Generierung von Prozessoren aus Instruktionssatzbeschreibungen

Ralf Dreesen
Institut für Informatik, Universität Paderborn
rdreesen@uni-paderborn.de

Abstract: Die Automatisierung der Prozessorentwicklung ist ein seit Langem untersuchtes Thema, das durch den zunehmenden Einsatz anwendungsspezifischer Prozessoren (ASIPs) in Ein-Chip-Systemen (SoCs) erheblich an Bedeutung gewonnen hat. Bei bisherigen Ansätzen werden Prozessoren auf mikroarchitektonischer Ebene beschrieben, wodurch ein Entwickler viele komplexe und fehleranfällige Aspekte definieren muss. Die Dissertation [Dre11] verfolgt einen bisher kaum untersuchten Ansatz, bei dem nur der Instruktionssatz eines Prozessors modelliert wird. Die Mikroarchitektur wird vollständig und automatisch durch neu entworfene Methoden erzeugt. Durch Variation eines Generatorparameters wird so ein Satz kompatibler Prozessoren mit verschiedenen physikalischen und dynamischen Eigenschaften erzeugt. Abhängig vom Einsatzgebiet kann aus diesem Entwurfsraum eine passende Implementierung ausgewählt werden.

1 Einleitung und Motivation

Zur Entwicklung anwendungsspezifischer Prozessoren werden heute Werkzeuge eingesetzt, die entweder eine Hardwarebeschreibung oder eine Prozessorbeschreibung verlangen. Bei einer *Hardwarebeschreibung* in Sprachen wie VHDL oder Verilog wird ein Prozessor auf mikroarchitektonischer Ebene definiert. Der Anwender muss sowohl Funktionseinheiten, als auch die Instruktions-Pipeline und ihre Kontrolle beschreiben. Letzteres ist dabei sehr zeitaufwendig und fehleranfällig.

Einzelne Instruktionen sind auf dieser Ebene der Beschreibung nicht mehr erkennbar. Die Semantik aller Instruktionen ist vereint und auf eine Vielzahl von Ressourcen verteilt. Eine Additionsinstruktion trägt zum Beispiel zum Decodierer, zur arithmetisch-logischen Einheit (ALU), zum Forwarding und zum Interlocking bei. Auf der anderen Seite vereint die ALU die Semantik aller arithmetischen und logischen Instruktionen. Der Instruktionssatz eines so beschriebenen anwendungsspezifischen Prozessors kann daher nur schwer nachträglich geändert werden.

Um diesen Problemen zu begegnen, bieten *Prozessorbeschreibungssprachen* wie zum Beispiel nMl [PLGG08] und Lisa [Inc08] spezielle Konzepte, welche helfen die Spezifikation eines Prozessors zu strukturieren und zu vereinfachen. Ein Prozessor wird in diesen Sprachen aber nach wie vor auf mikroarchitektonischer Ebene beschrieben. Da-

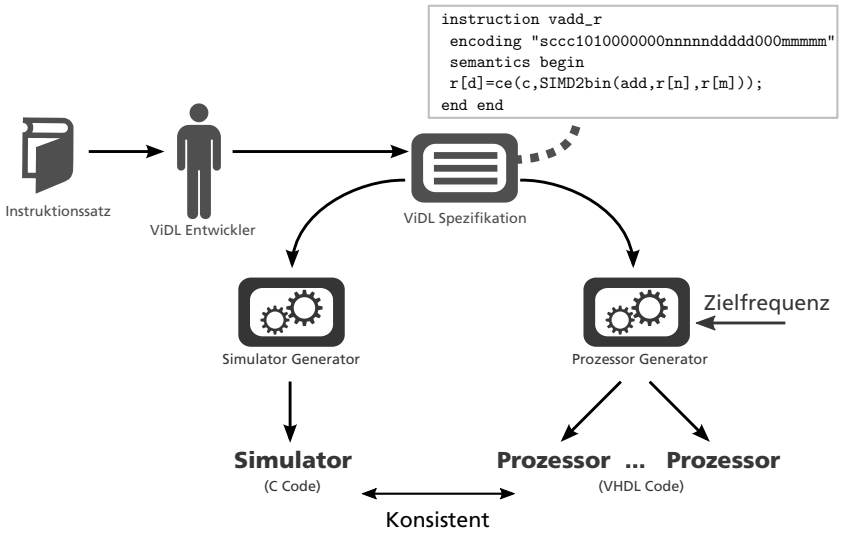

Abbildung 1: Überblick des Generator-Systems.

mit werden dynamische und physikalische Eigenschaften des Prozessors schon zu einem sehr frühen Zeitpunkt festgelegt. Eine späte Exploration dieses Entwurfsraums ist damit stark eingeschränkt. Zudem muss der Anwender selber Daten-, Kontroll- und Ressourcenkonflikte in der Instruktions-Pipeline auflösen. Diese Aufgabe ist fehleranfällig und ein aufwendiges Testen des Prozessors daher ratsam.

Im Gegensatz zu Hardwarebeschreibungen und Prozessorbeschreibungen modelliert eine *Instruktionssatzbeschreibung* in ViDL ausschließlich den Instruktionssatz eines Prozessors, nicht den Prozessor an sich. Dieser wird stattdessen, wie in Abbildung 1 dargestellt, generiert. Alle Aspekte der Mikroarchitektur werden dabei automatisch aus der Semantik des Instruktionssatzes und einer vorgegebenen Zielfrequenz des Prozessors hergeleitet. Dazu wurden im Rahmen der Arbeit eine Reihe von Analysen, Transformationen und Optimierungen entwickelt. Weitere Methoden wurden entworfen, um einen schnellen Instruktionssatzsimulator aus derselben Spezifikation zu erzeugen. Für die Sprache ViDL wurden domänenspezifische Sprachkonzepte entworfen, um Instruktionssätze prägnant und verständlich zu beschreiben.

2 Die Spezifikationssprache ViDL

Beim Entwurf der Sprache ViDL [Dre12b] wurden drei Ziele verfolgt. Erstes sollen typische Entwurfsabläufe für anwendungsspezifische Prozessoren unterstützt werden. Insbesondere die Exploration des Instruktionssatz-Entwurfsraums, sowie die Erweiterung

und Änderung bestehender Instruktionssätze ist von Relevanz. Zweitens sollen bewährte Richtlinien des Sprachentwurfs [Wat04, Seb09] befolgt werden, um eine hohe Sprachqualität zu erreichen. Eine Sprache die diese Richtlinien befolgt verringert typischerweise die Entwicklungszeit sowie die Fehleranfälligkeit und erhöht die Wartbarkeit. Drittens soll die Sprache mächtig genug sein, um realistische Instruktionssätze kompakt zu beschreiben. Eine Sprache, die hingegen nur eine kleine Klasse künstlicher Instruktionssätze abdeckt, würde sicherlich nicht vonseiten der Industrie akzeptiert. Um diese Ziele zu erreichen, wurde ein kleiner Satz kombinierbarer Sprachkonzepte so entworfen, dass sich auch die Eigenarten realistischer Instruktionssätze elegant beschreiben lassen. Diese Konzepte und Prinzipien werden im Folgenden kurz vorgestellt.

Die Semantik von Instruktionen wird in ViDL in einem Dialekt der funktionalen Sprache SML definiert. Eine ViDL Spezifikation ist dadurch frei von Seiteneffekten, was zum einen die Verständlichkeit erhöht und zum anderen die Fehleranfälligkeit verringert. ViDL unterstützt viele funktionale Konzepte, wie Funktionen höherer Ordnung (Funkionale), Lambda Ausdrücke, Polymorphie, Tupel und Closures. Die Konzepte ermöglichen einen hohen Grad an Wiederverwendbarkeit. So kann ein SIMD-Berechnungsmuster einfach als Funktional definiert werden. Die eigentliche arithmetische Operation einer konkreten SIMD-Instruktion wird dann wie in Abbildung 1 zu sehen in Form einer benannten Funktion oder eines Lambda Ausdrucks an das Funktional übergeben. Im Gegensatz zu anderen Ansätzen, kann das Prinzip SIMD also mit Sprachmitteln formuliert werden, was zu einem schlanken Sprachdesign beiträgt und Erweiterungen zulässt. Funktionen und Funktionale sind in der Regel polymorph, d.h. sie können auf beliebige n-Bit Daten angewendet werden, wodurch die Wiederverwendbarkeit weiter gesteigert wird. Zum Beispiel kann ein einmalig definierter Adressierungsmodus in mehreren Instruktionssätzen unterschiedlicher Bitbreite wiederverwendet werden.

Instruktionssätze verwenden häufig Strukturen wie virtuelle Adressräume, Sub-Wort Zugriffe auf Speicher und dedizierte Register für verschiedene Prozessormodi. Solche Strukturen sind generell Sichten auf physikalische Speicherelemente und I/O Schnittstellen. In ViDL werden solche Sichten durch architektonische Schnittstellen modelliert. Eine architektonische Schnittstelle beschreibt einen virtuellen Speicher mit einer definierten Anzahl an Elementen einer bestimmten Bitbreite. Die Relation zwischen virtuellen und physikalischen Elementen wird durch Abbildungen für Lese- und Schreibzugriffe spezifiziert. Die Relation selbst kann dynamisch sein, also zum Beispiel vom Prozessormodus abhängen. Architektonische Schnittstellen sind allgemein anwendbar und decken viele spezielle Konzepte anderer Sprachen ab. So zum Beispiel die Byte-Ordnung bei Speicherzugriffen, die sich bei manchen Ansätzen durch eine spezielle Direktive oder gar nicht definieren lässt.

Die meisten Instruktionssätze beinhalten Instruktionen, die Speicherelemente teilweise oder bedingt schreiben. Zum Beispiel ändert eine arithmetische Instruktion in der Regel nur bestimmte Bits des Status-Registers. Ein bedingter Sprungbefehl kann als ein bedingter Schreibzugriff auf den Programmzähler aufgefasst werden. Ähnliches gilt für die bedingte Befehlsausführung bei ARM und CoreVA. Um solches Verhalten einfach und

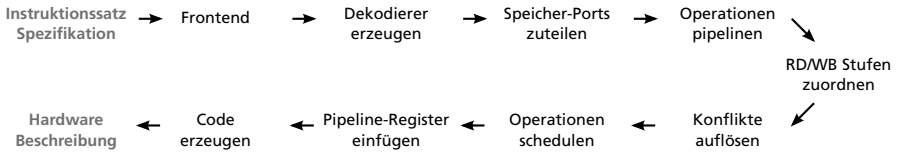

Abbildung 2: Überblick der Methoden zur Erzeugung eines Prozessors.

effektiv zu beschreiben, integriert ViDL eine dreiwertige Logik, ähnlich der Tri-State Logik für Buszugriffe. Anstelle des hochohmigen Zustandes "Z" drückt der Zustand "ϵ" jedoch aus, dass ein Bit eines Speichers unverändert bleibt. Epsilon-Logik harmoniert sehr gut mit ViDLs funktionalen Konzepten und kann sowohl in Funktionen, als auch in architektonischen Schnittstellen verwendet werden. Die Definition der bedingten Befehlsausführung bei ARM kann zum Beispiel in einer Funktion gekapselt werden. Beim CoreVA Instruktionssatz kann die bedingte Ausführung sogar auf SIMD-Ebene heruntergebrochen werden — eine Eigenschaft, die sich mit anderen Ansätzen nicht ausdrücken lässt.

ViDL operiert ausschließlich auf Bitketten, deren Breite nicht explizit durch den Entwickler definiert wird. Statt dessen wird die Bitbreite jeder Operation implizit durch den Generator hergeleitet. Dadurch wird die Spezifikation wesentlich vereinfacht und die Wartbarkeit erhöht. Zudem erhöht sich durch Polymorphie der Grad der Wiederverwendbarkeit. Um Bitbreiten herzuleiten, werden sie als Typen modelliert [DTK11]. ViDLs Typverband definiert dazu mehrere parametrisch polymorphe Typen, die in Subtyp-Relation stehen. Der Parameter ist dabei eine beliebige natürliche Zahl, die die Bitbreite beschreibt. Der Zusammenhang zwischen den Bitbreiten der Parameter und des Ergebnisses jeder Operation wird durch eine polymorphe Signatur beschrieben. Diese Form ist wesentlich prägnanter und verständlicher als zum Beispiel der Inferenz-Algorithmus der Hardware-Beschreibungssprache Verilog [Soc01]. Durch diese Modellierung können im Generator bewährte Methoden der Typinferenz angewendet werden, um die Parameter der polymorphen Typen, also die Bitbreiten, zu bestimmen. Widersprüchliche oder mehrdeutige Ausdrücke in ViDL führen dabei dank der statischen Typisierung zur Generierungszeit zu einem Typfehler. Dadurch wurde eine Mehrdeutigkeit im Pseudocode des ARM Handbuches aufgedeckt.

3 Methoden zur Prozessorgenerierung

ViDL abstrahiert stark von der Implementierungsebene eines Prozessors. Ein Prozessorgenerator bedarf daher vieler Analysen und Transformationen, die in diesem Abschnitt kurz vorgestellt werden.

Das Frontend des Generators führt die üblichen syntaktischen und statisch-semantischen

Analysen, wie Namensanalyse und Typanalyse durch. Zudem werden dort die meisten Sprachkonzepte in Datenfluss transformiert. Das Ergebnis ist eine Zwischendarstellung in Form eines Datenflussgraphen, der die Semantik aller Instruktionen vereint. Dieser Graph wird schrittweise durch viele Methoden transformiert, bis schließlich die Hardwarebeschreibung einer Prozessorimplementierung erzeugt wird. Abbildung 2 zeigt einen Überblick dieser Methoden, von denen im Folgenden einige kurz erläutert werden.

Eine Registerbank eines Prozessors hat in der Regel mehrere Lese- und Schreib-Ports, um gleichzeitig auf die Inhalte der Register zuzugreifen. Weil ViDL strikt von der Mikroarchitektur abstrahiert, wird dieser Aspekt nicht durch den Entwickler beschrieben, sondern durch den Generator hergeleitet. Dabei müssen den Zugriffen aus der Spezifikation so Ports zugeteilt werden, dass keine Ressourcenkonflikte bestehen und die Anzahl der Ports minimiert wird. Die Minimierung ist wichtig, weil Ports eine “teuere” Ressource in einer Hardware-Implementierung darstellen. Die in der Arbeit beschriebene Methode reduziert das Problem auf Graphfärbung und hat für alle evaluierten Instruktionssätze stets die minimale Anzahl an Ports erzeugt.

Um eine hohe Taktfrequenz zu erreichen, muss der Datenpfad eines Prozessors gepipelined werden. In der Dissertation werden zu diesem Zweck mehrere Methoden beschrieben, die auch erfolgreich implementiert wurden. Die Struktur der erzeugten Pipeline hängt dabei von drei Faktoren ab: dem Datenflussgraphen, der vorgegebenen Zielfrequenz und der Zieltechnologie. Letztere wird dabei durch eine Datenbank ihrer abgeschätzten Gatterlaufzeiten beschrieben. Bisher muss ein Entwickler all diese Faktoren berücksichtigen, um die Aspekte der Mikroarchitektur zu einem frühen Zeitpunkt selbst festzulegen. Diese Entscheidungen verlangen viel Erfahrung und eine gute Abschätzung von Signallaufzeiten. Die folgenden Methoden automatisieren diesen Prozess. Die Pipeline wird so konstruiert, dass die Zielfrequenz voraussichtlich erreicht wird, ansonsten aber die Pipeline-Tiefe und Instruktionslatenzen weitestgehend minimiert werden. Dadurch werden Ressourcen gespart, die andernfalls zu einem erhöhten Flächen- und Energiebedarf führen. Durch geringe Instruktionslatenzen wird der Instruktionsdurchsatz erhöht, bzw. die Anzahl der Zyklen pro Instruktion (CPI) verringert.

In einem ersten Schritt werden die Read und Write-Back Stufen der Pipeline separat für jedes Speicherelement bestimmt, also zum Beispiel für die freie (GP) Registerbank, das Status-Register und den Hauptspeicher. Die Methode berücksichtigt dabei die Gatterlaufzeiten der Operationen auf dem Datenpfad und die Zielfrequenz. Um unnötige Ressourcen für Bypässe zu vermeiden, versucht die Methode die Distanz zwischen Read und Write-Back Stufen zu minimieren. Für übliche Instruktionssätze wird nur ein Bypass für die GP Registerbank benötigt. Bei einer geringen Zielfrequenz fällt selbst dieser in der Regel weg. Die Mikroarchitektur wird also genau auf die Instruktionssemantik und die Zielfrequenz zugeschnitten.

Für den Fall, dass die Write-Back Stufe nicht der Read Stufe folgt, können Datenkonflikte auftreten. Diese Konflikte werden im generierten Prozessor wenn möglich durch Forwarding und andernfalls durch Interlocking behoben. Die Methode zur Erzeugung des Forwardings analysiert dabei, unter welchen Umständen ein Ergebnis bereits zu einem

früheren Zeitpunkt bekannt ist. Sie basiert auf einer Analyse kritischer Pfade im Datenflussgraphen, einer Analyse partiell-definierter Ausdrücke und einem Satz von Termersetzungregeln, um den Datenflussgraphen zu normalisieren. Die Methode liefert für jede Stufe zwischen Read und Write-Back das frühzeitige Ergebnis und eine Signalleitung, mit der das Interlocking gesteuert wird. Für den Sonderfall, dass ein Ergebnis nie frühzeitig bekannt ist, entfällt der entsprechende Bypass automatisch.

In ähnlicher Weise wird bestimmt, wann ein Kontrollflusswechsel durch einen (bedingten) Sprung ausgelöst wird. Zudem werden Steuersignale erzeugt, um spekulativ ausgeführte Instruktionen im Fall einer falschen Sprungvorhersage abzubrechen. Dabei ist sichergestellt, dass solche Instruktionen stets abgebrochen werden können, also noch keinen Einfluss auf den Prozessorzustand genommen haben.

Die vorgestellten Methoden kapseln das Expertenwissen eines Schaltungstechnikers. Der Entwurfsablauf ist dadurch vollständig automatisiert, ein Entwickler braucht also keinen Aspekt der Mikroarchitektur beisteuern. Wie die Auswertung im nächsten Abschnitt zeigt, treffen die Methoden gute Entscheidungen, die sich in den physikalischen und dynamischen Eigenschaften des erzeugten Prozessors widerspiegeln.

4 Evaluation

Zur Evaluation der Sprache ViDL wurden vier praktische Instruktionssätze (ARM, MIPS, Power und CoreVA) und zwei akademische Instruktionssätze (OISC und SRC) spezifiziert. Dank ViDLs mächtiger Konzepte, wie Epsilon-Logik, architektonischer Schnittstellen und Typinferenz, konnten die Instruktionssätze weitestgehend beschrieben werden. Das schließt Instruktionen mit ungewöhnlichen Bitbreiten, verzögerten Effekten und anspruchsvollen Adressierungsmodi ein. Durch die starke Abstraktion der Sprache und bewährte funktionale Konzepte zur Kapselung gemeinsamen Verhaltens, konnten die Instruktionssätze effizient beschrieben werden. So lag der Zeitaufwand eines erfahrenen Entwicklers zwischen 90 Minuten für SRC und einem Monat für ARM. Unerfahrene Benutzer (Studenten) haben den CoreVA und Power Instruktionssatz in 2 bzw. 3 Monaten spezifiziert. Dank der einfachen Struktur der Sprache, konnten sie sich schnell einarbeiten.

ViDL eignet sich gut zur Exploration des Entwurfsraumes eines Instruktionssatzes. Zur Evaluation wurden Instruktionssätze in kurzer Zeit erweitert und geändert. Zum Beispiel wurde eine bestehende 32-Bit ARM Spezifikation nachträglich in eine generische n-Bit Spezifikation überführt, wobei n ein statischer Parameter ist. Zwei Stellen in der Spezifikation, die dabei erweitert werden mussten, wurden durch die Typanalyse des Generators automatisch identifiziert.

Der ebenfalls generierte Instruktionssatzsimulator erreicht für übliche Instruktionssätze im Mittel eine Geschwindigkeit von 60 Mips auf einem typischen 3 GHz Rechner. Selbst der Simulator eines 256-Bit breiten ARM Instruktionssatzes erreicht noch eine Geschwin-

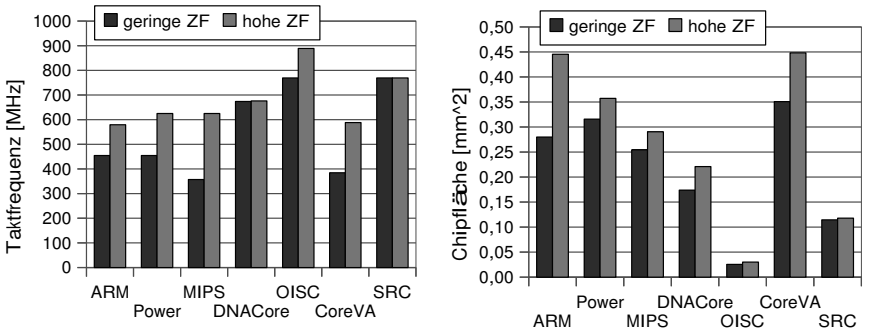

Abbildung 3: Taktfrequenz und Flächenbedarf generierter Prozessoren für verschiedene Zielfrequenzen (ZF).

digkeit von ca. 20 Mips. Die Evaluation hat gezeigt, dass die Optimierungen im Generator einen erheblichen Einfluss auf die Geschwindigkeit des Simulators haben.

Um nachzuweisen, dass die Methoden des Generators gute Prozessoren erzeugen, wurden für 13 Instruktionssätze (einschließlich Varianten) insgesamt 83 Prozessoren generiert. Für jeden Instruktionssatz wurden ca. 6 Prozessoren für unterschiedliche Zielfrequenzen erzeugt. Die Prozessoren werden automatisch generiert, also ohne Beitrag des Entwicklers. Die so erzeugten Prozessoren in Form von VHDL-Code wurden für eine Standardzellentechnologie[1] synthetisiert[2].

In Abbildung 3 ist die erreichbare Taktfrequenz und der Flächenbedarf generierter Prozessoren für verschiedene Instruktionssätze und zwei Zielfrequenzen (ZF) zu sehen. Mit steigender Zielfrequenz wird der Datenpfad auf eine zunehmende Anzahl an Pipeline-Stufen verteilt. Der kritische Pfad wird verringert und die erreichbare Frequenz des Prozessors steigt. Allerdings erhöht sich damit auch der Bedarf an Ressourcen (Chipfläche), zum Beispiel für Bypässe und Pipeline-Register. Es fällt auf, dass die DNACore und SRC Prozessoren auch für eine geringe Zielfrequenz schon eine hohe Taktrate erzielen. Beide Instruktionssätze enthalten keine Multiplikation, wodurch auch bei einer sehr einfachen Pipeline Struktur der kritische Pfad eine geringe Signallaufzeit hat.

Die Syntheseergebnisse liegen in etwa in der Größenordnung einer Handimplementierung. Für diesen Vergleich wurden generierte CoreVA Prozessoren mit einer optimierten VHDL Implementierung eines erfahrenen Schaltungstechnikers verglichen. Bei gleicher Taktfrequenz (ca. 350 MHz) hat der generierte Prozessor eine ca. 40% höhere Leistungsaufnahme bei doppeltem Flächenbedarf. Der Entwicklungsaufwand eines unerfahrenen Benutzers liegt mit zwei Monaten für ViDL jedoch deutlich unter dem Aufwand von einem Jahr für die VHDL Implementierung. Hinzu kommt, dass die Konfliktauflösung des

[1] 65 nm ST Microelectronics Low Power für Worst-Case ($1.1V$; $125°C$)
[2] Cadence RTL Compiler

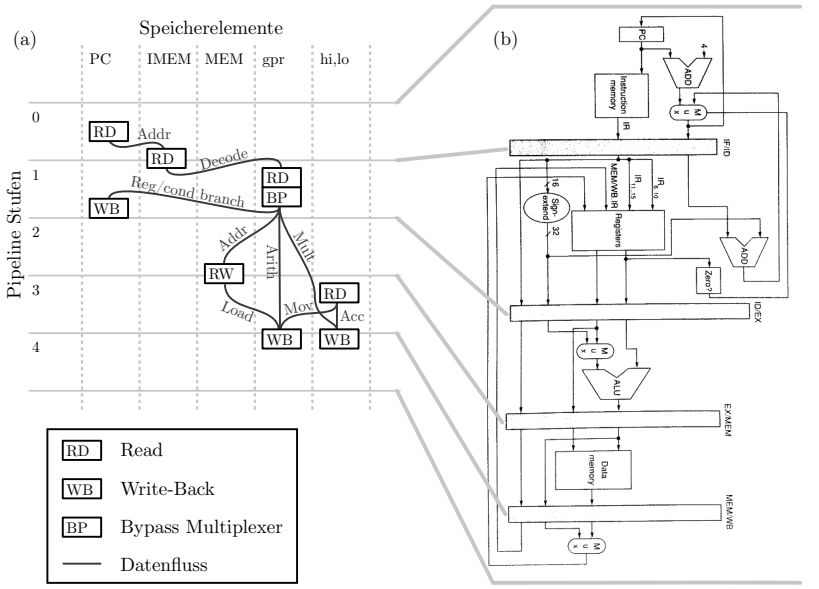

Abbildung 4: Vergleich der (a) generierten MIPS-Pipeline für eine Zielfrequenz von 400 MHz mit (b) der Pipeline aus dem Buch *Computer Architecture* [HP06].

generierten Prozessors nicht getestet[3] werden braucht, weil sie automatisch erzeugt wurde.

Die Struktur generierter Pipelines wird in der Dissertation am Beispiel des MIPS Instruktionssatzes eingehend untersucht. Abhängig von der Zielfrequenz ergeben sich dabei Pipeline-Tiefen zwischen zwei und sieben Stufen, wobei ein Prozessor mit zwei Pipelinestufen einem Prozessor ohne Pipelining entspricht. Wie in Abbildung 4 zu sehen ist, entspricht für eine Zielfrequenz von 400 MHz die automatisch hergeleitete Pipeline jener, die durch Hennessy und Patterson [HP06] beschrieben wird. Folglich ist das dynamische Verhalten (CPI) gleich. Wie üblich, haben nur Lade-Instruktionen und einige Instruktionen im Umfeld der Multiplikation eine Latenz von einem Zyklus. Durch eine geschickte Organisation der Pipeline kommen Multiply-Accumulate (MAC) Instruktionen sogar ohne Latenz aus. Auch diese Eigenschaft wird automatisch erzielt, ohne dass ein Entwickler Aspekte der Mikroarchitektur vorgeben muss. Falsch vorhergesagte Sprünge erzeugen in der hergeleiteten Pipeline zwei Strafzyklen, korrekt vorhergesagte unterbrechen die Ausführung hingegen nicht. Insgesamt setzt der generierte Prozessor die hohe Taktfrequenz also auch in einen hohen Instruktionsdurchsatz um. Für geringere Zielfrequenzen werden kürzere Pipelines erzeugt und das CPI verbessert sich entsprechend. So können für eine hergeleitete dreistufige MIPS-Pipeline keine Konflikte mehr auftreten. Alle In-

[3] Unter der Annahme, dass der Prozessorgenerator korrekt ist.

struktionen haben also ein CPI von 1 und falsch vorhergesagte Sprünge verursachen keine Strafzyklen. Zudem werden bei dieser Pipeline automatisch alle Bypässe und Steuerleitungen weggelassen, wodurch sich die Implementierung erheblich vereinfacht.

Um die Stärken des Systems zu demonstrieren, wurde im Rahmen der Dissertation der DNACore [Dre12a] Prozessor entwickelt. Der DNACore Instruktionssatz ist eine Erweiterung des MIPS Instruktionssatzes zur Beschleunigung des Smith-Waterman Algorithmus. Dieser Algorithmus wird in der Bioinformatik eingesetzt, um DNA, RNA und Proteinsequenzen zu vergleichen. Der Algorithmus berechnet eine sogenannte Scoring-Matrix, wodurch er eine hohe Komplexität hat, jedoch inhärent datenparallel ist. Diese Parallelität wird durch einen Satz anwendungsspezifischer SIMD-Instruktionen ausgenutzt. Zudem werden viele Speicherzugriffe durch einen kleinen Satz interner Register vermieden. Zusammen bilden SIMD-Instruktionen und interne Register konzeptuell ein Systolisches-Array, durch das ein Streifen der Matrix gepumpt wird. Die Spezifikation der Erweiterung in ViDL hat nur einen halben Tag gedauert. In der Praxis wird eine Auslastung der SIMD-Einheit von 97% erreicht, d.h. es werden in einem Takt durchschnittlich 3.8 Elemente der Scoring-Matrix berechnet. Zusammen mit den Syntheseergebnissen des generierten Prozessors ergibt sich so eine Geschwindigkeit von $2.6\,\mathrm{GCUPS}$[4] bzw. eine Energieeffizienz von $58\,\mathrm{GCUPS}/W$. Letztere liegt erheblich über den Ergebnissen alternativer Ansätze.

5 Zusammenfassung und Ausblick

In der Dissertation wurde gezeigt, wie effiziente Prozessoren aus einer abstrakten Instruktionssatzspezifikation automatisch generiert werden können. Im Gegensatz zu bestehenden Ansätzen abstrahiert die Sprache ViDL konsequent von der mikroarchitektonischen Ebene. Dadurch wird zum einen die Entwicklung erheblich vereinfacht und zum anderen die Fehleranfälligkeit in einem sensiblen Bereich deutlich gesenkt. Zudem können konsistente Simulatoren und Prozessoren mit sehr unterschiedlichen physikalischen und dynamischen Eigenschaften ohne zusätzlichen Aufwand aus derselben Instruktionssatzspezifikation generiert werden. Dies ist bei bestehenden Ansätzen in dieser Form nicht möglich. Zur Herleitung der Mikroarchitektur wurde Expertenwissen aus der Schaltungstechnik generalisiert und in Methoden gekapselt. Die Wirksamkeit der Methoden wurde an praktischen Instruktionssätzen belegt.

Es ist geplant, ViDL in Zukunft in zwei Richtungen voranzutreiben. Zum einen soll der Prozessorgenerator um weitere Optimierungen ergänzt werden. Vor allem Hardware-Sharing birgt hier ein bedeutendes Optimierungspotenzial. Zum anderen sollen Generatoren für Übersetzerwerkzeuge entwickelt werden. Ich habe bereits solche Generatoren für eine andere Beschreibungssprache mitentwickelt und schätze, dass sich einige der Konzepte gut übertragen lassen. Ein Entwurf für eine Erweiterung ViDLs wurde bereits fertiggestellt.

[4] Billion Cell Updates per Second

Literatur

[Dre11] Ralf Dreesen. *Generating Processors from Specifications of Instruction Sets*. Dissertation, University of Paderborn, Germany, 2011.

[Dre12a] Ralf Dreesen. DNACore: An Application Specific Processor for the Smith-Waterman Algorithm. In *39th International Symposium on Computer Architecture (ISCA 2012)*, 2012. (submitted).

[Dre12b] Ralf Dreesen. ViDL: A Versatile ISA Description Language. In *19th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS-19)*, April 2012. (accepted for publication).

[DTK11] Ralf Dreesen, Michael Thies und Uwe Kastens. Type Analysis on Bitstring Expressions. In *Proceedings of the 9th Workshop on Optimizations for DSP and Embedded Systems (ODES-9)*, April 2011.

[HP06] John Hennessy und David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2006.

[Inc08] CoWare Inc. *LISA Language Reference Manual*, 2008.

[PLGG08] Johan Van Praet, Dirk Lanneer, Werner Geurts und Gert Goossens. nML: A Structural Processor Modeling Language for Retargetable Compilation and ASIP Design. In Prabhat Mishra und Nikil Dutt, Hrsg., *Processor Description Languages*, Seiten 65–93. Morgan Kaufmann, 2008.

[Seb09] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley Publishing Company, USA, 9th. Auflage, 2009.

[Soc01] IEEE Computer Society. *IEEE Std 1364-2001 - IEEE Standard Verilog Hardware Description Language*. The Institute of Electrical and Electronics Engineers, Inc, 2001.

[Wat04] David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons, 2004.

Ralf Dreesen wurde am 20. Oktober 1980 in Soest geboren. Nach dem Abitur studierte er von 2001 bis 2006 in Paderborn Informatik mit Nebenfach Elektrotechnik. Im Anschluss, arbeitete er als Doktorand am Lehrstuhl für Programmiersprachen und Übersetzer von Professor Kastens, wo er 2011 mit Auszeichnung promovierte. Neben der Arbeit an seiner Dissertation entwickelte er in dieser Zeit im Rahmen einer Kooperation mit Infineon eine vollständige Compiler-Werkzeugkette für den CoreVA Mobilprozessor, die heute intensiv genutzt wird. Seine Forschungsinteressen liegen sowohl im Bereich des Übersetzerbaus, als auch im Gebiet der Schaltungstechnik.