and Exploration*; Proceedings of the First International Workshop on Low Power Design, S. 197-202, 1994; http://bwrc.eecs.berkeley.edu/Publications/1994/publications/
behav_lvl_pwr_est_/lpw94.pdf

[44] David Brooks, Vivek Tiwari, Margaret Martonosi: *Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*
http://citeseer.ist.psu.edu/brooks00wattch.html

[45] Avant! Corporation *Master Toolbox* (heute Synopsys: http://www.synopsys.com)

[46] Synopsys: *Design Compiler Reference Manual*

[47] IEEE Standard: *VITAL ASIC Modeling Specification*;
http://tams-www.informatik.uni-hamburg.de/vhdl/
doc/P1076.4/VITAL_LRM.pdf

[48] Synopsys *NanoSim*;
http://www.synopsys.com/products/mixedsignal/nanosim/nanosim.html

[49] Synopsys: *Power Compiler Reference Manual*

[50] Ashwin Balakrishnan: *An Experimental Study of the Accuracy of Multiple Power Estimation Methods* http://vlsi1.engr.utk.edu/ece/balash-thesis.pdf

[51] Cadence *Spectre*: http://www.cadence.com

[52] Vivek Tiwari, Sharad Malik, Andrew Wolfe: *Power Analysis of Embedded Software: A First Step Towards Software Power Minimization*; International Conference on Computer-Aided Design, San Jose, 1994;
http://citeseer.ist.psu.edu/tiwari94power.html

[53] Vivek Tiwari, Sharad Malik, Andrew Wolfe: *Compilation Techniques for Low Energy*; International Conference on Low-Power Electronics, San Diego, 1994;
http://citeseer.ist.psu.edu/tiwari94compilation.html

[54] Vivek Tiwari, Sharad Malik, Andrew Wolfe, Mike Tien-Chien Lee: *Instruction Level Power Analysis and Optimization of Software*; Journal of VLSI Signal Processing, 1-18, 1996; http://citeseer.ist.psu.edu/tiwari96instruction.html

[55] Amit Sinha, Anantha P. Chandrakasan: *JouleTrack - A Web Based Tool for Software Energy Profiling*; DAC 2001, Las Vegas; http://citeseer.ist.psu.edu/sinha01jouletrack.html

[56] Jeffry T. Russell, Margarida F. Jacome: *Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors*; Proceedings of ICCD 98, Austin; http://horizon.ece.utexas.edu/~jacome/papers/RuJ98.pdf

[57] A. Parikh, Soontae Kim, M. Kandemir, N. Vijaykrishnan, M. J. IRWIN: *Instruction Scheduling for Low Power*; Journal of VLSI Signal Processing 37, 129 149, 2004; http://www.inf.ethz.ch/personal/daniekro/classes/ 251-0207-00/ws2005-2006/papers/pari.pdf

[58] IAR Systemes: IAR Embedded Workbench for MSP430; http://www.iar.com/

[59] msp430-gcc; http://mspgcc.sourceforge.net/

[60] Brian W. Kernighan, Dennis M. Ritchie: *Programmieren in C (Zweite Ausgabe: ANSI C)*, Verlag Carl Hanser, ISBN: 3-446-15497-3 (Original: *The C Programming Language (2nd Edition)*; Prentice Hall Software Series)

[61] Advanced RISC Machines Ltd (ARM): *Writing Efficient C for ARM (Application Note)*; http://www.arm.com/pdfs/DAI0034A_efficient_c.pdf

[62] Steve Summit et. al.: *comp.lang.c Answers to Frequently Asked Questions*; http://www.faqs.org/faqs/C-faq/faq/

[63] CTC Documentation - Bentley's Rules: http://www.tc.cornell.edu/Services/docs/Prog/Performance/bentley.asp (Zusammenfassung von J.L. Bentley: *Writing Efficient Programs*, Prentice-Hall, ISBN 0-13-970244-X)

[64] Bret Victor: *Software Optimization Using Hardware Synthesis Techniques*; http://www-cad.eecs.berkeley.edu/~wjiang/ee219b/report/bret.pdf

[65] Jack Klein: *A Replacement For atoi() Using A State Machine* http://home.att.net/~jackklein/c/code/getint.html

[66] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: *Numerical Recipes in C*; Cambridge University Press; ISBN: 0-521-75033-4

[67] *GNU binutils*; http://www.gnu.org/software/binutils/

[68] *OProfile*; http://oprofile.sourceforge.net

[69] Intel *VTune*; http://www.intel.com/software/products/vtune

[70] *DDD: Data Display Debugger*; http://www.gnu.org/software/ddd

[71] *Kprof*; http://kprof.sourceforge.net

[72] *GNU Compiler Collection (GCC)*; http://gcc.gnu.org

[73] *Executable and Linkable Format (ELF)*;
http://www.x86.org/ftp/manuals/tools/elf.pdf

[74] Julia Menapace, Jim Kingdon, David MacKenzie (Cygnus Support): *The „stabs“ Debug Format*; http://docs.freebsd.org/info/stabs/stabs.pdf

[75] The Institute of Electrical and Electronics Engineers, Inc: *IEEE Standard for Binary Floating-Point Arithmetic*;
http://754r.ucbtest.org/standards/754xml.html

[76] http://www.cs.nmsu.edu/~pfeiffer/classes/473/notes/fp-extras.html

[77] OpenEXR; http://www.openexr.com/

[78] Li Xi, Wang Aifeng, Jian Dasheng: *High-level Low Power Technique & Its Application* http://research.microsoft.com/asia/ur/workshop05/
downloads/Lixi%20low-power%20design.pdf

Es sei darauf hingewiesen, dass aufgrund der Schnelllebigkeit des Internets aufgeführte Links eventuell im Laufe der Zeit nicht mehr gültig sind. In diesem Fall wird empfohlen Suchmaschinen, wie http://www.google.de oder http://citeseer.ist.psu.edu/ zu benutzen, um die Dokumente wieder aufzufinden.

Lebenslauf

Ralf Hildebrandt wurde am 22. 03. 1978 in Bautzen geboren. Nach der Polytechnischen Oberschule besuchte er das Gymnasium in Bautzen und erreichte die allgemeine Hochschulreife am 21. 06. 1996. Nach einem Jahr allgemeinem Grundwehrdienst studierte er ab 01. 10. 1997 Elektrotechnik an der TU Dresden. Das Studium schloss er am 05. 12. 2002 erfolgreich als Diplomingenieur ab. Anschließend arbeitete er als Doktorand beim Fraunhofer-Institut für Photonische Mikrosysteme (IPMS) in Dresden Klotzsche.