der Bundeswehr
Universität München

Universeller Virtueller Computer
Weiterentwicklung der Spezifikation zur effizienteren und vielseitigeren Verwendung in der Langzeitarchivierung

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inf. Nico Krebs
im Juli 2012

Tag der mündlichen Prüfung:	14. Dezember 2012
Vorsitzender der Kommission:	Prof. Dr. Peter Hertling
Betreuer und 1. Berichterstatter:	Prof. Dr. Uwe M. Borghoff
2. Berichterstatter:	Prof. Klaus Buchenrieder, Ph.D.
1. Prüfer:	Prof. Dr. Gunnar Teege
2. Prüfer:	Prof. Dr. Stefan Pickl

Universität der Bundeswehr München
Fakultät für Informatik

Für meine Familie,
ohne deren Rückhalt ich diese Arbeit nicht so hätte fertigen können, dass ich – so wie jetzt –
rundum zufrieden damit bin.

Zusammenfassung

Im Bereich der Langzeitarchivierung gibt es drei wesentliche Strategien: Migration, Emulation und Konservierung obsoleter Hardware. Die Migration stellt den zukünftigen Zugriff auf archivierte Daten durch fortgesetzte Konvertierung in jeweils aktuelle Formate sicher. Die Emulation versucht, mittels Software obsolete Maschinen nachzubilden und so alte Programme zur Darstellung archivierter Daten zu erhalten. Schließlich kann die Konservierung obsoleter Hardware helfen, übergangsweise auf archivierte Daten zuzugreifen, bevor diese Daten einer der ersten beiden Strategien zugeführt werden können. Ein ergänzender Ansatz wurde 2001 von Lorie vorgestellt. Er entwarf den Universellen Virtuellen Computer, der so einfach gehalten ist, dass er in kurzer Zeit – auch in ferner Zukunft – allein aus seiner Spezifikation heraus implementierbar sein sollte. Dadurch können heute Programme entwickelt werden, die zusammen mit dieser Spezifikation Jahrhunderte im Archiv überstehen können, ohne fortlaufend an neue Standards oder Formate angepasst werden zu müssen. Diese Programme können den Zugriff auf die archivierten Informationen für einen sehr langen Zeitraum gewährleisten.

Durch eine Machbarkeitsstudie hat Lorie belegt, dass sein Konzept in der heutigen Zeit funktioniert. Die vorliegende Arbeit geht an dieser Stelle deutlich weiter. Die Frage, ob das Konzept des UVC wirklich universell ist und einige Jahrhunderte später noch funktioniert, wird wie folgt ergründet: In der vorliegenden Arbeit wird zunächst eine Referenzimplementierung des UVC in einer anderen als den von den Entwicklern genutzten Programmiersprachen für heutige Systeme erstellt und die dabei gesammelten Erfahrungen sehr genau dokumentiert. Dadurch ergeben sich erste Hinweise auf Ungenauigkeiten und mögliche Fehlinterpretationsmöglichkeiten der Spezifikation. Eine Zeitreise in die Mitte des letzten Jahrhunderts – mit Hilfe der datArena – widmet sich der Universalität des Ansatzes und wird zeigen, ob und ggf. wo die Spezifikation hinsichtlich dieses Ziels weiter verbessert werden sollte. Zudem liefert diese erste empirische Evaluation für Archivare und Softwareentwickler gleichermaßen wichtige Daten und Erfahrungen zum Implementierungs- und Portierungsaufwand.

Die Entwicklung eigener Tools und komplexer Software für den UVC ermöglicht die Bewertung der Praxistauglichkeit und zeigt weitere sinnvolle Erweiterungen und Anpassungen der Spezifikation im Hinblick auf Effizienz und den bislang völlig außer Acht gelassenen Aspekt der Interaktivität.

Diese Arbeit identifiziert Schwächen der bestehenden Spezifikation und überwindet diese mit einer Weiterentwicklung der Spezifikation des Universellen Virtuellen Computers.

Inhaltsverzeichnis

1. Einführung

In einer Zeit der fortschreitenden Digitalisierung in nahezu allen Bereichen gibt es neben dem Wunsch nach schnellem Zugriff auf Informationen auch den Anspruch, diesen Zugriff dauerhaft für nachfolgende Generationen zu erhalten. Mit diesem Aspekt beschäftigt sich das Forschungsgebiet der Langzeitarchivierung digitaler Objekte. Die bisher gefundenen Ansätze sind jedoch nur eingeschränkt praktikabel. Aufgrund ihrer Vor- und Nachteile sind sie nur für bestimmte Klassen digitaler Objekte und auch nur für bestimmte Zeitspannen geeignet.

Diese Dissertation befasst sich mit einem speziellen Werkzeug zur Langzeitarchivierung digitaler Objekte: Dem Universellen Virtuellen Computer (UVC). Der sich auf diesen UVC stützende Ansatz der Langzeitarchivierung wurde mit dem Ziel entwickelt, statische digitale Objekte wie Bild-, Videodateien, eBooks oder Kalkulationstabellen über ein extrem langen Zeitraum unverändert zu bewahren.

Ziel der vorliegenden Arbeit ist die eingehende Überprüfung der Tauglichkeit dieses Ansatzes. Aufgrund der vorhandenen technischen Gegebenheiten der Universität der Bundeswehr in München, insbesondere der umfangreichen Sammlung funktionsfähiger Hardware (datArena), konnte sie weit über das hinausgehen, was den Entwicklern möglich war.

Dieses einführende Kapitel wird zunächst die Notwendigkeit der Langzeitarchivierung digitaler Objekte über sehr lange Zeiträume zu unvorhersehbaren Bedingungen motivieren. Der zweite Abschnitt erarbeitet Kriterien, die der UVC als Werkzeug, aber auch der Ansatz als Ganzes erfüllen muss, damit er tatsächlich zum Einsatz kommen kann. Aus diesen Kriterien leitet sich die Zielsetzung der Arbeit ab.

1.1. Motivation

Das folgende Zitat stammt von Kritias, einer Dialogfigur Platons. Eher als Fiktion denn tatsächliche Überlieferung der Sage von Atlantis belegt es eindrucksvoll, wie bewusst man die Probleme der Langzeitarchivierung bereits in der Antike sah:

> *Von diesen sind die Namen erhalten, ihre Thaten aber wegen des Unterganges Derer, die sie von ihnen überkamen, und der Länge der Zeit in Vergessenheit gerathen. Denn das jedesmal übrigbleibende Geschlecht pflegt, wie schon früher bemerkt wurde, das auf den Bergen lebende und der Schrift unkundige zu sein, welches bloß die Namen der Herrscher im Lande gehört hat und dazu etwas Weniges von ihren Thaten. Sie mußten sich daher damit begnügen, ihren Nachkommen diese Namen zu überliefern; die Tugenden und die Staatseinrichtungen ihrer Vorfahren aber kannten sie nicht, es sei denn einige dunkle Gerüchte über Einzelnes, und da sie überdies zusammt ihren Abkömmlingen viele Geschlechter hindurch*

> *an dem Nothwendigen Mangel litten und daher vielmehr auf die Ausfüllung dieses Mangels ihren Sinn richten mußten, so sprachen sie auch vielmehr hierüber mit einander und vernachlässigten das einst bei ihren Vorfahren und vor Alters Geschehene. Denn die Erzählung alter Sagen und die Erforschung der Vorzeit tritt erst mit der Muße in den Staaten ein, wenn sie die Sorge um die Nothdurft des Lebens bei Manchen als eine schon überwundene vorfindet, und nicht früher. Darum also sind uns die Namen der Alten ohne ihre Thaten erhalten geblieben.* (Übersetzung: Susemihl [1857])

Von Atlantis ist nur der Name überliefert. Sämtliche Zeugnisse gingen zusammen mit dem Wissen „unter“ oder gerieten aufgrund der Sorge um die *Nothdurft des Lebens* in Vergessenheit.

Losgelöst von Naturkatastrophen wie Erbeben oder Sintfluten steht hier die Erkenntnis, dass die „Erforschung der Vorzeit“ und der Erhalt des kulturellen Erbes nur in einem Staat möglich sind, der ausreichend Ressourcen dafür übrig hat. Diese Erkenntnis ist wenigstens 2500 Jahre alt. Demographische Entwicklungen, Klimawandel, große Naturkatastrophen und auch Finanzkrisen sind nur einige Ursachen, die bisher eingeschlagene Wege der Langzeitarchivierung schnell unbezahlbar machen und das digitale Erbe an sich bedrohen könnten.

Eine praktikable Archivierungsmethode, die diesen Aspekt berücksichtig, fehlt derzeit völlig. Der den UVC nutzende Ansatz ist grundsätzlich geeignet, diese Lücke zu schließen, verdient aber nach wie vor nur „Proof of Concept“ Status.

Dieser Ansatz verlagert den Großteil des Aufwands der Archivierung auf das Einstellen und auf den Zugriff. In der womöglich sehr langen Zeit dazwischen müssen lediglich die einzelnen Bits erhalten werden, was kostengünstig und über Generationen hinweg auch ohne weiteres Zutun möglich wäre.

Der UVC wird in diesem Zusammenhang deutlich zu wenig beachtet. Eine Forschergruppe um Raymund A. Lorie bei IBM Research hat diesen Ansatz in Kooperation mit der Koninklijke Bibliotheek der Niederlande (KB) verfolgt. Innerhalb dieser Kooperation ist der Ansatz anhand zweier Anwendungen für bestimmte Bilddateien einmal erfolgreich getestet worden; zum praktischen Einsatz kam er nach diesem Machbarkeitstest m. W. nicht.

Diese Arbeit setzt genau hier an. Zum einen bietet das Institut für Softwaretechnologie an der Universität der Bundeswehr eine hervorragende Basis, diesen Ansatz mit der angebrachten Distanz wissenschaftlich zu untersuchen, und zum anderen bietet die datArena einzigartige technischen Voraussetzungen.

1.2. Zielsetzung

Ein Werkzeug wie der UVC wird erst dann genutzt, wenn eine ausreichende Vertrauensbasis geschaffen und die Methode publik gemacht wurde. Der Ansatz ist noch neu; lediglich in einem sehr eng angelegten Szenario hat der UVC seine Tauglichkeit beweisen können.

Zur Etablierung des UVC Ansatzes müssten vor allem jene Softwarehersteller überzeugt werden, die Anwendungen erstellen, die wiederum archivierungswürdige Dateiformate bereitstellen. Denn für alle Formate potentieller Archivalien muss je eine Anwendung für den

UVC erstellt werden. Mit der zielgerichteten Spezifikation des Formats PDF/A[1] müsste auch eine UVC-Anwendung entwickelt werden, die den Zugriff auf die enthaltenen Informationen sicherstellt. Vertrauen die Nutzer dem UVC Ansatz, könnte dieser Mehraufwand durchaus werbewirksam genutzt und letztendlich auf den Kunden umgelegt werden.

Abseits dieser wirtschaftlichen Aspekte untersucht diese Arbeit kritisch die für einen erfolgreichen Einsatz notwendig erscheinenden Eigenschaften des UVC Ansatzes, die im Folgenden abgeleitet werden.

Der UVC ist virtuell und somit Software. Aufgrund ihrer Systemabhängigkeit kann Software nur in der Zeit überdauern, in der ein geeignetes Trägersystem bereit steht. Der UVC wird daher nicht als Software an zukünftige Generation weitergegeben, sondern in Form einer Spezifikation. Diese Spezifikation muss aus sich heraus verständlich sein, um dem Anspruch sehr langfristiger Archivierung zu genügen. Diese Eigenschaft lässt sich am glaubwürdigsten von einem bisher unbeteiligten Dritten überprüfen. Hierbei gefundene Ungenauigkeiten, Widersprüche oder gar Fehler in der Spezifikation helfen, die Spezifikation zu präzisieren und Fehler zu beseitigen.

Der Zugriff in ferner Zukunft gelingt nur, wenn ein UVC nach der Spezifikation erstellt werden kann. Damit das gelingt, muss der UVC universell spezifiziert sein. Im Kontext der vorliegenden Arbeit ist ein virtueller Computer dann universell, wenn keine seiner Eigenschaften, unabhängig vom Zeitpunkt und des genutzten Basissystems, seine Erstellung verhindert, zu stark verzögert oder zu aufwendig macht. Diesem bereits im Namen verankertem Aspekt der Universalität widmet sich ein Großteil dieser Arbeit. Zu prüfen ist, ob sich der UVC für verschiedene Hardwarearchitekturen mit vergleichbarem Aufwand schnell implementieren lässt. In diesem Zusammenhang entstehen für Archivare wichtige Richtwerte für „schnell."

Neben der Erstellung des UVC ist der Aspekt der Anwendungserstellung von besonderer Bedeutung. Hier ist zu prüfen, ob sich Anwendungen für den UVC mit vertretbarem Aufwand erstellen lassen und ob die spezifizierte Funktionalität des UVC ausreicht, um auch komplexere Anwendungen mit „erträglichen" Laufzeiten zu erstellen. Ist der Aufwand zur Entwicklung zu hoch, ist die Akzeptanz dieser Methode gefährdet.

Eine von den Entwicklern getroffene Einschätzung besagt, dass Anwendungen für den UVC nicht performant sein müssten. Sie berufen sich dabei auf immer schneller werdende Prozessoren und vertrauen hierbei auf eine Interpretation von „Moore's Law," nach der sich die Geschwindigkeit der Computer alle 18 Monate verdoppelt. Diese Arbeit durchzieht daher die Fragestellung, ob Performanz durch geschickte Implementierung des UVC erreicht werden kann oder ob diese implizit durch die Spezifikation ausgeschlossen ist.

Diese Arbeit untersucht daher in mehreren Experimenten folgende Fragestellungen:

- Ist die Spezifikation aus sich heraus verständlich und vollständig?
- Gilt Universalität (schnelle, systemunabhängige Implementierbarkeit)?
- Ist eine leichte und vollständige Programmierbarkeit gegeben?
- Ist eine angemessen schnelle Programmausführung möglich?

[1] ISO-19005-1 – Document management - Electronic document file format for longterm preservation - Part 1: Use of PDF 1.4 (PDF/A-1).

Als Ergebnis erarbeitet diese Dissertation eine weiterentwickelte Spezifikation, wobei gefundene Fehler ausgebessert und identifizierte Unklarheiten und enthaltene Systemabhängigkeiten beseitigt werden. Zudem umfasst die Weiterentwicklung eine Verbesserung der Teile der Spezifikation, die für die langsame Programmausführung verantwortlich sind.

Als Zielgruppe werden bewusst Archivare und in zweierlei Hinsicht Softwareentwickler angesprochen. Archivare bekommen einen kritischen Einblick in die den UVC nutzende Methode und zudem wichtige Richtwerte für die Implementierungszeit an die Hand, die Aufwand und Kosten für ein Archiv abschätzbar werden lassen. Softwareentwickler finden im Anhang eine Übersicht über effiziente Implementierungsmöglichkeiten, die innerhalb der Arbeit dargestellt wurden. Die erarbeitete umfangreiche Testsuite hilft bei der Implementierung eines spezifikationskonformen UVC. Entwickler von UVC-Anwendungen hingegen finden Hinweise darauf, wie sich bewährte Konstrukte in UVC-Assemblersprache realisieren lassen. Dies erleichtert den Einstieg und eröffnet den Weg zu komplexeren Anwendungen.

1.3. Aufbau der Arbeit

Die Arbeit besteht wie folgt aus vier Teilen.

Der erste Teil der Arbeit vermittelt die Grundlagen und verortet die bereits in Ansätzen beschriebene Archivierungsmethode innerhalb der Langzeitarchivierung. Hierfür werden zunächst die Archivierungsmethoden Migration und Emulation vorgestellt. Über deren Vor- und Nachteile motiviert sich der Ansatz mittels UVC, dessen detaillierte Darstellung folgt. Die damit geschaffenen Grundlagen dienen auch der Konkretisierung o.g. Fragestellungen. Eine abschließende Abgrenzung gegenüber anderen virtuellen Maschinen motiviert die Entwicklung des UVC.

Der zweite und umfangreichste Teil belegt, dass verschiedene Aspekte bei der Implementierung des UVC eine Rolle spielen. Dieser Teil ist von Darstellungen verschiedener Experimente geprägt. Im ersten wird eine Referenzimplementierung des UVC für ein aktuelles System erstellt, um die Verständlichkeit und Vollständigkeit der Spezifikation zu ergründen. Das sich daran anschließende, sehr umfangreiche Experiment umfasst die Implementierung auf verschiedene Hardwarearchitekturen vergangener Jahrzehnte und trägt im Rahmen der vorliegenden Arbeit das Synonym „Zeitreise.“ Die Frage nach der Universalität steht hierbei im Vordergrund. Ein abschließendes Experiment hinterfragt den interpretierenden Ansatz und wendet als Alternative den Crosscompiler-Ansatz auf UVC-Anwendungen an. Über den gesamten Teil hinweg finden sich Hinweise, wie sich ein UVC performant implementieren lässt.

Der dritte Teil beschäftigt sich mit der Anwendungsentwicklung für den UVC. Anhand verschiedener Anwendungen, wie einer Turing-Maschine, eines Assemblers, einer umfangreichen Testsammlung und eines Spiels, wird die Vollständigkeit und Nutzbarkeit des Befehlssatzes untersucht. Mit dem interaktiven Spiel wird zudem experimentell untersucht, inwiefern der UVC abseits statischer Objekte nutzbar ist.

Der vierte und letzte Teil fasst alle Ergebnisse bezüglich Integrität, Universalität und Performanz zusammen und motiviert so die weiterentwickelte Spezifikation. Darüber hinaus finden sich hier weiterführende Überlegungen, die im Rahmen dieser Arbeit nicht mehr verfolgt werden konnten.

Teil I.

Grundlagen

2. Langzeitarchivierung digitaler Objekte

Der UVC wurde speziell für den Einsatz innerhalb der Langzeitarchivierung entwickelt. Sowohl die Methode der fortwährenden Formatüberführungen (Migration) als auch die der Nachahmung obsoleter Hardware (Emulation) haben ihre Vor- und Nachteile. Beide Methoden werden hier vorgestellt. Unter dem nicht ganz passenden Begriff des Computermuseums wird in diesem Zusammenhang auch die Funktion einer Sammlung obsoleter, aber noch funktionsfähiger Hardware als wichtiger Baustein innerhalb der Langzeitarchivierung beschrieben. Die den UVC nutzende Methode braucht diesen Baustein nicht. Jedoch nutzt diese Arbeit in nicht unerheblichem Maße eine solche Sammlung, um in einem Experiment die Universalität zu untersuchen.

2.1. Bitstromkonservierung

Ein Bitstrom ist eine feststehende Folge von Nullen und Einsen. Bei der Konservierung kommt es auf die Bereitstellung dieser Bits in unverfälschter Folge an. Sowohl die Migration als auch die Emulation bauen auf einer funktionierenden Bitstromkonservierung auf. Im Kontext der Langzeitarchivierung oft zitierten OAIS-Referenzmodell wird die Migration in vier Bereiche eingeteilt [CCSDS, 2002]. Die ersten drei sind der Bitstromkonservierung gewidmet: *Refreshment*, *Replication* und *Repackaging*. Das *Refreshment* dient dabei dem Erhalt der Lesbarkeit des Bitstroms auf dem Datenträger. Die *Replication* sorgt dagegen durch regelmäßiges Umspeichern auf neuere und eventuell auch aktuellere Speichermedien für den Erhalt der einzelnen Bits. Jedes Speichermedium ist in seiner Lebensdauer begrenzt. Die in Archiven oft genutzten CDs erreichen selbst unter idealen Bedingungen nur eine Haltbarkeit von wenigen Jahren [Righi, 2008; Slattery et al., 2004; Byers, 2003]. Mit dem Umspeichern geht ggf. ein *Repackaging* einher, das wiederum für eine den aktuellen Systemen verständliche Datenorganisation sorgt. Darunter fällt die Verwendung von komprimierten Ordnern ebenso wie die Nutzung aktuellerer Dateisysteme. Echte Probleme entstehen hierbei nur, wenn kopiergeschützte Daten darunter sind [Borghoff et al., 2006], Beschriftungen der alten Datenträger von Bedeutung sind [Rothenberg, 1999] oder aber der „richtige“ Zeitpunkt verpasst wurde.

2.2. Migration

Die Migration wird von den meisten Computeranwendern unbewusst immer wieder praktiziert. Bei nahezu jedem Anwender finden sich Bilder, Filme, digitale Bücher oder Musik in

digitaler Form. Das regelmäßige Umspeichern zum Erhalt des Bitstroms erzeugt zwar Kosten, stellt aber wie o.a. kein Problem dar. Der vierte Bestandteil der Migration, nämlich die *Transformation*, ist dagegen nicht trivial. Werden Dateiformate, in denen die Information digital abgelegt ist, von aktuellen Anwendungen nicht mehr ausreichend unterstützt, müssen diese in aktuellere Formate überführt, also transformiert werden. Besonders gegenwärtig sind wohl die mit der Einführung neuer Versionen von Office-Anwendungen einhergehenden Veränderungen in der Darstellung.

Je nach genutztem Werkzeug, kann bei der Transformation Information verloren gehen. Es ist möglich, dass das Werkzeug, also das dazu genutzte Programm, bestimmte, scheinbar nicht relevante Informationen ignoriert und somit die Information selbst verfälscht. Auch ist es möglich, dass Zielformate nicht die gesamte Funktionalität abdecken. Klassisches Beispiel sind Formateigenschaften geschriebener Texte. Das müssen, wie in Abbildung 2.1 angedeutet, keine großen Veränderungen sein. Viele solcher kleinen Veränderungen über Jahre hinweg führen fast zwangsläufig zu Authentizitätsverlusten [Borghoff et al., 2006].

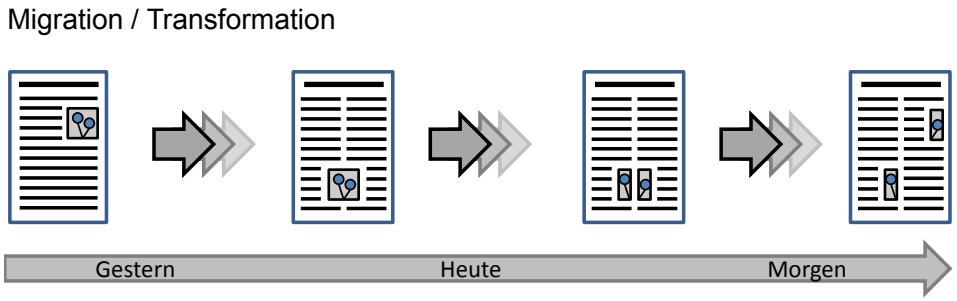


Abbildung 2.1.: Schleichender Informationsverlust bei der Migration.

Im Falle einer Transformation eines Filmes sehen wir zunächst wahrscheinlich keine Veränderung, wenn aufgrund einer nicht mehr unterstützten Komprimierung die Bilder einzeln neu codiert werden müssen. Passiert das aber im Laufe einer langfristig angelegten Archivierung einige hundert Mal, könnte enthaltene Information wie in eingeblendeten Texten aufgrund zunehmender Verpixelung bereits verloren gegangen sein.

Neben diesem gravierendsten Nachteil bietet diese Methode jedoch den Vorteil, dass Daten nahezu ohne Verursachung großer Kosten in ein Archiv eingestellt werden können. Anfängliche Migrationsschritte sind problemlos durchführbar, zum Teil setzt man hier auf automatisierte Prozesse. Den schleichenden Informationsverlust, vor allen aber den Totalverlust durch Fehler bei der Automatisierung kann man nur durch kostenintensive, manuelle Kontrollen und Nachbearbeitungen vermeiden, oder durch den Einsatz sehr aufwendiger Strategien [Triebsees und Borghoff, 2006]. Mit stetig wachsenden Datenbeständen wird das Problem akut. Mit jedem Migrationsschritt steigt die Anzahl aller betroffenen Dokumente, die transformiert werden müssen. Daher ist eine manuelle Überprüfung oder gar Nachbearbeitung der Ergebnisse zunehmend abwegig [Borghoff et al., 2006].

2.3. Emulation

Digitale Dokumente werden in ihrer originalen Abspielumgebung authentisch widergegeben. Für Dokumente im PDF-Format ist das z.B. ein PDF-Betrachter. Ziel des Emulationsansatzes ist, die originalen Abspielumgebungen der archivierten digitalen Objekte so exakt wie möglich durch Software nachzubilden. Im Gegensatz zur Migration sind die archivierten Objekte somit keinen fortlaufenden Veränderungen ausgesetzt. Nachgebildet werden können die Abspielumgebungen auf verschiedenen Ebenen. So kann ein Emulator ein einzelnes Programm, ein Betriebssystem oder auch die zugrunde liegende Hardware nachbilden [Granger, 2000]. Sofern ein einzelnes Programm emuliert wird, besteht die erhaltene Abspielumgebung nur aus dem Emulator. Nutzbar ist ein solcher Emulator nur für die Widergabe weniger Dateiformate. Wird ein Betriebssystem emuliert, können zusammen mit den originalen Programmen mehrere Abspielumgebungen erhalten werden. Auf der untersten Ebene wird die Hardware obsoleter Systeme emuliert. Auf diese Weise lassen sich alle Abspielumgebungen bilden, die für diese Hardware existierten. Vorausgesetzt die Betriebssysteme und die Wiedergabeprogramme wurden ebenfalls archiviert. Je tiefer daher die Emulation ansetzt, desto umfassender kann der entwickelte Emulator eingesetzt werden [Krebs und Borghoff, 2010]. Die Abbildung 2.2 veranschaulicht diesen Zusammenhang.

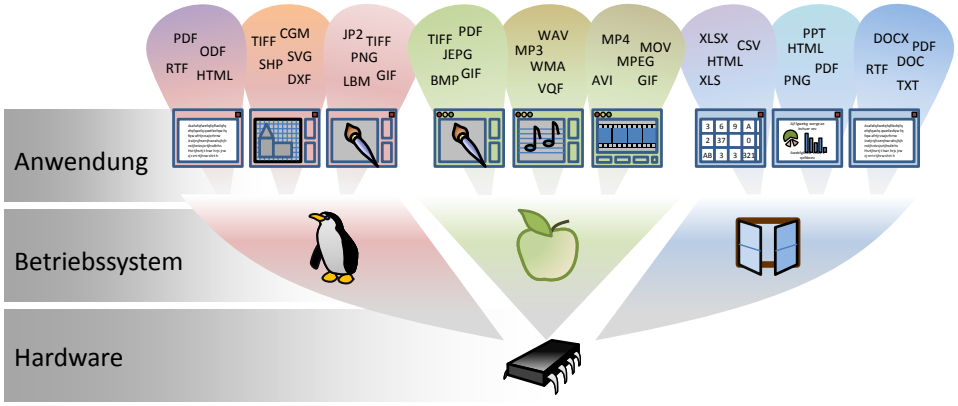


Abbildung 2.2.: Ebenen der Emulation.

Die Ebene ist für den Anwender selbst nicht wichtig, er interagiert nach wie vor mit einer Anwendungssoftware. Ob die zugrunde liegende Hardware oder das genutzte Betriebssystem original oder aber emuliert ist, kann er im Idealfall nicht unterscheiden. Für ein Archiv ist die Ebene äußerst relevant. Hier geht es nicht zuletzt um die damit verbundenen Kosten. Mit nur einem Emulator für ein x86 basierendes System wie dem Dioscouri [van der Hoeven et al., 2007] können viele verschiedene Anwendungsprogramme für mehrere Betriebssysteme genutzt werden, sofern diese geeignet archiviert wurden. Folgt man der schlüssigen Argumentationskette von Rothenberg [1999], so kommen Emulatoren für die Langzeitarchivierung vorrangig für die unterste Ebene in Betracht.

Im Vergleich zur Migration können die Kosten für das fortlaufende Transformieren gespart werden. Zudem können die Informationen authentisch und unverändert im Originalformat archiviert werden. Aber auch der Emulationsansatz hat seine Nachteile. So muss auch der

Emulator regelmäßig für neue Systeme portiert oder gar neu implementiert werden. Eine erfolgreiche Entwicklung eines Emulators erfordert zudem eine sehr genaue Spezifikation der Hardware (im Idealfall zusammen mit einem noch lauffähigem Original zum Vergleich) und ausreichend Ressourcen, denn die Entwicklung ist extrem kostenintensiv [Oltmans und Kol, 2005]. Wie in Abbildung 2.3 dargestellt, können Emulatoren auch in bereits emulierten Systemen ausgeführt werden. Durch derartige Emulator-Ketten lassen sich Kosten minimieren, aber die Anzahl möglicher Fehler steigt [Funk, 2010].

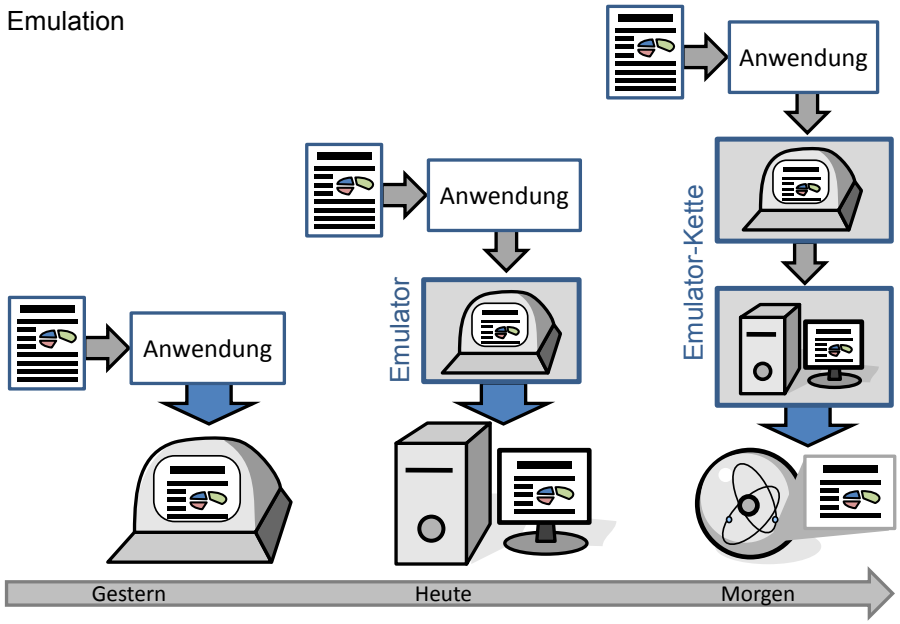


Abbildung 2.3.: Emulation von Hardware.

2.4. Computermuseum

Dieser Abschnitt betrachtet Computermuseen im Kontext der Langzeitarchivierung. Diese Abgrenzung vorab ist wichtig, denn es gibt auch klassische Museen mit regulären Öffnungszeiten, Glasvitrinen, pädagogischem Konzept und der gleichen, die auch Computer ausstellen. Ein Computermuseum im Sinne der Langzeitarchivierung hat dagegen andere Schwerpunkte und verfolgt in erster Linie das Ziel der Aufrechterhaltung der Funktionsfähigkeit obsoleter Hardware zusammen mit der originalen Software. Der Begriff Museum ist diesbezüglich irreführend, aber leider im deutschen Sprachraum verfestigt [Huth, 2010]. Zwar waren die ersten Institutionen, die dieses Ziel verfolgten Museen, aber es finden sich heute Hardwaresammlungen auch abseits des Museumsbetriebs, wie z.B. an der Universität der Bundeswehr in München.

Dieser Abschnitt geht auf die Motivation und die Notwendigkeit von Computermuseen bzw. Sammlungen historischer Hardware ein und stellt anschließend mit der datArena die Einrichtung vor, die für diese Arbeit genutzt wurde.

2.4.1. Motivation und Notwendigkeit

Es gibt mehrere Gründe für den funktionsfähigen Erhalt bedeutender Hardware vergangener Generationen. Auf die wichtigsten wird im Einzelnen eingegangen.

Ein Grund liegt in **übersehenen Datenbeständen**, die nicht rechtzeitig der Langzeitarchivierung, z.B. der Migration zugeführt wurden. Datenformate sind dann nicht mehr auf aktuellen Systemen interpretierbar, die Informationen darauf verloren. Dazu kommt das Problem der fehlenden Hardware zum Auslesen obsoleter Datenträger. Nicht immer erkennt man sofort die Archivierungswürdigkeit der Daten. Diese landen im „Keller“ sind noch lesbar, aber es fehlt der Zugriff. Die über die lauffähig erhaltene Hardware wieder zugänglichen Informationen können auf aktuelle Datenträger in noch lesbare Datenformate überführt werden. Falls nötig, müssen hierzu eigens Programme für die betagten Systeme neu entwickelt werden, die neuere Formate noch nicht unterstützen. Eine solche Hardwaresammlung muss daher auch die entsprechenden Entwicklungsumgebungen und den Zugang zum notwenigen Know-How, z.B. über Bedienungsanleitungen bereit halten.

Während der oben aufgeführte Grund am ehesten einem nachgeholten Migrationsschritt gleich kommt, kann eine funktionsfähige Hardwaresammlung auch im Bereich der **Emulation unterstützen**. Zum einen kann die Zeit bis zur Herstellung eines Emulators überbrückt werden. Üblicherweise muss die aktuelle Hardware vielfach leistungsfähiger als die zu emulierenden Systeme sein, damit zufriedenstellende Ergebnisse erzielt werden können. Selbst mit einem verfügbaren Emulator kann es daher notwendig sein, die originale Hardware weiterhin zu nutzen, bis aktuelle Systeme entsprechend leistungsfähig sind. Zum anderen dient die obsolete Hardware dem Vergleich bzw. der Verifizierung. Nur im direkten Vergleich kann das Verhalten des Emulators als authentisch überprüft werden Rothenberg [1999]. Ggf. kann, solange die obsolete Hardware verfügbar ist, der Emulator weiter verfeinert werden. Emulatoren können in Kombination mit funktionsfähig erhaltener Hardware automatisiert getestet werden. Dabei sind einfach, schnell und kostengünstig umfassende Ergebnisse möglich [Martignoni et al., 2009].

Ein eher pragmatischer Grund liegt dann vor, wenn man um die **zeitlich begrenzte Archivierungswürdigkeit** seiner Daten weiß, die man nur für einen bestimmten Zeitraum, z.B. aufgrund geltenden Rechts, verfügbar halten muss. Hier kann es sich lohnen, große Mengen dieser Daten unangetastet auf den obsoleten Datenträgern zusammen mit der obsoleten Hardware einzulagern, wenn diese nur noch in wenigen Ausnahmefällen gebraucht werden. Somit hat auch die Strategie des Nichtstuns eine Berechtigung (*“an archive’s option to do nothing”* [Janée et al., 2008]).

Ein Grund, der den ansonsten nicht vorhandenen Museumscharakter rechtfertigen könnte, ist in der möglichen **Vermittlung historischer Arbeitswelten** gegeben. So beeindruckt die Vorführung einer funktionierenden Z3 im Deutschen Museum in München regelmäßig die Besucher [Petzold, 2000].

Ein letzter Grund sei schon allein deshalb aufgeführt, weil er für diese Arbeit von großer Bedeutung ist. **Für die Wissenschaft** sind solche funktionsfähigen Hardwaresammlungen von Interesse; umso mehr, je umfangreicher eine solche Sammlung vergangene Hardwarearchitekturen umfasst. Hier kann man in verschiedensten Forschungsfeldern aus der Vergangenheit auf Neues schließen, Tendenzen erkennen und so Prognosen für die Zukunft erarbeiten und

letztlich, wie im noch folgenden Kapitel 6 der vorliegenden Arbeit gezeigt, Erfahrungen generieren. Es kann sich auch für andere Forschungsgebiete lohnen, bestimmte Konzepte in die Vergangenheit zu projizieren, um daraus Schlüsse für die Zukunft abzuleiten.

2.4.2. datArena

Die Gesellschaft für historische Rechenanlagen e.V.[1] betreibt das Großprojekt Computermuseum Muenchen, um die Geschichte des Computers zu dokumentieren und weiterzugeben. Neben dem Sammlungsbetrieb sind Restaurationen und die funktionsfähige Erhaltung historischer Rechenanlagen primäre Aktivitäten. Außer der Hardware werden auch Software, Speichermedien, Handbücher, Kataloge, Werbung und Literatur gesammelt und archiviert.[2]

Die Großrechnerabteilung des Computermuseums Muenchen wurde 2002 gegründet. Sie umfasst ca. 100 Großrechner. Ihr Fokus liegt auf Maschinen zum technisch-wissenschaftlichen Rechnen. Sammlungsschwerpunkt sind Modelle der Firmen Control Data Corporation, Cray Research Incorporated sowie Sun Microsystems Incorporated. Vertreten sind aber auch die Hersteller NEC, Fujitsu und andere. Die Zeitlinie reicht dabei von 1964 bis 2001 und wird kontinuierlich erweitert werden. Die letzten 10 Jahre fehlen dabei absichtlich. Zum einen sollen Neuzugänge kaum Anschaffungskosten verursachen, im Idealfall sind es Spenden. Zum anderen müssen sich die Großrechner bewährt und als für die Computergeschichte bedeutend erwiesen haben. Um den letzten Aspekt zu beurteilen, bedarf es zeitlicher Distanz.

Einige der wichtigsten Vertreter der funktionsfähig erhaltenen Großrechner finden sich in der datArena, einer Einrichtung, die auf dem Campus der Universität der Bundeswehr in Neubiberg in einer denkmalgeschützten Halle entsteht. Derzeit befindet sich die datArena noch im Aufbau. Neben den Großrechnern wird die finale Ausbaustufe auch eine umfangreiche Mikrocomputerabteilung, eine profunde, gut sortierte Sammlung an Anleitungen und Beschreibungen und ein Software-Archiv von internationalem Rang aufnehmen.[3]

Die vorliegende Dissertation fokussiert bei der Nutzung der datArena den Schwerpunkt Großrechner. Welche Großrechner dabei zum Einsatz kamen, wurde von mehreren Faktoren bestimmt. Zum einen sollten die Großrechner z.B. für das angedachte Szenario tatsächlich in Frage kommen und zudem eine mögliche Kette im Sinne von Nachfolgegenerationen darstellen und dabei einen möglichst großen und lückenlosen Zeitraum abdecken. Zum anderen spielen auch ganz pragmatische Gründe eine Rolle wie die verlässliche Verfügbarkeit innerhalb der aktuellen Ausbaustufe oder verfügbare Emulatoren. Detaillierte Angaben zur Motivation der getroffenen Auswahl finden sich in Kapitel 6.

[1] http://www.gfhr.de
[2] http://www.computermuseum-muenchen.de
[3] Derzeit befinden sich diese Komponenten in Lagern des Computermuseums München.

3. Der UVC für die Langzeitarchivierung

Die im Kapitel 2 dargestellten Nachteile der bisher verfügbaren Archivierungsmethoden lassen sich wie folgt zusammenfassen: Sie sind aufgrund des geringen initialen Aufwands attraktiv, aber im laufenden Betrieb zunehmend wahlweise teuer oder anfällig für Informations- und Authentizitätsverlust.

Die auf dem UVC basierende Archivierungsmethode soll das o.a. Problem der Langzeitarchivierung lösen. Um die dabei relevanten Aspekte dieser Methode bewerten zu können, wird dieses Kapitel daher zunächst die Methode zusammen mit den technischen Grundlagen vorstellen. Anhand dieser Darstellung und eigener Erfahrungen werden notwendige Voraussetzungen abgeleitet. Auf Basis dieser Voraussetzungen werden die eingehend formulierten Fragen der Zielsetzung konkretisiert. Dieses Kapitel schließt mit einem Überblick über den aktuellen Forschungstand ab.

3.1. Der Ansatz

Die zugrunde liegende Idee wird in allen Quellen[1] sehr kurz dargestellt. Das liegt zum einem daran, dass sie einfach und leicht zu verstehen ist. Und zum anderen ist sie noch nicht wirklich ausgereift. Zunächst wird der Ansatz an sich vorgestellt und im Anschluss anhand eines Beispiels die noch offenen Punkte herausgearbeitet.

3.1.1. Der Ansatz im Allgemeinen

Der Grundgedanke liegt in dem Bestreben, während der unbestimmt langen Zeit des Archivierens kaum Kosten zu verursachen. Der Aufwand für den Archivar bzw. für das Archiv muss möglichst gering sein. Andernfalls können die Kosten mit dem rasant anwachsenden Archivbeständen unbezahlbar werden. Die Folge: Authentizitäts-, Informations- oder auch Totalverlust. Abgeleitet ist dieser Gedanke wohl aus der Geschichte überlieferter Artefakte. Höhlenmalereien oder ägyptische Hieroglyphen haben sich erhalten, weil über Jahrhunderte kein Archivar bezahlt werden musste, der sich regelmäßig um den Erhalt kümmern musste. Allerdings ist die heute mögliche Interpretation der überlieferten Hieroglyphen eher einem Zufall zu verdanken, nämlich dem Fund des Steins von Rosetta.[2]

[1] [Lorie, 2001, 2002a,b; Wijngaarden und Oltmans, 2004; Oltmans und Kol, 2005; Lorie und van Diessen, 2005; van der Hoeven et al., 2005; van der Hoeven et al., 2005; Gladney und Lorie, 2005; Kol et al., 2006]

[2] Eine knappe Zusammenfassung ist bei Heminger und Robertson [2000] zu finden.

Bei der Entwicklung aktueller Archivierungsmethoden wird versucht, den Faktor „Zufall“ weitestgehend auszuschließen. Um die Lesbarkeit/Interpretierbarkeit digitaler Objekte langfristig zu erhalten, setzt man auf Dokumentenbeschreibungssprachen wie SGML bzw. XML. Dabei steigt der erforderliche Speicherplatzbedarf um mehr als das Sechsfache [Olson, 2008]. Allerdings sind die Daten in einer auch von Menschen lesbaren Form und können zugleich die notwendigen Metadaten enthalten.

Die den UVC nutzende Methode setzt dagegen auf Programme. Programme sind das ideale Medium, um kompakt den Zugriff auf enthaltene Informationen zu sichern [Wijngaarden und Oltmans, 2004]. Sie machen das Archivieren im komprimierten Originalformat möglich, ohne komplexe Beschreibung der Dateiformate oder der genutzten Kompressionsverfahren.

Der über Programme gesicherte Zugriff ähnelt dem Ansatz der Emulation. Der Unterschied besteht darin, dass kein reales System emuliert wird [Lorie, 2002b]. Vielmehr wird ein virtueller Computer, also der UVC emuliert. Da er jedoch nicht real existiert und aufgrund seiner Eigenschaften auch nicht gebaut werden wird, passt der Begriff Emulation nicht. Es sei angemerkt, dass, wie mit der Emulation auch, neben dem Zugang zu komplexen Formaten auch das Verhalten von Programmen selbst erhalten werden kann, sollten sie für den UVC migriert werden.

Der UVC soll universell und sehr einfach spezifiziert sein. Dadurch soll er sich sehr schnell und für jede (auch zukünftige) Hardwarearchitektur implementieren lassen. Der Grad der Authentizität herkömmlicher Emulatoren kann mit Hilfe der realen Hardware (noch) gemessen und überprüft werden. Dem UVC fehlt diese Möglichkeit. Nach Lorie [2002a] ist es jedoch eine wesentlich einfachere Aufgabe einen UVC zu implementieren als die Entwicklung eines Emulators für eine reale Maschine. Somit sollte auch das Prüfen auf Korrektheit gemäß einer recht kurzen und übersichtlichen Spezifikation wesentlich einfacher und in ferner Zukunft möglich sein.[3] Der gesamte Ansatz basiert auf der Annahme, dass der UVC bzw. die Spezifikation diese Eigenschaften aufweist. Der zweite Teil der vorliegenden Arbeit wird diese Voraussetzungen untersuchen.

Durch die Abkehr von realen Systemen kann man vorhandene Software wie bekannte PDF-Betrachter nicht nutzen. Der Ansatz basiert daher darauf, dass UVC-Programme speziell für den Zweck der Archivierung erstellt werden. Es wird also nicht versucht, die originale Abspielumgebung zu erhalten. Vielmehr wird eine neue geschaffen, die dann aber zeitlich unbegrenzt verfügbar sein soll.

Der entscheidende Unterschied gegenüber dem Emulationsansatz ist, dass mit dem UVC nur noch eine Plattform für die jeweils neuen Systeme migriert werden muss.

Dabei wird in Kauf genommen, dass die Programme für den UVC nicht die volle Funktionalität der originalen Abspielumgebung bieten werden, sondern sich auf das „Wesentliche“ konzentrieren. Im Beispiel eines PDF-Betrachters könnte das die Fähigkeit sein, den Text zu extrahieren oder Einzelseiten als Pixelgrafik auszugeben. Wobei die nicht-triviale Bestimmung des „Wesentlichen“ vom Archivar auf die Entwickler bzw. Nutzer der zu archivierenden Formate übergeht. Damit ist nicht die eingehende Bewertung der Archivwürdigkeit gemeint, die weiterhin Aufgabe des Archivars bleiben wird, sondern die Bewertung der vom UVC-

[3] Z.B. mit der Unterstützung einer umfangreichen und speziell hierfür entwickelten UVC-Testprogrammsammlung.

Programm erreichten Authentizität.

Ein wichtiges Merkmal archivierungsfähiger Dateien wird somit ein zugehöriges UVC-Programm sein, das den Zugriff auf die enthaltene Information sichert. Daraus resultiert ein sehr hoher Aufwand beim erstmaligen Aufnehmen eines digitalen Objekts mit neuem Format. Es wird so aber auch sichergestellt, dass der Grad der Authentizität bereits mit der Archivierung festgelegt wird. Durch diesen hohen Aufwand wird zudem der verhältnismäßig niedrige Folgeaufwand während der Bestandserhaltung „erkauft."

Zum Zeitpunkt der Archivierung sollte bereits eine Implementierung des UVC verfügbar sein. Für diese werden UVC-Programme entwickelt, getestet und der erreichte Grad an Authentizität bewertet. Sofern das Ergebnis überzeugt, können digitale Objekte der so unterstützten Formate im Archiv abgelegt werden. Während der Archivierung fällt lediglich die Bitstromkonservierung an, die kostengünstig, zuverlässig und unter Beachtung der Haltbarkeit genutzter Speichermedien ohne Informationsverlust möglich ist.

In ferner Zukunft muss mit dem ersten Zugriff auf archivierte Objekte ein UVC vorhanden sein, oder aber für diesen Zugriff einmalig neu implementiert oder portiert werden. Der damit verbundene Arbeitsaufwand ist aufgrund der einfach gehaltenen Spezifikation eine nicht zu schwierige Aufgabe.[4] Es finden sich aber bisher keine verlässlichen Angaben dazu, was „nicht zu schwierig" eigentlich bedeutet.

Aus der noch jungen Geschichte der Computer ist uns ein ähnlicher Fall bekannt. Allein durch die vielen Aufzeichnungen und Konstruktionspläne konnte mehr als ein Jahrhundert später ein Emulator für die mechanische Analytical Engine von Babbage [1837] entwickelt[5] und ein Programm für diese von Ada Lovelace [1843] zur Berechnung von Bernoulli Zahlen korrekt ausgeführt werden.

Den UVC in der Zukunft zu implementieren sollte deutlich einfacher werden. Die Spezifikation umfasst lediglich 18 Seiten und ist im einfachen Englisch verfasst [Lorie und van Diessen, 2005].

Mit einem verfügbaren UVC kann das gewünschte digitale Objekt geladen und das dazu passende UVC-Programm gestartet werden. Danach erzeugt der UVC eine Ausgabe, die mit Hilfe eines logischen Datenschemas (LDS) in eine logische Datensicht (LDV) überführt werden kann. Hierbei wird eine einfache, aber vielfach bewährte Methode genutzt: Die Ausgabe eines Elements mit einem *Tag*. Dieses *Tag* gibt an, welche Rolle dieses Element innerhalb der Datenstruktur spielt [Lorie, 2002a]. Eine damit mögliche Form der Ausgabe ist das XML-Format. Es ist zugleich maschinen- und menschenlesbar und daher gut für die Langzeitarchivierung geeignet. Der Grund, warum nicht einfach Bilder bereits als XML-Dateien archiviert werden, liegt aber auf der Hand: die Größe des beanspruchten Archivspeichers.

Gerade dieser letzte Schritt der Aufbereitung und Interpretation der Ausgabe wird in den verfügbaren Quellen sehr kurz und leider auch widersprüchlich dargestellt. Das Beispiel eines real erprobten Anwendungsfalls soll helfen, den bisher eher trocken vermittelten Ansatz greifbarer zu machen.

[4] 'A not too difficult task.' Lorie [2002a]

[5] `http://www.fourmilab.ch/babbage/contents.html` (eingesehen am 12.07.2011)
Neben den seinerzeit begrenzten mechanischen Möglichkeiten führte vor allem die mangelnder Weitsicht der britischen Regierung dazu, dass keine seiner Maschinen gebaut wurde [Fuegi und Francis, 2003].

3.1.2. Der Ansatz am Beispiel

Ein leicht umzusetzender Anwendungsfall ist die Archivierung von Bildern mit dem UVC-Ansatz. Dieser Anwendungsfall auf Grundlage der Formate JPEG und GIF ist auch die Basis des Proof of Concepts von IBM [Lorie, 2002b].

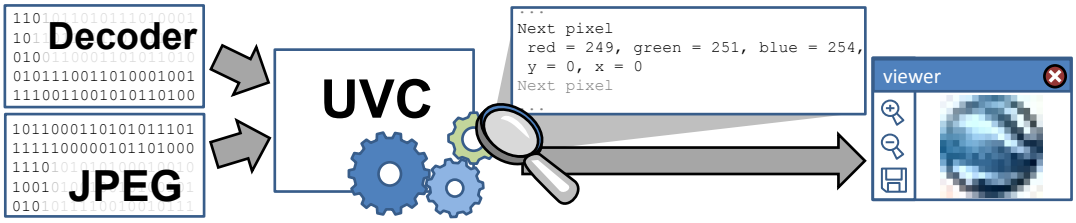


Abbildung 3.1.: UVC-Methode am Beispiel der Archivierung von Bildern im JPEG-Format.

In der Abbildung 3.1 wird das links oben dargestellte UVC-Programm (der JPEG-Decoder) zusammen mit dem archivierten JPEG-Bild an den UVC übergeben. Das ausgeführte Decoderprogramm liefert die enthaltenen Pixelinformationen, ohne dass zusätzlich Informationen zum Aufbau des JPEG-Formats oder der genutzten Komprimierung auf Basis der diskreten Cosinustransformation archiviert wurden. Diese Pixelinformationen enthalten die RGB-Farbwerte und die Position des Pixels im Bild. Vorweg wurden die Ausmaße des Bildes in Pixeln übermittelt. Ein nur wenige Programmzeilen umfassendes Programm, die sogenannte *Restore Application*, kann diese Informationen direkt darstellen.

Soweit die Theorie. In der Praxis lassen sich nur die ersten Schritte am verfügbaren Prototypen von IBM[6] nachvollziehen. Zusammen mit dem veröffentlichten UVC gibt es ein JPEG-Decoderprogramm und auch zwei in Java geschriebene *Restore Applications* zum Verarbeiten der erzeugten Ausgabe: Eines zum direkten Anzeigen der decodierten Bildinformationen und ein weiteres zum Erstellen einer XML-Datei.

Beim Nachvollziehen der Grundlagen anhand des Prototyps scheitert man schnell. Weder gibt der Prototyp Aufschluss über die tatsächlich ausgetauschten Nachrichten noch ist ersichtlich, wie diese Nachrichten interpretiert bzw. verarbeitet werden.

Die Ausgabe erschloss sich erst mit einer eigenen Implementierung des UVC, die den JPEG-Decoder ausführen konnte. Die Formatierung der Ausgabe ist bewusst nicht spezifiziert, der Inhalt aber ist zum Nachvollziehen der folgenden Interpretation notwendig. Die Abbildung 3.2 zeigt in der Mitte den Anfang der Ausgabe für den JPEG-Decoder (hexadezimal codiert) ausgehend vom Bild der kleinen Kugel.

Bei der Verwendung der eigenen Implementierung wurde (neben anderen Abweichungen von der Spezifikation) auch deutlich, dass die Bitfolge des zu decodierenden Bildes direkt in den Speicher des UVC zu laden ist. Die Spezifikation des UVC sieht ein solches Vorgehen nicht vor.

Bei der Interpretation der Ausgabe soll ein LDS helfen. Von außen sichtbar wird die Verwendung eines solchen LDS nicht, auch findet sich keine in Form einer Datei. Quellcode ist im Paket ebenfalls nicht enthalten, weder zur Implementierung des UVC noch zu den beiden Ver-

[6] www.alphaworks.ibm.com/tech/uvc

arbeitungsprogrammen. Ob wenigstens intern ein solches LDS genutzt wird, erschließt sich daher nicht.

Dieses LDS wird aber benötigt, um die vom UVC ausgegebene „Information“ korrekt interpretieren zu können. In der Abbildung 3.2 ist auf der rechten Seite die im XML-Format konvertierte Information dargestellt, wie sie das im Paket enthaltene Java-Programm LDV-Viewer generiert.

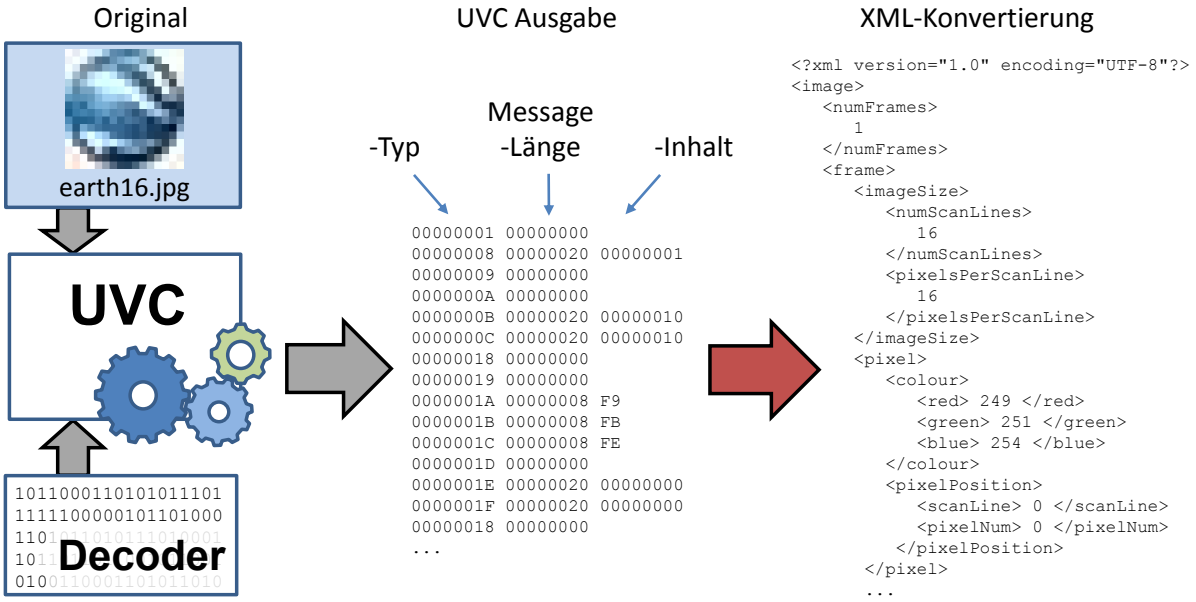


Abbildung 3.2.: Ausgabe des UVC-Decoder-Programms und die erzeugte XML-Datei.

Keine der verfügbaren Quellen zum UVC enthält eine Spezifikation der Syntax eines LDS, dagegen aber verschiedene Angaben.

Lorie [2001] stellte seinen Ansatz zunächst ohne die Nutzung eines LDS vor. Die ursprüngliche Idee war einfach eine DTD[7] zu nutzen, mit der der Aufbau von XML-Dateien beschrieben werden kann. Zur Veranschaulichung lieferte er eine DTD für Bilder in Graustufen (wobei diese Angabe keine konforme DTD ist):

```
DOCTYPE Building_collection [
! This is a collection of gray scale pictures of historical
  buildings in Mycity.
! A building has an address, and an (optional) name; it can have
  several pictures (for different years).
! The gray value is between 0 (white) and 255 (black).
ELEMENT Collection (building+)
ELEMENT building (name?, address, picture+)
ELEMENT picture (year, nbr_lines, dots-per-line, line+)
ELEMENT line (nbr, gray_value+)
ELEMENT name (CHARr)
ELEMENT address (CHAR)
ELEMENT year (NUM)
```

[7] Document-Type-Definitions

```
ELEMENT nbr_lines (NUM)
ELEMENT dots_per_line (NUM)
ELEMENT nbr (NUM)
ELEMENT gray_value (NUM)
```

In einer anderen Quelle gibt Lorie [2002a] eine LDV (!) für Bilddateien vor:

```
collection
collection_name,
owner,
picture*
picture_name,
creation_date,
width,
height,
pixels*
R
G
B
```

Abgesehen davon, dass der Aufbau nicht zur abgebildeten Ausgabe passt und der Aufbau sich nicht klar ableitet, ist diese LDV offensichtlich älteren Ursprungs, da sie die Pixel ohne Positionsangabe erwartet. Das setzt eine zeilen- bzw. spaltenweise Ausgabe voraus, die aber mit der blockweisen Ausgabe des aktuellen JPEG-Decoders nicht konform ist.

Der „Proof of Concept“ von Lorie [2002b] enthält eine nicht hilfreiche Darstellung eines LDS für Bücher zusammen mit der Bemerkung, dass der Prozess zur Erstellung eines LDS nicht Bestandteil der „UVC Convention“ sei. Wendet man die aus dem Beispiel der Bücher erkennbare Logik auf die Ausgabe in Abbildung 3.2 an, so kann man folgendes LDS ableiten:

```
ELEMENT image (1,2+)
ELEMENT 1 [numFrames]
ELEMENT 2 [frame] (3,6+)
ELEMENT 3 [imageSize] (4,5)
ELEMENT 4 [numScanLines]
ELEMENT 5 [pixelsPerScanLine]
ELEMENT 6 [pixel] (7,11)
ELEMENT 7 [colour] (8,9,10)
ELEMENT 8 [red]
ELEMENT 9 [green]
ELEMENT 10 [blue]
ELEMENT 11 [pixelPosition] (12,13)
ELEMENT 12 [scanLine]
ELEMENT 13 [pixelNum]
```

Dieses LDS ist wie folgt zu interpretieren. Ein Bild (`image`) besteht aus den Elementen `numFrames` und `frame`. Die Zahlen in der Klammer hinter `image` beziehen sich dabei auf die Nummern der Elemente, das `+` weist auf eine nicht leere Folge des Elements hin. Ein Bild enthält daher immer genau eine Anzahl der enthaltenen Frames (`1`) und mehrere Frames (`2+`). Die Zahlen verweisen auf die jeweiligen Elemente, die eine laufende Nummer haben. Die Messagetypen der Ausgabe passen aber nicht zu den laufenden Nummern, was entweder darauf schließen lässt, dass das tatsächlich verwendete LDS komplexer ist, oder die intuitiv angenommene fortlaufende Nummerierung nicht zwingend ist.

Eine recht gut passende, aber gekürzte Darstellung eines LDS gibt van der Hoeven et al. [2005]:

```
ELEMENT 1 [Image] (10, 24+)
ELEMENT 10 [Image Size] (11, 12)
ELEMENT 11 [Number Scan Lines]
ELEMENT 12 [Number Pixels Per Scan Line]
ELEMENT 24 [Pixel] (25, 29)
ELEMENT 25 [Colour] (26, 27, 28)
ELEMENT 26 [Red]
ELEMENT 27 [Green]
ELEMENT 28 [Blue]
ELEMENT 29 [Pixel Position] (30, 31)
ELEMENT 30 [Scan Line]
ELEMENT 31 [Pixel Number]
```

Die Quelle beschreibt grob die Entwicklung des UVC in Java und liefert dieses LDS. Leider ist das vollständige LDS nicht im Paket enthalten. Ob diese vollständig zur Ausgabe passen würde, bleibt somit offen.[8] Auch liefert die Quelle weder eine allgemeine Syntax noch die Lösung für die Interpretation dieser Angaben. Nach wie vor ist z.B. offen, wie vermittelt werden soll, dass die Angaben für Rot, Grün und Blau Werte zwischen 0 und 255 annehmen und 255 die volle Intensität darstellt. Die o.a. Kommentare in der DTD lassen lediglich die ersten Überlegungen von Lorie sichtbar werden.

Verwirrend ist zudem, dass in einem weiteren Aufsatz von van der Hoeven et al. [2005] die LDV, die ja eigentlich mit Hilfe des LDS erzeugt werden soll und z.B. Bilder oder vollständig konforme XML-Dateien darstellen, direkt mit der Ausgabe des UVC gleichsetzt. Ebenfalls für Verwirrung sorgt die mehrfach in den aufgeführten Quellen angesprochene Nutzung des UVC zur Verarbeitung der LDS. So sollen nach Lorie [2002a,b] nicht nur die Daten decodiert werden, sondern in gleicher Weise auch das LDS. Betrachtet man die begrenzten Ausgabemöglichkeiten des UVC, bedürfte es eines LDS, um die decodierte Information in ein LDS zu überführen. Leicht zu erkennen, dass das wenig hilfreich ist.

Es ist jedoch nur fair zu erwähnen, dass die Interpretation der Ausgabe des eigenen UVC, trotz fehlendem LDS, innerhalb nur weniger Minuten mit einer eigenen in Java programmierten *Restore Application* von nur wenigen Zeilen Umfang gelang. Schwieriger könnte sich das jedoch bei komplexeren Objekten wie PDF-Dateien gestalten. Hier kommen neben Bildern noch sehr viele andere Komponenten vor, deren intuitive Zuordnung innerhalb von Minuten sicher nicht mehr möglich ist.

Neben dieser Zuordnung ergibt sich auch die Frage nach dem Verständnis der erzeugten Informationen in ferner Zukunft. Für uns sind heute Bilder als Ansammlung von Pixeln geläufig. Auch die Farbzerlegung in Rot, Grün und Blau ist den meisten heute nichts Unbekanntes. Aber was, wenn in 1000 Jahren anstatt selbstleuchtender Pixel nur noch schillernd reflektierende Wabenstrukturen genutzt werden?

Kommentare allein werden dieses Problem nicht abschließend lösen können. Was hierzu notwendig ist, ist eine umfassende Ontologie, die Bedeutungen aller genutzten Begriffe wie Bild, Pixel und Farbcodierung liefert. Nach Staab und Studer [2009] wird der Begriff Ontologie in zwei Formen gebraucht. Im vorliegenden Fall ist der ursprüngliche Ontologiebegriff

[8] Die gekürzte Fassung unterstützt zumindest die Frames nicht.

aus dem Gebiet der Philosophie gemeint, der sich mit der Natur und der Struktur der Realität auseinandersetzt. Dieser Aspekt wird zugunsten des Umfangs der vorliegenden Arbeit nicht weiter verfolgt. Zumal eine solche Ontologie nicht Bestandteil der Archivierungsmethode selbst ist, sondern generell im Archiv vorhanden sein muss, um überhaupt Metadaten interpretieren zu können.

3.1.3. Folgerungen

Aus dem bisher Dargestellten leiten sich bereits Voraussetzungen ab, die zwingend erfüllt sein müssen, damit der vorgestellte Ansatz funktioniert. Aufgabe dieses Abschnitts ist auch die Konkretisierung der eingangs formulierten Fragen der Zielsetzung.

Damit die Bitströme der Originale zusammen mit denen der UVC-Programme über unbestimmte Zeit unverfälscht bleiben, müssen diese geeignet gespeichert werden können. Die Forschung und der Praxisbetrieb geben in diesem Bereich kaum Grund, an der Machbarkeit zu zweifeln. Aus erfolgreichen Backupstrategien leiten sich bereits wichtige Voraussetzungen für die Langzeitarchivierung von Bitströmen ab, z.B. die örtlich verteilte Speicherung zeitgleich auf unterschiedlichen Medien unter Beachtung der rechtzeitigen Umspeicherung auf neue Medien in Anbetracht der begrenzten Haltbarkeit. Die Risiken dabei können als gering eingestuft werden [CCSDS, 2002]. Archivalien, die absehbar keinen Veränderungen mehr unterliegen, aber auf Dauer für die Menschheit von Interesse sind, könnten „in Stein gemeißelt“ werden. Nach Wijngaarden und Oltmans [2004] zählt potentiell auch die Spezifikation des UVC dazu. Dieses „Meißeln“ findet eine reale Anwendung unter Nutzung von Nickelplatten [Sisson, 2008] oder auch von Mikrofilmen [Wendel, 2006].

Damit die Implementierung eines UVC in ferner Zukunft gelingt, muss die Spezifikation des UVC vollständig und korrekt sein, d.h. allein aus sich heraus verständlich und eindeutig. Diese Eigenschaften wurden bisher nicht umfassend geprüft und zudem dürfen sie in Frage gestellt werden, da die Spezifikation mit Absicht recht kurz gehalten und zudem in Englisch (anstatt rein formal) verfasst wurde. Ein Teil der Arbeit befasst sich mit diesem Aspekt. Bei der eigenen Referenzimplementierung sind Reibungspunkte aufgefallen und wurden entsprechend dokumentiert. Auch sind verfügbare Programme für Tests am eigenen UVC sehr hilfreich, idealerweise sind spezielle, aufeinander abgestimmter Testprogramme der Spezifikation beigelegt. Dieser Aspekt wird später in einem eigenen Kapitel aufgegriffen.

Die in Englisch verfügbare Spezifikation muss in ferner Zukunft korrekt interpretiert werden können. Mit Blick auf die Übersetzung des Zitats Platons wird eins sehr deutlich: Sprachen verändern sich. Wie am Beispiel des Rosetta Steins sichtbar, kann es helfen, die Spezifikation zusätzlich in andere Sprachen zu übersetzen und zu archivieren. Auch dieser Aspekt fällt in den Bereich der Ontologieforschung, der in der vorliegenden Arbeit nicht weiter verfolgt wird.

Der UVC muss sich leicht implementieren lassen. Ist der Aufwand absehbar zu groß, sinkt die Wahrscheinlichkeit, dass in ferner Zukunft jemand diese Hürde angeht. Den Aufwand für eine Implementierung für heutige Systeme abzuschätzen, ist nach einer gelungenen Implementierung möglich. Hierbei darf aber die Möglichkeit einer impliziten Hard- oder Softwareabhängigkeit nicht vernachlässigt werden. So könnte die Spezifikation Bestandteile enthalten, die sich besonders einfach mit heute verfügbaren Programmiersprachen und mitgelieferten

Bibliotheken umsetzen lassen. Allein vom „Heute“ auf das „Morgen“ zu schließen ist daher gefährlich. Diese Arbeit greift im Schwerpunkt diese Frage der Universalität unter Einbeziehung von „Gestern“ auf.

Die Ausgabe eines jeden UVC-Programms, und somit die erwarteten Eingaben, müssen in einer klar spezifizierten Form erfolgen. Die bisherigen Ansätze bzgl. des logischen Datenschemas und der daraus ableitbaren Datensicht sind vielversprechend, bedürfen aber in einem ersten Schritt einer Spezifikation des LDS, die von den Entwicklern erwartet und somit in der vorliegenden Arbeit nicht weiter verfolgt wird. Sinnvoll wäre die Aufnahme der Syntax in die Spezifikation des UVC. Mit weiteren UVC-Anwendungen kann dann die Vollständigkeit evaluiert werden. Fehlende klare Vorgaben werden potentielle Nutzer jedoch abschrecken.

Es ist vorstellbar, dass noch viele Generationen nach uns ebenfalls Dokumente erstellen, die sich auf die gleiche Art archivieren lassen. Unwahrscheinlich ist jedoch die fortwährende Nutzung unserer Formate. Es werden sicher andere Anforderungen an die Dokumente gestellt, die wiederum zu neuen Dateiformaten führen werden. Damit der UVC auch weiterhin nutzbar bleibt, muss er programmierbar bleiben. Zukünftige Generationen müssen in der Lage sein, neue UVC-Anwendungen zu entwickeln. Eine Möglichkeit ist, Assembler oder Compiler als UVC-Anwendungen bereitzustellen. Diesen Aspekt greift die Arbeit zusätzlich unter dem Blickwinkel speziell modifizierbarer Testprogramme auf.

UVC-Programme werden idealerweise für Dateiformate entwickelt, wenn sich eine weitverbreitete Nutzung abzeichnet. Vielleicht sogar, um einem Dokument das Attribut „für die Langzeitarchivierung geeignet“ zu geben. Im ungünstigen Fall werden diese Anwendungen erst mit der absehbar fehlenden Interpretierbarkeit erstellt. Entwickler einer solchen UVC-Anwendung legen den Grad der Authentizität fest. Dieser wird beeinflusst von der verfügbaren Entwicklungszeit und wohl auch von der Ausdauer beim Testen. Entgegen der Ansicht der Entwickler[9] leitet sich aus dem letzten Punkt die Notwendigkeit eines performanten UVC ab. Zumal, wie dargestellt, nur wenig Zeit bis zur Erstellung der UVC-Anwendungen vergeht, in der deutlich schnellere Rechnergenerationen verfügbar werden könnten. Nur wenn die Entwickler pro Testlauf nicht mehrere Minuten oder gar Stunden warten müssen, werden sie die zur Verfügung stehenden Mittel voll auf das Ziel der Authentizität ausrichten können. Der Grad der erreichten Authentizität sollte weitestgehend unabhängig von der Performanz sein. Über die gesamte Arbeit verteilt finden sich Hinweise, wie ein UVC performant implementiert werden kann. Sie sind gebündelt und mit Verweisen versehen im Anhang A.2 aufgeführt.

3.2. Die Besonderheiten des UVC gemäß Spezifikation

Statt einer bloßen und wenig hilfreichen Übersetzung aller 18 Seiten der Spezifikation werden in diesem Abschnitt die Besonderheiten des UVC herausgearbeitet. Die Kenntnis dieser Besonderheiten ist Voraussetzung zum Verständnis der folgenden Kapitel.

Wie in Abbildung 3.3 zu sehen, lässt sich die Architektur des UVC sehr übersichtlich darstellen. Er wurde bewusst so einfach spezifiziert. Im Gegensatz zu anderen, realen Computern gibt es jedoch gravierende Unterschiede. Mit dem ausschließlich virtuellen Einsatz konnten

[9] ”Performance is of secondary importance“ [Lorie, 2002b]

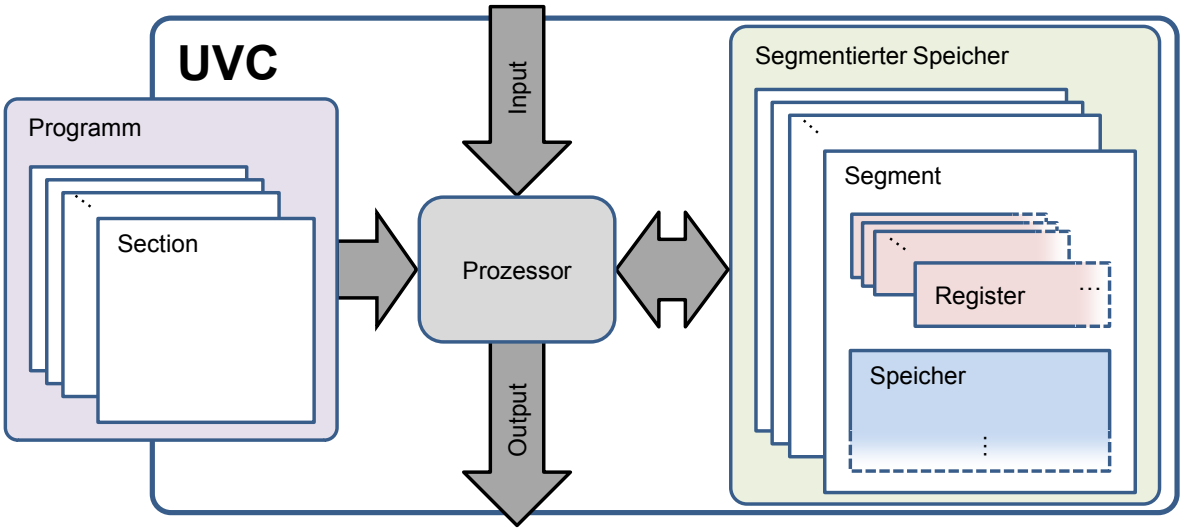


Abbildung 3.3.: Die Architektur des UVC im Überblick

zwei Ziele verfolgt werden. Zum einen sollten Programme für den UVC leicht zu implementieren sein. Zum anderen sollte der UVC bei Bedarf recht schnell allein aus der Spezifikation heraus implementiert werden können.

Auf den ersten Blick wirkt der UVC wie ein gewöhnlicher Computer. Er hat einen zentralen Prozessor, einen segmentierten Speicher und Register. Bei genauer Betrachtung fallen darüber hinaus nur noch Besonderheiten auf, die im Folgenden im Fokus stehen. Gerade im Kontext dieser Besonderheiten werden viele Betrachtungen innerhalb folgender Kapitel überhaupt erst nachvollziehbar. Die Spezifikation selbst betont diese dagegen nicht.

Eine Begründung warum der UVC im Detail so aufgebaut ist, findet sich nicht. Lediglich eine allgemeine Motivation ist von Lorie [2001] schlüssig dargestellt: Er schließt nicht aus, dass sich der UVC an realer Hardware oder anderen virtuellen Maschinen orientiert. Grundgedanke ist jedoch, dass sich der UVC nicht physisch implementieren lassen muss. Er ist daher z.B. auf keine feste Registeranzahl begrenzt. Es gibt keine Bytes, alles ist bitorientiert. Dadurch können leicht verschiedenste Computer emuliert werden. Dagegen soll der eingeschränkte Befehlssatz den Aufwand einer Implementierung reduzieren.

Ein wesentlicher Grund für die unverhofft langsame Ausführung von UVC-Programmen wird an dieser Bitorientierung des UVC festgemacht [van der Hoeven et al., 2005]. Böse Zungen werden behaupten, der UVC sei ein Ergebnis unserer Überflussgesellschaft. Niklaus Wirth bemerkte bereits 1995, dass „Software schneller langsamer als Hardware schneller wird.“ Im Prinzip folgt die Spezifikation diesem „Trend.“ Das Ziel jedoch ist die Universalität der Maschine, die auch in 1000 Jahren noch zum Leben erweckt werden können soll.

Wäre eine bestimmte Wortlänge spezifiziert, müssten Basismaschinen mit abweichender Wortlänge erheblichen Mehraufwand leisten – und zwar unabhängig davon, ob ihr Wort länger oder kürzer ist. Ist das Wort der Basismaschine kürzer, muss der Prozessor mit jeder Operation mehrere Worte „anfassen“ und ggf. Teilergebnisse miteinander „verrechnen.“ Bei längeren Worten müssen nach einer Operation die höherwertigen, also die „überstehenden“ Bits geprüft und ggf. gelöscht werden. Auch lassen sich hier Überläufe (z.B. nach der Addition oder

Multiplikation) nur sehr umständlich feststellen. Dadurch sind wesentlich mehr Befehle auf der Basismaschine notwendig. Bei effizienter Programmierweise sind es zwischen zwei und vier zusätzlichen Befehlen.[10] Noch größer (und leider kaum messbar) wird der Mehraufwand bei der Implementierung in Hochsprachen, wie das hier der Fall ist.[11]

Unabhängig von der Wahl der Wortlänge würde es daher immer Anpassungen geben müssen, die viel Rechenzeit verschlingen. Die Bitorientierung lässt dagegen einen konstanten, von der Hardware weitestgehend unabhängigen Implementierungsaufwand erwarten.

Von dem Ziel andere Computer mit dem UVC vollständig emulieren zu können, ist in aktuelleren Quellen nichts mehr zu finden. Dagegen wird die gezielte Ausrichtung auf die Fähigkeit, mit Bitströmen umgehen zu können, in der Spezifikation [Lorie und van Diessen, 2005] betont. Der UVC ist in mehreren Quellen beschrieben. Während sich die Eigenschaften dabei immer wieder weiterentwickeln, ist der Grundgedanke der unlimitierten Eigenschaften dauerhaft.

Im Folgenden werden die Besonderheiten des UVC gemäß der Spezifikation vorgestellt.

Prozessor Der Prozessor ist die zentrale Einheit, die das Programm Schritt für Schritt abarbeitet. Dabei hat der Prozessor einen Zustand, der sich aus dem aktuellen Befehlszähler (*Instruction Counter*), also einer gespeicherten Adresse der nächsten zu bearbeitenden Instruktion und einem Statusbit (*Condition Flag*) zusammensetzt.[12] Dieses Bit wird nur von zwei Vergleichsinstruktionen gesetzt bzw. überschrieben. In vielen realen Prozessoren ist der interne Zustand wesentlich komplexer. Der Prozessor selbst hat keine Register, insbesondere kein Statuswort.

Rücksprungadressen, die reale Prozessoren an bestimmten Speicherstellen oder in Registern erwarten, verwaltet der Prozessor intern selbst. Fehler durch fehlende oder falsche Rücksprungadressen sind somit ausgeschlossen, was die Programmierung des UVC stark erleichtert. Wie die zugehörige Implementierung des UVC dazu aussehen soll, ist nicht spezifiziert. Diese Eigenschaft lehnt sich an die Architektur kellerbasierter virtueller Maschinen an [Craig, 2006].

Befehlssatz Der Prozessor kennt nur 25 Instruktionen in jeweils nur einer Form. Reale Prozessoren können zumeist mehrere Hundert zusammen mit mehreren möglichen Operandentypen verarbeiten. Diese Minimierung soll die Implementierung in sehr kurzer Zeit gewährleisten. Zum Befehlssatz gehören bedingte und unbedingte Sprünge, Unterprogrammaufrufe, arithmetische und logische Operatoren, Ein- und Ausgabeinstruktionen, Speicher- und Vergleichsinstruktionen. Alle Instruktionen erwarten eine festgelegte Anzahl von Registern als Operanden. Einzige Ausnahme ist eine Instruktion, mit der konstante Werte in Register geladen werden können.

[10] Einen entsprechend umfangreichen und mächtigen Befehlssatz vorausgesetzt.

[11] Erfahrungen aus dem Bereich der Emulation zeigen, dass ein Befehl der emulierten Maschine ca. 500 Befehle der Basismaschine benötigt.

[12] Die Spezifikation erwähnt noch ein *Error Indicator*, der einen Fehlercode enthalten kann und bei Programmende ausgegeben wird. Leider spielt das für den Prozessorstatus und bei der Programmausführung keine Rolle.

Speicher Wie es bei realen Computern auch oft der Fall ist, gibt es einen segmentierten Speicher. Während jedoch für gewöhnlich die Segmente bestehend aus Speicherworten fester Größe, in irgendeiner Form begrenzt sind, kennt der UVC nur bitweise adressierbare Segmente unbeschränkter Größe. Segmente sind dabei keine Teilbereiche des real, in linearer Form verfügbaren Speichers, die sich überlappen könnten. Die Segmente des UVC sind vollständig unabhängig und können theoretisch unbegrenzt groß werden. Leider ist in keiner Quelle dargelegt, woraus sich die Verwendung dieser Speicherstruktur ableitet. Ebenso wenig findet sich eine Motivation der vordefinierten Bedeutungen einzelner Segmente mit den folgenden Nummern:

- *Segment 0* kann von jedem Unterprogramm aus genutzt und bearbeitet werden. Die Spezifikation sieht eine Möglichkeit vor, die Register dieses Segments beim Laden des Programms vorzubelegen.
- *Segment 1* ist für jedes Unterprogramm separat verfügbar. Jedes Unterprogramm hat somit sein eigenes Segment, wobei alle Instanzen eines Unterprogramms, die durch Rekursionen entstehen, dasselbe Segment als Segment 1 sehen.
- *Segment 2* ist das Parametersegment. Ein aufrufendes (Unter-)Programm bekommt das Segment mit der übergebenen Segmentnummer als Segment 2 eingeblendet. Dieses kann bearbeitet werden und dient somit auch der Rückgabe von Ergebnissen.
- *Segmente 3-999* sind gemeinsam nutzbare Segmente. Wie Segment 0 sind auch diese Segmente für alle Unterprogramme in gleicher Weise nutzbar. Es fehlt jedoch die Möglichkeit zur Vorbelegung.
- *Segmente 1000-*∞ sind private Segmente, die jeweils nur für ein bestimmtes Unterprogramm verfügbar sind. Andere Segmente können gleiche Segmentnummern nutzen, sehen und bearbeiten aber dann nur ihr eigenes Segment ohne Auswirkungen auf andere. Der Inhalt privater Segmente ist mit dem Rücksprung aus dem Unterprogramm verloren. Wie alle anderen Segmente auch, können sie aber an aufgerufene Unterprogramme übergeben und dort weiter bearbeitet werden. Das Symbol ∞ steht für eine unendlich große Zahl. In der Theorie ist die Anzahl nicht begrenzt. Im Code selbst müssen die Segmente aber mit einer vorzeichenlosen, 32 Bit großen Zahl angegeben werden. Mit dem Aufruf von Unterprogrammen wird jedoch eine in einem Register abgelegte Nummer übergeben, die ein Segment identifiziert. Die Spezifikation verbietet Segmente mit größeren Nummern nicht.

Register Ein Register enthält einen ganzzahligen Wert und ein separates Vorzeichen. Die sonst übliche Komplementdarstellung findet hier keine Anwendung, was mehrere Darstellungen der Null möglich macht. Der Wert eines Registers wird durch eine Bitfolge repräsentiert. Sie ist nicht begrenzt, aber dem Bedarf angepasst. Somit sind theoretisch unbegrenzt große Registerinhalte möglich. Gemäß der Spezifikation können die Bitfolgen der Register auch führende Nullen enthalten. Lediglich arithmetische Operationen liefern immer Registerinhalte ohne führende Nullen.

Es gibt keine Register im Prozessor. Register sind jeweils in theoretisch unbegrenzter Anzahl ausschließlich den Segmenten zugeordnet. Das macht die Programmierung äußerst flexibel, jedoch sind so Zugriffszeiten auf Register nicht mit denen anderer realer Maschinen vergleichbar.

Register werden immer mit 64 Bit angegeben. Dabei geben die ersten 32 Bit die Segmentnummer an. Die folgenden 32 Bit codieren die Registernummer, wobei das erste Bit das *Indirection Flag* darstellt. Ist dieses gesetzt, wird der Inhalt des angegebenen Registers selbst als Registernummer interpretiert. Die somit verfügbare indirekte Registerangabe ist mit jedem Operanden nutzbar und in dieser Form ein Alleinstellungsmerkmal des UVC. Diese ermöglicht den Zugriff auf ein Register über eine in einem anderen Register gespeicherte Nummer. Einzige Einschränkung: Beide Register sind dem gleichen Segment zugeordnet. Auf den ersten Blick lässt obige Darstellung auf einen 31 Bit großen Wert zur Identifizierung eines Registers in einem Segment schließen. In einem Register lassen sich aber unbegrenzt große Nummern ablegen. Über den indirekten Zugriff lassen sich also Registernummern mit mehr als 2^{31} Bits nutzen. Eine ähnliche Situation wurde bereits für die Segmente aufgezeigt.

Unterprogramme Ein UVC-Programm setzt sich aus verschiedenen Unterprogrammen (*Sections*) zusammen. Jede Section enthält somit einen Teil des Programms. Der Programmfluss beginnt immer am Anfang der ersten Section und kann nur per Unterprogrammaufruf auf andere Sections übergehen. Dies ermöglicht eine sehr intuitive prozedurale Programmierweise und erinnert an Hochsprachenkonstrukte. In einigen Programmiersprachen wie PL/I [Sturm, 2009] oder FORTRAN [Metcalf et al., 2011] gibt es sogenannte *Entry Points*, die einen Sprung direkt an eine bestimmte Stelle des Unterprogramms ermöglichen. Ähnliches wird hier durch die Angabe einer Adresse erreicht, an der der Programmfluss in der aufgerufenen Section beginnt.

Der Bitstrom eines UVC-Programms enthält die Bitcodes der einzelnen Sections. Diese Bitcodes werden im bitweise adressierbaren Speicher jeweils separater Segmente abgelegt. Die Opcodes und auch die meisten Operanden belegen 8, 32 oder 64 Bit im Speicher. Es gibt jedoch die Möglichkeit, Konstanten beliebiger Länge zu codieren. So kommt es regelmäßig vor, dass Opcodes an ungerader Adresse im Speicher abgelegt werden. Eine aus heutiger Sicht effiziente Ausrichtung an Byte- oder Wortgrenzen könnte sich aber in Zukunft als ungünstig herausstellen, wenn ein Byte mehr oder weniger Bits umfasst.

Zu den bereits aufgeführten Sichtbarkeiten der Segmente hat ein Unterprogramm auch Zugriff auf das Segment, das den eigenen Code enthält. Eine genaue Ausgestaltung fehlt aber in der Spezifikation. Selbstmodifizierender Code ist aber nicht klar ausgeschlossen.

3.3. Notation von UVC-Programmen

Die Spezifikation des UVC definiert keine Syntax für UVC-Assemblersprache. Dafür liefert sie im Anhang ein Beispielprogramm, aus dem in Grundzügen eine Syntax ableitbar ist. Das folgende Beispielprogramm besteht aus zwei nebeneinander dargestellten Sections und basiert auf der ableitbaren Syntax, die vollständig im Anhang A.1 angegeben ist.

```
Main
1001

LOADC 1,1    10 1002  # aufzuruf. Sect.
LOADC 1,2     0    0  # entry point
LOADC 1,3    10 1003  # Parametersegm.

LOADC 1003,1  7  100  # Startwert
LOADC 1003,2  8  200  # Zielwert

CALL  1,1 1,2 1,3

STOP
```

```
Call
1002

LOADC 1,1  1  1   # Inkrement
LOADC 1,2  Loop   # Sprungmarken
LOADC 1,3  Ret    # ...vorladen

label: Loop
GRT   2,1  2,2    # Abbruch?
JUMPC 1,3         # Ret
ADD   2,1  1,1    # +1
LOADC 2,1* 16 0xCAFE
JUMP  1,2         # Loop

label: Ret
BREAK
```

Jede Section hat einen Namen, der in der ersten Zeile steht. Obwohl dieser für die Assemblierung keine Rolle spielt, enthalten alle vollständig aufgeführten UVC-Programme innerhalb dieser Arbeit einen. Der Name dient somit lediglich der intuitiven Zuordnung. So steht `Main` innerhalb der vorliegenden Arbeit immer für die Section, die als erstes vom UVC ausgeführt werden soll.

Die Nummer in der zweiten Zeile ist die Nummer der Section, die gebraucht wird, um diese Section aufzurufen. Sie dürfen innerhalb eines UVC-Programms nur einmal vergeben werden.

Das der Spezifikation beliegende Beispiel gibt in der dritten Zeile eine Nummernsequenz an, die gemäß der Spezifikation nur für den Assembler gedacht ist und alle innerhalb der aktuellen Section genutzten Segmente angibt. Wie der Assembler diese Information nutzen soll, wird nicht erklärt. Aus später genannten Gründen (3.4.1) ist diese Liste bei vollständigen UVC-Programmen jedoch obligatorisch, wird aber innerhalb der vorliegenden Arbeit nicht aufgeführt.

Abweichend zu dem Beispiel im Anhang der Spezifikation wird ein Komma genutzt, um Register deutlicher zu kennzeichnen. Registerangaben bestehen immer aus einer Segmentnummer gefolgt von einer Registernummer. Bei einigen Instruktionen sind so sechs Zahlen aneinander gereiht. Das Komma erleichtert die direkte Zuordnung.

Ein wichtiger Aspekt lässt sich aus der Spezifikation nicht ableiten: die Darstellung des indirekten Registerzugriffs. Lediglich in einer Grafik ist eine Registernummer gefolgt von einem Stern dargestellt. Daher wird der indirekte Registerzugriff durch ein Stern (`*`) gekennzeichnet, der direkt der Registernummer folgt. In der rechts dargestellten Section findet sich ein solcher Registerzugriff.

Das abgebildete UVC-Programm demonstriert einige Grundkonzepte des UVC. Das Hauptprogramm ist links abgebildet.[13] Darin wird mit den ersten drei Instruktionen ein Unterprogrammaufruf vorbereitet. Es soll die Section mit der Nummer `1002` aufgerufen werden. Als

[13] Ein Unterprogramm, also eine Section, wird zum Hauptprogramm, weil dessen Bitcode als erstes im Bitstrom des UVC-Programms gespeichert ist. Der Name oder die Nummer spielen dabei keine Rolle. Im selbst entwickelten Assembler werden die verwendeten Unterprogramme in einem Reiter organisiert. In der resultierenden Reihenfolge werden sie assembliert und gespeichert.

entry point, also als Startadresse des ersten auszuführenden Befehls, wird 0 angegeben. Mit dem Aufruf eines Unterprogramms wird auch eine Nummer eines Parametersegments übergeben. Hier ist es die Nummer `1003`. Eine Segmentnummer über 1000 weist ein privates Segment aus, das nur von der aktuellen Section gesehen und bearbeitet werden kann. Die einzige Ausnahme ist die Verwendung als Parametersegment, die, wie in diesem Beispielprogramm, auch dem aufgerufenen Unterprogramm den vollen Zugriff ermöglicht.

Die beiden folgenden Instruktionen belegen zwei Register des erwählten Segments mit den Werten `100` und `200`. Diese Werte können im Unterprogramm direkt genutzt werden. Diese beiden Zeilen demonstrieren somit die Parameterübergabe bei einem Unterprogrammaufruf.

Der eigentliche Aufruf des Unterprogramms erfolgt mit der Instruktion `CALL`. Als Operanden werden drei Register erwartet, die zu Beginn mit Werten vorgeladen wurden. Die Verwendung von konstanten Werten ist nur mit `LOADC` möglich. Daher finden sich sehr viele dieser Instruktionen im Code. Alle anderen Instruktionen erwarten ausschließlich Register einer festen Anzahl.

Nach dem Unterprogrammaufruf wird die Programmabarbeitung in der rechts abgebildeten Section fortgesetzt. Hier wird ein Register mit einer `1` geladen, um im Programm als Inkrement zu dienen. Es folgen zwei Instruktionen, die Sprungmarken in Register laden. Auch bedingte (`JUMPC`) und unbedingte (`JUMP`) Sprünge erwarten immer ein Register als Operanden.

Das Unterprogramm zählt in einer Schleife den Wert des Registers `2,1` hoch. Im aufgerufenen Unterprogramm wird das Parametersegment als das Segment mit der Segmentnummer `2` eingeblendet. Der Wert des Registers ist eingangs `100` und wird bis auf `200` hochgezählt. Diese Werte sind die übergebenen „Parameter:“

Eine weitere Besonderheit des UVC ist innerhalb des Schleifenrumpfes zu sehen. Mit Hilfe des indirekten Registerzugriffs werden die Register mit den in der Schleife hochgezählten Nummern mit dem hexadezimalcodierten Wert `CAFE` belegt. Diese 101 Register können als Rückgabewerte betrachtet werden.

Die Abarbeitung des Unterprogramms wird mit `BREAK` beendet. Der Programmfluss wird im aufgerufenen Unterprogramm, in diesem Fall das Hauptprogramm, fortgesetzt. `STOP` schließlich stoppt die Programmausführung.

3.4. Forschungsstand Teil I

Dieser erste Teil der Darstellung des Forschungsstands fokussiert die von IBM entwickelten Prototypen und stellt die damit erzielten Ergebnisse dar. Dem zweiten Teil, nämlich der Abgrenzung hinsichtlich anderer virtueller Maschinen, widmet sich ein anschließendes Kapitel.

3.4.1. IBM’s Prototypen und deren Nutzung

Der UVC wurde bereits implementiert und evaluiert als die Spezifikation noch weiterentwickelt wurde. Einzelne Details werden durch die Aufsätze von Lorie deutlich. Anhand dieser Details wird auch der Charakter einer sich fortentwickelnden Spezifikation sichtbar. Auch mit der Veröffentlichung der Spezifikation machen die Autoren Lorie und van Diessen [2005]

deutlich, dass sie nicht daran glauben, dass die Spezifikation keinen Änderungen mehr unterliegen wird.

Prototyp im Rahmen des „Proof of Concept“

Dieser Prototyp wird in zwei Quellen mit kleineren Abweichungen beschrieben, sodass auf eine zwischenzeitliche Weiterentwicklung geschlossen wird. Die Beschreibungen sind aber so ähnlich, dass es sich hierbei um nur einen Prototypen handeln kann.

Das Konzept wurde vor der Entwicklung des ersten Prototypens bereits sehr ausführlich beschrieben. Lorie [2001] wies mit der ersten Beschreibung seines Konzepts als nächsten Schritt die Implementierung eines ersten Prototypen aus, der dazu dienen sollte, die bisherigen Designentscheidungen zu überprüfen und die Tauglichkeit des Konzepts zu validieren.

In Kooperation mit der *Koninklijke Biblitheek*, der Nationalbibliothek der Niederlande, kam schließlich ein „eingeschränkter“ Prototyp zum Einsatz [Lorie, 2002a]. Der Prototyp unterstützte die unbegrenzten Dimensionen nicht, sondern unterstützte maximal 100 Segmente mit je maximal 8 MB Speicher und je 100 je nur 32 Bit großen Registern. Das wurde als ausreichend bewertet.[14] Diese Bewertung wurde mit der Idee gerechtfertigt, solche Grenzen in der Versionsnummer des UVC mitzuführen zu können. Folgende Implementierungen des UVC müssten mindestens diese Grenzen erreichen. Diese Idee wurde später zu Recht verworfen.

Welche Programmiersprache genutzt wurde, wie umfangreich der Quellcode gewesen ist und vieles mehr bleibt dabei offen.

Der Prototyp wurde dazu eingesetzt, um GIF- und JPEG-Bilder zu decodieren und *“some aspects of PDF”* Lorie [2002a]. Was genau Letzteres bedeutet, schreibt Lorie nicht. Leider ist weder die komplette Spezifikation noch der Prototyp selbst verfügbar. Auch sind die Programme nicht veröffentlicht worden. Der Auszug aus der Spezifikation lässt aber deutliche Abweichungen zur später veröffentlichten Spezifikation erkennen. So war je eine Vergleichsoperation für Zeichen und normale Zahlen vorgesehen, die den *condition code* auf -1, 0 oder 1 setzen können. Die einzige Instruktion für bedingte Sprünge war entsprechend komplex spezifiziert. Aufgrund der von Lorie durchgeführten Tests wurden bereits drei Instruktionen überarbeitet.

Bis zur Erstellung des Berichts zum „Proof of Concept“ wurde der UVC und die Spezifikation weiterentwickelt. Die Leistungsdaten des UVC blieben gleich, aber kleinere Details des spezifizierten Befehlssatzes änderten sich. So bietet dieser 23 statt 21 Befehle und eine „entschärfte“ Syntax für einen bedingten Sprung [Lorie, 2002b]. Die nur schwer verständliche Spezifikation der Schnittstelle und die kompliziert zu speichernden UVC-Programme blieben dagegen gleich.

Mit der Vorstellung, dass zu archivierende PDF-Dokumente mit aktuellen Werkzeugen seitenweise in einfache Bilddateien überführt und Texte zusammen mit den Positionen im Bild extrahiert werden könnten, bewertet die Studie das Ergebnis insgesamt als Erfolg:

> *We successfully demonstrated that the concepts are quite sound, can be applied to archiving PDF documents, and can be implemented.* [Lorie, 2002b]

[14] *“This is quite enough for the applications at hand.”* [Lorie, 2002a]

In den detaillierteren Ausführungen wird angemerkt, dass sie planen, zur Vermeidung von unnötig langen Sequenzen von UVC-Instruktionen zwei neue Instruktionen einzuführen. Die Speicherstruktur hingegen wurde noch während der Testphase angepasst. Ein gemeinsam genutztes Segment ermöglicht einen deutlich effizienteren Code.

Im *Nationaal Archief* wurde 2001 für das *Digital Preservation Testbed* eine initiale Version des UVC entwickelt. Im anschließenden Test wurde die Archivierungsmethode mit anderen Strategien wie der Migration oder der Archivierung auf Basis des XML-Formats verglichen. Für den Anwendungsfall zur Erhaltung von Spreadsheets, also von Tabellen, erwies sich der UVC als wenig attraktiv. Allerdings wird angemerkt, dass der UVC noch nicht vollständig entwickelt war und somit nicht sein ganzes Potential zeigen konnte.

Prototyp V1.0 im Rahmen des Demonstration Tools

Während der fortgesetzten Forschung am UVC und der damit verknüpften Archivierungsmethode wurde ein weiterer Prototyp in der Programmiersprache Java implementiert. Dieser ist Bestandteil eines von IBM veröffentlichten Testpakets.[15] Neben dem UVC sind zwei UVC-Anwendungen zur Decodierung von JPEG- und GIF-Bildern und entsprechende Treiberanwendungen enthalten. Das Paket enthält keinerlei Quellcode. Einzige Ausnahme ist der Quellcode für eine UVC-Test-Anwendung, die rekursiv die Fakultät einer übergebenen Zahl berechnet.

Diese Implementierung entstand zwischen 2003 und 2004, was aus den Textdateien hervorgeht, die dem Paket beiliegen. Maßgeblich beteiligt waren demnach van der Hoeven et al. [2005], die diese Implementierung in einem Aufsatz beschrieben. Die Version des UVC wird in den Textdateien mit 1.3.2 angegeben, zudem protokollieren sie einige Veränderungen und damit Entwicklungsschritte, die teilweise vom Gedanken der Effizienz und der Vereinfachung der Programmierung getrieben wurden. Immer wieder waren dadurch die UVC-Anwendungen anzupassen und erneut zu assemblieren.

Die vorhandene Java-Implementierung von IBM passt nicht zur gegebenen Spezifikation des UVC [Lorie und van Diessen, 2005]. Im Folgenden werden diese Abweichungen beschrieben.

Das dem Paket beiliegende Beispiel zur Berechnung der Fakultät liegt im Quellcode und in Assemblersprache vor. Anhand dessen konnte der abweichende Programmaufbau und einige Abweichungen der Opcodes festgestellt werden. Der UVC V1.3.2 nutzt entgegen der Spezifikation offenbar die im Assemblerprogramm geforderte Liste der genutzten Segmente pro Section. Sie wird genutzt, um Zugriffe auf andere Segmente mit Fehlermeldungen zu behandeln. Weitere Motive erschließen sich nicht.

Ein selbstentwickelter Assembler wurde an die ebenfalls abweichende Syntax und an die veränderten Opcodes angepasst. Mit einigen Tests wurden drei bereits in früheren Versionen genutzte Instruktionen gefunden. Die komplette Opcodetabelle findet sich im Anhang A.1.3. Alle in dieser Arbeit aufgeführten Beispiele nutzen die dort fixierten Anpassungen der Befehlsnamen.

Bemerkenswert ist dabei die Anpassung der Sprungbefehle. So wird jetzt `JUMP` anstatt `BR`

[15] http://www.alphaworks.ibm.com/tech/uvc

verwendet. Für gewöhnlich sind *branches* Sprünge, deren Reichweite auf eine kleine Region im Code begrenzt ist, dafür aber weniger Platz zur Codierung benötigen, da sie die Sprungziele relativ angeben [Craig, 2006]. Diese und weitere Anpassungen der abgewandelten Syntax wurden komplett übernommen, auch wenn insbesondere die Instruktion `REGLENGTH` extrem lang und umständlich erscheint. Das im Beispielprogramm bereits eingeführte Komma ist eine eigene, davon unabhängige Erweiterung.

Ebenfalls im Assembler berücksichtigt wurde die Schreiberleichterung bei den Sprungbefehlen. So finden sich im Beispiel von IBM Anweisungen der Art:

```
JUMP  1,1 LOOP
```

Aus dieser Anweisung werden vom Assembler zwei generiert, wobei `LOOP` durch einen realen Wert der Sprungmarke des Quellcodes ersetzt wird:

```
LOADC 1,1 32 LOOP
JUMP  1,1
```

Zwei Instruktionen wurden nicht spezifikationskonform umgesetzt. `LOADC` setzt in der Version 1.3.2 immer das Vorzeichen auf positiv. Der Zustand eines Registers ist somit nach dem Laden einer Konstante definiert. Gemäß Spezifikation würde das vorhergehende Vorzeichen bestehen bleiben.

Die Instruktion `AND` bekommt zwei Register als Operanden übergeben. Sollte dabei ein Register kleinerer Länge sein, so würde es gemäß Spezifikation virtuell um führende Einsen erweitert. Das führt allerdings zu wenig intuitiven Ergebnissen. So wäre `10101` das Ergebnis einer UND-Verknüpfung von `10101` und `111`. Die Version 1.3.2 weicht hier ab, korrigiert diesen offensichtlichen Fehler und liefert `101`.

Offensichtlich beeinflusste auch hier die Implementierung des UVC und die aus den Tests gewonnenen Erfahrungen die Spezifikation fortlaufend. Die Diskrepanz lässt vermuten, dass der UVC noch weiter angepasst wurde, während die Spezifikation bereits fixiert und veröffentlicht war. Einige erprobte Verbesserungen und erkannte Fehler wurden so nicht mehr berücksichtigt.[16]

Zu Beginn der vorliegenden Arbeit war keine weitere Implementierung verfügbar. Und damit insgesamt keine mit der Version 0 kompatible Implementierung, die sich für Vergleichstest hätte heranziehen lassen. Da aber durchaus neben der Vollständigkeit und Korrektheit der Spezifikation auch ein Vergleich der daraus resultierenden Ergebnisse erfolgen sollte, bestand ein erster Schritt in der Anpassung der Spezifikation zur Konformität mit dem verfügbaren UVC der Version 1.3.2. Alle eigenen Implementierungen berücksichtigen die aufgeführten Abwandlungen zum Zweck der Vergleichbarkeit.

Dieser UVC wurde innerhalb der Kooperation mit der KB erfolgreich zur Archivierung von Bildern getestet. Das Ergebnis wurde von Wijngaarden und Oltmans [2004] als erfolgreich bewertet.

Der UVC ist innerhalb des Archivsystems der KB (das Digital Information Archiving System – DIAS), das vorrangig die Emulation nutzt, eine von mehreren Übergangslösungen. Bewertet wird die den UVC nutzende Archivierungsmethode als Möglichkeit, den Inhalt digitaler

[16] Van Diessen ließ in persönlicher Korrespondenz die Notwendigkeit einiger kleiner Updates erkennen, verwies jedoch auf die offizielle Version: *We know that there are some small updates to be made to the definition but we have an official version of the specification.*

Objekte zu erhalten, aber nicht das Verhalten oder die Funktionalität. Diese Methode stellt derzeit einen aktiven Pfad für den Erhalt von Bildern innerhalb des Archivs dar, der auch für PDF Dateien geeignet ist, falls deren Inhalt zuvor entsprechend seitenweise aufbereitet wurde. Wobei Wijngaarden und Oltmans [2004] lediglich die Konvertierung in JPEG Dateien ansprechen und die noch im Proof of Concept angesprochene Textextrahierung nicht mehr erwähnen. Vor diesem Hintergrund ist das Ergebnis zu relativieren.

Prototyp V1.1 im Rahmen des Updates

Mit der Veröffentlichung eines neuen Updates im August 2010 sind weitere Tools und eine Implementierung des UVC in C++ verfügbar. Dieser UVC ist entgegen der Java-Implementierung konform zur Spezifikation und entstand im Rahmen einer Bachelorarbeit im Jahr 2006.[17]

Bis auf die Instruktion **AND**, die wie oben beschrieben intuitiv korrigiert umgesetzt wurde, ist diese Implementierung vollständig spezifikationskonform. Aufgrund des zur Version 1.3.2 abweichenden Programmaufbaus, der vollständig veränderten Opcodes und der abweichenden Semantik der Instruktion **LOADC** sind die Programme nicht kompatibel.

Im Gegensatz zur Java-Implementierung ist der Quellcode zur Implementierung in C++ vorhanden.[18] Dieser Quellcode dient der plattformabhängigen Kompilierung, die jedoch ein tieferes Verständnis der Programmentwicklung in C++ voraussetzt. Für die ungeübten Nutzer ist eine Anleitung beigelegt, die die Nutzung einer virtuellen Maschine zur Virtualisierung eines Ubuntu-Betriebssystems beschreibt. Trotz dieser Umgehung der plattformabhängigen Einrichtung erfordert dieses Update einen enormen Zeitaufwand von mindestens 4-6 Stunden. Zusätzlich müssen sämtliche Quellen – auch die der Beispielprogramme – zunächst kompiliert bzw. assembliert werden. Das ausführliche Testen und „Begreifen“ dieser Version des UVC bleibt somit leider nur einer sehr kleinen Gruppe vorbehalten.

Bei genauer Betrachtung des Codes fällt auf, dass diese Implementierung nicht die technischen Möglichkeiten ausschöpft. So wurde zwar der Speicher eines Segments über einen balancierten Suchbaum abgebildet, allerdings wurde jeweils nur ein 32 Bit großes Wort abgelegt. Gerade beim Verarbeiten des Bitcodes, der auch innerhalb eines Segments gespeichert ist, treten so im Schnitt etwa 1,6 Zugriffe auf den Baum mit jedem gelesenen Byte oder Wort auf. Für Instruktionen mit drei Operanden sind das im Schnitt 4,8 Zugriffe, mit der Komplexität $O(log(n))$, wobei n die Anzahl der vom Programm im Segment genutzten Speicherworte ist. Eine weitere Fehlentscheidung ist die Organisation aller Segmente in einer Struktur und die Nutzung eines zusammengesetzten Schlüssels. Nur in wenigen Fällen werden alle Bestandteile des Schlüssels genutzt. Sinnvoller ist die Organisation der verschiedenen Typen der Segmente in jeweils eigenen Strukturen. Der Code überzeugt in seiner Qualität nicht.

Von einer in C++ implementierten UVC würde man sich eine erheblich effizientere Version erwarten. Dem direkten Vergleich entzieht sich dieser UVC jedoch aufgrund der o.a. Inkom-

[17] In einer begleitenden `readme.txt` ist der Verweis auf den Autor Xander Tamminga und das Jahr zu finden. Auf seinem Profil der Verweis auf die Bachelorarbeit: `http://nl.linkedin.com/in/xandert`
Aus der Korrespondenz mit van Diessen erschließt sich, dass ursprünglich zwei Studenten mit der Entwicklung beauftragt wurden, die auf Basis der fixierten Spezifikation arbeiteten.

[18] Dieser diente als Basis des im Rahmen einer vom Autor der vorliegenden Arbeit betreuten Masterarbeit weiter verfolgten Ansatzes der Crosskompilierung von UVC-Programmen (siehe Kapitel 7).

patibilität. Erst nach Anpassung des C++ Codes des UVC und des JPEG-Decoders ist der Vergleich möglich. Hierbei erweist sich dieser UVC gegenüber der Java-Implementierung um den Faktor 4,85 langsamer. Das spricht für die Java-Implementierung.

Mit dem Paket wurde auch ein Assembler und ein Compiler für einen MiniJava Dialekt veröffentlicht. Die im Paket enthaltenen Java-Quellcodes gehen auf N.J.C. Kol und J.R. van der Hoeven zurück und wurden von Tamminga teilweise angepasst.

Der Assembler unterstützt in der verfügbaren Version die Syntax und die Opcodes der Spezifikation. Die Autoren Kol et al. [2006] veröffentlichten einen Aufsatz zur Entwicklung des Compilers. Dieser Compiler unterstützt eine Form von MiniJava, die zwar Klassen und Methoden kennt, aber keine Klassen vorhandener Bibliotheken einbinden kann. Die Ein- und Ausgabe ist dabei auf die Möglichkeiten des UVC ausgerichtet. So erzeugt die Java-Anweisung `System.out.println(7)` die Ausgabe `00000001 00000003 7`. Der Ganzzahltyp wird in seiner Länge nicht begrenzt. Der Compiler unterstützt, wie die Java Virtual Machine [Dalheimer, 1997], Fließkommazahlen doppelter Genauigkeit nach IEEE 754, wofür eine Library in Form einer UVC-Section dem UVC-Programm eingefügt wird. Darin werden Fließkommazahlen immer mit 64 Bit dargestellt, die mögliche beliebige Genauigkeit wird nicht genutzt. Die Anweisung `System.out.println(13.7)` erzeugt bei der Ausführung des UVC-Programms die Nachricht `00000002 00000040 402B666666666666`. Der Messagetyp 2 deutet jetzt auf eine Fließkommazahl hin. Neben den Datentypen `int` und `float` bietet der MiniJava-Dialekt seit 2007 auch den Typen `char`.

Dieses auch erst seit der Veröffentlichung 2010 verfügbare Tool wird genutzt, um einen in diesem MiniJava-Dialekt verfassten Decoder für Kalkulationstabellen in UVC-Assemblersprache zu übersetzen. Der Assembler wiederum erzeugt daraus das UVC-Programm.

Die Übersetzung ist alles andere als effizient, da sie sich sehr stark am *Operandenstack* der Java Virtual Machine orientiert. Mit den beliebig vielen Registern des UVC könnte das effizienter umgesetzt werden. So aber werden die Register genutzt, um ausschließlich einen Keller abzubilden.

Obwohl der Compiler keinen effizienten Code erzeugt, demonstriert er dennoch überzeugend die Machbarkeit. Ob sich allerdings dieser MiniJava-Dialekt zur Erstellung komplexerer UVC-Anwendungen eignet, darf bezweifelt werden.

Sowohl der Assembler als auch der Compiler nutzten mit den Tools JavaCC[19] und JTB[20] automatisch generierte Komponenten. So lassen sich Assembler und Compiler leicht an eine fortentwickelte Spezifikation anpassen.

Die früheren Versionen der Tools kamen bereits mit dem UVC V1.3.2 zum Einsatz. Deren aktualisierten und spezifikationskonformen Versionen dienten nur der Demonstration einer einzigen Anwendung: Der Archivierung von Kalkulatiostabellen. Es finden sich keine Belege für eine aktive Nutzung der in C++ implementierten Version. Auch gibt es keine neuen Belege, die die mit dem Proof of Concept getroffene Einschätzung revidieren: *„[...] the UVC approach was stated less attractive for spreadsheets than migration and XML [...]“* [van der Hoeven et al., 2005].

[19] Java Compiler Compiler: `javacc.java.net`
[20] Java Tree Builder: `compilers.cs.ucla.edu/jtb`

3.4.2. Kritische Wertung der bisherigen Entwicklung

Es ist festzustellen, dass der UVC nur im Umfeld des gemeinsamen Projekts von IBM und der KB zur Anwendung kam. Die erste vollständige Entwicklung von van der Hoeven et al. [2005] wurde parallel zur Entwicklung der Spezifikation erstellt. Über die zweite Implementierung in C++ sind Einzelheiten nur über den dürftig dokumentierten Quellcode zu entnehmen. Die dabei gesammelten Erfahrungen bzgl. Entwicklungszeit, notwendiger Rückfragen oder gar Fehlinterpretationen der Spezifikation sind bisher weder veröffentlicht noch flossen sie in eine weiterentwickelte Spezifikation ein. In Hinblick auf die Vollständigkeit der Spezifikation sind diese beiden Implementierungen daher kein Beleg. Eine kritische Betrachtung der Spezifikation findet sich nicht.

Bezüglich der zu erwartenden Entwicklungszeiten finden sich keine konkreten Angaben, auf die sich Archivbetreiber verlassen könnten. Bei Gladney und Lorie [2005] ist zu lesen, dass der UVC mit dem Ziel entwickelt wurde, in weniger als ein Arbeitsjahr implementiert zu werden. Auch die Angaben von Oltmans und Kol [2005] sind wenig hilfreich. Zwar geben sie für die Erstellung des UVC, inklusive Forschung und Design, 32 volle Arbeitswochen an. Auch die Implementierungszeit für den UVC wird mit 160-Arbeitstunden recht konkret angegeben. Basierend auf einer 40-Stunden-Woche wären das vier Arbeitswochen. Aber auch mit diesen vagen Angaben lässt sich der reale Aufwand nicht verlässlich abschätzen. Was offen bleibt sind folgenden Fragen: Auf welche der Implementierungen beziehen sich die Autoren? Ist es die qualitativ hochwertige Implementierung in Java? Gemäß den begleitenden Textdateien ist jedoch von stetigen Erweiterungen bis zur Version 1.3.2 zu lesen. Oder ist damit die initiale, sehr beschränkte Version mit den 100 Segmenten gemeint? Hier wären verlässliche Angaben, die sich auf einen definierten Funktionsumfang des UVC beziehen, äußerst wichtig für eine realistische Kostenabschätzung.

Dass der UVC zweimal für heutige 32-Bit-Systeme implementiert wurde, ist kein Indiz für Universalität, ebenso wenig die Verwendung der Programmiersprache Java. Zumal die zwei genutzten Sprachen Java und C++ auf dem Paradigma der Objektorientierung beruhen. Das Nutzen bereits verfügbarer Klassen für die Arithmetik großer Zahlen und für die Organisierung von dynamischen Strukturen macht die Implementierung in kurzer Zeit möglich. Wie sieht es in anderen Sprachen mit anderen Paradigmen aus? Gibt es unentdeckte Abhängigkeiten von den genutzten Programmiersprachen? Ist der UVC tatsächlich universal spezifiziert oder lassen sich bestimmte Systemabhängigkeiten erkennen, die sich stark auf Entwicklungszeiten oder Laufzeiten der UVC-Anwendungen auswirken können? Antworten kann IBM so nicht liefern.

Die verfügbaren Assembler und Compiler sind hilfreich für zukünftige UVC-Anwendungsentwickler unserer Generation. Allerdings ist der Assembler sehr unkomfortabel und der Compiler erzeugt extrem ineffizienten Code. Zudem sind über den unterstützten MiniJava-Dialekt viele Vorzüge des UVC verborgen. Die indirekte Registeradressierung, die unbegrenzten Register und der unbegrenzte, lineare Speicher sind vom Anwendungsentwickler nicht nutzbar. Auch macht der Compiler von effizienten Möglichkeiten des UVC-Befehlssatzes keinen Gebrauch. Die Basis der Werkzeuge steht nur teilweise zur Verfügung. So findet sich nur die Definition der MiniJava Syntax für JavaCC, jedoch ohne Beschreibung. Die Definition für den Assembler fehlt komplett. Interessierte Nutzer können sich so nicht bei der Verfeinerung

und Weiterentwicklung einbringen. Zu beantworten ist auch die Fragestellung, ob sich solche Tools als UVC-Anwendungen implementieren lassen, um ebenfalls langfristig nutzbar zu bleiben. Auch wäre es aufschlussreich zu untersuchen, ob sich anstatt MiniJava eine neue Sprache besser eignen würde – eine Hochsprache mit unbegrenzten Datentypen und Konstrukten geeignet, um die Vorteile des UVC auszunutzen. Nur der erste Aspekt kann innerhalb dieser Arbeit weiter verfolgt werden, der zweite Aspekt hingegen würde eine eigenständige Arbeit rechtfertigen.

Neben dem Erhalt von statischen Dokumenten, wie Bildern und Kalkulationstabellen wurden bisher keine weiteren Anwendungsfälle betrachtet. Die Verwendung des UVC, um andere Maschinen zu emulieren ist nach van der Hoeven et al. [2005] ein interessanter Anwendungsfall, für den aber noch einige wichtige Fragen neben der Performanz zu klären sind, wie z.B. die Erweiterung der Ein- und Ausgabefunktionalität. Hier wäre Pionierarbeit zu leisten, weitere Anwendungsbereiche aufzudecken und deren Tauglichkeit für den Bereich der Archivierung zu werten.

4. Abgrenzung des UVC zu anderen virtuellen Maschinen

Dieses Kapitel verortet den UVC im Bereich virtueller Maschinen (VM). Dazu wird zunächst der für die vorliegende Arbeit relevante Zusammenhang zwischen Emulatoren und VM erarbeitet. Darauf folgt die Darstellung verschiedener Anwendungsgebiete VM mit jeweils relevanten Vertretern. Anhand deren Eigenschaften werden sie für die Nutzung im Bereich der Langzeitarchivierung allgemein und für die von Lorie angedachte Archivierungsmethode im Speziellen bewertet. Dieses Kapitel beantwortet abschließend die Frage, ob der UVC neu entwickelt werden musste, oder ob eine bereits verfügbare VM ebenfalls geeignet gewesen wäre.

4.1. Virtuelle Maschinen

Im Kapitel 2 wurden Emulatoren als Software beschrieben, die versucht eine andere Umgebung nachzubilden. Dabei wurden verschiedene Ebenen der Emulation vorgestellt. VM sind ebenfalls Programme, also Software.

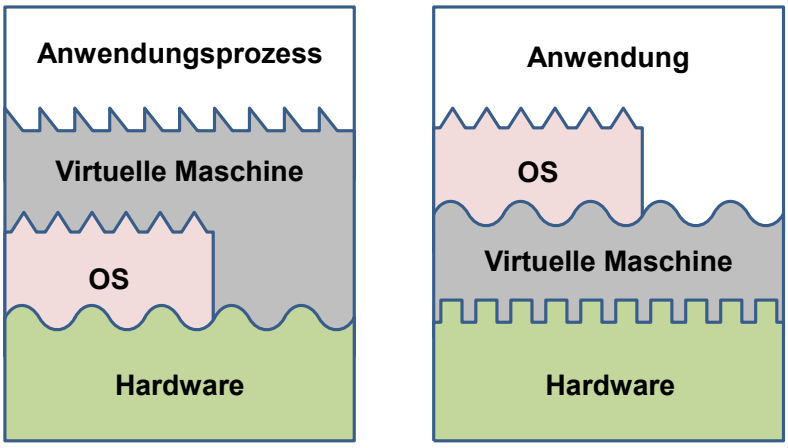
Eine Maschine ist größtenteils zusammengesetzt aus einem oder mehreren Prozessoren mit einem spezifischen Befehlssatz, einem Speicher fester Größe und Ein- und Ausgabegeräten. Aus der Sicht eines Anwendungsprogramms ist eine Maschine eine Kombination eines Betriebssystems und der aus Anwendersicht verfügbaren Hardware [Smith und Nair, 2005].

VM sind Programme, die die Komponenten einer gewünschten Maschine auf eine verfügbare Maschine abbilden (virtualisieren) und Anwendungen über Schnittstellen (Interfaces) zur Verfügung stellen. Anwendungsprogramme erwarten vom System bestimmte Schnittstellen, um auf Speicher, Ein- und Ausgabegeräte usw. zugreifen zu können. Zudem muss deren Code, der für einen bestimmten Befehlssatz erstellt wurde, korrekt verarbeitet werden können.

Aufgrund der verschiedenen Sichten können VM auf verschiedenen Ebenen zum Einsatz kommen. Smith und Nair [2005] unterscheiden die Prozess VM und die System VM.

Die in Abbildung 4.1(b) dargestellte System VM zeichnet aus, dass sie direkt auf der Hardware ausgeführt wird, also ohne Unterstützung eines Betriebssystems auskommt. Somit lässt sich eine „eckige“ Hardware so nutzen, als wäre sie „gewellt.“ Es sind auch andere Szenarien auf dieser Ebene möglich. So könnte eine System VM genutzt werden, um mehrere virtuelle Instanzen einer einzelnen Hardware zu Verfügung zu stellen. Auf diese Weise könnten parallel mehrere (auch verschiedene) Betriebssysteme auf einer Hardware ausgeführt werden.

Die Prozess VM ist schematisch in Abbildung 4.1(a) dargestellt. Diese werden in einem bestehenden System gestartet und können daher die Schnittstellen des Betriebssystems (OS für *Operating System*) nutzen. Sie selbst bieten im Gegensatz zur System VM keine Schnittstellen für komplette Systeme, sondern für einzelne Prozesse. Die Java Virtual Machine (JVM) ist



(a) Prozess VM (b) System VM

Abbildung 4.1.: Virtuelle Maschinen auf zwei Ebenen (nach Smith und Nair [2005])

ein prominentes Beispiel einer Prozess VM. Diese Anwendungprozesse sind gegenüber den Anwendungen der System VM nur in der Laufzeit limitiert. So können Anwendungsprozesse, in Abhängigkeit der durch die Prozess VM zur Verfügung gestellten Schnittstellen, ebenfalls andere Systeme abbilden. Beispiele hierfür sind JaC64 und JPC.[1]

Wie gehören VM und Emulatoren zusammen? Emulatoren, die eine Abspielumgebung auf der Ebene der Hardware nachbilden, sind VM. Sie bilden ebenfalls Komponenten wie Speicher, Prozessoren und Ein- und Ausgabegeräte einer Maschine ab. Hardware-Emulatoren können als Prozess VM oder als System VM realisiert werden. Im Bereich der Langzeitarchivierung finden sich jedoch keine Beispiele genutzter System VM. Ein bestimmender Grund ist die anvisierte Weiterverarbeitung der über die Emulation zugreifbaren Informationen in aktuellen Anwendungen. Werden originale Abspielumgebungen und aktuelle Anwendungen im gleichen System ausgeführt, lassen sich Schnittstellen einfacher gestalten bzw. bereits vorhandene einfach nutzten.

Wo ist der UVC einzuordnen? Der UVC wird von Lorie und van Diessen [2005] Computer genannt, weil seine Architektur auf Konzepten wie Speicher, Register und Basisbefehlssatz beruht, die seit dem Beginn der Computerära existieren. Diese Verwendung des Begriffs Computer ist zwar nachvollziehbar, da aber sehr viele VM auf den gleichen Konzepten beruhen, könnten sich viele andere VM auch VC nennen. Betrachtet man die Verwendung der Konzepte Register und Speicher näher, fällt auf, dass in der gesamten Computerära derartige Ausprägungen noch gar nicht vorkamen. Der UVC ist eine VM.

Das Anwendungskonzept des UVC sieht vor, dass er Eingaben in Form von Messages von externen Anwendungen bekommt und auch solche ausgibt. Das diese Ausgaben aufbereitende Programm kann dabei nicht auf dem UVC ausgeführt werden, da es wiederum nur solche Messages ausgeben kann. Ein Programm zur Weiterverarbeitung der Informationen, die sogenannte *Restore Application*, muss daher auf einem aktuellen System ausgeführt werden.

[1] JaC64 ist ein in Java implementierter Emulator für den C64: `http://www.jac64.com` Der Emulator JPC ist ebenfalls in Java implementiert und emuliert eine x86-Architektur [Newman und Dennis, 2009].

Denkbar wäre, die für den UVC notwendigen Schnittstellen über hardwarebasierte Ein- und Ausgabekanäle zu realisieren und Programme zur Aufbereitung und Weiterverarbeitung auf separaten Systemen auszuführen. Der Aufwand wäre jedoch enorm, weil zu einer maschinennahen Implementierung die Hardware sehr aufwendig entwickelt werden müsste und nur von begrenzter Haltbarkeit wäre. Der UVC ist daher nicht als System VM geeignet.

Obwohl von Lorie [2001] angedacht, ermöglicht die aktuelle Spezifikation des UVC nicht, für den UVC auch Emulatoren realer Hardware zu entwickeln. Wie Gladney und Lorie [2005] richtig anmerken, müssten UVC-Anwendungen (mindestens) eine Systemuhr lesen und Interrupts handhaben können, zusätzlich müsste der UVC Anwendungsprogrammen einen *fork*-Mechanismus für Mehrfachprogrammbetrieb und Synchronisationsmechanismen bieten.

Neben diesen fehlenden Eigenschaften, sind auch die Ein- und Ausgabekanäle dahingehend hinderlich spezifiziert. Sobald ein `IN` abzuarbeiten ist, blockiert der UVC so lange, bis er eine Nachricht empfängt. Das Prüfen, ob überhaupt eine Nachricht zum Lesen ansteht, ist einer UVC-Anwendung nicht möglich.

4.2. Einsatzgebiete virtueller Maschinen – Forschungsstand Teil II

Der UVC wurde speziell für die Langzeitarchivierung entwickelt. Dieser Abschnitt stellt verschiedene Einsatzgebiete virtueller Maschinen vor. Dabei werden relevante Vertreter untersucht, um damit die Frage zu beantworten, ob eine eigenständige Entwicklung tatsächlich notwendig war oder ob eine bestehende VM zu diesem Zweck tauglich wäre. Auf diese Weise gelingt die Abgrenzung des UVC aufgrund seiner Eigenschaften gegenüber anderen VM.

4.2.1. Langzeitarchivierung

Gibt es speziell für die Langzeitarchivierung entwickelte VM? Nach Hoeven und Wijngaarden [2005] wurden Emulatoren weder speziell für die Langzeitarchivierung entwickelt noch werden sie produktiv in Archiven genutzt.

Im Bereich der Langzeitarchivierung kommen mittlerweile vereinzelt Emulatoren zum Einsatz. Für die meisten Emulatoren gilt jedoch, dass sie unabhängig von angedachten Archivierungsmethoden mit spezieller Orientierung auf Computerspiele entwickelt wurden. Im Fall des Emulators MAME, der zahlreiche Arcade Spiele emulieren kann, ist die vorhandene Gemeinschaft der interessierten Nutzer so groß, dass er ohne Bereitstellung von Geldmitteln als Open Source Projekt entstehen konnte [Suter, 2010]. So beschäftigen sich Studien ausschließlich damit, die Tauglichkeit solcher verfügbaren Emulatoren zu evaluieren. Z.B. untersuchen Guttenbrunner et al. [2008] die Emulatoren MESS, ZSNES und SNES9X.

Der erste direkt für die Langzeitarchivierung entwickelte Emulator ist Dioscouri [van der Hoeven et al., 2007]. Dieser emuliert Hardware auf der Basis der x86-Architektur mit einigen üblichen Ein- und Ausgabegeräten für z.B. Grafik- und Sound. Nach mehreren Jahren Entwicklungszeit ist dieser in Java implementierte Emulator in der Lage, alle Arten von Programmen der MS-DOS Ära auszuführen [Suchodoletz et al., 2010].

4.2.2. Lehre

In diesem Abschnitt werden drei VM aus dem Bereich der Lehre betrachtet. Bei allen verfügbaren Implementierungen dieser VM handelt es sich um Simulatoren. Der wesentliche Unterschied ist der damit erzielte Zweck. Ein Emulator versucht die originale Maschine so effizient wie möglich nachzubilden, um eine möglichst authentische Abspielumgebung zu schaffen. Simulatoren betrachten vorrangig andere Aspekte, wie die Möglichkeit, jederzeit den Inhalt der Register, des Speichers, der Keller usw. einsehen und eventuell manipulieren zu können.

Dass für diese VM nur Simulatoren existieren, bedeutet aber nicht, dass keine effizienten Emulatoren dafür implementierbar wären, die man so wie den UVC einsetzen könnte.

MIX

Die im Bereich Lehre wohl bekannteste virtuelle Maschine ist MIX. Sie wurde von Knuth [1969b] eingeführt, um begleitend zu den dargestellten Algorithmen in seiner Buchreihe Beispielimplementierungen zu bieten. Dabei wurden die Beispiele maschinennah formuliert und zugleich über eine abstrakte Ebene vermittelt. So waren sie leicht für die Hardware dieser Zeit umsetzbar. Schon die Herleitung des Namens „MIX“ lässt erahnen, woher viele der genutzten Konzepte stammen.[2] Knuth führt seine „mythische“ Maschine als eine Maschine ein, die „ist wie jede andere, nur vielleicht ein wenig schöner.“ Sie sollte kompakte Programme ermöglichen und zugleich leicht zu verstehen sein. Im Wesentlichen erinnert das an den UVC. Im Gegensatz zum UVC liegt aber der Schwerpunkt bei „schön“ zu formulierenden Programmen, was sich stringent im Befehlssatz wiederfindet. So kennt MIX 64 Opcodes. Vereinzelt wird ein weiteres Byte herangezogen, um den eigentlichen Befehl zu codieren. So gibt es insgesamt 149 Befehle, davon 60 Sprungbefehle (alle springen ausschließlich zu im Speicher abgelegten Adressen) und neun Vergleichsbefehle. Entsprechend weniger „schön“ lassen sich Programme für den UVC mit nur jeweils zwei Sprung- und Vergleichsbefehlen formulieren. MIX bietet neun Register, davon ein Akkumulator und ein Zusatzregister. Diese zwei Register bestehen aus 5 Bytes und einem separaten Vorzeichen. Sechs weitere Indexregister bestehen aus nur zwei Bytes und haben auch ein separates Vorzeichen. Das letzte Register dient zur Speicherung einer Adresse, z.B. für die Rücksprungadresse zur Realisierung von Unterprogrammen. Die zwei großen Register sind am Speicher ausgerichtet. Dieser umfasst 4000 Speicherworte. Jedes Speicherwort umfasst ebenfalls 5 Bytes und ein separates Vorzeichen. Jede Instruktion belegt im Speicher genau ein Wort.

Während sich solche Strukturen tatsächlich in der Hardware der damaligen Zeit wiederfinden, bietet MIX eine Besonderheit, die an die Universalität des UVC erinnert. Gemeint ist das Byte. Es besteht aus z.B. 6 Bits, was zu den Großrechnern von CDC passt. Ein Byte kann aber auch zwei Dezimalstellen umfassen. Für ein Byte gilt, dass es mindestens 64 aber maximal 100 verschiedene Werte annehmen kann. Für ein Wort, das aus mehreren Bytes besteht, ergeben sich dadurch unterschiedliche Belegungen. Die meisten von Knuth aufgeführten Algorithmen berücksichtigen das und verwenden maximal 64 verschiedene Werte. Von der Bytegröße abhängige Programme zu schreiben, bezeichnet er als illegalen Akt, der nicht toleriert wird.

[2] Die römische Zahl MIX enstand durch Mittelung der Nummern seinerzeit aktueller Computersysteme.

Erreicht wird dadurch die einfache Simulation von MIX für die Hardware damaliger Zeit. Interessant ist die folgende Formulierung diesbezüglich:

> *Those programmers accustomed to a binary machine can think of MIX as binary; those accustomed to decimal may regard MIX as decimal. Programmers from another planet might choose to think of MIX as a ternary machine.* [Knuth, 1969b]

Das unterstreicht den Gedanken der Universalität. Jedoch ist es kein „Alien," sondern ein gewöhnlicher Programmierer der heutigen Zeit, der mit seiner gewohnten Hexadezimaldarstellung diese „Universalität" sprengt.

Aus heutiger Sicht ist MIX weder universell noch besonders einfach zu simulieren. MIX Programme könnten nur einen Bruchteil der möglichen Performanz der Basismaschine nutzen.

Anders als der UVC wird MIX von einer spezifizierten Assemblersprache begleitet. Obwohl die Programme dadurch lesbar bleiben, sind viele der implizit enthaltenen Hinweise auf eine effiziente Implementierung bereits heute bedeutungslos.

Die Spezifikation ist mit ebenfalls 18 Seiten ähnlich umfangreich wie die des UVC. Die Spezifikation der Assemblersprache umfasst weitere 12 Seiten und enthält erklärende Beispiele. Beide sind in Englisch verfasst.

Die Implementierung eines Emulators könnte ähnlich viel Zeit beanspruchen. Der Prozessor ist nur unwesentlich komplexer und der Speicher und die Registerverwaltung sind dagegen einfach zu realisieren. Allerdings könnte die Implementierung der anzusteuernden Ein- und Ausgabegeräte die eingesparte Implementierungszeit wieder aufbrauchen. Es finden sich keine Angaben zum Implementierungsaufwand eines MIX-Emulators. Zu den Simulatoren, die eine schrittweise Ausführung und eine detaillierte „Innenansicht" ermöglichen, gibt es auch keine konkreten Angaben diesbezüglich.

Mit seinem begrenzten Speicher, seiner Orientierung der Ein- und Ausgabegeräte auf Lochkarten und Bandlaufwerke und seiner nicht mehr zeitgemäßen Programmierbarkeit ist MIX deutlich ungeeignet für die Langzeitarchivierung.

MMIX

MIX ist nach obiger Darstellung veraltet und verlangt vom Leser ein unnötig hohes Maß an Abstraktion. Weil selbst eine Weiterentwicklung von MIX, der MIXMaster, ebenfalls „hoffnungslos veraltet" ist, entwickelte Knuth [1999] einen neuen virtuellen Computer für das dritte Jahrtausend. Erneut ist die Idee, einen anschaulichen und effizienten Computer in Anlehnung an aktuelle Hardware zu entwickeln. Universalität spielt hierbei erneut nur in der Form eine Rolle, dass sich MMIX für aktuelle Hardware gut simulieren lassen soll.

MMIX bietet 256 64 Bit große Register, 30 Spezialregister und einen 64-Bit-Adressraum. Für arithmetische Berechnungen können Register neben Zahlen im Zweierkomplement auch 64-Bit-Fließkommazahlen nach IEEE 754 enthalten. Opcodes nutzen ein Byte, das wiederum belegt 8 Bit. Wie auch schon bei MIX werden ausnahmslos alle Opcodes genutzt. Wesentliche Erweiterungen finden sich in der Arithmetik und im Bereich der Nebenläufigkeit. So finden sich viele bitweise Operationen wie `AND` und `OR` (aber kein `NOT`), Schiebebefehle und zusätzliche Anweisungen zur direkten Unterstützung von Fließkommazahlen. Ein- und Ausgabegeräte werden über spezielle Operationen unterstützt.

So wie heutige Hardware auch, kann MMIX ein vollständiges Betriebssystem ausführen. Bisher wurde aber noch keins entwickelt. MMIX ist entsprechend umfangreich spezifiziert. Allein die Spezifikation umfasst 60 mit insgesamt ca. 23.000 Wörtern beschriebene Seiten.

Aufgrund der zugenommenen Komplexität gegenüber MIX dürfte eine MMIX-Implementierung deutlich mehr Zeit in Anspruch nehmen.

Zusammen mit der Spezifikation veröffentlichte Knuth [1999] eine Anleitung zur Implementierung. Diese ist jedoch stark an einer 32-Bit-Architektur angelehnt und muss bereits heute von Programmierern 64-Bit-Systeme „interpretiert“ werden. Ob MMIX tatsächlich für das gesamte Jahrtausend aktuell bleibt, darf bereits heute bezweifelt werden.

Für die Langzeitarchivierung ist MMIX weniger geeignet als der UVC. Zwar lassen sich Programme effizient schreiben, aber nur auf dem derzeit effizient implementierbaren MMIX auch effizient ausführen. Eigenschaften, aus denen sich eine schnelle und effiziente Implementierung auch in ferner Zukunft ableiten lassen würde, finden sich keine.

MI

Die Maschine für die Informatikausbildung (MI) wurde bereits 1987 entwickelt. Sie ist wesentlich moderner als MIX und komplexer als MMIX. Das Vorgehen ist insgesamt anders als bei Knuth. Während Knuth die Spezifikationen bewusst in seinen Büchern abdruckt, um den einen oder anderen Interessierten die eigene Implementierung seiner Maschinen zu ermöglichen, geben Borghoff et al. [1987] ihren Studenten eine Anleitung zur Nutzung ihrer virtuellen Maschine an die Hand. Entsprechend knapp fällt die Spezifikation aus. Von den 96 Seiten gehören nur 50 zur Spezifikation. Die restlichen Seiten geben eine ebenfalls sehr knappe Anleitung zur Nutzung des verfügbaren Simulators. Dabei ist die MI mit ihren vier Rechnerkernen und den zwei verschiedenen Arbeitsmodie komplexer als MMIX. Zudem bietet sie 16 32 Bit große Arbeitsregister pro Rechnerkern und zwei parallel arbeitende EA-Prozessoren.

Die MI wurde im Rahmen zweier Diplomarbeiten implementiert. Die gesamte Bearbeitungszeit betrug zweimal 6 Monate. Allerdings ist anzunehmen, dass die Implementierung einen einzelnen Programmierer keine vollen 12 Monate beanspruchen würde. Wie viele Arbeitsstunden exakt aufgewendet wurden, ist nicht bekannt.

Die vom Autor der vorliegenden Arbeit betreute Bachelorarbeit von Oehme [2012] gibt jedoch einen konkreten Anhaltspunkt für die reale Implementierungszeit auf heutigen Systemen. Oehme entwickelte im Rahmen seiner Arbeit ein Programm mit grafischer Oberfläche, das die Eingabe und das Editieren von Quellcode zulässt und diesen direkt vor der Ausführung übersetzt. Neben der eigentlichen Maschine implementierte Oehme daher einen umfangreichen Assembler und eine detailreich gestaltete Oberfläche mit Syntaxhighlighting. Insgesamt benötigte er, samt ausführlicher Dokumentation, nur drei Monate. Von allen geschriebenen Zeilen Quelltext, lassen sich „nur“ ca. 10.000 der implementierten Maschine zuordnen. Das sind 59%. Der Rest kann der Grafik (18%) und dem Assembler (23%) zugeordnet werden. Die implementierte Maschine umfasst jedoch nicht den vollständigen in der Spezifikation beschriebenen Funktionsumfang. Die Restriktion erlaubt die elegante Formulierung komplexer Algorithmen und die Demonstration diverser Speicheradressierungsarten. Für die Nutzung innerhalb der Langzeitarchivierung fehlen jedoch noch geeignete Ein- und Ausgabekanäle. Die MI ist somit weniger gut für die Langzeitarchivierung geeignet als der UVC. Dabei benötigte

selbst die restriktive Implementierung von Oehme allein für den Anteil der Maschine über 85 Stunden, wobei viele davon der Vielfalt der zu implementierenden Befehle geschuldet sind. Die bereits nur eingeschränkt umgesetzte MI bietet einen mehr als viermal so umfangreichen Befehlssatz wie der UVC.

Die MI wurde wie die Maschinen von Knuth vorrangig für die Lehre entwickelt. Wobei die MI den Schwerpunkt auf maschinennahe Konzepte legt, zu denen auch die Programmierung einer Mehr-Prozessor-Maschine gehört. Der Schwerpunkt bei Knuth liegt dagegen in der Vermittlung eines „schönen“ Programmierstils und der Darstellung effizienter Algorithmen.

Alle diese VM orientieren sich stark an den Hardwarearchitekturen und den Ein- und Ausgabegeräten ihrer Zeit. Ihre Obsoleszenz in ferner Zukunft ist absehbar. Für die Langzeitarchivierung eignen sie sich daher nicht, auch wenn der erkennbare Implementierungsaufwand deutlich unter dem liegt, was für Emulatoren realer Maschinen notwendig wäre.

Einen weiteren Nachteil teilen sie mit dem UVC: Anwendungen müssen extra für sie implementiert werden, und zwar in ihrer speziellen Assemblersprache. Eine entsprechende Hochsprache oder gar Compiler bieten sie nicht. Ein nur geringer Vorteil ist die hohe Anzahl geübter Studenten, die im Umgang mit diesen VM geschult sind.

Portabilität und Ausführung von Hochsprachen

Craig [2006] identifiziert zwei wesentliche Anwendungsgebiete: Sicherstellen der Portabilität von Anwendungssoftware und die Ausführung von Programmiersprachen, die nicht gleichermaßen für alle Zielarchitekturen geeignet sind.

In diese Kategorie fallen sehr viele VM, die zusammen mit einer Programmiersprache verwendet werden. Nach Craig ist die zu unterstützende Hochsprache ein wesentliches Designkriterium einer VM. Im Idealfall existiert diese zuerst. In einem zweiten Schritt wird die Architektur der VM maßgeschneidert. So richten sich z.B. der Befehlssatz und die Größe der Register nach den zu unterstützenden Datentypen.

Nach Smith und Nair [2005] wurden solche Hochsprachen-VM mit dem zur Pascal Programmierumgebung gehörenden USCD p-System populär [Bowles, 1980]. Ein heute nicht mehr wegzudenkender Vertreter ist die Java Virtual Machine (JVM).

Leider steigt die Komplexität solcher Hochsprachen-VM sehr schnell, wenn die zugehörige Hochsprache vielseitig nutzbar sein soll. Die JVM ist wie viele andere Hochsprachen-VM kellerbasiert [Craig, 2006]. MIX nutzt einen Akkumaltor und die MI eine feste Anzahl Arbeitsregister, um Berechnungen durchzuführen. Die kellerbasierte JVM nutzt einen *Operandenstack*. In diesem Keller werden zunächst die Parameter abgelegt. Eine Operation wie `ADD` holt die obersten zwei Parameter aus dem Keller und legt nach der Addition das Ergebnis wieder im Keller ab. Die Anzahl der Register spielt bei kellerbasierten VM eine untergeordnete Rolle. Eine Eigenheit der JVM macht die Ausrichtung auf die Sprache Java sehr deutlich: Die „Hardware“ der JVM unterstützt direkt das *Class File Format*.

Die vielseitige Verwendung der JVM und die Breite der bereits verfügbaren Anwendungen sprechen für eine Verwendung im Bereich der Langzeitarchivierung. Dagegen sprechen jedoch gleich zwei Gründe. Zum einen ist die JVM trotz ihrer erreichten Portierbarkeit stark abhängig vom Basissystem. Die gute Portierbarkeit fußt auf der Ähnlichkeit aktueller Hardware, die allesamt 32-Bit-Architekturen sehr gut emulieren können oder selbst sind. Einige

Besonderheiten, wie die Unterstützung von nativem, also speziell für eine Basismaschine erzeugtem Code, sind weitere Nachteile in Bezug auf die Langzeitarchivierung.

Zum anderen ist die Komplexität der JVM ein Nachteil. Die Spezifikation der JVM ist im einfachen Englisch [Lindholm und Yellin, 1999] verfasst und lässt sich daher quantitativ vergleichen. Sie umfasst insgesamt über 67.000 Wörter. Für eine volle Implementierung für eine neue Architektur muss aber nicht nur die JVM implementiert werden, sondern zusätzlich alle Bibliotheken, die aus Gründen der Performanz nativen Code enthalten.

Eher denkbar wäre eine eingeschränkte JVM passend zu einem eingeschränkten Sprachumfang. Dadurch ließe sich der Implementierungsaufwand erheblich verringern. Das hätte den Vorteil, dass bisherige Java-Programmierumgebungen genutzt werden könnten. Geeignete Programme ließen sich auf den reduzierten Sprachumfang anpassen. Dieser Gedanke findet sich bei MiniJava [Appel und Palsberg, 2002] oder C−− [Holdsworth, 2001].

Allerdings bleibt fraglich, wie beständig die JVM ist. Mit dem Einzug immer mehr 64-Bit-Maschinen wird eine Weiterentwicklung der 32-Bit-JVM immer wahrscheinlicher. Für die Langzeitarchivierung ist die JVM aufgrund der permanenten Abhängigkeit von den aktuellen Anwenderwünschen nicht geeignet.

4.2.3. Sicherheit

Geht es darum, Systeme gegenüber fremder Software zu schützen, kommen ebenfalls VM zum Einsatz. Die JVM ist auch in diesem Bereich ein prominenter Vertreter.

Java und die JVM wurden nach Dalheimer [1997] entwickelt, um in verteilten Umgebungen eingesetzt zu werden. Schon beim Entwurf wurde daher großer Wert auf Sicherheit gelegt. So greift ein mehrstufiges Sicherheitssystem auf allen Ebenen der Programmierung, der Programmübertragung und der Programmabarbeitung.

Besonders sichtbar wird das mit der Verwendung von Applets, das sind kleine Java-Programme, innerhalb von Webseiten. Diese werden über das Internet übertragen und in der lokalen JVM ausgeführt. Dabei wird aufgrund der potentiellen Gefahr jedes Applet zunächst in einer *Sandbox* ausgeführt. In einem solchen „Buddelkasten“ kann sich der übertragene Code sinnbildlich austoben, Zugriff auf das reale System bekommt er nicht.

Neben der sicheren Ausführung von Code können VM auch dazu dienen, bereits identifizierte Malware, also Schadsoftware zu untersuchen. Wurde Malware entdeckt, sind zwei Dinge besonders wichtig: wie befreit man das System davon und was macht die Malware eigentlich. Letzteres stellt man am einfachsten durch Beobachten fest. Dazu werden zunehmend VM genutzt. Virtuelle Systeme lassen sich nach einem bösartigen Zugriff sehr leicht wieder herstellen. Einige VM wie z.B. QEMU, die eine x86-Architektur abbilden, bieten sogar für den Zweck der Codeinspektion besondere Schnittstellen [Dinaburg et al., 2008].

Aktuelle Malware nutzt allerdings zunehmend Routinen zum Erkennen einer virtuellen Ausführung und versucht sich dann durch abweichendes Verhalten der Analyse zu entziehen. Möglich wird das, weil virtuelle Maschinen das Original scheinbar nie vollständig fehlerfrei nachbilden können [Dinaburg et al., 2008]. Auf Basis dieser Erkenntnis entwickelten Paleari et al. [2009] ein automatisiertes Verfahren, das im Falle von QEMU in kürzester Zeit 53926 Testroutinen generierte, mit deren Hilfe die Ausführung innerhalb von QEMU entdeckt

werden kann. Diese enorme Anzahl an Testroutinen lässt sich auf abweichende Implementierungen von 405 Opcodes zurückführen. Allerdings merken Martignoni et al. [2009] an, dass oftmals eine laxe Spezifikation der Hardware Ursache war.

Daraus lässt sich schließen, dass VM in Form von Nachbildungen realer Hardware für die Langzeitarchivierung denkbar ungeeignet sind. Selbst wenn umfangreiche Spezifikationen wie im Fall der x86-Architekturen vorhanden sind, lassen sie sich nicht fehlerfrei emulieren.

Ebenso wie QEMU ist Dioscouri quelloffen und daher für die Langzeitarchivierung interessant. QEMU ist in C++, Dioscouri in Java geschrieben. Neben diesen emuliert auch JPC eine x86-Architektur und ist ebenfalls in Java geschrieben. Allerdings liegt der Schwerpunkt nicht auf Modularisierung, sondern auf Effizienz. Mit verschiedenen Tricks, die alle auf der Kenntnis interner Abläufe verfügbarer JVM beruhen, wird die Ausführung von Code in *„native hardware speed“* erreicht. Dieses Projekt gibt darüber hinaus Aufschluss über den notwendigen Aufwand. Für Newman und Dennis [2009] ist der modernen PC ein *„complicated beast“*: Allein die Prozessorbeschreibung umfasst ca. 1.500 Seiten und beschreibt u.a. ca. 65.000 mögliche Instruktionen. Mit der Implementierung waren durchschnittlich 2,5 Programmierer über 30 Monate lang beschäftigt. Zum Vergleich: Das Projekt Dioscouri startete bereits 2005 und 2010 ist Dioscouri ebenfalls wie JPC nur in der Lage, DOS-Applikationen auszuführen.

4.2.4. Performanz

Eine virtuelle Maschine, die für den performanten Einsatz in Browser-Umgebungen entwickelt wurde, ist der *Native Client* [Yee et al., 2010]. Um hohe Ausführungsgeschwindigkeiten zu erreichen, wird x86-kompatibler Code genutzt und von der Basismaschine direkt ausgeführt. Davor wird der Code nach bestimmten *Constraints* untersucht und z.B. ein Befehl zum direkten Systemzugriff nicht zugelassen. Durch die Verwendung von x86-Instruktionen ist dieser Ansatz zwar für heutige Systeme einfach umzusetzen, aber sehr systemabhängig und mit Blick auf den Zuwachs der ARM-basierten Internetgeräte auch nicht zukunftsträchtig.

Auf unterster Ebene können System VM genutzt werden, um z.B. mehrere OS auf einer Maschine auszuführen. Darauf lassen sich verschiedene Anwendungen ausführen. Je nach Bedarf können die verfügbaren Ressourcen zugeteilt werden [Smith und Nair, 2005]. Hier geht es um effiziente Ressourcenauslastung.

Für die Langzeitarchivierung haben VM dieser Anwendungsgebiete keine Bedeutung.

4.2.5. Tiny

Tiny VM finden vor allem in Sensornetzwerken Verwendung [Levis und Culler, 2002]. Aufgrund der beschränkten Ressourcen müssen diese VM besonders „schmal“ spezifiziert sein [Shaylor et al., 2003]. Bei der Verwendung in Smart Cards liegt der Fokus auf den *two main basic features: security and communication* [Chen, 2000]. Als Software wären derartige *tiny* VM sehr einfach zu realisieren. Solche VM sind aber auf die engen Grenzen ihrer Umgebung hin optimiert. Die Registerbreite oder der verfügbare Adressraum lassen komplexere Anwendungen wie PDF-Betrachter nicht zu.

Für die Langzeitarchivierung sind *tiny* VM ungeeignet.

4.2.6. Universal

Auf der Suche nach anderen universellen VM stößt man schnell auf die UVM. Nach [Folliot et al., 2002] plant IBM die Entwicklung einer in ihrer Komplexität der JVM vergleichbaren Maschine mit dem Namen *Universal Virtual Machine*. Sie soll in Java, Smalltalk und Visual Basic geschriebene Programme ausführen können. Abgesehen von der zu erwartenden Komplexität sind bisher keine Spezifikation und auch kein Prototyp zu finden.

Deutlich universeller ist der eigene Ansatz von Folliot et al. Diese nennen ihre Maschine *Virtual Virtual Machine* (VVM). Im übertragenen Sinne virtualisiert sie auf zwei Ebenen. Auf der unteren Ebene wird eine virtuelle Basismaschine abgebildet. Auf der darauf aufsetzenden Ebene wird über VMLets die für Anwendungen sichtbare VM spezifiziert. Wenn jede Applikation nicht nur ihren Code, sondern auch die zugehörigen VMLets enthält, kann die VVM sehr flexibel und ebenfalls schmal ausfallen. Mittels dieser VMLets wird nicht nur die Abbildung der jeweiligen Operationen auf die der unteren Ebene definiert, auch ist es möglich, völlig neue Befehle für die virtuelle Basismaschine zu liefern [Folliot et al., 1998]. Diese müssen dann vor der Programmausführung übersetzt werden [Folliot et al., 2002].

Leider lässt der letzte Aspekt erkennen, dass die VVM zwar dynamisch, schlank und auf ihre Weise universell ist, aber für jede Zielplattform neu implementiert werden muss. Folliot et al. [2002] nennen viele relevante Einsatzbereiche wie aktive Netzwerke, Mobiltelefone und eingebettete Systeme, erwähnen aber zu Recht die Langzeitarchivierung nicht.

Interessant ist die Idee für die Langzeitarchivierung dennoch. In abgewandelter Form findet sie sich bei Hoeven und Wijngaarden [2005] wieder. Entwickelt wird Dioscouri, um flexibel aus Modulen zusammengesetzte Hardware emulieren zu können. Solche Module bilden auch Prozessoren ab, virtualisieren also auf gleicher Ebene wie VMLets. Die untere Ebene bildet bei Dioscouri allerdings die JVM, die keine neuen Befehle „lernen“ kann. Hoeven und Wijngaarden [2005] kündigten an, Dioscouri für eine universelle virtuelle Maschine (UVM) migrieren zu wollen, sobald es eine solche gibt. Bis heute nutzen sie die JVM.

Damit eine solche Migration gelingt, müsste eine derartige UVM ähnlich komplex aufgebaut sein wie die JVM. Das eigentliche Problem der JVM aus Sicht der Langzeitarchivierung wird damit aber nicht gelöst: ihre Komplexität.

4.3. Zusammenfassung

VM werden in den verschiedensten Anwendungsgebieten eingesetzt, auch bereits im Bereich der Langzeitarchivierung. Die den UVC nutzende Archivierungsmethode benötigt eine VM mit sehr speziellen Eigenschaften: Der perfekte UVC soll sich durch eine knappe, aber vollständige Spezifikation auszeichnen, eine Implementierung für verschiedenste (auch zukünftige) Hardwarearchitekturen in kürzester Zeit ermöglichen, darüber hinaus einfach programmierbar sein und auch komplexe und umfangreiche Anwendungen ausführen können.

In keinem anderen Bereich findet sich eine VM mit den erforderlichen Eigenschaften in dieser Kombination. Eine Entwicklung von Grund auf erscheint infolge der obigen Betrachtung gerechtfertigt. Die folgenden Kapitel widmen sich der mit der aktuellen Spezifikation des UVC erreichten Zielsetzung.

Teil II.

Implementierungen des UVC

5. Referenzimplementierung in Ada

Dieser zweite Teil der vorliegenden Arbeit untersucht die verschiedenen Aspekte, die bei der Implementierung des UVC eine Rolle spielen. Dazu zählen die Universalität, die Portierbarkeit und die zu erwartende Entwicklungszeit. Während der erste Teil die Grundlagen zum Verständnis und der Einordnung des UVC mit der zugehörigen Archivierungsmethode legte, ist dieser Teil von Experimenten geprägt. Das erste Experiment ist dabei als „Grundsteinlegung“ für die im nächsten Kapitel beschriebene *Zeitreise* zu sehen.

In diesem Kapitel wird dieses erste Experiment, in Form der Erstellung einer Referenzimplementierung, beschrieben. Der erste Abschnitt motiviert diese Implementierung zunächst ausführlich. Der zweite Abschnitt begründet die Wahl der dazu genutzten Programmiersprache. Der dritte, für die weitere Arbeit wesentliche Abschnitt, beschreibt die Implementierung der wichtigsten Komponenten. Aufgrund der hier gemachten Erfahrungen wurden verschiedene Entscheidungen für die Implementierung für die historischen Großrechner getroffen. Abschließend werden der zeitliche Aufwand, die Universalität sowie die entdeckten Unzulänglichkeiten der vorhandenen Spezifikation des UVC prägnant zusammengefasst.

5.1. Wozu eine Referenzimplementierung?

Der UVC ist ein sehr einfach gehaltener virtueller Computer [Lorie, 2002a]. Dennoch zählt seine Implementierung zu den komplexeren Aufgaben, die ohne gründliche Vorbereitung missglücken können.[1] Zu Beginn der Erstellung der im Folgenden beschriebenen Referenzimplementierung lag nur die Java-Implementierung des UVC von IBM vor. Der Quellcode wurde nicht veröffentlicht, aber diese Implementierung umfasst bereits 15 Klassen und geschätzte 6000 Zeilen Code.[2]

Um Prognosen für die Portierbarkeit auf künftige Systeme abgeben zu können, wird die umgekehrte Richtung eingeschlagen und der UVC auf einer Reihe obsoleter Großrechner implementiert, die in der datArena lauffähig erhalten wurden.

Die Referenzimplementierung des UVC wurde in einer vertrauten Umgebung, also für einen derzeit aktuellen Computer mit einem vertrauten Betriebssystem und einer bekannten Programmiersprache erstellt. Für diesen Ausgangspunkt der *Zeitreise* sprechen gleich mehrere Gründe. Bei der Implementierung für einen obsoleten Großrechner wird es ohne Zweifel zu einigen Problemen kommen, die bedingt durch verlorengegangenes Wissen oder durch eigene Unerfahrenheit entstehen werden. Wird hierbei gleichzeitig versucht, einen UVC korrekt zu implementieren, können entstehende Schwierigkeiten nicht klar zugeordnet werden. Ein

[1] Eine gescheiterte Diplomarbeit demonstrierte dies anschaulich.

[2] Anhand der Größe der kompilierten Klassen geschätzt. Die erst seit August 2010 verfügbare Implementierung in C++ ist ähnlich umfangreich.

Beispiel soll das verdeutlichen. Die eigene Implementierung liefert bei einer arithmetischen Operation ein anderes Ergebnis als die vorhandene Java-Implementierung von IBM. Wo liegt jetzt der Fehler? Wurde die Spezifikation falsch gedeutet, oder wurde der Großrechner schlicht falsch programmiert? Gerade die erste Frage soll durch diese Arbeit klar beantwortet werden. Daher ist der zweite Aspekt, so weit das möglich ist, auszugrenzen. Wird der UVC auf einem vertrauten System implementiert, kann zusätzlich eine Aussage zum tatsächlichen Implementierungsaufwand gemacht werden. Zudem ist sichergestellt, dass sich der UVC allein mit Hilfe der Spezifikation korrekt implementieren lässt.

Bereits bei der Implementierung entstehen Testszenarien und Testprogramme, die für die Überprüfung der Komponenten genutzt werden. Diese wurden teils sehr leicht in Programme für den UVC überführt und dienten dem Vergleich mit der vorhandenen Java-Implementierung. Für die in der *Zeitreise* genutzten Großrechner wurden sie problemlos wiederverwendet. Hier bietet es sich an, die Programme so zu gestalten, dass sie bei entsprechender Archivierung als „Benchmarks“ für Implementierungen und Portierungen in ferner Zukunft nutzbar erhalten bleiben (siehe Kapitel 10).

5.2. Warum in Ada83?

Jede Epoche hatte bisher ihre eigene meistgenutzte Programmiersprache. So war es 1960 Algol, 1970 Pascal, 1980 C und 1990 C++. Heute dominiert Java und wäre aufgrund ihrer Verbreitung auch meine erste Wahl für die Implementierung des UVC.

Es sprechen jedoch zwei Gründe gegen diese Wahl. Zum einen ist der vorhandene UVC, der 2004 von IBM implementiert wurde, ebenfalls in Java implementiert. Eine erneute Implementierung könnte nur zeigen, dass auch ein unabhängiger Programmierer, also einer an der Entwicklung der Spezifikation für den UVC Unbeteiligter, einen korrekten UVC nach dieser Spezifikation implementieren kann. Die vorliegende Arbeit zielt aber auf die Frage, ob sich dieser universelle Computer mit den verschiedensten Sprachen, auf den unterschiedlichsten Maschinen mit vergleichbarem Aufwand realisieren lässt. Zum anderen spricht dagegen, dass sich vorhandene, kompilierte Java-Klassen nutzen ließen und sogar Source-Code daraus zurückgewonnen werden könnte. Die getroffene Wahl half dieser Versuchung zu widerstehen und unvoreingenommene Designentscheidungen zu treffen.

Die Unabhängigkeit von Spezifikation und Sprache zu zeigen ist ein Ziel der Implementierung. Das andere Ziel ist die Vorbereitung auf die Implementierungen innerhalb der *Zeitreise*. Das heute wohl jedem Anwendungsentwickler in Fleisch und Blut übergegangene Konzept der objektorientierten Programmierung lässt sich nicht auf alle historischen Großrechner übertragen, da geeignete Programmiersprachen fehlen.

Die Wahl fiel schließlich auf Ada83, weil bei dieser mir vertrauten Programmiersprache die Hoffnung groß ist, dass sich hier entwickelte Algorithmen und Strukturen leicht in frühere Programmiersprachen zurück übertragen lassen. Ob diese Strategie aufging, wird das Kapitel 6 zeigen.

5.3. Die zu implementierenden Komponenten

Die Spezifikation macht keine Vorgaben, ob und wie der UVC strukturiert implementiert werden kann. Die Implementierung des verfügbaren UVC in Java wurde leider von van der Hoeven et al. [2005] nur sehr oberflächlich beschrieben. Sie schreiben von drei wesentlichen Aufgaben, aus denen sich dann die grobe objektorientierte Struktur ableiten lässt: *Memory Management*, *Execution Management* und *Input/Output Management*.

Schon bei den ersten Vorüberlegungen zur Implementierung wird deutlich, dass diese Dreiteilung zwar passt, aber zu grob ist. Einen wesentlichen Punkt stellen die unbeschränkten Register und deren Arithmetik dar, den van der Hoeven et al. nicht erläutern. Diesen interessanten Punkt betrachten wir daher vor der Implementierung der erwähnten drei Aufgabenbereiche.

5.3.1. Datenstruktur der Register

Die Register des UVC sollen unabhängig von jeglicher Hardware universell gestaltet sein. Aus diesem Grund kennt der UVC, wie sonst bei realer Hardware üblich, keine Worte von 8, 16, 32 oder 64 Bit, sondern einfach nur Register mit Werten einer beliebigen Anzahl von Bits. Diese Anzahl wird jeweils individuell für jedes Register zur Laufzeit, abhängig von der ausgeführten Operation, angepasst. Die Länge der Register wird vom UVC verwaltet. Zusätzlich hat ein Register ein Vorzeichen, das in einem separaten Bit gespeichert ist. Dies sind Vorgaben der Spezifikation für den UVC [Lorie und van Diessen, 2005].

An zwei anderen Stellen finden sich weitere Hinweise. So dient ein 31 Bit großer Wert der direkten Identifizierung eines Registers innerhalb des Segments und beim Laden einer Konstante wird die Länge mit einem 32 Bit großen Wert angegeben. Beide Angaben spezifizieren jedoch die Register nicht, sondern nur den Umgang mit ihnen (siehe 3.2).

Für eine Implementierung müssen jedoch Grenzen festgelegt werden, damit sich Datenstrukturen definieren lassen. Die Länge eines Registers mit einem 32 Bit großen Wert anzugeben, erscheint mehr als ausreichend. Damit kann ein einzelnes Register theoretisch 4 GigaBit, also 512 MB aufnehmen. Der Wert eines Registers wird in einer Liste gespeichert, die unbegrenzt viele 32-Bit-Worte aufnehmen kann. Aufgrund möglicher führender auf Null gesetzter Bits, lässt sich die Länge eines Registers nicht aus der Liste bestimmen und muss zusätzlich abgelegt werden. So ergibt sich die in Abbildung 5.1(a) dargestellte Datenstruktur.

Bei dieser Implementierung geht es nicht allein darum zu zeigen, in wie kurzer Zeit ein UVC implementiert werden kann, sondern es wird auch auf eine effiziente Implementierung geachtet. Gerade für die anschließende *Zeitreise* wird die effiziente Nutzung des Speichers und der Rechenleistung eine wesentliche Rolle spielen.

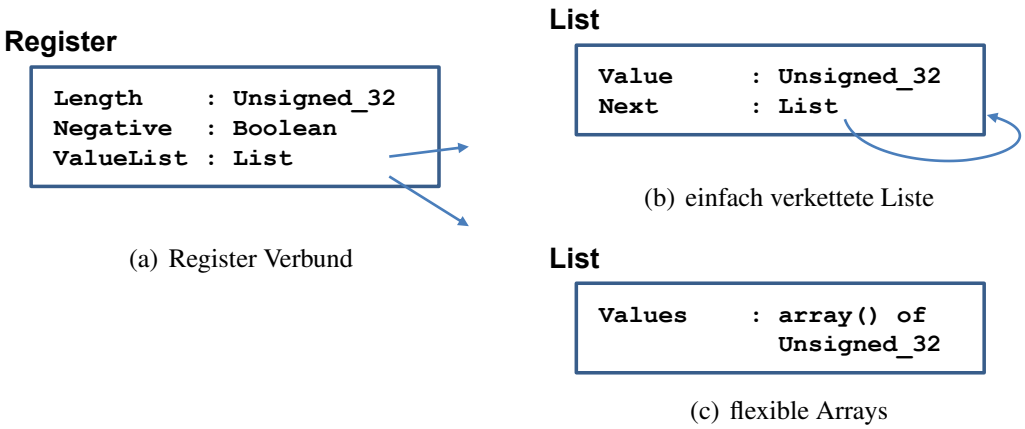
Ein Register soll effizient gespeichert und verwendet werden können. Großen Einfluss hat hierbei die Länge eines Wortes der Basismaschine. Bei heutigen Maschinen sind das 32 oder 64 Bit. Solange der UVC mit Zahlen arbeitet, die mit 32 Bits darstellbar sind, wird die Ausführung nur durch das Prüfen dieser Voraussetzungen verlangsamt. Ein fortwährendes Zusammensetzen und Zerlegen von Bits, wie es in einigen Paketen,[3] die zur Darstellung großer

[3] Z.B. Paul E. Dunne, Karatsuba Multiplication Algorithm, http://www.csc.liv.ac.uk/~ped/teachadmin/algor/karatsuba.ada, eingesehen am 11.08.09

Zahlen Strings benutzen, notwendig ist, entfällt. Für die Ada-Implementierung bedeutet das die Verwendung der Typen `Unsigned_32` und `Unsigned_64` aus dem Paket Interfaces.

Obwohl zu diesem Zeitpunkt feststeht, dass einer der während der *Zeitreise* genutzten Großrechner 60 Bit Worte unterstützt, soll dennoch die aktuelle Basismaschine mit ihren 32 Bit Worten berücksichtigt werden. Die implementierten Algorithmen lassen sich jedoch problemlos für andere Wortlängen adaptieren.

Operationen passen bei Bedarf die Anzahl der Bits an. Ein Überlauf eines Registers, wie er für alle realen Maschinen gängige Praxis ist, wird somit ausgeschlossen. Zum Beispiel kann bei einer Addition das Ergebnis mehr oder auch weniger Bits benötigen, als die Register der Summanden selbst hatten. Das Ergebnis wird soviele Bits beanspruchen, wie es zur fehlerfreien Darstellung benötigt. Als Folge davon müssen Register dynamisch gestaltet werden, da sie in schneller Folge Werte verschiedenster Länge aufnehmen können sollen. Programmiersprachen bieten hier verschiedene Konzepte an. Ada kennt Arrays, die aber spätestens zum Zeitpunkt ihrer Initialisierung eine feste, unveränderliche Länge haben müssen. Das Arbeiten mit flexiblen Arraygrößen ist etwas kompliziert, aber technisch möglich. Eine weitere Möglichkeit stellen dynamisch verkettete Listen dar, die in Ada mittels Verbunden und Zeigern realisiert werden können.



(a) Register Verbund

(b) einfach verkettete Liste

(c) flexible Arrays

Abbildung 5.1.: Implementierung der Register

Diese beiden Möglichkeiten (Abb. 5.1(b) und 5.1(c)) lassen sich in Ada gut realisieren und wurden auch implementiert. Beide wurden als Package `ArrayList` implementiert und haben exakt die gleiche Schnittstellenspezifikation, so dass sie beliebig gegeneinander austauschbar sind. In abschließenden Laufzeittests sollte sich die geeignetere durchsetzen. Leider nutzt keine der verfügbaren UVC-Anwendungen Register mit mehr als 32 Bit. Testprogramme, die sehr große Zahlen nutzen, werden jedoch mit der einfach verketteten Liste etwas schneller ausgeführt. Diese kommt daher in der Referenzimplementierung zum Einsatz.

Bei Operationen mit großen Zahlen werden die einzelnen Werte sequenziell vom am wenigsten signifikanten Wort beginnend verarbeitet. Daraus ergibt sich die Reihenfolge innerhalb der Liste. Nur die Division braucht die entgegengesetzte Richtung. Eine zweifach verkettete Liste nur für die Division erscheint wenig zweckmäßig, da beide Operanden für den Algorithmus normiert werden müssen und bei dieser Gelegenheit die Reihenfolge initial leicht

wechseln können. Bei der anschließenden Denormierung kann die Reihenfolge wieder korrigiert werden. Beides geht beinahe ohne Laufzeitverluste. Einer fundierten Bewertung fehlen jedoch passende UVC-Anwendungen.

5.3.2. Register und deren Arithmetik

In der Referenzimplementierung wird die Länge eines Registers mit 32 Bit codiert und somit der Registerinhalt auf 512 MB begrenzt. Die effiziente Speicherung eines solchen Registers wurde bereits dargestellt, ist aber gemessen an den folgenden Herausforderungen, ein eher kleines Problem. Es muss auch möglich sein, mit derart großen Registern zu „rechnen."

Die Implementierung der booleschen Funktionen, wie `AND`, `OR` und `NOT` richtet sich stark nach dem, was die Programmiersprache für den gewählten Datentyp zur Verfügung stellt. Die von Ada bereitgestellten booleschen Operatoren können Wort für Wort angewendet werden. Sonderfälle gibt es bei der Verknüpfung zweier unterschiedlich langer Werte, da die Länge des Ergebnisregisters angepasst werden muss, und bei der Negation von Registerinhalten einer nicht durch 32 teilbaren Länge, da hier „überstehende" Bits ausmaskiert werden müssen. An dieser Stelle wird die Spezifikation der Instruktion `AND` wie in Abschnitt 3.4.1 beschrieben korrigiert.

Darstellung der Zahlen

Die Wahl der Darstellung der Zahlen durch einen Wert und ein separates Vorzeichen ist nicht ungewöhnlich, bringt aber bekannte Probleme wie die mehrfache Darstellung der Null mit sich. Zudem werden führende Nullen nicht ausgeschlossen. Vergleiche von solchen Zahlen müssen einige Fälle berücksichtigen. So müssen z.B. folgende Vergleiche zutreffen, bei denen vereinfachend angenommen wird, dass jeweils eine Dezimalstelle in einem Wort der Liste gespeichert ist:

```
 0 =  0
 0 =  _          # Register mit leerer Liste
 0 = -0
 0 =  00000000   # Register mit 8 Werten, alle 0
27 > -1234352
54 >  0003
-27 > -28
```

Ein erster Blick auf das Vorzeichen löst im Normalfall die Hälfte alle Vergleiche, ein weiterer auf die Anzahl der verwendeten Worte sollte einen Großteil der Vergleiche abdecken. Nicht aber beim UVC, denn sonst wäre `0 > -0` und `54 < 0003`.

An dieser Stelle drängt sich die Frage auf, warum führende Nullen zugelassen wurden. Würden diese konsequent entfernt, ließe sich viel Rechenzeit sparen.

War die angedachte Emulation anderer Computer ausschlaggebend? Wie emuliert man z.B. einen 20-Bit-Computer? Ein Ergebnis einer arithmetischen Operation müsste auf einen Überlauf getestet und bereinigt werden:

```
LOADC 1002,3  20 0xFFFFF # Maske für 20-Bit-Computer
ADD   1002,1  1002,2
COPY  1002,4  1002,1
AND   1002,1  1002,3     # Ergebnis

GRT   1002,4  1002,1     # Überlauf Testen
JUMPC ...                # Überlaufbit setzen
```

Führende Nullen spielen in diesem Beispiel keine Rolle. Bereits kleine Erweiterungen in der Implementierung schließen führende Nullen vollständig aus. Es gibt nur wenige Möglichkeiten, wie führende Nullen in Register gelangen. Zum einen über das Laden einer frei bestimmbaren Anzahl Bits über **LOADC** oder aus dem Speicher mit (**LOAD** und **IN**). Zum anderen sind es die Instruktionen **AND** und **NOT**, die führende Nullen erzeugen können. An allen Stellen kann einfach im Anschluss die Länge korrigiert werden. Die vorhandenen Decoderprogramme von IBM werden durch diese Erweiterungen nicht beeinflusst. Einzig der Befehl **NOT** kann jetzt unzweckmäßige Ergebnisse liefern:

```
# ohne Erweiterungen

LOADC  1002,1 16 0x00FF0F00    # 00FF0F00
NOT    1002,1                  # FF00F0FF

# mit Erweiterungen

LOADC  1002,1 16 0x00FF0F00    # FF0F00
NOT    1002,1                  # F0FF
```

Auf den ersten Blick sind die Auswirkungen dramatisch. Eine doppelte Negation sollte das Original hervorbringen. Die Negation wird für gewöhnlich gern genutzt. (Bei den zur Verfügung stehenden Decoderprogrammen wird jedoch auf die Negation ganz verzichtet.) Die obigen Beispiele sind zudem konstruiert. Die Negation wird hier direkt nach dem Laden einer Konstante ausgeführt. Wird die Konstante in irgendeiner Form bearbeitet, z.B. durch arithmetische Operationen, so passt sich ihre Länge automatisch an:

```
# ohne Erweiterungen

LOADC  1002,1 32 0x00FF0F00    # 00FF0F00
LOADC  1002,2 16 0x0001        # 0001
ADD    1002,1 1002,2           # FF0F01
NOT    1002,1                  # 00F0FE

# mit Erweiterungen

LOADC  1002,1 32 0x00FF0F00    # FF0F00
LOADC  1002,2 16 0x0001        # 1
ADD    1002,1 1002,2           # FF0F01
NOT    1002,1                  # F0FE
```

Wenn die geladene Konstante vor der Negation nicht mehr verändert werden darf, macht die Negation aber keinen Sinn, da sie bereits durch das Laden der negierten Konstante ersetzt werden kann.

Daraus leitet sich die Notwendigkeit eines geänderten Befehlssatzes ab, falls auf führende Nullen verzichtet wird. Dem Negations-Befehl müsste zusätzlich die Anzahl der Bits des zu erzeugenden Ergebnisses als Parameter übergeben werden können, z.B. als konstanter 32-Bit- oder als Registeroperand. Alternativ könnte man auch auf die Negation verzichten und stattdessen bei Bedarf auf `SUB` ausweichen.

Arithmetik

Bei der Implementierung der Arithmetik fiel auf, dass diese alles andere als trivial ist. Noch mehr verwunderte, dass dieser Punkt nicht in den publizierten Dokumenten erwähnt wird. Die Ursache dafür liegt wohl in der Wahl der Programmiersprache. Java ist unter anderem so erfolgreich, weil sie mit einer Vielzahl von *Packages* geliefert wird, die die unterschiedlichsten Funktionalitäten vereinen. Dabei ist auch die Klasse `java.util.BigInteger`. Diese stellt genau das von der UVC-Spezifikation geforderte Verhalten zur Verfügung. In vielen anderen Programmiersprachen ist sie jedoch nicht vorhanden und muss selbst implementiert werden, so auch in Ada. Die Implementierung in C++ (siehe Abschnitt 3.4.1) adaptiert eine frei verfügbare `BigInt`-Implementierung.[4] Allein die Änderungen umfassen über 2.800 Zeilen. Dass die Implementierung mehrfacher Genauigkeit nicht trivial ist, erst recht wenn es effizient implementiert werden soll, belegen zahlreiche Quellen, die sich diesem Thema widmen.

Knuth [1969] stellt effiziente Algorithmen für die Arithmetik von großen Zahlen vor. So ist je ein Algorithmus zum Addieren von großen positiven Zahlen und zur Subtraktion einer kleineren von einer größeren Zahl beschrieben und in der Maschinensprache von MIX angegeben. Nach Abdecken aller möglichen Fälle, wie z.B. der Addition von negativen Zahlen, sind die Algorithmen leicht selbst zu implementieren.

Für die Multiplikation von großen Zahlen liefert Knuth auch einen Algorithmus, der die Schulmethode umsetzt. Für große Zahlen gibt es effizientere Verfahren, wie er selbst feststellt. Zwei zweistellige Zahlen A und B lassen sich auch wie folgt darstellen: $A = a_1 * d + a_2$ und $B = b_1 * d + b_2$, wobei d die Basis des gewählten Zahlensystems ist (z.B. 10). Die Schulmethode berechnet das Ergebnis über die vier Produkte $b_2 * a_2$, $b_2 * a_1$, $b_1 * a_2$ und $b_1 * a_1$, die dann entsprechend ihrer Wertigkeit verschoben und addiert werden:

$$C = A * B = (b_2 * a_2) * d^2 + (b_2 * a_1 + b_1 * a_2) * d + b_1 * a_1.$$

Die Multiplikationen mit d lassen sich einfach durch Verschieben auf die nächsten Worte darstellen. Wie in der Schule schreiben wir die Ergebnisse versetzt untereinander, bevor wir sie addieren. Die Multiplikationen der verschiedenen a_i und b_j jedoch werden real durch die Multiplikation der Basismaschine durchgeführt und benötigen im Vergleich zu den anderen Operationen viel Zeit. Ein russischer Mathematiker entdeckte in den 60'ern, dass sich die Formel zur Berechnung von C auch anders schreiben lässt:

$$C = A * B = (b_2 * a_2) * d^2 + (b_2 * a_2 + b_1 * a_1 - (b_2 - b_1) * (a_2 - a_1)) * d + b_1 * a_1.$$

Auf den ersten Blick sieht das komplizierter aus. Auf den zweiten Blick hingegen fällt auf, dass hier durch die mehrfache Verwendung der Produkte $b_2 * a_2$ und $b_1 * a_1$ tatsächlich nur drei statt vier Multiplikationen benötigt werden, wenn die Zwischenergebnisse erhalten bleiben. Diese Idee ist Grundlage für einen Algorithmus zur schnellen Multiplikation großer Zahlen,

[4] www.rossi.com

der ausnutzt, dass d frei wählbar ist [Karatsuba und Ofman, 1963]. So können Zahlen, die sich fortwährend in zwei gleich große Teile teilen lassen, rekursiv über dieses Verfahren multipliziert werden. Mit jedem Rekursionsschritt sind so nur drei Multiplikationen notwendig. Dieses Verfahren benötigt viel Speicher, da große Zwischenergebnisse entstehen. Zudem erfordert die fortwährende Teilbarkeit der Zahlen, dass sie (zumindest virtuell) mit führenden Nullen aufgefüllt werden, bis sie eine Länge haben, die einer Potenz von 2 entspricht.

Beide Verfahren wurden implementiert und getestet. Die Abbildung 5.2 zeigt das Laufzeitverhalten von 5 Algorithmen, die jeweils die gleichen Zahlen der auf der X-Achse aufgetragenen Länge miteinander multiplizieren.[5] Die Wortlänge beträgt 32 Bit.

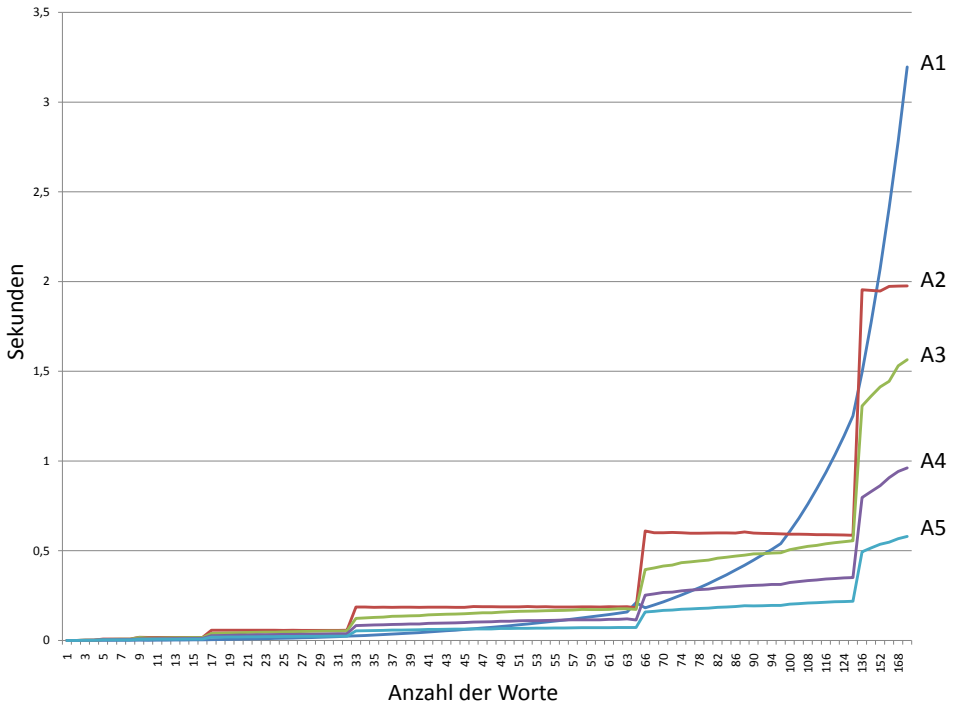


Abbildung 5.2.: Laufzeiten der Multiplikationen durch verschiedene Implementierungen

Algorithmus A1 implementiert die Schulmethode und zeigt deutlich das Verhalten eines Algorithmus mit quadratischem Aufwand. Hier werden in zwei geschachtelten Schleifen alle Worte der einen Zahl mit jedem Wort der anderen Zahl multipliziert und die Ergebnisse ihrer Wertigkeit entsprechend verschoben addiert.

Die Algorithmen A2 bis A5 implementieren das Karatsuba-Verfahren. Bei A1 werden die beiden Zahlen zunächst auf eine einheitliche, fortwährend durch 2 teilbare Länge gebracht und dann das rekursive Verfahren gestartet. Die Kurve zeigt eine signifikante Treppenform, da große Zahlen mit 65 Worten genauso gehandhabt werden, wie jene mit 128.

[5] Bei den Laufzeiten geht es weniger um die realen Zeiten als mehr um das Verhältnis zwischen den verschiedenen Algorithmen. Messungen auf einem Intel(R) Core(TM)2 Quad CPU Q6600, 2.6 GHz. (Nur ein Kern genutzt.)

Algorithmus A3 versucht die „Treppen“ etwas zu glätten, in dem die Zahlen nur virtuell erweitert werden und die Rekursion sofort mit dem Ergebnis Null beendet wird, wenn ein Faktor die reale Länge Null aufweist. Dennoch bleiben deutliche Sprünge bei den Zweierpotenzen.

Algorithmus A4 und A5 nutzen jeweils einen internen Datentyp zur Darstellung der großen Zahlen und sparen somit viel „Overhead“ bei der Abfrage einzelner Werte, die sonst in einer durch das Package `ArrayList` bereitgestellten Datenstruktur gehalten werden. Algorithmus A4 nutzt verkettete Listen und A5 Arrays, die in Ada sehr effizient unterstützt werden.

Als *break even point* wird der Punkt bezeichnet, an dem ein Algorithmus von seinem Laufzeitverhalten her besser wird, als ein anderer. Das Verfahren nach Karatsuba kann sehr effizient implementiert und so dieser Punkt bereits mit Zahlen bestehend aus 5 bis 50 Worten erreicht werden [Jebelean, 1997]. Algorithmus A2 erreicht diesen Punkt erst mit 150 Worten, also 4800 Bits und damit deutlich im Bereich von Kryptographie-Anwendungen. Der effizientere Algorithmus A5 dagegen erreicht diesen Punkt bereits ab einer Wortlänge von etwa 45, also 1440 Bits. Ein möglicher *break even point* bei 5 Worten gelingt nur mit einer Implementierung in Maschinensprache und einem geeigneten Befehlssatz. In einer Hochsprache wie Ada, sind 45 Worte ein sehr brauchbares Ergebnis.

Wie Laufzeittests zeigen, ist es unzweckmäßig, immer nur eines dieser Verfahren zu verwenden. Stattdessen ist eine Kombination dieser Verfahren anzustreben. Die Tabelle in Abbildung 5.3 zeigt die Laufzeiten des Algorithmus A5 kombiniert mit dem Algorithmus A1, wobei die jeweilige Zahl angibt, ab welcher Wortlänge der Algorithmus A5 zur Anwendung kommt (Schwelle). D.h. die mit „0” gekennzeichnete Spalte stellt die Laufzeiten des Algorithmus A5 wie in Abbildung 5.2 dar (nur für deutlich größere Wortlängen). Die Zahlen 32, 64 und 128 meinen den Algorithmus A5, der mit jedem Rekursionsschritt prüft, ob die zu multiplizierenden Zahlen eine kleinere Wortlänge als die vorgegebene Zahl haben. Ist dem so, wird das (Teil-)Ergebnis mit Algorithmus A1 berechnet. Diese Kombinationen sind alle besser, als der Algorithmus A5 und benötigen sogar weniger Speicher. Mit den Schwellwerten 64 und 128 ist die Multiplikation zweier Zahlen möglich, die jeweils aus über 130.000 Worten bestehen (je 4 MegaBit). Dazu werden 1,8 bzw. 1.4 GB Speicher und 694 bzw. 720 Sekunden zur Berechnung benötigt.[6] Zahlen mit 2^{22} Bit liegen derzeit weit außerhalb unserer Vorstellungskraft, decken aber dennoch gerade mal $0,1\%$ des theoretisch Möglichen ab. Obwohl das Laufzeitoptimum bei einer Schwelle von 32 Worten liegt, wird eine Schwelle von 64 implementiert, um besonders große Zahlen speicherschonender berechnen zu können.

Es soll an dieser Stelle nicht verschwiegen werden, dass noch weitere effiziente Verfahren existieren [Cook, 1966; Schönhage und Strassen, 1971]. Diese können noch effizienter große Zahlen multiplizieren. Allerdings sind sehr laufzeitintensive Initialisierungen notwendig, die eine Laufzeitsteigerung erst für sehr große Registerlängen möglich machen [Knuth, 2001]. In einer Implementierung in Ada ist mit keiner Verbesserung des implementierten Algorithmus zu rechnen.

Zur Division großer Zahlen wird oft ein Verfahren genutzt, das der aus der Schule bekannten schriftlichen Division ähnelt. Das Problem hierbei ist, die intuitiv getroffene Wahl des jeweiligen Quotienten bei großen Worten durch einen Algorithmus zu beschreiben. Knuth [1969] hat eine sehr effiziente Methode für die Berechnung eines Quotienten gefunden, die er zuvor so

[6] Hardware entspricht der letzten Fußnote, 3 GB Arbeitsspeicher.

Wortlängen der Faktoren	Laufzeiten in Abhängigkeit der Anzahl der Worte (Schwelle)				
	0	16	32	64	128
1024	0,5829	0,1378	0,0951	0,1004	0,1132
2048	1,5642	0,3321	0,3185	0,3339	0,3719
4096	4,8339	1,1378	1,0950	1,1448	1,2590
8192	15,0338	3,9369	3,8150	3,9608	4,3206
16384	47,5146	13,9105	13,5995	13,9703	15,0360
32768	-	50,2132	49,2331	50,3920	53,4578
65536	-	184,7720	181,3394	184,9638	194,1570
131072	-	-	-	694,5104	720,8059

Abbildung 5.3.: Laufzeiten des Algorithmus A5 kombiniert mit A1 abhängig einer Schwelle für die Wortlänge in Sekunden

überprüft, dass nur in sehr wenigen Fällen eine nachträgliche (und zeitaufwendige) Korrektur notwendig ist.

Das Verfahren nach Knuth wurde hier implementiert, auch wenn es Techniken gibt, die eine noch effizientere Division großer Zahlen möglich machen. Jebelean [1997] beschreibt einen Algorithmus, der unter Nutzung der Karatsuba-Multiplikation den *break even point* erst oberhalb einer Anzahl von 1000 Worten erreicht. Zudem berechnet dieser Algorithmus keinen Rest. Dieser müsste separat berechnet werden.

Die Erkenntnis aus diesem Schritt ist wichtig auch für zukünftige Programmierer, die den UVC implementieren werden. Soll die Implementierung möglichst schnell und einfach vollzogen werden können, so muss eine Programmiersprache gewählt werden, die bereits die Funktionalität zum effizienten Rechnen mit unbegrenzt großen Zahlen bereithält.

Die Programmiersprache Java enthält für die Arithmetik mit unbegrenzt großen Zahlen die Klasse `java.util.BigInteger` und `java.util.MutableBigInteger`. Letztere hat die Fähigkeit, große Zahlen zu multiplizieren. Hier wurde jedoch nur die triviale Multiplikation nach der Schulmethode implementiert.[7]

Für die Programmiersprachen C und C++ gibt es Literatur, die eine Implementierung einer kompletten Applikation beschreibt, die effiziente Algorithmen für die Arithmetik für große Zahlen enthält [Welschenbach, 2005].

Für eine sehr effiziente Implementierung in Maschinensprache muss sehr viel Zeit veranschlagt werden. Hier können die Algorithmen, die in der Sprache für MIX angegeben sind, hilfreich sein. Knuth [1969] hat in diesem Bereich viele Grundlagen geschaffen, die von allen bekannten Anwendungen, die Arithmetik für sehr große Zahlen bereitstellen, aufgegriffen werden. Obwohl die Implementierung eine Verbesserung um den Faktor 10 gegenüber einer Implementierung in einer Hochsprache verspricht, stellt sie für eine weitere Portierung die größte Hürde dar. Maschinencode wird auch mit guter Dokumentation schnell unübersichtlich und ist generell schwer lesbar, gerade, wenn einzelne Befehle auf der aktuellen Basismaschine nicht mehr vorhanden sind oder von ihrer Funktionalität abweichen. Da diese Implementierung als Referenz dienen soll, wird auf maschinennahe Implementierungen verzichtet.

[7] Sourcecode eingesehen der Version 1.2 vom 19.12.03

5.3.3. Speicher

Der UVC ist mit einem segmentierten Speicher ausgestattet. Im Code werden genutzte Segmente über eine 32-Bit-Nummer angesprochen. Die mögliche Umgehung dieser Grenze über die Parametersegmente wurde bereits dargestellt (siehe Abschnitt 3.2).

Implementierungen des UVC müssen in der Lage sein, sehr viele Segmente zu verwalten. Als Schlüssel zur Ablage bieten sich die Segmentnummern an. Besonders effizient lässt sich eine Verwaltung implementieren, wenn Vergleiche der genutzten Schlüssel wenig Aufwand verursachen. Die Implementierung nutzt daher als Schlüssel den Datentyp `Unsigned_32`. Damit lassen sich theoretisch 2^{32} Segmente verwenden.

Gemäß Spezifikation bietet jedes Segment einen unbegrenzten bitweise adressierbaren Speicher und unbegrenzt viele Register.

Registerverwaltung

Register werden über eine Nummer angesprochen. Im Code wird dazu ein 32-Bit-Wert genutzt, wobei das erste Bit den indirekten Registerzugriff anzeigt. Durch diese Indirektion sind theoretisch unbegrenzt viele Register nutzbar (siehe Abschnitt 3.2). Register sollten möglichst schnell im laufenden Programm zugreifbar sein. Eine entsprechende Datenstruktur zur Verwaltung einer unbegrenzten Anzahl von Registern, die zudem beliebig, also auch in nicht fortlaufender Folge genutzt werden kann, sind balancierte Suchbäume. Solche Suchbäume benötigen jedoch einen Schlüssel, um die eigentlichen Werte abzulegen. Auch im Fall der Register bieten sich die Registernummern als Schlüssel an. Um einen effizienten Vergleich zu ermöglichen, wird erneut der Datentyp `Unsigned_32` verwendet. Somit sind 2^{31} Register direkt zugreifbar und noch einmal soviel indirekt.

Speicherverwaltung

Auf den unbegrenzten Speicher kann ebenfalls wie auf Register wahlfrei zugegriffen werden. Niemand schreibt einer UVC-Anwendung vor, den Speicher am Stück oder immer bei Adresse 0 beginnend zu nutzten. Auch hier bieten sich zur Realisierung balancierte Suchbäume an. Als Schlüssel kann die Adresse genutzt werden. Jedes genutzte Bit einzeln im Suchbaum abzulegen, würde keine effiziente Implementierung ermöglichen, aber einen Punkt der Spezifikation umsetzbar machen:

> *The sequential memory can be initialized by first loading a value in a register and then storing the register value in the memory through a store instruction. Of course, a register can also be initialized by loading an already initialized value from the sequential memory.* [Lorie und van Diessen, 2005]

Eine greifbare, daraus ableitbare Regel wäre, dass nur bereits geschriebene Bits auch gelesen werden können. Denn noch nicht initialisierte Bits werden nicht spezifiziert.

Zugunsten einer effizienten Realisierung des Speichers müssen statt einzelner Bits größere Speicherblöcke genutzt werden. Eine Überwachung, welche Bits innerhalb eines Blocks bereits gesetzt wurden und welche nicht, ist nicht sinnvoll umsetzbar. Untermauert wird diese

Einschätzung durch die verfügbare Java-Implementierung, die offensichtlich auch Speicherblöcke nutzt und nur den Verstoß gegen obige Regel außerhalb dieser Blöcke mit Fehlermeldungen ahndet. Noch nicht initialisierte Bits stehen hier auf 1. Die so feststellbare Größe eines Speicherblocks liegt bei exakt 50.000 Byte. Mehr als 63 MB pro Segment sind in der Java-Implementierung nicht adressierbar, auch nicht bei Nutzung nur jedes zehnten Speicherblocks.

Der Speicher der eigenen Implementierung soll den tatsächlich verfügbaren Speicher ausnutzen dürfen und der Zugriff darauf möglichst schnell sein. Die daher genutzten Speicherblöcke umfassen je 1024 Worte (4.096 Byte) und werden in einem B-Baum verwaltet. Dieser spezielle Suchbaum eignet sich hervorragend, um bei extremer Speichernutzung einzelne Teil-Bäume komplett auszulagern [Bayer und McCreight, 1970]. Die eigene Implementierung wird diese Fähigkeit nicht haben, sie ließe sich jedoch aufgrund der verwendeten B-Bäume leicht einarbeiten. Eine effiziente B-Baum Implementierung für die Sprache Modula wird von Wirth [1975] angegeben. Nach leichter Anpassung[8] und Ausbesserung kleinerer Fehler,[9] konnte diese Implementierung übernommen werden.

Auch zur Verwaltung der Speicherblöcke wird ein Schlüssel benötigt und erneut wird der Datentyp `Unsigned_32` genutzt. Die Bitadresse des jeweils ersten Bits ist eine hervorragende Wahl. Sie enthält aufgrund der Blockgröße und der Wortlänge aber immer 15 auf Null stehende Bits. Werden nur die relevanten Bits der Adresse in den 32 Bits des Schlüssels gespeichert, sind 2^{47} Bits, also 16 TB pro Segment adressierbar.

Das Speichern und Laden von Bits aus und in den Speicher ist nichts der Informatik Unbekanntes. Die Implementierung dessen ist aber nicht zu unterschätzen. Gerade in diesem Bereich kam es während der Implementierung zu den häufigsten Fehlern. Grund dafür sind die zahlreichen Sonderfälle. So muss es möglich sein, eine beliebige Anzahl von Bits an jede beliebige „ungerade“ Adresse zu speichern. Zudem können mehr oder weniger Bits zu speichern sein, als im Register vorhanden sind. (Hier wurde die Annahme getroffen, dass dann mit Nullen aufzufüllen ist bzw. rechtsbündig abgezählt wird.) Sonderfälle entstehen auch, wenn über Wort- oder Speicherblockgrenzen hinweg gespeichert oder geladen werden soll. Einen Algorithmus in einer Hochsprache zu entwickeln, der effizient ist und alle diese Fälle berücksichtigt, ist nicht trivial.

Besonders unzweckmäßig ist in diesem Zusammenhang die Spezifikation des UVC, der die Ein- und Ausgabe nur über den Speicher zulässt. Werte werden generell in Registern gehalten. Sollen diese ausgegeben werden, müssen sie zuvor gespeichert werden, da der Befehl `OUT` nur eine frei wählbare Anzahl Bits aus dem Speicher ausgibt. Zur Illustration folgt ein Programm-Ausschnitt des JPEG-Decoders von IBM (mit eigenen Kommentaren):

```
IN    0,1     1,500   1,400  # Empfänger bereit? (Nachricht
                             # selbst wird ignoriert)
STORE 1,65*   1,420   0,108  # speichert den Wert
OUT   0,40080 0,108   1,420  # gibt Wert aus (Pixel Rot)
IN    0,1     1,500   1,400  # bereit?
STORE 1,66*   1,420   0,108  # speichert den Wert
```

[8] Die Sprachen Modula und Ada sind sich sehr ähnlich.

[9] In der Implementierung von N. Wirth wird ein Zeiger so umgehangen, dass Kreise im Baum entstehen können (S. 337, Behandlung des Unterlaufs). Zudem wird die Status-Variable `h` nach dem Löschen der Knoten nicht neu belegt (S. 338).

```
OUT   0,40090 0,108   1,420  # gibt Wert aus (Pixel Grün)
IN    0,1     1,500   1,400  # bereit?
STORE 1,67*   1,420   0,108  # speichert den Wert
OUT   0,40100 0,108   1,420  # gibt Wert aus (Pixel Blau)
```

Der nicht vollständig vermeidbare Engpass des Speicherzugriffs wird somit auch auf die Ein- und Ausgabe übertragen. Hier bietet sich die Erweiterung um weitere Befehle an, die eine Ein- und Ausgabe in und aus Registern ermöglichen.

Segmentverwaltung

Ein Segment verwaltet somit ausschließlich zwei Suchbäume. Aber wie werden die Segmente verwaltet? Die Spezifikation gibt hier leider irreführende Hinweise, indem sie physische Segmente explizit erwähnt. Von einer UVC-Anwendung werden ausschließlich logische Segmentnummern genutzt, die zur Laufzeit auf physische Segmente abgebildet werden. Bis auf die Segmentnummern 0 und 3 bis 999 können jedem Unterprogramm andere physische Segmente unter gleicher logischer Nummer zugeordnet werden.

Schaut man sich die verschiedenen Segmente genauer an, stellt man fest, dass jede Art von Segmenten eigenständig verwaltet werden kann und physische Segmentnummern dabei keine Rolle spielen. Diese Erkenntnis ist nicht trivial. Die eigene Implementierung nutzte zunächst physische Segmentnummern aufgrund der suggestiven Vorgabe. Auch die Java-Implementierung nutzt eine komplexere Abbildung [van der Hoeven et al., 2005]. Die Implementierung in C++ schließlich speichert alle Segmente in nur einem Suchbaum.

- Das Segment 0 und die Segmente 3 bis 999 sind *shared*, d.h. sie können von allen Unterprogrammen aus genutzt werden. Die physische Segmentnummer muss daher immer der logischen entsprechen. Zur Verwaltung dieser Segmente bietet sich ein Array an, das zentral im Prozessor verwaltet wird. Als Index kann direkt die logische Segmentnummer genutzt werden. Der Zugriff auf diese Segmente ist somit sehr schnell.

- Das Segment 1 ist für jedes Unterprogramm ein anderes, aber für jede Instanz das gleiche. Es bietet sich also an, jeweils ein Segment einem Unterprogramm fest zuzuordnen. Auch der Zugriff auf dieses Segment ist damit sehr schnell.

- Das Segment 2 ist das übergebene Parametersegment. Jede Instanz eines Unterprogramms sieht somit ein möglicherweise anderes Segment, das aber bereits an anderer Stelle verwaltet wird. Diese Verknüpfung kann im aktuellen Prozessorzustand gespeichert werden. Mit einem Unterprogrammaufruf ist neben dem *Instruction Counter* und dem Verweis zum aktuellen Unterprogramm auch diese Zuordnung zu sichern und mit der Rückkehr wiederherzustellen. Der Zugriff auf dieses Segment wird so ebenfalls sehr schnell. Die Spezifikation enthält hier einen schwerwiegenden Fehler, der die richtige Interpretation stark erschwert, potentielle Programmierer abschreckt und Autoren wie Gladney [2007] falsch abschreiben lässt. Die korrekte Anwendung des Segments erschließt sich erst mit der Auswertung des UVC-Programmbeispiels im Anhang der Spezifikation. Ein Beleg dafür, dass in Assembler-Sprache vorliegende Testprogramme eine unabdingbare Erweiterung zur Spezifikation darstellen.

- Die Segmente mit Nummern größer 999 sind private Segmente und nur für die jeweilige Instanz eines Unterprogrammes sichtbar. Diese Segmente sind nach Beenden, also mit dem Rücksprung aus dem Unterprogramm unerreichbar und sollten gelöscht werden. Mit dem Aufruf eines neuen Unterprogramms werden die aktuellen privaten Segmente unsichtbar und das aufgerufene Unterprogramm kann seine eigenen Segmente nutzen. Somit ist die Verwaltung dieser potentiell zahlreichen Segmente in einem Suchbaum sinnvoll, der seinerseits ebenfalls Bestandteil des aktuellen Prozessorzustands ist. Als Schlüssel kann die logische Segmentnummer verwendet werden. Die Schnelligkeit des Zugriffs auf private Segmente ist so nur abhängig von der Anzahl der im aktuellen Unterprogramm genutzten privaten Segmente.

Eine komplexe Abbildung auf physische Segmentnummern entfällt hierbei vollständig. Diese in Abbildung 5.4 dargestellte verteilte Segmentverwaltung ermöglicht Laufzeiten von nur 80% gegenüber der suggerierten Variante. Zweckmäßig ist auch der erreichte Nebeneffekt: Alle zu löschenden Segmente (ausschließlich die rot dargestellten privaten Segmente) werden in einer separaten Struktur gespeichert und können so „auf einen Schlag“ gelöscht werden. Zum Löschen ist kein Suchen im Baum notwendig. Auch kann auf die Implementierung des komplexeren Algorithmus zum Löschen einzelner Elemente aus dem balancierten Suchbaum vollständig verzichtet werden. Gleiches gilt nämlich auch für die Verwaltung der Register und der Speicherblöcke.

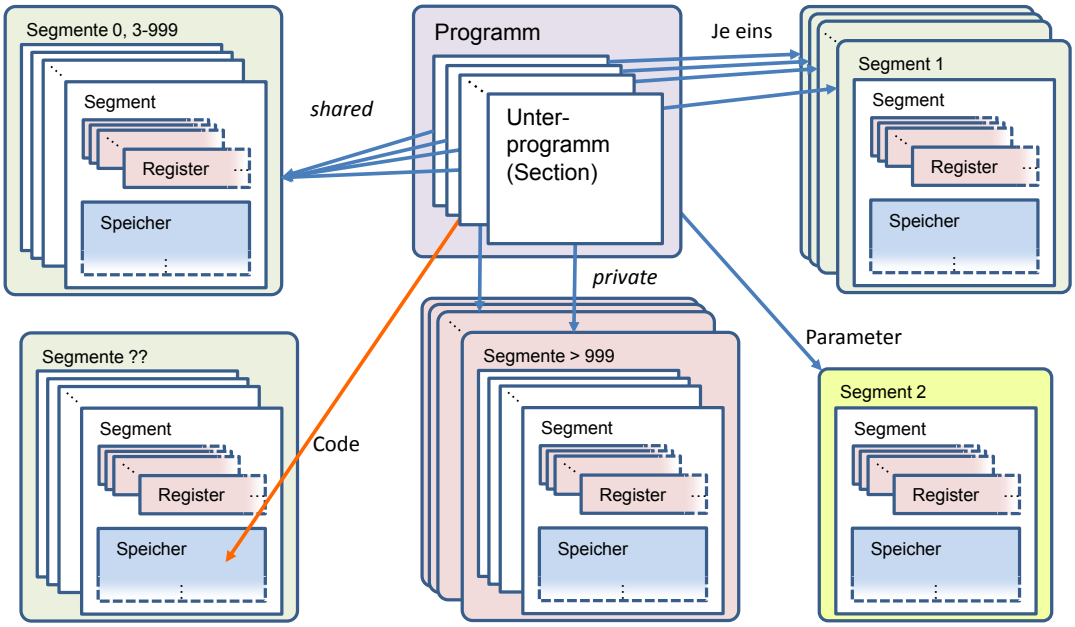


Abbildung 5.4.: Verteilte Segmentverwaltung

Verwaltung der Unterprogramme

Eine wichtige Frage lässt sich aus der Spezifikation heraus nicht vollständig beantworten: Was ist der Unterschied zwischen einem Unterprogramm und einem Segment?

In der Spezifikation steht, dass der Code eines Unterprogramms in einem Segment gespeichert ist (orangener Pfeil in Abb. 5.4). Damit ist der bitweise adressierbare Speicher gemeint. Daraus lässt sich die Möglichkeit ableiten, dass der Code zur Laufzeit veränderbar wäre, oder neue Unterprogramme erstellt werden könnten. Weiter heißt es:

> *The address space of a section always contains segments 0, 1, and 2, plus a segment that contains the section code, plus an arbitrary number of segments containing variables and data.*

Das bestärkt diesen Schluss. Aber wie bekommt ein Unterprogramm Zugriff auf seinen Code? Den Nummern 0, 1 und 2 sind Segmente mit konkreten Funktionen zugewiesen. Über alle übrigen Segmentnummern werden „gewöhnliche“ Segmente zugreifbar.

Für jedes Unterprogramm wird vom Programmierer eines UVC-Programms eine eindeutige Nummer angegeben. Man könnte erwarten, dass diese Nummer der Segmentnummer entspricht. Zumindest Segmente mit den Nummern zwischen 3 und 999 sind für alle Unterprogramme in gleicher Weise sichtbar. Der Speicher und somit der Code könnte manipuliert werden. Tests mit der verfügbaren Java-Implementierung scheiterten jedoch. Es lassen sich nur solche Programme ausführen, die Nummern größer 1000 nutzen. Der Zugriff auf das jeweilige private Segment mit dieser Nummer ermöglicht den Zugriff auch nicht. Ein weiterer Versuch, selbst ein Programm in den Speicher eines Segments zu schreiben und anschließend aufzurufen, scheiterte ebenfalls.

Selbstmodifizierender Code ist mit der Java-Implementierung nicht möglich. Der erst nach der Erstellung der Referenzimplementierung verfügbare, in C++ implementierte UVC unterstützt dagegen eingeschränkt selbstmodifizierenden Code in *shared* Segmenten.

Soll selbstmodifizierender Code nicht unterstützt werden, so kann eine völlig neutrale Struktur für die Speicherung der Unterprogramme gewählt werden. Die Speicherung der Befehle im Bitcode ist sehr ineffizient. Der Zugriff auf den Speicher ist wie bereits dargestellt langsam. Das Laden einzelner an „ungeraden“ Adressen gespeicherter Befehle verursacht eine fortwährende Verschiebung einzelner Bits.

Sinnvoller ist die Speicherung in fertigen Strukturen, die sehr schnell aufzunehmen sind. Wie der Abschnitt 5.3.4 detailliert darstellen wird, wurde in der Implementierung auch so vorgegangen.

Speichernutzung

Wie nutzt man den gigantischen Speicher mit den Speicherbefehlen? Dem UVC stehen zwei Befehle für den Zugriff auf den Speicher zur Verfügung:

```
LOAD    Reg1 (Ziel)     Reg2 (Adresse)   Reg3 (Länge)
STORE   Reg1 (Quelle)   Reg2 (Adresse)   Reg3 (Länge)
```

Aber auf welches Segment greifen diese Befehle eigentlich zu? Ganz sicher ist es nicht das Segment, in dem der Code des Unterprogramms steht. Nach einigen Tests wurde klar, dass das Register `Reg2` nicht nur die Adresse innerhalb des Segments angibt, sondern mit seiner Zugehörigkeit zu einem Segment auch dieses festlegt. Ein Beispiel:

```
LOADC  10,1 32 0        # Die Adresse (offset) im Segment
LOADC  20,1 32 42       # Der zu speichernde Wert
LOADC  30,1 32 17       # Anzahl der zu speichernden Bits

STORE  20,1 10,1 30,1   # Speichert den Wert 42 an die
                        # Adresse 0 mit insgesamt 17 Bits
```

Dieses Beispiel, ausgeführt mit der IBM-Implementierung in Java, speichert den Wert in das Segment `10`, da das Register, das die Adresse enthält, zum Segment 10 gehört. Dass an dieser Stelle überhaupt Fragen aufkommen, ist ein Problem der Spezifikation. In diesem Punkt ist sie zu vervollständigen. Der hier implementierte UVC richtet sich nach dieser angenommenen Vorgabe, die durch spätere Tests mit der IBM-Implementierung in C++ bestätigt wurde.

5.3.4. Prozessor

Durch Verzicht auf selbstmodifizierenden Code, wird es möglich, Anweisungen nicht als Bitstrom in Segmenten zu speichern, sondern in leicht zugreifbaren Datenstrukturen. Opcodes, Segment- und Registernummern aus dem bitorientierten Speicher zu laden, ist eine komplexe Aufgabe, die Rechenzeit verschlingt. Der Zugriff auf einzelne Speicherblöcke ist von der Komplexität $O(log\ n)$. Zudem bedarf es mehrerer Bitoperationen. Dieser Zugriff wird im Idealfall nicht mit jedem Bestandteil der Instruktionen einzeln notwendig sein. Schneller ist in jedem Fall die Organisation der Instruktionen in maßgeschneiderten Datenstrukturen in einem Array. Hier wurden drei verschiedene Versionen implementiert, die jeweils Umsetzungen während der Entstehung der vorliegenden Arbeit angeeigneter Techniken sind.

Trivial geparster Bitcode

Die Datenstruktur der Instruktionen ist wesentlich für die Geschwindigkeit deren Abarbeitung. Um den langsamen Zugriff auf den bitadressierbaren Speicher zu umgehen, müssen alle zur Interpretation notwendigen Informationen direkt in dieser Struktur gespeichert werden. In Abbildung 5.5 ist die genutzte Struktur an einem Beispiel skizziert. Deutlich wird an dem Beispiel auch die im UVC-Bitstrom sehr häufig auftretende Bitverschiebung.

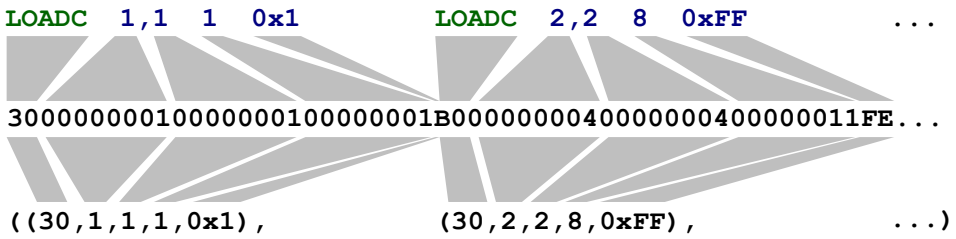


Abbildung 5.5.: Mögliche Struktur zur beschleunigten Interpretation des Codes

Dieses Beispiel macht deutlich, dass hierbei nur der langsame Speicherzugriff umgangen wird, indem die einzelnen Bits einmalig beim Lesen des Programms in diese Struktur übertragen werden. Bei der Interpretation der einzelnen Instruktionen müssen dann die einzelnen Zahlen z.B. als Opcode oder als Segmentnummer interpretiert werden.

Besonders profitiert dabei der Befehl **LOADC**, der eine variable Anzahl Bits als Parameter bekommt, die passend der genutzten Datenstruktur für Register rechtsbündig ausgerichtet wird. **LOADC** bildet einen deutlichen Schwerpunkt eines UVC-Programms (siehe Abb. 5.6).

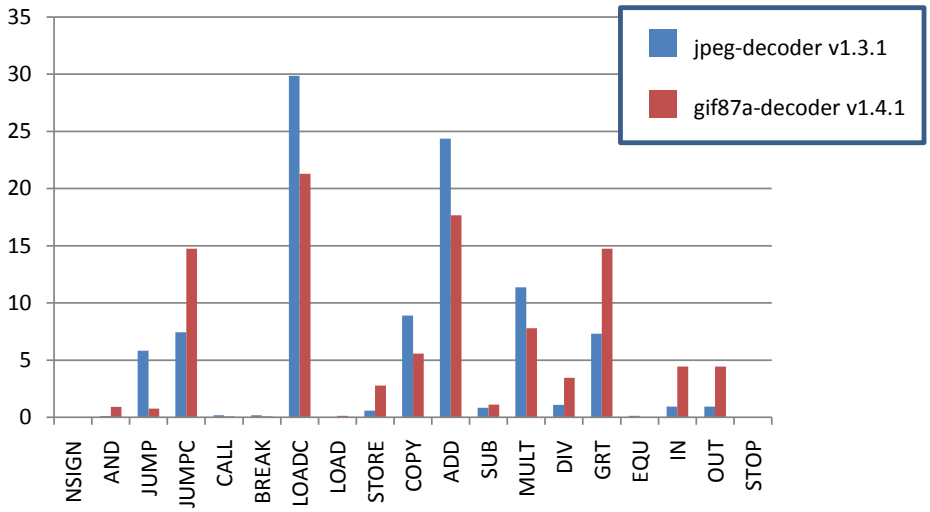


Abbildung 5.6.: Häufigkeiten der Befehle am Beispiel der beiden Decoder-Programme

Diese Version des trivial geparsten Bitcodes wurde als erstes implementiert.

Mit erstem Zugriff verlinkte Register

Einige der Zahlen in der in Abbildung 5.5 dargestellten Struktur kodieren immer gleiche Segmente bzw. Register darin. Im kurzen Beispielcode werden zwei Konstanten in Register geladen. Im Gegensatz zum Segment 1 im ersten Fall, das während der gesamten Programmausführung gleich bleibt, ist das Segment 2 abhängig vom jeweiligen Parametersegment, das vom aufrufenden Programm übergeben wurde. Die Segmente, die aus Sicht eines Unterprogramms gleich bleiben, sind die Segmente 0 bis 999 mit Ausnahme des Segments 2 (grün gefärbte Segmente in Abb. 5.4). Die privaten Segmente werden mit jedem Aufruf des Unterprogramms neu instanziiert.

Im Falle erstgenannter Segmente lässt sich ein weiterer Schritt eingesparen. Mit der Interpretation der Segment- und Registernummern muss das Segment aus dem Array geholt und aus dem darin gespeicherten balancierten Baum das Register geladen werden. Da dieses Register jedoch immer das gleiche ist, kann ein „Link“ in der Struktur abgelegt werden. Neben der Referenz auf Register über Nummern und per direktem Link, muss ein weiterer Fall berücksichtigt werden. Der indirekte Registerzugriff ist auch in öffentlichen Segmenten möglich. Hierbei kann das Register, das die Nummer speichert, zusammen mit dem Segment direkt verlinkt werden. So wird zwar nicht das Ziel-Register verlinkt, gespart wird aber immerhin einer von zwei Zugriffen auf den balancierten Baum im Segment.

Eine Referenz auf ein Register innerhalb einer Instruktion könnte folgende Struktur haben, in der diese drei Fälle in Form eines varianten Verbundtyps realisiert wurden:

```
type ReferenceType is (legal,indirect,linked);
type RegisterReference(RegRefType : ReferenceType := legal) is record
   case RegRefType is
      when legal =>
         segn : Interfaces.Unsigned_32;
         regn : Interfaces.Unsigned_32;
      when indirect =>
         segm : Segment.Segment;
         regi : Register.Register;
      when linked =>
         regl : Register.Register;
   end case;
end record;
```

Innerhalb einer Instruktion können diese Referenzen entsprechend zugeordnet werden. Mit varianten Verbundtypen lassen sich zwar Konstanten von Registern unterscheiden, jedoch ist es in Ada83 nicht sinnvoll, alle Operandenkombinationen zu unterscheiden, da auch intern jeweils unterschiedliche Bezeichnungen notwendig werden. Beim Laden des Programms enthalten alle Referenzen nur Segment- und Registernummern. Während der Programmausführung können Register der gleich bleibenden Segmente geladen und dauerhaft in der Struktur verlinkt werden. Auf diese Weise werden nur tatsächlich benötigte Register bearbeitet. Nachteilig wirkt sich die Fallunterscheidung aus, die zum Differenzieren der Varianten notwendig ist. Newman und Dennis [2009] geben einen „Trick“ an, wie sie Fallunterscheidungen effizient implementieren können. Voraussetzung ist, dass einer den Regelfall darstellt, also viel wahrscheinlicher ist. Im Programmcode wird dann der wahrscheinlichere direkt implementiert. Liegt doch der andere Fall vor und es kommt aufgrund nicht geprüfter Voraussetzungen zu einem Fehler, kann der entsprechende Code als Ausnahmebehandlung implementiert werden. Die Funktion `getReg` lädt ein Register gemäß der Referenz und nutzt diesen „Trick“:

```
function getReg(Reg : RegisterReference) return Register.Register is
   begin
      return Reg.regl;
   exception when others =>
      if Reg.RegRefType = legal then ...
```

Diese Implementierung benötigt lediglich 80% der Laufzeit und zeigt damit eine deutliche Verbesserungen. Beide Decoderprogramme von IBM benutzten jedoch im Schwerpunkt öffentliche Segmente und profitieren daher enorm von den direkt verlinkten Registern.

Sinnvoll kann auch das Verlinken der Register privater Segmente sein. Allerdings muss mit jedem Unterprogrammaufruf eine Kopie der Struktur erstellt und entsprechend aufbereitet werden. Nach Verlassen des Unterprogramms ist diese temporäre Struktur zu verwerfen. Dieser Aufwand lohnt sich daher nur, wenn Register privater Segmente entsprechend häufig innerhalb eines Unterprogramms genutzt werden, z.B. innerhalb einer Schleife. Die verfügbaren Anwendungen für den UVC lassen eine solche Nutzung privater Segmente nicht erkennen.

Übersetzung für eine interne virtuelle Maschine

Die finale Referenzimplementierung setzt einen weiteren „Trick“ um. Newman und Dennis [2009] erzeugen während der Ausführung ihrer VM intern neue Klassen, deren generierter

Code die auszuführenden Programme abbildet. Zur Generierung des Codes nutzen sie mehrerer kleinere Instruktionen. Durch den in der JVM integrierten JIT-Compiler (*Just in Time*) wird dieser Code zuverlässig optimiert. Die Idee, die zu emulierende Maschine zunächst auf eine interne virtuelle Maschine abzubilden, ließ sich gewinnbringend für den UVC anpassen.

Die Instruktionen des UVC nutzen maximal drei Register. Nur eine Instruktion (`LOADC`) lädt eine Konstante, insgesamt sechs Instruktionen nutzen zusätzlich die implizite Segmentangabe. Im Falle des UVC kennt der Prozessor der intern genutzten virtuellen Maschine daher zusätzlich drei feste Register `R1`, `R2` und `R3`, eine Konstante und ein Segment. Die Instruktionen des UVC werden direkt für die festen Register implementiert; `ADD` addiert z.B. immer `R2` auf `R1`. Zusätzliche Instruktionen können die Register entsprechend vorladen. Im folgenden Beispiel ist der Code eines kleinen Unterprogramms gegeben. Dieser addiert zwei Register.

```
UProg
1001

LOADC  1,1  1  0x1
LOADC  2,2  8  0xFF
ADD    2,2  1,1
BREAK
```

Im folgenden Code sind die Instruktionen für die interne virtuelle Maschine gegeben. Die violett eingefärbten Instruktionen gewährleisten den effizienten Zugriff auf die Register ohne Fallunterscheidungen im laufenden Programm. Die Implementierung der einzelnen Anweisungen fällt zudem sehr knapp aus. Bereits beim Laden des Programms werden die passenden Instruktionen in den Code geschrieben. Die Fallunterscheidungen sind somit nur einmalig vorab nötig. Der so erzielte Laufzeitvorteil beträgt zusätzliche 10%.

```
GETR1  (1,1)    # bereits verlinkt
GETC   (0x1)    # bereits für Register optimiert
LOADC

LOADR1  2,2     # reguläre Registerangabe
GETC   (0xFF)
LOADC

LOADR1  2,2     # ist hier überflüssig!
GETR2  (1,1)
ADD
BREAK
```

Eine verbliebene Fleißaufgabe wäre, die Implementierung weiterer komplexerer interner Instruktionen, die bei Bedarf gleich zwei oder drei Register in einem Schritt laden. Sollen diese ebenfalls keine Fälle unterscheiden müssen, so sind zusätzlich weitere 9 bzw. 27 Einzelinstruktionen zu implementieren.

Newman und Dennis [2009] nutzen auch den Umstand aus, dass in den meisten JVM interne JIT-Compiler verwendet werden, die solchen Code zur Laufzeit übersetzen und dabei optimieren. Auch in unserem Fall wäre eine Optimierung möglich. Recht einfach ist das Identifizieren einer überflüssigen Anweisung wie sie im Beispiel bereits gekennzeichnet ist. So könnte im Code eine interne Sprunganweisung dazu dienen, diese zu überspringen:

```
GETR1   (1,1)    # bereits verlinkt
GETC    (0x1)    # bereits für Register optimiert
LOADC

LOADR1   2,2     # reguläre Registerangabe
GETC    (0xFF)
LOADC
BRI      +2      # überspringt nächste Instruktion
LOADR1   2,2     # verbleibt zur korrekten Abarbeitung für Sprung zu ADD
GETR2   (1,1)
ADD
BREAK
```

Gute Compiler könnten den vorliegenden Code weiter optimieren:

```
GETR2   (1,1)    # bereits verlinkt
GETC    (0x1)    # bereits für Register optimiert
LOADC2           # lädt direkt nach R2

LOADR1   2,2     # reguläre Registerangabe
GETC    (0xFF)
LOADC
ADD
BREAK
```

Allerdings müssen diese Compiler den Kontrollfluss kennen und ausschließen können, dass es einen Sprung direkt zur `ADD`-Instruktion gibt. Wie noch detailliert im Kapitel 7 dargestellt wird, ist das im Falle des UVC (noch) nicht realisierbar. Algorithmen dieser Art sind in der Referenzimplementierung nicht realisiert.

Aufgrund des Verzichts des selbstmodifizierenden Codes wurden die vorgestellten Techniken möglich. Der Umkehrschluss aber gilt nicht. Keine dieser Techniken schließt selbstmodifizierenden Code aus: Der Bitcode eines Unterprogramms kann in einem Segment gespeichert werden, parallel zur erstellten Struktur. Finden Schreibzugriffe auf dieses Segment statt, so muss vor der nächsten Ausführung die Struktur neu erstellt bzw. angepasst werden. Dieses Verfahren gehört zum Repertoire einer VM-Implementierung mit JIT-Compiler und wäre auch für die Referenzimplementierung realisierbar.

Für gewöhnlich werden Prozessoren virtueller Maschinen mit einer Schleife implementiert, die jeweils einen Befehl abarbeitet [Phillips, 2010]. Innerhalb der Schleife wird per Fallunterscheidung der Opcode des Befehls ausgewertet. Die meisten Programmiersprachen bieten `case`-Konstrukte, die z.B. in Java effizient mittels Sprungtabellen umgesetzt werden. Nachdem eine eigene Struktur für die Instruktionen verwendet wird, muss nicht zwingend der Opcode gespeichert werden. Stattdessen könnten direkt die Adressen der Routinen hinterlegt sein. Leider ist eine solche Optimierung mit der gewählten Sprache Ada83 nicht umsetzbar.

Im vorliegenden Fall kann dennoch optimiert werden, zudem auf äußerst einfache Weise. In der Abbildung 5.6 sind die Häufigkeiten der Befehle dargestellt, wie sie bei der Ausführung der Decoderprogramme auftreten. Aus diesen lassen sich allgemeingültige Zusammenhänge ableiten. `LOADC` hat in jedem UVC-Programm eine dominante Rolle und `STOP` wird immer nur ein einziges Mal ausgeführt. Sprünge sind deutlich häufiger als Unterprogrammaufrufe. Durch die derart angepasste Fallunterscheidung konnten 10% der Laufzeit eingespart werden.

Die weiterentwickelte Spezifikation gibt mit der Vergabe der Opcodes eine implizite Reihenfolge vor, die den zu erwartenden Häufigkeiten entspricht. Soweit möglich werden dabei sinnvolle Gruppierungen beibehalten.

5.3.5. Ein- und Ausgabe

Der UVC ist für die Ausführung von Filterprogrammen konzipiert. Das bedeutet im Fall des UVC die Ausführung eines Programms, das von außen Daten als Bitstrom bekommt, diese verarbeitet und einen Bitstrom als Ausgabe produziert. Dafür stehen die zwei Befehle `IN` und `OUT` zur Verfügung. Wie diese implementiert werden, ist offen gelassen. Z.B. ist es möglich, den UVC mit Schnittstellen auszustatten, über die kommuniziert wird (van der Hoeven et al. [2005] nutzen z.B. ein Java-*Interface*). Ebenso kann über die Standardein- und -ausgabe kommuniziert werden, was für Filterprogramme nützlich ist und daher implementiert wurde.

Intuitiv wurde angenommen, dass ein Decoderprogramm – in welcher Form auch immer – Daten über `IN`-Befehle einliest und die decodierte Information über `OUT`-Befehle ausgibt. Ein Aufruf wie der folgende ist jedoch nicht möglich:

```
> uvc jpeg_decoder_v1.3.1.uvc < test_1.jpg > bild.dat
```

Der erste Entwurf des UVC von Lorie [2002a] sah keinerlei Ein- bzw. Ausgabe für den UVC vor. Kommuniziert werden sollte einzig über den Speicher des UVC. Eingabedaten sollten direkt in den Speicher des UVC geladen und auszugebenden Daten direkt entnommen werden. Nach eigenen Erfahrungen ist der direkte Zugriff auf den Speicher eines unter Umständen noch laufenden UVC-Programms sehr gefährlich. Zweifellos ist dieser Zugriff schneller, aber bedarf einer sehr genauen Spezifikation und Kenntnis um die vorliegende Implementierung des UVC. Im Jahr 2010 hat diese Kenntnis keiner über den UVC des Jahres 2050.

Die Zugriffszeit ist wohl dafür verantwortlich, dass die beiden von IBM veröffentlichten Decoder nach wie vor Bilddaten im Speicher des Segments 0 erwarten. Angegeben ist das in der Spezifikation der Decoder allerdings nicht. Auch ist in der Spezifikation des UVC dazu nichts zu finden. Erst eigene Tests des eigenen UVC mit diesem Decoder ließen das notwendige „manuelle“ Laden der Bilddaten erkennen. Es ist seltsam, dass nichts über die Art der Eingabe festgehalten wurde. Dabei wäre mit einem einzigen Befehl das Laden des Bildes spezifikationskonform möglich. Dem Befehl `IN` kann eine frei wählbare Anzahl an Bits übergeben werden. Diese wird dann an ein durch das Programm vorgegebenen Speicherort geschrieben:

```
LOADC 0,1 0 0       # Adresse 0 im Segment 0
IN    0,2 0,3 0,1   # Laden der Daten an die gegebene Adresse
...                 # Länge der Daten steht in 0,3 und der Typ in 0,2
```

5.3.6. Garbage Collection

Viele moderne Programmiersprachen wie Java bieten eine sogenannte *Garbage Collection*, einen Algorithmus, der den Speicher zur Laufzeit bereinigt. Ada83 jedoch nicht. Das heißt, dass in Ada-Programmen kein Algorithmus im Hintergrund läuft und prüft, ob angeforderter Speicher noch genutzt wird. Wird zur Laufzeit eines Programms Speicher benötigt, z.B. weil dynamische Datenstrukturen erzeugt werden, wird Speicher vom Betriebssystem angefordert

und bereitgestellt. Programme haben einen Zeiger (Verweis) auf diese Daten. Werden neue Datenstrukturen erzeugt, verweist der ursprüngliche Zeiger auf die neuen Daten. Die alten Daten sind unerreichbare „Speicherleichen“. Eine *Garbage Collection* prüft regelmäßig während der Laufzeit eines Programms, ob angeforderter Speicher vom Programm noch zugreifbar ist, also ob weiterhin Verweise darauf vorhanden sind. Falls nicht, wird der nicht mehr genutzte Speicher „eingesammelt“ und erneut bei Bedarf zur Verfügung gestellt. Da Ada-Programme, wenn sie über den freien Ada-Compiler erzeugt werden, keinen Prozess zur Speicherbereinigung laufen haben, wächst bei der Verwendung von dynamischen Datenstrukturen der Speicher stetig an. Das auf dem eigenen UVC früherer Version ausgeführte Decoder-Programm von IBM bricht bei dem Versuch ein 500x375 Pixel großes Bild aufzubereiten nach ca. 20% ab. Zu diesem Zeitpunkt ist der genutzte Heap auf 2 GB angewachsen. Die Implementierung von IBM benötigt dafür 64 MB, von denen 6 MB zur Laufzeit dazukommen.[10]

An verschiedenen Stellen wird von der Referenzimplementierung Speicher dynamisch allokiert und nur temporär genutzt. Wird eine Konstante in ein Register geladen, ersetzt diese den bisherigen Wert, der in einer dynamischen Liste gespeichert war. Wird diese Liste zuvor Element für Element freigegeben, wächst der Speicherbedarf nur noch halb so schnell an. Eine weitere Stelle liefern abermals die Register bei den verschiedenen arithmetischen Operationen und beim Kopieren. Das Vorgehen hier ist gleich: Vor dem Aufnehmen eines neuen Wertes muss die dynamische Liste, die den alten Wert enthält, zunächst freigegeben werden. Das entsprechende Paket ist **`ArrayList`**. Mit dem Aufruf eines Unterprogramms werden sehr viele dynamische Datenstrukturen angelegt, viele davon als B-Baum. Nutzt ein Unterprogramm private Segmente, werden diese mit dem Aufruf erzeugt. Nach Rückkehr zur aufrufenden Section werden diese Segmente nicht mehr genutzt. Die Register und der Speicher dieser Segmente können und müssen wieder freigegeben werden. Die entsprechenden Pakete sind **`BTree`**, **`Section`** und **`Segment`**.

Für das Funktionieren ist der speicherneutrale Umgang mit dynamischen Strukturen nicht notwendig; er kostet vielmehr wertvolle Laufzeit.[11] Das Beispiel des großen Bildes zeigt jedoch, dass beim unachtsamen Umgang mit dem Speicher Programmfehler auftreten können. Nach Implementierung der genannten Algorithmen benötigt der eigene UVC für das Anzeigen des oben genannten Testbildes 3 MB Speicher, unverändert über die gesamte Laufzeit.

Für künftige Entwickler des UVC bedeutet das, dass entweder eine Programmiersprache bzw. ein Compiler gewählt wird, der *Garbage Collection* bereitstellt, oder zusätzlicher Aufwand eingeplant werden muss.[12]

5.4. Zusammenfassung der Ergebnisse

Als Ergebnis dieses Experiments werden Indizien geliefert, die zwei Fragen betreffen. Zum einen die Frage nach dem tatsächlichen Implementierungsaufwand und zum anderen die Frage nach der Integrität der Spezifikation.

[10] Java SE Runtime Environment (JRE) Version 6

[11] Im Fall des JPEG-Decoders entspricht das etwa 11% Leistungsverlust.

[12] Das Anpassen der eigenen Implementierung erweiterte den Quellcode um 169 zusätzliche Zeilen reinen Code und inklusive ausführlichem Testen wurden 4 Tage benötigt.

5.4.1. Implementierungsaufwand

Die Basis dieses Abschnitts bildet eine Tabelle, in der auf Stunden genau dokumentiert ist, wie viel Zeit für die Implementierung der einzelnen Komponenten des UVC benötigt wurden. Anhand dieser Tabelle wird eine erste Abschätzung entwickelt, wie lange tatsächlich eine Implementierung eines UVC dauert. Berücksichtigt werden dabei verschiedene Voraussetzungen wie die Vertrautheit mit der gewählten Entwicklungsumgebung sowie Kenntnisse der notwendigen Algorithmen. Nicht zu vergessen ist der Aufwand, um bestimmte Voraussetzungen zu schaffen. Hier soll daher zwischen der reinen Implementierung und der Implementierung von notwendigen „Helferlein“ unterschieden werden, wenn diese möglicherweise in anderen Programmiersprachen bereits verfügbar sind (z.B. hält Java einiges mehr bereit als Ada83).

Die folgende Tabelle (Abb. 5.7) erscheint geschönt. Das Studieren der Spezifikation und der vorhandenen Beispielprogramme, das Entwickeln eigener UVC-Testprogramme sowie das Entwickeln notwendiger Zusatzprogramme (wie ein Assembler) sind nicht aufgeführt. Einzig die reine Programmierzeit für die einzelnen Pakete und die Entwicklung von Testprogrammen für diese Pakete sind berücksichtigt. Die erste Version der Referenzimplementierung entstand in sechs Wochen, in denen ein Programmierer nahezu ausschließlich daran arbeitete.

	UVC				UVC mit GC			
	Zeiten in Stunden		Lines of Code		Zeiten in Stunden (zus.)		Lines of Code	
	Implementierung	Testen	adb	ads	Implementierung	Testen	adb	ads
UVC	2	7	160	15	0	2	163	15
MemMan								
Register	15	9	1025	32	1	4	1060	34
Segment	7	4	345	39	3	8	417	44
Processor	4	3	320	12	1	3	332	12
Section	6	2	217	44	0	0	217	46
UVCStack	3	2	22	25	1	1	26	28
Util								
ArrayList	6	4	129	35	2	2	140	35
BTree	4	3	261	35	1	3	279	37
Gesamt	47	34	2479	237	9	23	2634	251
		81		**2716**		**32**		**2885**

Abbildung 5.7.: Implementierungszeiten und Anzahl der „echten“ Programmzeilen.

Für die Referenzimplementierung wurden keine vorgefertigten Pakete genutzt. Sämtliche Funktionalität wurde selbst implementiert. Zum einen gibt es keine Pakete, die die gewünschte Funktionalität liefern, zum anderen wären solche Pakete nicht zwingend für die Implementierung in FORTRAN vorhanden, deren Basis hier gelegt wurde. Zukünftige Programmierer werden sehr wahrscheinlich Pakete für das effiziente Arbeiten mit Listen, Bäumen und für die beliebig genaue Arithmetik vorfinden. Java bietet bereits umfangreiche Funktionalität mit den Klassen `ArrayList`, `Map` und `BigInteger`. Sollten sich in ferner Zukunft andere Programmiersprachen durchsetzen, werden diese nicht mit weniger Funktionalität ausgestattet sein.

In der Abbildung 5.7 sind die benötigten Stunden und die Programmzeilen (ohne Leerzeilen und Kommentare) aufgeführt. Rechnet man aus diesen die in modernen Programmiersprachen zur Verfügung stehende Funktionalität heraus, umfasst die Implementierung der Register weniger als 200 Zeilen. Zudem kann die Implementierung der Pakete `ArrayList` und `BTree` entfallen. Übrig bleiben dann gerade noch ca. 1431 Zeilen Code und weniger als 50 Stunden.

Die zwei effizienteren Versionen (siehe Absschnitt 5.3.4) wurden jeweils innerhalb von zwei Tagen implementiert, wobei der jeweils zweite Tag dem ausführlichen Testen diente.

Der Ansatz von Gladney und Lorie [2005] legt einen UVC zugrunde, der weniger als ein Mann-Jahr zur Implementierung benötigt. Die Referenzimplementierung hat gezeigt, dass für Rechnerarchitekturen, wie sie zum Zeitpunkt der Erstellung der Spezifikation genutzt wurden, eine Implementierung innerhalb von sechs Wochen sehr deutlich unter einem Jahr gelingt. Und das ohne Verwendung vorhandener Programmpakete. Ein eingespieltes Team könnte in ferner Zukunft den Zugang zu längst verschüttetem Wissen womöglich noch schneller freilegen. Werden darüber hinaus Programmiersprachen genutzt, die bereits teilweise die geforderte Funktionalität bereitstellen, gelingt die Implementierung abermals schneller. Erhaltene UVC-Testprogramme könnten die Spezifikation leichter und schneller verständlich machen und die Fehlersuche erheblich vereinfachen. Das Kapitel 10 untersucht diesen Aspekt eingehend.

5.4.2. Integrität der Spezifikation

Dieses Experiment führte zu einer Referenzimplementierung in Ada83. Es ebnet den Weg für die im Anschluss angestrebte *Zeitreise*. Die hierbei aufgefallenen Unzulänglichkeiten der Spezifikation sollen die *Zeitreise* als solches nicht mehr gefährden. Dieses Experiment liefert Lösungen für jede offene Frage, die die Spezifikation lässt. Diese Erfahrungen sind besonders wichtig für eine Weiterentwicklung der Spezifikation, denn eines wird aus den bisherigen Darstellung deutlich: Die Spezifikation ist nicht aus sich heraus eindeutig interpretierbar.

Als Referenz für die in diesem Experiment gemachte Erfahrung sollte ein Student in einer Diplomarbeit den UVC implementieren. Leider scheiterte er an dieser Aufgabe. Statt der Untermauerung des durch die Referenzimplementierung erkennbaren Implementierungsaufwandes, wurde die Komplexität des UVC sichtbar. Offensichtlich enthält die Spezifikation keine implizite Anleitung zur effizienten und schnellen Implementierung. Das Scheitern des Studenten basiert nur zum Teil auf Fehldeutungen der Spezifikation, vielmehr waren es falsche Ansätze wie die Realisierung des bitadressierbaren Speichers mit jeweils einer auf Festplatte gespeicherten Datei, die den Speicher lückenlos abbildet. Der Student blieb ohne Abschluss.

Daraus leitet sich zum einen ab, nur qualifiziertes Personal mit der Implementierung eines UVC zu beauftragen. Zum anderen wird deutlich, dass die Spezifikation wie im Falle der physischen Segmentnummern einerseits irreführende Vorgaben macht und andererseits die mögliche Nutzung weit auseinanderliegender Teile des Speichers nicht betont. Ähnlich wie Ausführungsbestimmungen Verordnungen beigelegt sind, könnte die vorliegende Arbeit eine sinnvolle Ergänzung zur Spezifikation sein.

Wie die Ausführungen erkennen lassen, besteht Anpassungsbedarf für die vorhandene Spezifikation. Zum einen wurden Fehler und unklare Passagen gefunden, die sich erst nach Auswertung des Beispielprogramms oder im direkten Vergleich mit der – allerdings nicht vollständig spezifikationskonformen – Implementierung von IBM aufklären ließen. Zum anderen ist aufgefallen, dass die jetzige Spezifikation die Programmierung des UVC unnötig kompliziert macht. Zudem liegen an genau diesen Stellen die Gründe einer langsamen Programmausführung. Anpassungen müssen jedoch so gestaltet werden, dass sie der Universalität des UVC nicht abträglich sind. Die aufgrund der hier gesammelten Erfahrungen angedachte Weiterentwicklung wird im nächsten Kapitel dahingehend untersucht.

6. Projektion des UVC in die Vergangenheit - eine Zeitreise

Wäre das Konzept des UVC bereits vor 50 Jahren erdacht worden, wäre es bereits damals realisierbar gewesen? Es ist die Frage nach der Universalität. Dieses Alleinstellungsmerkmal des UVC ist bislang nur argumentativ belegt. Die hier angetretene *Zeitreise* ist die erste empirische Evaluation dieser für die Langzeitarchivierung wichtigsten Eigenschaft.

Bei näherer Betrachtung erscheint der *Proof of Concept* von IBM [Lorie, 2002b] nicht sehr tiefgründig: Eine Spezifikation in der heutigen Zeit erdacht, mit modernen Entwicklungswerkzeugen erstellte Software, getestet auf der Hardware unserer Generation. Genau genommen ist dieser Beweis ein Beleg für die Tauglichkeit des Ansatzes für die heutige Zeit – mehr nicht. Der zukunftsweisende Ansatz, den Lorie mit seinem Ansatz verfolgt, geht aber viel weiter. So soll doch gerade sein UVC so universell sein, dass er für beliebige Systeme implementierbar ist. Diese für den sinnvollen Einsatz innerhalb der Langzeitarchivierung zwingend notwendige Eigenschaft wurde in keinster Weise „bewiesen." Lediglich der Hinweis auf die Verwendung der Programmiersprache Java ist zu finden.

Diese Lücke wird mit diesem Kapitel zwar nicht mit einem Beweis, aber zumindest mit handfesten Indizien geschlossen. Funktionierende „Glaskugeln" gibt es nicht. Prognosen sind zwar teilweise möglich, aber gerade im Bereich der Entwicklung von Hardware sind zuverlässige Prognosen nur für die nächsten 10 bis 20 Jahre möglich. Bestimmte Annahmen allein durch die Erfahrungen zu bestätigen, ist spätestens seit „Moore's Law" auch der Informatik nichts Unbekanntes mehr.

Die Indizien werden in diesem Kapitel sehr aufwendig auf einer *Zeitreise* gesammelt. Im ersten Abschnitt wird die Idee dieser *Zeitreise* im Detail vorgestellt. Während dieser *Zeitreise* wird gezeigt, dass sich der UVC – mit kleineren Einschränkungen – für sehr verschiedene Hardwarearchitekturen implementieren lässt. Das ist ein sehr starkes Indiz dafür, dass er tatsächlich universell ist, bilden doch gerade die ersten Hardwaregenerationen ein enormes Spektrum des Möglichen und Vorstellbaren ab.

Sehr interessant sind auch die gewonnenen Ergebnisse bezüglich der Entwicklungszeiten. Hier können Archivare bzw. beauftragte Softwareentwickler den initialen und auch den kontinuierlichen Aufwand abschätzen, den dieser Ansatz einem Archiv auf Dauer abverlangt.

6.1. Die Idee hinter der Zeitreise

Unter der Annahme, dass die Spezifikation des UVC bereits vor ca. 50 Jahren vorhanden gewesen wäre, wie hätte eine Implementierung dessen ausgesehen? Mit welchen Hürden wäre zu kämpfen gewesen? Welche Einschränkungen hätten getroffen werden müssen? Wie hoch wä-

re der Entwicklungsaufwand für die Hardware bzw. Entwicklungsumgebungen für diese Zeit? Welcher Aufwand ist bei der Portierung einer vorhandenen Implementierung zu erwarten?

Antworten auf diese Fragen sind für mit Langzeitarchivierung betraute Personen äußerst interessant und bislang noch nirgends zu finden.

Zur Beantwortung dieser Fragen wurde eine Einrichtung benutzt, die einem Computermuseum nahe kommt. In einem Computermuseum werden historisch bedeutsame Computer und deren Peripherie zu Schau gestellt. Nur im sehr geringen Umfang ist diese Hardware noch voll einsatzfähig. An der Universität der Bundeswehr entsteht eine Einrichtung, die historisch relevante Hardware, insbesondere Großrechner, sammelt und für Forschungszwecke einsatzfähig bereit stellt: die datArena. Der Museumscharakter spielt hierbei eine sehr untergeordnete Rolle (siehe Abschnitt 2.4.2).

Die zur Entstehung der vorliegenden Arbeit vorhandene Ausbaustufe der datArena bietet lauffähige Großrechner verschiedener Generationen. Der Fokus liegt auf den Großrechnern, da in größeren Archiven – egal zu welchem Zeitpunkt der letzten 50 Jahre – keine handelsüblichen Computer zum Einsatz gekommen wären, um digital archiviertes Material auszuheben und für die Nutzung aufzubereiten. Sowohl das Archivsystem selbst als auch der UVC wird heute noch auf einem Großrechner zum Einsatz kommen, wie der Praxistest der Königlichen Bibliothek der Niederlande zeigt [Wijngaarden und Oltmans, 2004].

6.1.1. Die Großrechner der datArena

Während der Entstehung dieser Arbeit befand sich die datArena in einer Ausbaustufe, in der der älteste noch funktionsfähige Großrechner von der Firma Control Data Corporation stammt. Mit der CDC Cyber 180/860A-DP verfügt die datArena über einen funktionsfähigen Großrechner aus dem Jahr 1982. Leider war dieser wassergekühlte Großrechner aufgrund seiner komplexen Installation noch nicht einsatzbereit. Mit der CDC Cyber 180/960-31 aus dem Jahr 1988 stand jedoch ein entsprechender ca. 25 Jahre alter Großrechner zur Verfügung, um als Startpunkt der angetretenen *Zeitreise* zu dienen. Dieser Großrechner hat einen Hauptprozessor, 60 bzw. 64 Bit große Speicherworte und insgesamt 128 MByte Speicher. Wird er mit dem Betriebssystem NOS betrieben, so ist er im Cyber-170-Mode [Cyb, 1987] und mit seinen 60-Bit-Worten binär-kompatibel zur CDC 6600 aus dem Jahr 1964. Für die kompatiblen Geräte der 170er Baureihe gibt es zudem ein Emulator, der nahezu identische Laufzeiten wie die genutzte Cyber erreicht. So war es möglich, den UVC zu entwickeln und ausführlich zu testen, bevor der stromfressende Gigant zu den finalen Testläufen hochgefahren werden musste. Die Einschränkungen, die sich bei der Entwicklung des UVC auswirkten, gehen, bis auf die Größe des Hauptspeichers, alle auf die Architektur der CDC 6600 zurück. Mit der genutzten Cyber wird somit eine Zeitspanne beginnend im Jahr 1964 abgedeckt. Auch endet diese Zeitspanne nicht mit der Einführung erster 180er Modelle. Mit den moderneren Cyber-Modellen ließ sich das ältere Betriebssystem NOS parallel betreiben. Viele Anwendungen konnten so ohne Anpassungen noch über das Jahr 1990 hinaus genutzt werden.

Der CDC 6600 wurde von Seymour Cray entwickelt und prägte 1964 erstmals den Begriff „Supercomputer“ [Ceruzzi, 2003]. Damit ist der Start der *Zeitreise* passend platziert. Dieser skalare Supercomputer übertraf die Computer von anderen Anbietern wie IBM um einige Größen. Dennoch blieben viele Kunden diesen Herstellern treu, schon allein aufgrund der Softwa-

rekompatibilität. Für eine Betrachtung einer Zeitspanne von mehr als diesen knapp 50 Jahren, müsste der UVC initial auf einem der anderen Systeme implementiert werden. Abgesehen von den zeichenkodierten Programmen und Daten, die zur damaligen Zeit auf Lochkarten zwar digital, aber weiterhin menschenlesbar gespeichert wurden, gab es eine ausschließlich digitale Speicherung relevanter Informationen erst mit dem Beginn der Raumfahrt. Die Masse der wissenschaftlichen Primärdaten, die z.B. während den ersten Mondlandungen ab 1969 aufgezeichnet wurden, liegen ausschließlich digital vor [Drexler et al., 1990].

Unter der Annahme, dass der UVC initial auf einem skalaren Supercomputer implementiert wurde, so wäre die nächste Station innerhalb der *Zeitreise* die Ära der Vektorrechner. Während die skalaren Systeme zunehmend ihre Grenzen erreichten, suchte man nach anderen Wegen, die Leistungen der Supercomputer zu steigern. Einen Ausweg fand man in der Verarbeitung mehrerer Daten „gleichzeitig“ mit ein und derselben Operation. Die Zeit, die benötigt wurde, um Daten aus dem Speicher zu laden bzw. zu speichern, wurde jetzt, wann immer möglich, zum Rechnen benutzt. Bei vielen Anwendungen für Großrechner wie Windkanalberechnungen und Wettervorhersagen sind fortlaufende Daten jeweils mit derselben Operation zu bearbeiten. Sogenannte Vektorrechner können hier ihr Potential voll ausschöpfen.

Aufgrund der skalaren Architektur des UVC lässt sich die Vektorfähigkeit nicht gewinnbringend einsetzen. Für die den UVC nutzende Archivierungsmethode eignen sich Vektorrechner daher nicht. Diese Einschätzung basiert auf der aktuellen Nutzung des UVC durch die verfügbaren Applikationen, die alle die mögliche mehrfachgenaue Arithmetik nicht ausnutzen. In diesem Bereich wären durch Vektorisierung, Vorteile gegenüber skalaren Systemen – auch automatisiert – möglich. Die Routinen der mehrfachen Genauigkeit basieren auf Schleifen, die für einzelne Iterationen nahezu identische Schritte ausführen. Der FORTRAN-Compiler der verfügbaren Cray Y-MP kann solche Schleifen erkennen und automatisiert für die Vektorarchitektur optimieren.

Die datArena bietet neben dem Bull DPS 6000 aus dem Jahr 1990 und dem Fujitsu VPP 300 aus dem Jahr 1996 noch viele weitere Vektorrechner der Firma Cray. Zum Teil aus Platzgründen und zum Teil aus organisatorischen Gründen, die mit der Ausbaustufe zusammenhängen, war jedoch kein einziger Vektorrechner einsatzbereit.

Vom Rhythmus der sich abwechselnden Computergenerationen bietet sich ein Großrechner an, der im Vergleich zur genutzten Cyber eher klein ausfällt, aber bereits über zwei Prozessoren, 32 Bit Wortbreite und 256 MB Hauptspeicher verfügt und mehr als dreimal so schnell ist: die CDC 4680. Dieser zudem äußerst stabil laufende Großrechner wurde tatsächlich als Archivsystem genutzt. Auch wenn zur Zeit der Nutzung die Ansteuerung von Bandrobotern und die längerfristige Speicherung sehr großer Datenmengen im Vordergrund stand[1] und weniger die langfristige Wahrung der Authentizität.

Die Wortbreite von nur 32 Bit hatte im Bereich der Großrechner nur kurzzeitig Erfolg. Der verwendete MIPS-Prozessor wurde nur in sehr wenigen Systemen verbaut. Im Rahmen der *Zeitreise* kann über die CDC 4680 daher nur ein vergleichsweise kleiner Zeitbereich abgedeckt werden. Betrachtet man zusätzlich die über die genutzte Programmiersprache kompatiblen Systeme, müssen die verwendeten Erweiterungen berücksichtigt werden. So wurde eine dynamische Speichernutzung mit den von Cray eingeführten, aber nicht standardisierten

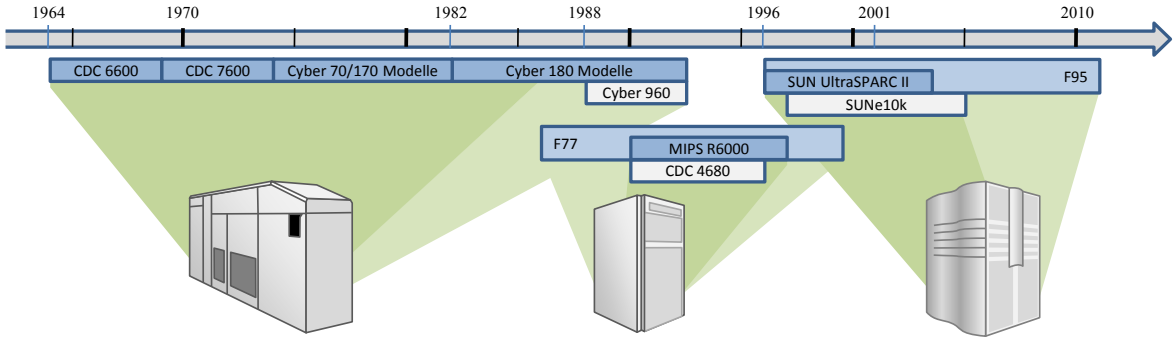
[1] www.lrz.de/wir/geschichte, eingesehen am 10.05.2011

Zeigern realisiert [Cra, 1986].

Aus heutiger Sicht wissen wir, dass Vektorrechner nur einen temporären Erfolg erzielen konnten. Bis heute wirklich erfolgreich in der Sparte der Großrechner sind (massiv) parallele Systeme. Die Entwicklung dieser Systeme überschnitt sich mit der Zeit der Vektorrechner. Aus dem Rahmen fällt z.B. die in der datArena verfügbare Cray T3D aus dem Jahr 1993, die als „Zwitter“ gesehen werden kann.

Interessanter, weil in der vorhandenen Ausbaustufe funktionsfähig verfügbar, sind dagegen die Origin 2000 von Silicon Graphics und die Enterprise 10000 der Firma SUN, beide aus dem Jahr 1997. Die in der datArena befindliche SUN hat den Vorteil, dass sie mit dem installierten SUN Studio 11 bereits über einen sehr komfortablen FORTRAN 95 Compiler verfügt und erhält daher innerhalb der *Zeitreise* den Vorrang. Die genutzte UltraSPARC II-Architektur gab es 1996 ebenso wie Compiler für FORTRAN 95. Da es solche Compiler auch noch für aktuelle Systeme gibt, deckt die genutzte SUN einen bis in die heutige Zeit hineinreichenden Zeitraum innerhalb der *Zeitreise* ab, auch wenn die genutzte SUN bereits 2005 gespendet wurde.

Die genutzten Systeme sind in der Abbildung 6.1 aufgeführt. Dabei sind die direkt durch die Hardware abgedeckten Zeiträume dunkler gekennzeichnet als die indirekt durch den Kompatibilitätsmodus oder kompatible Compiler abgedeckten. Insgesamt ergibt sich so ein abgedeckter Bereich von knapp 50 Jahren.



Name:	Cyber 180/960-31	CDC 4680-MP	SUN Enterprise 10000
Hersteller:	Control Data Corp.	Control Data Corp.	SUN Microsystems
Baujahr:	1988	1990	1997
Betriebssystem:	NOS	EP/IX	Solaris 5.10
Compiler:	FORTRAN IV	FORTRAN 77	FORTRAN 95 (SUN Studio 11)
Prozessoren:	1 (Cyber 180 Type P3)	2 (MIPS R6000 @ 60 MHz)	60 (UltraSPARC II 400 MHz)
Wortbreite:	60 Bit (170-Mode)	32 Bit	64 Bit
Speicher:	2.131.000 Worte (~15MB)	256 MB	60 GB
Leistung:	2,5 MFlops	11 MFlops pro CPU	800 MFlops pro CPU

Abbildung 6.1.: Die genutzten Großrechner im Überblick

6.2. Implementierung für CDC Cyber 180/960-31

Dieser Abschnitt skizziert die Implementierung des UVC für die mittlerweile über 25 Jahre alte Cyber 180/960-31 der Firma Control Data Corporation, kurz *Cyber 960*. Diese Maschine wurde mit einem FORTRAN IV Compiler für NOS ausgeliefert. Auf dem genutzten Modell

gibt es auch einen Compiler für FORTRAN V. Das parallel installierte NOS/VE mit seinen vielen Vorzügen spielt in diesem Schritt keine Rolle. Vorrangig ist die Abwärtskompatibilität zur CDC 6600 und somit ist die Nutzung des älteren FORTRAN Compilers obligatorisch.

Für die folgend beschriebenen Implementierungen sind die Unterschiede zwischen den FORTRAN-Versionen nicht wesentlich. Ab der Version V gibt es Strings, die Nutzereingaben einfacher und Ausgaben übersichtlicher hätten werden lassen [For, 1986]. Neu sind auch Blockstrukturen innerhalb der IF Anweisungen, deren Nutzung große Teile des Spaghetticodes mit einer erheblichen Anzahl an `GOTO`s hätte vermeiden können.

Nach einer Beschreibung der für die Implementierung wesentlichen Merkmale der Maschine, folgt eine Darstellung der Möglichkeiten und Einschränkungen der verwendeten Sprache FORTRAN IV. Den Schwerpunkt in diesem Abschnitt bildet die Beschreibung der Implementierung, die von der Speicherverwaltung und den virtuell geschaffenen Datentypen alles beschreibt, was zum Verständnis des Programmcodes unbedingt notwendig ist. Als Ergebnis liefert dieser Abschnitt Aussagen zum Arbeitsaufwand (vgl. Abschnitt 5.4.1) und zu erkannten Abhängigkeiten, sowohl seitens der Hardware als auch der Software in Form des Betriebssystems oder der verwendeten Programmiersprache.

6.2.1. Details zur Maschine

Die verwendete Cyber 960 stammt aus dem Jahr 1988. Sie wurde hauptverantwortlich von Seymor Cray entwickelt, der zu dieser Zeit noch für die Control Data Corporation arbeitete. Sie ist bereits vollständig auf Transistorbasis und im Vergleich zu den wassergekühlten Vorgängern luftgekühlt. Einzig der hohe Strombedarf (ca. 400 A), der zum „Anlassen" nötig ist, hindert an einer flexiblen Nutzung.

Aus Sicht des Programmierers gibt es hinsichtlich der CDC 6600 aus dem Jahr 1964 kaum Veränderungen. Lediglich die zugrunde liegende Technologie und damit die verfügbare Rechenleistung änderten sich erheblich. Im Cyber-170-Modus bietet die Cyber 960 die gleiche Anzahl an Registern unveränderter Wortbreite (60 Bit). Teilweise findet man in den verfügbaren Anleitungen dieser Zeit auch den Begriff *Byte*, der jedoch 6 statt der heutigen 8 Bit meint. Der verwendete Zeichensatz nutzt ebenfalls 6 Bit, wodurch 10 Zeichen in einem Speicherwort darstellbar sind. Die Hardware kann mit den 60 Bit in Form von Gleitkommazahlen die Grundrechenarten durchführen. Für wissenschaftliche Anwendungen mit intensiver Gleitkommanutzung wurde sie speziell entworfen. Für Ganzzahlen steht ein eigener Addierer zur Verfügung, für die Multiplikation muss die Gleitkommazahlmultiplikation genutzt werden, ebenso für die Division. Gleitkommazahlen nutzen 12 Bit für Exponent und Vorzeichen. Für die Ganzzahl-Multiplikation und Division stehen somit nur die 48 Bit der Mantisse zur Verfügung [Cyb, 1980]. Sowohl Operanden als auch Ergebnisse dürfen diese 48 Bit Grenze nicht überschreiten [For, 1983]. Details, die es trotz vorhandener Abstraktion durch die Sprache FORTRAN zu beachten gilt.

Ein weiteres Detail ist die Speicherarchitektur. Insgesamt bietet die Cyber 960 128 MB „Hauptspeicher." Jedoch ist dieser im Cyber-170-Modus vergleichsweise umständlich und zudem von einer einzelnen Anwendung nicht vollständig nutzbar. Lediglich 2^{17} Worte sind im „Random Access" zugreifbar (*Central Memory*). Das ist fast ein MB. Der Extended Core Storage (ECS) bildet den restlichen Speicher, dessen Inhalt jedoch über spezielle Funktio-

nen zunächst in den *Central Memory* kopiert werden muss, um darauf zugreifen zu können [For, 1983]. Damit ein Programm mehr als das eine MB nutzen kann, muss es selbst eine Speicherverwaltung implementieren, die wir heute unter dem Begriff *Paging* kennen, nur dass die Seiten nicht auf einer Festplatte, sondern in anderen Bereichen des Speichers ausgelagert werden.

Die CDC 6600 wurde mit unterschiedlich großem *Central Memory* ausgeliefert. Dieser konnte 16, 32, 65 oder 131 Tausend Worte umfassen. Mit der Einführung der CDC 6600 gibt es auch den erweiterten Speicher in unterschiedlichen Ausführungen. Ein solches Extended Core Storage (ECS) System wurde in einem oder zwei raumfüllenden Schränken (*cabinets*) geliefert und konnte 125, 300, 500, 1000 oder 2000 Tausend Worte aufnehmen [CDC, 1968]. Die maximale Konfiguration besteht somit aus einem knappen MB *Central Memory* und ca. 14 MB ECS. Dies ist auch die obere Schranke für Anwendungen im Cyber-170-Modus.

6.2.2. NOS FORTRAN Extended Version 4

FORTRAN IV, in der Literatur auch FORTRAN 66, ist seit 1975 standardisiert (DIN 66027). Allerdings ist dieser Standard nicht annähernd so umfangreich, dass er alle damaligen Systeme effizient nutzbar machte. So entwickelte jeder Hersteller von Großrechnern neben dem Betriebssystem auch entsprechende Entwicklungsumgebungen. Ausgelieferte Compiler unterstützten den Standard und enthielten eine schier unüberschaubare Anzahl von Erweiterungen. Sehr genau ist im genutzten Handbuch gekennzeichnet, was zum Standard gehört und was erweitert wurde. Die grau unterlegten Seiten der FORTRAN-Anleitung, die Erweiterungen kennzeichnen, haben einen sehr großen Anteil. Ohne diese Erweiterungen sind effiziente Programme kaum möglich, sie wären aber portabel. Der letzte Aspekt scheint in den 70'ern und 80'ern noch keine große Rolle gespielt zu haben. Interessant ist aber, dass viele dieser Erweiterungen in zeitlich folgende Standards einflossen.

Im Folgenden werden die Besonderheiten der genutzten erweiterten Sprache FORTRAN IV vorgestellt, sofern sie Einfluss auf die Implementierung des UVC für die Cyber 960 hatten und zum Verständnis des Quellcodes notwendig sind.

Programmcode

Der Programmcode wurde mit Hilfe eines heute üblichen Editors erstellt. So ließen sich gewohnte Funktionen wie Kopieren, Einfügen, Zeichenersetzungen usw. bequem nutzen. Unter NOS vorhandene Editoren sind dagegen unkomfortabel. Solange der Code über ein Terminalprogramm zur Cyber 960 transferiert wird, werden auch die Zeichen von ASCII in den von NOS genutzten 6-Bit-Zeichensatz konvertiert. Darin stehen nur Großbuchstaben zur Verfügung. Der entwickelte Quelltext enthält jedoch Groß- und Kleinbuchstaben zur besseren Lesbarkeit auf heutigen Systemen.

Deutlich mehr Einfluss auf die Programmentwicklung hat die Orientierung des Quellcodes an den Lochkarten. Eine solche Karte kann jeweils 80 Zeichen aufnehmen und enthält je eine Zeile des Quellcodes. Die ersten sechs Zeichen dienen dabei als Platz für Sprungmarken oder für spezielle Compileranweisungen. Die letzten acht Zeichen dienen in einem Lochkartenstapel der Sortierung und sind somit nicht für Quellcode nutzbar. Diese Besonderheiten werden

von keinem aktuellen Editor korrekt berücksichtigt. Hier ist der Programmierer auf sich allein gestellt.

Bei der Übertragung des Quellcodes werden von dem NOS Tool (`copycr`) zwei aufeinanderfolgende Zeilenumbrüche als Eingabeende gedeutet [NOS, 1988]. Der Quellcode enthält somit keine Leerzeilen. Um den Code übersichtlicher zu gestalten, finden sich stattdessen Kommentarzeilen, die nur ein `C` enthalten.

Ein FORTRAN IV Programm besteht aus einem **`PROGRAM`**, das sich wiederum auf Unterprogramme abstützen kann (**`FUNCTION`** und **`SUBROUTINE`**) [For, 1983]. Module oder Packages kennt FORTRAN IV nicht. Sämtliche Namen werden mit nur sieben Zeichen kodiert, wobei der erste Buchstabe eine Typisierung impliziert, die sich aber übersteuern lässt. Der vorhandene Compiler überprüft weder die Typen innerhalb eines Programmteils noch gibt es Fehler zur Laufzeit. Variablen werden nicht initial definiert, sondern können im Programm jederzeit genutzt werden. Zudem wird jeder Teil des Programms eigenständig übersetzt. So können auch größere Programme speicherschonend übersetzt und üblicherweise fortlaufend auf Band geschrieben werden. Jedoch führt das dazu, dass Namen der Unterprogramme und auch die Anzahl und Typen der Parameter nicht geprüft werden können. Einige Tippfehler im Quellcode werden somit vom Compiler nicht gefunden. Deren manuelles Auffinden hatte einen signifikanten Anteil an der Implementierungszeit.

Datentypen

FORTRAN IV kennt im Vergleich zu heutigen Programmiersprachen nur wenige Datentypen. Die wenigen verfügbaren orientieren sich sehr stark an der zugrunde liegenden Hardware, also an 60 Bit. Es gibt Fließkommazahlen einfacher und doppelter Genauigkeit und komplexe Zahlen, die jedoch keine Verwendung bei der Implementierung des UVC finden. Ganzzahlen sind über den Typ **`INTEGER`** verfügbar, der ebenfalls 60 Bit nutzt - jedoch im Einerkomplement. Neben den Zahlentypen gibt es noch einen Typen zur Darstellung von konstanten Zeichenfolgen. Die Hollerith-Konstanten kommen beim UVC jedoch nur zur Ausgabe von Status- oder Fehlermeldungen in Betracht und werden in allen Fällen vom Compiler direkt als Zeichenfolgen akzeptiert. Sie spielen daher zum Verständnis des Codes keine Rolle. Strings bzw. variable Zeichenketten gibt es nicht.

Sehr entscheidend ist der Umstand, dass es keine Möglichkeit gibt, dynamisch Variablen zu verwenden. Alle verwendeten Variablen müssen zum Programmstart feststehen, also an irgendeiner Stelle im Programm vorkommen. Insbesondere gibt es noch keine Speicherallokierung zur Laufzeit und keine Zeiger. Programmierer dieser Zeit wussten scheinbar immer, wie viele Variablen sie benötigten, oder konnten zumindest eine obere Grenze abschätzen und mit einem Array arbeiten. Nachvollziehbar ist dies nur in Anbetracht der noch stark limitierten Ressource „Computer“.

Arrays in FORTRAN IV dienen dem indizierten Zugriff auf eine vorher festgelegte Anzahl von Werten eines einheitlichen Typs. Dabei sind nur die wenigen primitiven Typen zugelassen. Die Indizierung beginnt immer mit 1. Heute übliche Verbunde gibt es nicht.

Programmstruktur

FORTRAN IV kennt zwei verschiedene Unterprogrammtypen. Mit **SUBROUTINE** werden Unterprogramme definiert, denen Parameter übergeben werden können. Werden Variablen übergeben und deren Werte im Unterprogramm überschrieben, so ändert sich auch der Inhalt der Variablen im umgebenen Programm. Mit dem Schlüsselwort **CALL**, dem vereinbarten Namen und einer Liste von Parametern wird ein Unterprogramm aufgerufen. Eine **FUNCTION** liefert dagegen immer ein Ergebnis, das direkt einer Variablen zugewiesen oder wieder als Parameter übergeben werden kann.

Bei dieser Implementierung des UVC wurde auf Funktionen verzichtet. Das vereinheitlicht den Code und erspart die sonst notwendige Typisierung jeder genutzten Funktion innerhalb eines jeden Programmabschnitts. Für zukünftige Implementierungen ist diese Beschränkung jedoch nicht sinnvoll. Zum einen können in jüngeren FORTRAN-Versionen Programme interne Subroutinen und Funktionen enthalten, die eine Übersetzung in einem Schritt ermöglichen und somit eine zusätzliche Typisierung unnötig machen. Zum anderen werden Programme durch den gezielten Einsatz von Funktionen lesbarer.

Speichernutzung

Der UVC ist hochgradig von dynamischen Strukturen abhängig. Es ist nicht vorhersagbar, wie intensiv der Speicher oder aber die Register von der jeweiligen Anwendung genutzt werden. Folglich sollte ein UVC im Extremfall den gesamten verfügbaren Speicher der Basismaschine für Register oder aber zum Abbilden des Speichers nutzen können. Dynamische Strukturen gibt es bei FORTRAN IV nicht. Einzig ein sehr großes Array kann genutzt werden, um darauf eine eigene Speicherverwaltung mit dynamischer Allokierung zu implementieren.[2] Diese muss für nahezu alle Programmteile gleichermaßen zugänglich sein. Hierzu bietet FORTRAN zwei Möglichkeiten. Zum einen können Variablen mit dem Aufruf eines Unterprogramms übergeben werden. Solange auf diese Weise nicht zu viele Parameter notwendig werden, ist das eine vernünftige Lösung. Zum anderen bietet die Verwendung eines **COMMON**-Blocks eine Lösung. Ein solcher Block enthält global definierte Variablen. Nachteilig ist dabei, dass jedem Unterprogramm dieser Block neu angegeben werden muss, da der Compiler jeden Abschnitt einzeln übersetzt. Vom Schreibaufwand her ist die erste Möglichkeit etwas umfangreicher, sorgt aber für leicht lesbaren Code und wurde daher anfangs genutzt.

Die Cyber 960 verfügt bereits über 128 MB Speicher. Ein einzelnes Array ist aber aufgrund der Speicherorganisation der Hardware an die Grenze von 2^{17} Worten gebunden. Das knappe MB teilen sich zudem Programmcode, weitere Variablen und einige Daten des Betriebssystems. Intensive Laufzeittests der kompletten UVC-Implementierung wiesen ein Array mit noch 80000 Worten als zuverlässig nutzbar aus.

Soll der erweiterte Speicher genutzt werden, so ist das mit speziellen Anweisungen nur in speziellen **COMMON**-Blocks möglich. Solche Blöcke werden benannt und sind dem ECS zugeordnet.[3] Variablen in diesem Bereich sind nicht direkt zugreifbar und müssen vor der Nutzung

[2] Das stellte eine durchaus übliche Vorgehensweise dar.

[3] Das ist eine spezifische Erweiterung der Sprache FORTRAN nur für diese Baureihe. Das zugehörige Anweisung: **LEVEL 3**

mit der Subroutine **READEC** in lokale Variablen kopiert werden. Dieses Kopieren wird von eigenen Prozessoren bewerkstelligt und ist besonders effektiv bei größeren zu bewegenden Speicherblöcken, also insbesondere bei Arrays. Werden diese Daten verändert, ist ein Rückschreiben mittels **WRITEC** notwendig. Jeder Block kann wiederum maximal 2^{17} Worte umfassen.

Daraus motiviert sich die zu realisierende Speicherverwaltung, die Grundlage für die gesamte Implementierung des UVC ist.

6.2.3. Details zur Implementierung

Das gesamte Programm lässt sich in sechs in der Abbildung 6.2 unterschiedlich eingefärbte, aufeinander aufbauende Bestandteile zerlegen. In heutigen Paradigmen würden diese Bestandteile in einzelnen Paketen gebündelt. Im Quellcode ist dieser Zusammenhang nur durch Kommentare sichtbar. Der Abbildung ist zu entnehmen zu welchem „Paket“ die einzelnen Unterprogramme gehören und was diese im Einzelnen leisten.

Im Folgenden wird auf die Eigenheiten dieser „ältesten“ Implementierung eingegangen. Von besonderem Interesse ist hierbei die Speicherverwaltung, die dynamische Datenstrukturen erst möglich macht, um UVC-Programmen die flexible Nutzung von Registern und bitweise adressierbarem Speicher zu gestatten. Neben der Speicherverwaltung sind die genutzten Datenstrukturen und die damit abgebildeten Datentypen aufgezeigt. Ohne diese Dokumentation ist der Quelltext nicht bzw. nur sehr schwer verständlich.

Jede UVC-Implementierung unterstützt Ein- und Ausgaben auf eigene Art und Weise. Sie sind in der Spezifikation wohl absichtlich offen gestaltet. Zumindest erweist sich das bei dieser Implementierung als vorteilhaft. Dieses spezielle Verhalten einer UVC-Implementierung muss daher ebenfalls sorgfältig dokumentiert werden, damit z.B. darauf aufbauende Programme wie Bildbetrachter die „Schnittstellen“ nutzen können.

Der Abschnitt schließt mit den gewonnenen Erkenntnissen aus diesem ersten Schritt der *Zeitreise*.

Primitive Speicherverwaltung

Die Speicherverwaltung ist zweistufig entwickelt worden. Zunächst stand die flexible Nutzung des verfügbaren Speichers unterhalb der Grenze von 2^{17} Worten im Vordergrund, also die ausschließliche Nutzung eines Arrays im Hauptspeicher. In einem weiteren Schritt wurde das „Paging“ unter Nutzung des erweiterten Speichers (Extended Core Storage - ECS) implementiert. Dieser Schritt erfolgte erst nach der Implementierung aller anderen Programmteile. Dieses „Nachflicken“ der ECS-Nutzung wurde notwendig aufgrund des genutzten Emulators, der zunächst keinen ECS unterstütze. Neben einigen verschenkten Stunden der Fehlersuche, erwies sich dieses Vorgehen jedoch auch als nützlich. So sind jetzt zwei Versionen verfügbar, die im Vergleich aufzeigen, wie viel Laufzeit mehr Speicher kostet, ohne genutzt zu werden.

Speicherverwaltung ohne ECS Die Speicherverwaltung basiert zunächst auf nur einem sehr großen Array. In diesem Array sind die ersten 9 Werte für die Organisation des eigentlichen Speichers reserviert. Aber was soll eigentlich in diesem Speicher abgelegt werden?

Package	Subroutine	Beschreibung
Main	UVC	Programm, das den UVC startet
Memory	INITMEM	Initialisiert den vom Programm genutzten Speicher
	GETVAL	Gibt ein bestimmtes Speicherwort
	SETVAL	Schreibt ein bestimmtes Speicherwort
	NEWELEM	Erzeugt ein neues Speicherelement (Listenelemt)
	FREELEM	Gibt Speicherelement frei
	NEWBLOC	Erzeugt neuen Speicherblock
	FREBLOC	Gibt Speicherblock frei
List	NEWLIST	Erzeugt neue (einfach verkettete) Liste (Kopf)
	ADDLIST	Fügt der Liste einen Wert (immer 40-Bit) an
	GETLIST	Gibt einen Wert einer Liste an geg. Position
	SETLIST	Setzt einen Wert einer Liste an geg. Position
	INSLIST	Fügt einen Wert an erster Position ein
	POPLIST	Entfernt ersten Wert aus Liste
	CLRLIST	Entfernt alle Werte aus der Liste
	FRELIST	Gibt gesamte Liste frei – inklusive Kopf
	LENLIST	Gibt die Anzahl der enthaltenen Werte
Register	NEWREG	Erzeugt neues Register
	INIREG	Initialisiert Register aus Liste und Vorzeichen
	PRNREG	Gibt ein Register aus
	BITLENG	Bestimmt zu einem 60-Bit-Wort das höchste genutzte Bit
	SADDREG	Addiert die Beträge zweier Register
	SSU1REG	Subtrahiert die Beträge zweier Register, R1 > R2
	SSU2REg	Subtrahiert die Beträge zweier Register, R1 <= R2
	ADDREG	Addiert zwei Register vorzeichenrichtig
	SUBREG	Subtrahiert zwei Register vorzeichenrichtig
	ANDREG	Verknüpft zwei Register mit AND
	ORREG	Verknüpft zwei Register mit OR
	NOTREG	Negiert alle Bit gemäß Länge eines Registers
	EQUREG	Vergleicht zwei Register auf Gleichheit
	GRTREG	Vergleicht zwei Register auf Größe (R1 > R2)
	MULREG	Multipliziert zwei Register vorzeichenrichtig
	SHLREG	Schiebt Register bitweise um max. 40 Bit nach links
	SHRREG	Schiebt Register bitweise um max. 40 Bit nach rechts
	CORREG	Korrigiert die Länge eines Registers entsprechend der Bits
	SDIVREG	Dividiert ein Register mit einem 20-Bit-Wert
	SMULREG	Multipliziert ein Register mit einem 20-Bit-Wert
	DIVREG	Dividiert R1 durch R2 und speichert den Rest in R3
Segment	NEWSGM	Erzeugt neues Segment
	GETSGM	Gibt ein physisches Segment zur logischen Nummer
	CLRSGM	Löscht ein Segment mit genutzten Speicher und Registern
	GETSGMR	Gibt ein Register eines bestimmten physischen Segments
	SSIGN	Speichert das Vorzeichen eines Registers als Bit im Speicher
	LSING	Lädt ein Bit als Vorzeichen eines Registers
	STORE	Speichert die Bitfolge eines Registers im Speicher
	LOAD	Lädt Bitfolge aus Speicher in Register
	STORWRD	Speichert 60-Bit-Wort im Speicher
	LOADWRD	Lädt 60-Bit-Wort aus Speicher
	LOADC	Lädt Konstante (auf Programmcode) in Register
Processor	GETREG	Gibt Register passend zu Segment- und Registernummer
	GETSEG	Gibt Segment passend zu Segmentnummer
	RUNPRG	Startet die Programmausführung
UVC	READWRD	Liest 60-Bit-Wort von der Eingabe
	GETWORD	Liest 60-Bit-Wort ab virtueller Eingabeposition
	LOADPRG	Lädt auszuführendes UVC-Programm
	LOADFIL	Lädt initial im Speicher des Segments 0 zu ladende Daten

Abbildung 6.2.: Hierarchisch geordnete Subroutines des Cyber960-Implementierung

Bei genauerer Betrachtung des UVC fällt auf, dass er mit lediglich zwei verschiedenen Typen auskommt: dynamische Listen und Speicherblöcke. Dynamische Listen können über einfach verkettete Listen realisiert werden, die pro Listenelement ein Speicherwort belegen.

Speicherblöcke hingegen sollten zur Realisierung des wahlfreien Zugriffs aus mehreren zusammenhängenden Speicherworten bestehen. Aufgrund der effizienten Berechnung des adressierten Blocks ist eine Größe in Zweierpotenz anzustreben. Eine solche Anzahl Bits lässt sich jedoch nicht zweckmäßig mit 60-Bit-Worten darstellen, es bleibt ein Überhang. Die gefundene Blockgröße von 1024 Bit benötigt 18 60-Bit-Worte. Es bleiben 56 Bit zur Markierung des jeweiligen Blocks, der somit selbst die Information tragen wird, welchen Speicherbereich er abdeckt. Somit ist der gesamte Speicher auf $2^{56} \cdot 1024$ Bit „beschränkt."

Bei einer gleichmäßigen Nutzung von Speicherblöcken und Listenelementen, ist mit einer schnellen Fragmentierung des genutzten Arrays zu rechnen, die trotz ausreichend ungenutzter Indizes keine Allokierung neuer Speicherblöcke zulässt. Eine grundlegend geteilte Nutzung des Speichers für Speicherblöcke und Listenelemente ist somit anzustreben, wobei eine optimale Lösung beide Extremfälle (siehe 6.2.2) abdecken muss. Dieses Prinzip der zweigeteilten Speichernutzung ist ebenfalls nichts Neues. In betagteren Systemen findet man häufig die Nutzung einer Halde und eines Kellers, die auf einander zu wachsen. Hier sind es Listenelemente und Speicherblöcke (siehe Abb. 6.3).

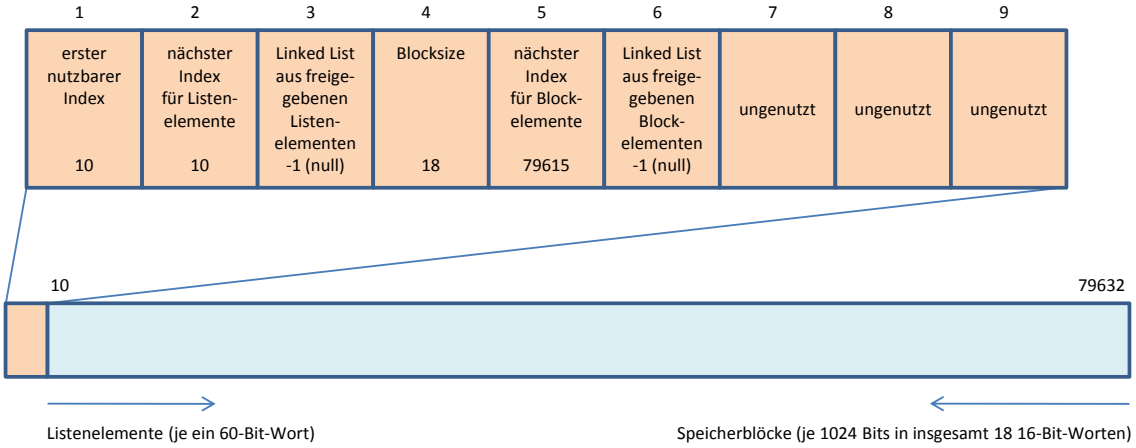


Abbildung 6.3.: Organisation des Speichers mit Hilfe eines Arrays

Da lediglich zwei verschiedene Typen zu speichern sind und zudem in zunächst unterschiedlichen Bereichen, eignen sich zwei verwaltete und überdies zeitgemäße Freilisten zur effizienten Speichernutzung [Cormen et al., 2001]. Die für die Verwaltung von Listenelementen genutzte Freiliste ist anfangs leer. Jedes freigegebene Element wird durch einfaches „Umzeigern" der Liste vorangestellt. Wird ein neues Listenelement gebraucht, so wird immer das erste Element der Freiliste entfernt und genutzt. Ist die Freiliste leer, wird im Array das Feld nach dem zuletzt auf diese Weise zugewiesenen Feld genutzt und dieser Maximalwert entsprechend angepasst. Zur Verwaltung von freien Listenelementen werden somit zwei Werte genutzt: ein Zeiger auf die Freiliste und der Index des zuletzt zugewiesenen Feldes. Die Verwaltung von Speicherblöcken funktioniert ebenso einfach, nur befindet sich der erste Speicherblock am Ende des Arrays und umfasst mehrere Worte. Weitere Blöcke liegen direkt davor.

Der „extrem einfache, aber elegante Trick" der Verwaltung freier Speicherzellen in einer verlinkten Liste war zur Einführung der CDC 6600 bereits bekannt [Newell, 1964]. Die hier implementierte Speicherverwaltung ist somit durchaus „zeitgemäß." Für modernere Systeme

bzw. Programmiersprachen ist dagegen eine effiziente *Garbage Collection* obligatorisch, die vor dem Nutzer verborgen wird.

Speicherverwaltung mit ECS Die wesentliche Erweiterung besteht in der Nutzung von weiteren Arrays, die im ECS abgelegt werden. Da darauf nicht direkt zugegriffen werden kann, müssen Teile davon jeweils in das eigentliche Array „eingeblendet“ werden. Dazu stellt die FORTRAN-Implementierung die zwei intrinsischen Funktionen **`READEC`** und **`WRITEC`** zur Verfügung. Mit dem Kopieren der Daten sind separate Prozessoren beschäftigt. Diese müssen zuvor zeitintensiv initialisiert werden, was sich nur für größere Speicherbereiche lohnt. In kleineren Tests zeigte sich, dass etwa 1000 Speicherworte eine vernünftige Größe für eine solche „Page“ darstellen. Damit insbesondere Speicherblöcke nicht auf verschiedenen Speicherseiten verteilt sind, wird eine Größe von 1008 Worten genutzt, die 56 Speicherblöcke aufnehmen kann. Im ECS können Arrays ebenfalls maximal 2^{17} Worte umfassen, allerdings müssen sie sich den Platz nicht mehr teilen. Damit eine gerade Anzahl an Speicherseiten ablegbar ist, umfassen die Arrays exakt 131040 Worte. Drei solcher Arrays werden in der Implementierung genutzt, maximal 8 wären aufgrund der intern zur Adressierung genutzten 20 Bit sinnvoll. Somit kann die Implementierung knapp 8 der maximal verfügbaren 14 MB nutzen.

Die virtuellen Seiten werden in der Mitte des im Hauptspeicher liegenden Arrays eingeblendet (siehe Abb. 6.4). Dadurch können UVC-Anwendungen, die nur die Hälfte des Speichers für Register oder bitweise adressierten Speicher nutzen, ohne zeitraubenden Wechsel von Speicherseiten ausgeführt werden. Es soll aber nicht verschwiegen werden, dass sich mit der Virtualisierung des Speichers die Zugriffszeiten spürbar verlängern. Statt des direkten Zugriffs müssen jetzt zwei Unterprogramme (**`GETVAL`** und **`SETVAL`**) bemüht werden, die den Zugriff steuern und ggfs. Speicherseiten austauschen. Letzteres fällt aufgrund der separaten Prozessoren aber kaum ins Gewicht.

Realisierung dynamischer Listen

Dynamische Listen sind die Grundlage der dynamischen Speicherverwaltung. Alle genutzten Datenstrukturen – mit Ausnahme der Speicherblöcke – basieren darauf, um eine effiziente Speichernutzung weitgehend ohne Fragmentierung zu erreichen.

60 Bit große Speicherworte sind eigentlich zu umfangreich für den UVC, der sich in vielen Fällen auf 32 Bit beschränkt. Zudem ist die Ganzzahlarithmetik der Basismaschine nicht in der Lage, mit den vollen 60-Bit-Worten umzugehen. Daher werden die ersten 20 Bit eines Speicherwortes genutzt, um die Adresse, also den Index des nächsten Speicherwortes zu speichern. Für den in der Liste aufzunehmenden Wert bleiben dabei 40 Bit. Eine Liste ist etwas, das erzeugt und an Unterprogramme übergeben werden muss, um dort manipuliert zu werden. Es bedarf daher eines speziellen Kopfelements, das erhalten bleibt, solange die Liste – auch die leere – gebraucht wird.

Kopfelemente einer Liste enthalten neben dem Index des nächsten Elements auch den Index auf das letzte Element, um ein effizientes Anfügen zu ermöglichen. Die vorangegangene Implementierung in Ada hat gezeigt, dass bei den arithmetischen Operationen oft Elemente an Listen angefügt werden müssen. Die verbleibenden 20 Bit enthalten zudem die Länge dieser

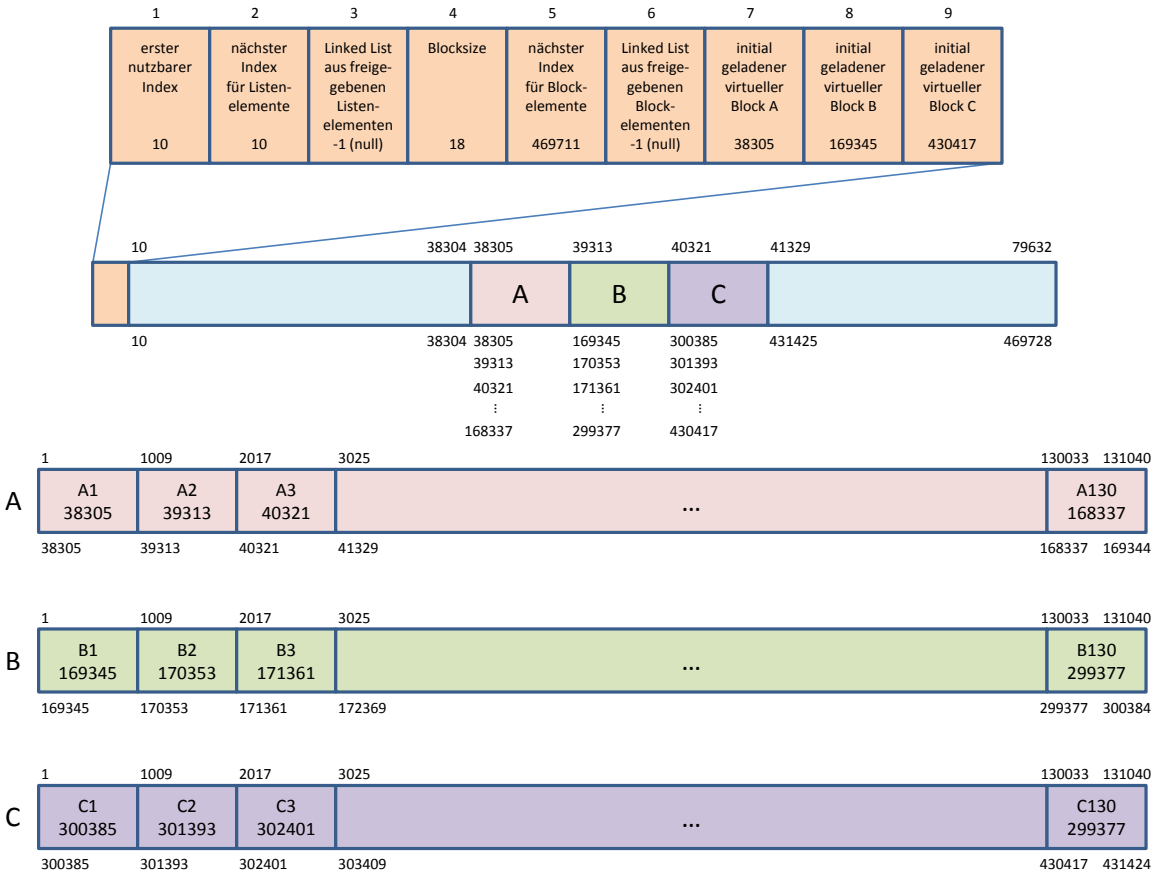


Abbildung 6.4.: Organisation des virtuellen Speichers mit Hilfe weiterer Arrays im ECS

Liste, die sich somit ohne Iteration bestimmen lässt, aber auch (z.B. bei den arithmetischen Operationen) gepflegt werden muss. Die Abbildung 6.5 zeigt den schematischen Aufbau.

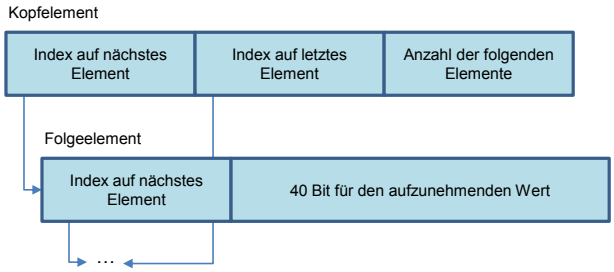


Abbildung 6.5.: Schematischer Aufbau dynamischer Listen mit Kopfelement

Verwendete „Datentypen“

Auch wenn es so etwas wie selbstdefinierte Datentypen in FORTRAN IV nicht gibt, werden Strukturen gebraucht, die ihrerseits dynamisch erzeugt, angepasst und gelöscht werden

können. Die Kenntnis dieser Strukturen ist überaus hilfreich beim Lesen des Quellcodes, der aufgrund der – aus heutiger Sicht – rückständigen Kommentierfunktionen nicht selbst erklärend sein kann. Sie werden daher im Folgenden vorgestellt.

Register Wie alle Strukturen dieser Implementierung sind auch Register Listen. Ein Register hat immer ein Vorzeichen und kennt seine Länge, die sich gemäß Spezifikation nicht aus dem gespeicherten Bitstrom ableitet und daher separat gespeichert werden muss. Diese Information wird im ersten Folgeelement einer Liste abgelegt. Eventuell folgende Elemente beinhalten dann die Bits, die den Inhalt eines Registers ausmachen. Dabei werden sie zu je 40 Bit portioniert und rechtsbündig in aufsteigender Reihenfolge in der Liste abgelegt. Die Iteration zum höchstwertigen Bit hin erwies sich bereits bei der Ada-Implementierung (siehe Kapitel 5) als zweckmäßig. Ein Register belegt somit immer mindestens zwei Speicherworte; Register, die nicht mehr als 40 Bit aufnehmen, belegen drei. Maximal können Register damit 2^{20} Elemente zu je 40 Bit umfassen, was einen Umfang von über 5 MB pro Register bedeutet. Eine Multiplikation derart großer Register ist weder auf einer kompatiblen Maschine „abwartbar“ noch vom Speicher her realisierbar.

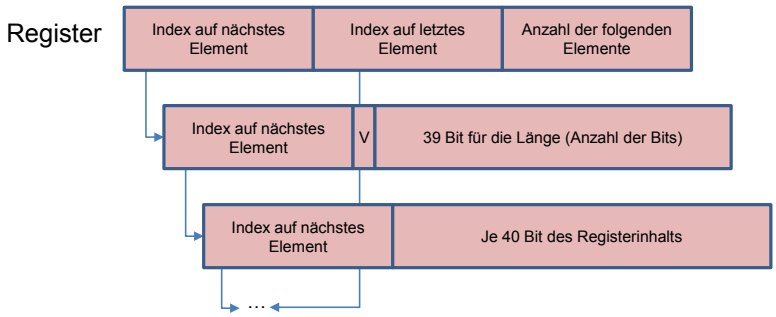


Abbildung 6.6.: Schematischer Aufbau der Register

Segmente Jedes Segment wird über eine Liste mit drei Einträgen abgebildet. Zu einem Segment gehören eine physische Segmentnummer, unbegrenzt viele Register und unbegrenzt viele Speicherblöcke. Die physische Segmentnummer wird direkt gespeichert und wird nur zum Löschen bestimmter Segmente gebraucht, wenn die logische von der physischen Segmentnummer abweicht.[4]

Der zweite Eintrag enthält den Index, ab dem die Liste der Register gespeichert ist. Zu jedem Register, das zum Segment gehört, gibt es zwei aufeinanderfolgende Einträge in dieser Liste. Der erste enthält immer die Registernummer, der zweite den Index, ab dem die Liste gespeichert ist, die das Register abbildet.

Der dritte und letzte Eintrag der Segmentliste enthält den Index einer Liste, die zu allen genutzten Speicherblöcken jeweils den ersten Index gespeichert hat.

[4]Diese Implementierung entstand auf Grundlage der ersten Ada-Implementierung, die noch die suggerierten physischen Segmentnummern nutzt. Folglich ist diese Implementierung ein Beleg für die mögliche Irreführung durch die Spezifikation, die sich sowohl in der Laufzeit als auch in der Implementierungszeit deutlich bemerkbar macht.

Segmente selbst werden auch in einer Liste verwaltet. Für jedes Segment werden zwei aufeinanderfolgende Einträge genutzt. Der erste enthält die physische Segmentnummer, der zweite den Index der Segmentliste. Die vollständige Struktur ist in Abbildung 6.7 illustriert.

Eine Verwaltung mittels balancierter Bäume wird nicht umgesetzt, weil der in der Ada-Implementierung verwendete B-Baum 1964 noch nicht erfunden war und der von Adelson-Velsky und Landis [1962] erfundene AVL-Baum wohl erst lange nach der Beschreibung Newells eleganten Tricks in einer Hochsprache umgesetzt worden wäre. Davon zeugt auch die „späte“ Einführung von Zeigern in FORTRAN 77 [Cra, 1986]. Ein weiterer Grund sind die in einem Praktikum gesammelten Erfahrungen. Dabei wurde die Karatsuba-Multiplikation (siehe Abschnitt 5.3.2) basierend auf dynamischen Strukturen umgesetzt. Aufgrund der komplexen Verwaltung des Speichers erwies sich diese Implementierung für „erwartbare“ Zahlengrößen stets langsamer als die einfache Multiplikation nach der Schulmethode. Diese Implementierung besitzt noch Optimierungspotential, lässt aber erahnen, dass die zu erwartenden Laufzeitvorteile den erforderlichen Aufwand nicht rechtfertigen werden.

Unterprogramme Ein Unterprogramm (Section) ist ein logischer Teil des Programmcodes. Zumeist besteht eine UVC-Anwendung aus mehreren Sections. Die jeweils erste dient als Programmeinstieg. In dieser Implementierung sind Sections spezifikationskonform im Speicher von Segmenten abgelegt. Diese Segmente werden in einer separaten Liste verwaltet und enthalten keine Register. Diese Liste ist nach dem Laden des Programms fest und keinen weiteren Veränderungen unterworfen. Selbstmodifizierender Code ist aufgrund des fehlenden Zugriffs der UVC-Anwendung zudem nicht möglich.[5]

Prozessor Beim Aufruf von Unterprogrammen muss der Zustand des Prozessors gesichert werden, um den Programmfluss nach der Rückkehr fortzusetzen. Viele reale Maschinen nutzen dazu einen reservierten Speicherbereich. Bei der Implementierung eines UVC muss dagegen eine interne Datenstruktur geschaffen werden, die einer UVC-Anwendung verborgen bleibt. Was exakt gesichert werden muss, ist nicht spezifiziert (siehe Abschnitt 5.3.4). Die Adresse des nach Rückkehr aktuellen Befehls zusammen mit der ID der Section gehört in jedem Fall dazu. Ebenfalls unverzichtbar sind die Zuordnungen der logischen Segmentnummern zu den physischen in den Fällen der Nutzung privater Segmente. Das Parametersegment wird erst mit dem Aufruf einer Section als Segment 2 eingeblendet. Diese spezielle Zuordnung muss ebenfalls gesichert werden.

Praktisch ist es, das jeweilige Segment 1 im Keller zu speichern. Dieses Segment ist zwar für eine bestimmte Section immer gleich und könnte zusammen mit dieser gespeichert werden, jedoch würde das eine weitere Datenstruktur und zusätzlichen Code verursachen. Für den schnelleren Kontextwechsel nach einem Rücksprung ist es zudem sinnvoll, neben der ID der Section auch den zugehörigen Index mit zu sichern.

Der Keller selbst wird als Liste realisiert. Die Bestandteile des Prozessorzustands werden in

[5] Hier finden sich noch Optimierungsmöglichkeiten. Z.B. könnten die Bitfolgen der Sections im ersten oder letzten Teil des Speicherarrays geladen werden – unter Verschieben des Starts bzw. Endes für die dynamische Zuweisung von Indizes. Programme müssten jedoch ohne „Paging“ im Speicher abgelegt werden und dürften einen Umfang von ca. 300 kB nicht überschreiten.

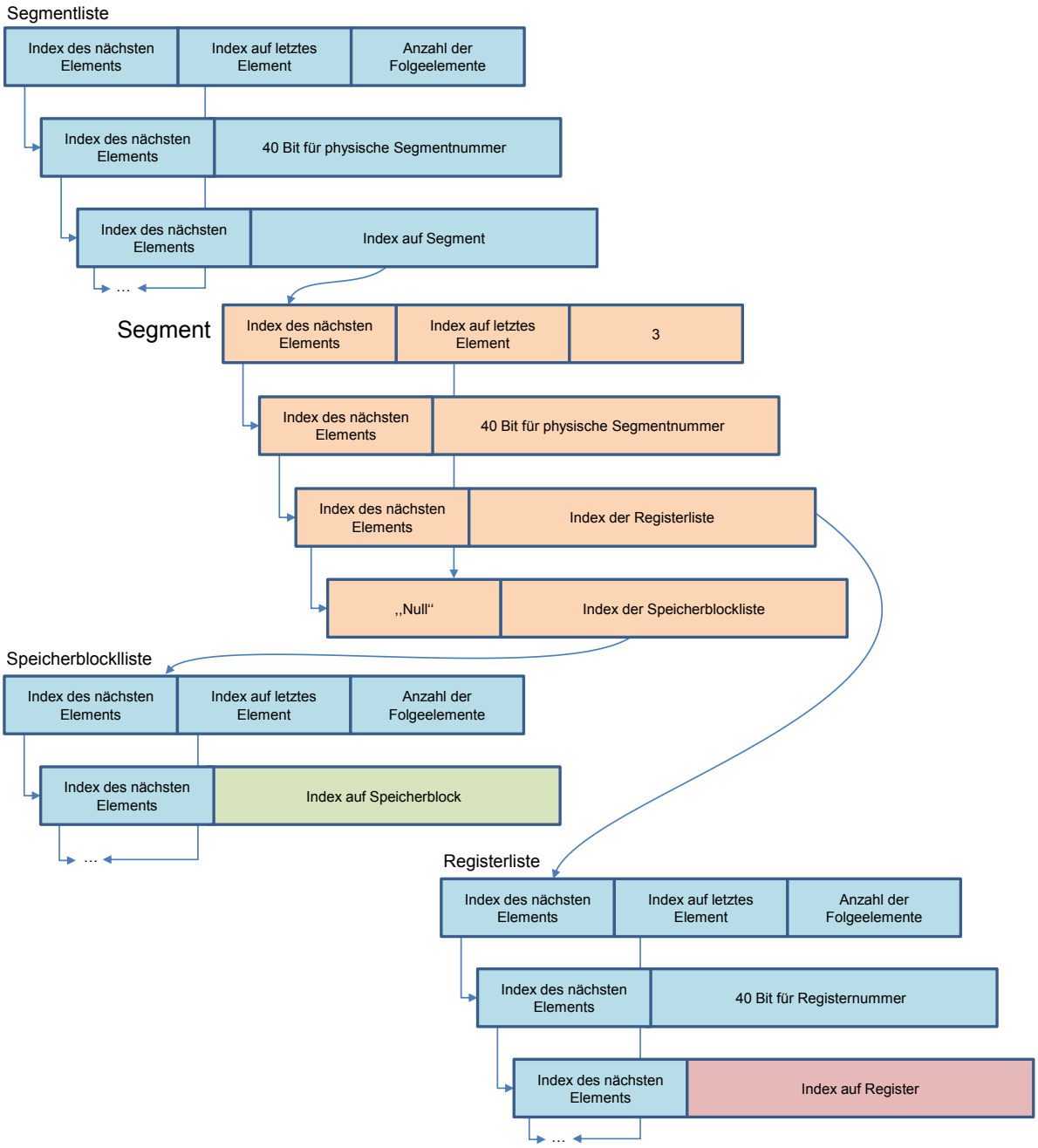


Abbildung 6.7.: Schematischer Aufbau der Segmente

fester Reihenfolge jeweils vorn in diese Liste eingefügt bzw. wieder entfernt. Die Implementierung der Liste realisiert hierzu die Unterprogramme **INSLIST** und **POPLIST**. Die Reihenfolge der Kellereinträge ist in der FORTRAN-Implementierung direkt ersichtlich.

Optimierungen wie sie in der Ada-Implementierung vorgestellt wurden, finden hier noch keinen Eingang. Sie sind 1964 ebenso wenig „zeitgemäß“ wie es balancierte Bäume sind.

Einfluss der Wortbreite auf Algorithmen

Wie zuvor beschrieben ist die Integerarithmetik der CDC 6600 sowie der verwendeten Cyber 960 im Cyber-170-Modus eingeschränkt. Zwar gibt es einen Addierer, der die vollen 60 Bit verarbeiten kann, aber die Multiplikation und die Division stützt sich auf die Fließkommaarithmetik ab, wodurch nur 48 Bit insgesamt nutzbar sind. Aufgrund der genutzten Speichergröße werden 20 Bit eines Speicherwortes zum Speichern eines Indizes genutzt. Die verbleibenden 40 Bit können Registerinhalte aufnehmen. Das ist im Falle der Addition eine handliche Größe. Für die Division und Multiplikation gilt, dass sowohl Operanden als auch das Ergebnis die 48-Bit-Grenze nicht überschreiten dürfen [For, 1983]. Für die implementierte Multiplikation nach der Schulmethode (siehe Abschnitt 5.3.2) wurde daher intern eine auf 20 Bit basierende und 40 Bit umfassende Multiplikation implementiert. Die Division hingegen basiert auf einer alternierenden Schleife, die abwechselnd die vorderen und anschließend die folgenden 20 Bit verarbeitet und so in jedem Schritt nur 40 durch je 20 Bit dividiert.

Die vertrauten Algorithmen konnten somit genutzt werden, auch wenn die Implementierungen, insbesondere die der Division, zugunsten der Laufzeiten viel umfangreicher geworden sind. Die Multiplikation nach Karatsuba und Ofman [1963] wurde im Rahmen eines Praktikums separat implementiert. Sie ist zwar „zeitgemäß“ aber leider mit FORTRAN IV ebenso wenig effizient umsetzbar wie balancierte Baumstrukturen. Die Algorithmen mussten zunächst entrekursiviert werden, da FORTRAN IV keine Rekursion kennt. Zudem verschlingt die dynamische Speicherverwaltung viel der zu gewinnenden Laufzeit durch die vielen Zwischenprodukte, die temporär zu speichern sind. Der *break even point* liegt damit weit oberhalb sinnvoll nutzbarer Registerlängen. Eine solche Implementierung enthält die FORTRAN IV-Implementierung des UVC daher nicht.

Die implementierte Division nach Knuth [1969] ist genau genommen nicht „zeitgemäß.“ Die verfügbare Alternative ist die in der Schule vermittelte Division. Die Herausforderung bei dieser Methode ist das Finden des jeweiligen Teilergebnisses, mit dem der Divisor jeweils multipliziert und vom Dividenden abgezogen werden muss. Ist dieser zu groß gewählt, wird der verbleibende Dividend negativ und der Divisor muss entsprechend oft „zurück“ addiert werden. Knuths Algorithmus ist deshalb effizienter, weil er das Teilergebnis mit einer wesentlich höheren Wahrscheinlichkeit korrekt bestimmt und das nachträgliche Korrigieren fast eliminiert. Eine Einarbeitung dieser Erweiterung ist aber sehr leicht möglich und wäre wohl auch 1969 nach Knuths Veröffentlichung sofort geschehen.

Anders dagegen verhält es sich mit dem Algorithmus nach der Indischen Methode, die von Parthasarathi und Jhunjhunwala [1995] beschrieben wurde. Diese ist zwar nicht so effizient wie die von Knuth, aber speziell für kleinere Systeme gedacht, die wenig Platz für die fortwährend anfallenden großen Zwischenergebnisse haben. Da diese Methode ohne intensive Nutzung des dynamischen Speichers auskommt, könnte sie im Falle der FORTRAN IV Implementierung Laufzeitvorteile aufweisen. Sie war jedoch 1964 noch nicht publiziert.

Einfluss der verfügbaren Ein- und Ausgabe

Der Einfluss der verfügbaren Ein- und Ausgabe spielt immer eine Rolle sobald es um Portierung von Software geht. Bekannte Probleme entstehen z.B. durch veränderte Eigenschaften

von Anzeigegeräten oder durch Nutzung anderer Eingabegräte wie Joystick oder Maus.

Der UVC hingegen versucht sich solchen Einflüssen zu entziehen, indem er lediglich zwei Mittel hat, um mit seinem Umfeld zu kommunizieren: die Instruktionen `IN` und `OUT`. Und obwohl die Spezifikation an dieser Stelle viel Freiraum lässt, ist eine Abhängigkeit bereits spürbar. Die beiden Instruktionen werden mittels `READ` und `WRITE` bzw. `PRINT` umgesetzt. Zwar können mit den sehr umfangreichen Formatierungsmöglichkeiten nahezu alle Ergebnisse erzielt werden, jedoch sind dazu vom Compiler umfangreiche Prozeduren anzusteuern, die sich negativ auf die Laufzeit auswirken würden. Zugunsten der Laufzeit wird daher ein Ausgabeformat gewählt, das alle spezifizierten Informationen enthält, aber nicht identisch ist mit dem der Ada-Implementierung. Ebenso weicht das Eingabeformat ab. Anwendungsprogramme, die den UVC ansteuern, müssen das berücksichtigen. Die Anpassungen sind unkritisch, die Notwendigkeit soll aber nicht verschwiegen werden. Es ist sehr wahrscheinlich, dass jede den UVC nutzende Generation ihr eigenes zeitgemäßes Format nutzen wird. Wichtig ist, dass die Bitfolgen unverändert im UVC ankommen. Ob sie dabei gestückelt, hexadezimal codiert und in einzelnen Zeichenketten übertragen werden, spielt keine Rolle.

Eine Sprachabhängigkeit führt schließlich zur Notwendigkeit von der Spezifikation abzuweichen. Diese schreibt Binärdateien zur Speicherung von UVC-Applikationen vor. Die Sprache FORTRAN unterstützt bis zum Standard 2003 ausschließlich Record-basierte Dateizugriffe. Ohne das Einbetten von in anderen Sprachen geschriebener Zugriffsroutinen ist eine spezifikationskonforme Implementierung nicht möglich. Bei dieser Implementierung wurde eine andere Lösung gefunden. Programme werden in fortlaufenden 60 Bit Worten einfach per `READ` eingelesen. Das hat auch den Vorteil, dass UVC-Programme sehr einfach und flexibel per *Copy and Paste* über die Konsolenanwendung übertragen werden können, ebenso die zu verarbeitenden Daten.

6.2.4. Erkenntnisse

Die größten Hürden bei der Implementierung entstanden durch den nur unflexibel zur Verfügung gestellten Speicher. Hierdurch musste zunächst eine eigene Speicherverwaltung implementiert werden. Die gefundene Lösung ist einfach und zeitgemäß zugleich, hält aber von der Nutzung effizienter Strukturen und Algorithmen ab. Hier könnte sicher durch viel Arbeit und dem Einsatz von moderneren Algorithmen noch der ein oder andere Laufzeitgewinn erzielt werden, jedoch würde das die Ergebnisse enorm verzerren. Eine perfekt optimierte Implementierung benötigt viel Entwicklungszeit und steht schnell nicht mehr im Verhältnis zum erzielten Laufzeitgewinn.

Die Implementierung hat insgesamt sechs Wochen in Anspruch genommen. Nicht mit eingerechnet ist die Einarbeitung in das Betriebssystem und das Sammeln erster Erfahrungen mit der Sprache FORTRAN IV, da sich das im Rahmen eines Praktikums vereinzelt über mehrere Wochen erstreckte. Die Implementierung selbst als auch die vorangegangene Entwicklungsphase, in der die „Pakete" und deren Funktionalität geplant wurden, war innerhalb dieser sechs Wochen am Stück abgeschlossen.

Die erkannten Abhängigkeiten sind nahezu alle zu vernachlässigen. Dass die Geschwindigkeit von der Hardware abhängt oder dass die Algorithmen zur Umsetzung der Arithmetik auf die verfügbaren Bits maßgeschneidert werden müssen, ist für alle Maschinen gleich. Anders

wäre es, wenn der UVC eine feste Wortbreite nutzen würde. Dann wären Maschinen mit gleicher Wortbreite deutlich schneller mit einer UVC Implementierung versorgt. Die Abbildung der spezifizierten Unbegrenztheit ist jedoch so immer eine gleichbleibende Herausforderung. Der Umfang der Entwicklungszeit ist dem der Ada-Implementierung nahezu gleich.

Dass die Spezifikation UVC-Applikationen in Form einer Binärdatei vorschreibt, wäre nach der Erfahrung hier wohl 1964 nicht passiert. Das Format wäre als eine Record-basierte Struktur spezifiziert worden, mit der Konsequenz, dass heutige Implementierungen damit zu kämpfen hätten. Daher ist auch die Frage berechtigt, ob die archivierten UVC-Applikationen im Laufe der Jahre tatsächlich keinen Veränderungen unterliegen werden. Der Aufwand der hier notwendigen Migration ist noch nicht alarmierend, kann aber ein Indiz dafür sein, dass im Laufe der Archivierung mit einem geringen zusätzlichen Aufwand gerechnet werden muss. In jedem Fall müssen aufgrund des stets aufzufüllenden letzten Speicherwortes die Füllbits spezifiziert werden, damit Klarheit bei Migrationsvorgängen herrscht.

6.3. Portierung für CDC 4680-MP

Die Idee der Nutzung des UVC zur Langzeitarchivierung von digitalen Objekten stützt sich auf zwei arbeitsintensive Phasen, um einen möglichst geringen Aufwand während der Archivierung selbst zu erreichen. Die Phase der Einstellung eines digitalen Objekts in das Archiv soll hierbei absichtlich so vollständig sein, dass die letzte Phase, also die des Objektzugriffs durch die Nutzer, so einfach wie nur immer möglich ist. Für diese Phase sind zwingend der UVC und geeignete, aber äußerst einfach zu erstellende Wiedergabeprogramme notwenig. Da die Implementierung eines UVC deutlich aufwendiger eingeschätzt wird, liegt die Hoffnung nicht zu Unrecht auf Softwareportierungen. Software, die für ein bestimmtes System erstellt wurde und zu der noch Quellcode vollständig erhalten ist, kann für Nachfolgesysteme ebenfalls kompiliert und ausgeführt werden, wenn entsprechende Compiler vorhanden sind. Die größten Probleme entstehen dabei durch sich weiterentwickelnde Compiler, die nicht immer vollständig abwärtskompatibel sind, aber auch durch Teile des Quellcodes, die sehr hardwarespezifisch gestaltet sind. In solchen Fällen ist mit einem erheblichen Aufwand zu rechnen, wie auch dieser und der folgende Abschnitt zeigen werden.

Zunächst wird die Maschine kurz vorgestellt und dann die wenigen notwendigen Schritte erläutert, die zur Anpassung des FORTRAN-Codes notwendig waren.

6.3.1. Details zur Maschine

Die verwendete CDC 4680-MP wurde im Jahr 1990 gebaut, verwendete aber nicht mehr ausschließlich Komponenten „made by CDC“. So wurden die zwei verbauten Hauptprozessoren (R6000) von MIPS Computer Systems entwickelt. Obwohl der Prozessor laut verschiedenen Quellen mit 80 Mhz betrieben werden könnte, sind es in der hier genutzten Info-Server-Variante nur 60 MHz [CDC, 1990]. Zum Chipset gehört auch ein Coprozessor für die Fließkommazahlarithmetik, die hier aber keine Rolle spielt. Auch unterblieb bisher der Versuch Ganzzahlarithmetik auf Fließkommaarithmetik abzubilden, um die Vorteile eines vorhandenen Coprozessors auszunutzen.

Zur Implementierung der Arithmetik mehrfacher Genauigkeit bietet die Hardware spezielle Befehle. So kann sie zwei volle 32-Bit-Register multiplizieren (`multu`) und das Ergebnis in zwei speziellen 32-Bit-Registern ablegen, deren Inhalt sich mit zwei weiteren Maschinen-Anweisungen in normale Register kopieren lässt [CDC, 1993a]. Auch für die Division stehen Befehle dieser Art zur Verfügung.

Die CDC 4680 war im Vergleich zu ihren Zeitgenossen recht erschwinglich, bei ansehnlicher Rechenleistung.[6] Zudem waren die umfangreichen Anschlussmöglichkeiten für Peripherie ausschlaggebender Grund. Sie wurde daher in den Jahren von 1992 bis 1996 überwiegend in Rechenzentren als Server für Archivierungsaufgaben verwendet.[7]

Es findet sich zwar keine dokumentierte Ablösung einer Cyber 960 durch eine CDC 4680, wohl aber sind mehrere Indizien zu finden, die ein solches Szenario wahrscheinlich machten. Zum einen ist die CDC 4680 etwa dreimal schneller als die Cyber, wenn man nur die Anzahl der Fließkommazahl-Operationen pro Sekunde (FLOPS) betrachtet [CDC, 1990] und stammt zudem vom selben Hersteller. Zum anderen existiert das Werkzeug `f477`, das die Portierung von FORTRAN IV-Code, speziell der Hersteller Cray und CDC, unterstützt.

Die Ergebnisse der hier durchgeführten Portierung haben daher enorme Aussagekraft bezüglich Aufwand und Erfolg.

6.3.2. Details zu FORTRAN 77

FORTRAN 77 bietet mit der aktuell installierten Version neben dem Datentyp `INTEGER*4` auch den 64 Bit breiten `INTEGER*8`. Hiermit lässt sich mehrfache Genauigkeit auf Hochsprachenebene realisieren – auch wenn damit den Möglichkeiten der Basismaschine nicht Rechnung getragen werden kann. Um zwei vorzeichenlose 32-Bit-Zahlen zu multiplizieren, muss bereits eine 64 Bit umfassende Variable genutzt werden. Entsprechende Anweisungen auf Hardwareebene kennt die vorhandene MIPS-II-Architektur noch nicht. Folglich muss diese Arithmetik über komplexere Funktionen abgebildet werden, was Rechenzeit kostet. Die Möglichkeit, einzelne Programmteile in C oder Assembler zu implementieren, wird aber nicht genutzt, um dem Konzept der Portierung nicht entgegenzuwirken.

6.3.3. Details zum Portierungsaufwand

Wie bereits eingangs erwähnt gibt es ein Werkzeug, das speziell für Umsteiger der Cyber 170 Modelle gedacht war. Die Darstellung des Ergebnises der Arbeit mit diesem wird dem Einstieg dienen, um schließlich den tatsächlichen manuellen Portierungsaufwand darzustellen.

[6] Die CDC 4680 war als Supercomputer, wenn auch nicht als schnellster seiner Zeit, schon für 400.000 DM zu haben. www.computerwoche.de/heftarchiv/1990/12/1144847 (23.05.2011)

[7] Darunter das LRZ, www.lrz.de/wir/geschichte,
das RRZE, www.computerwoche.de/heftarchiv/1993/27/1128758/ und das HRZ,
www.uni-giessen.de/hrz/service/veroeff/zeitschrift/LOGIN-93-1/6.txt
(23.05.2011).

Vorhandenes Portierungswerkzeug

Für das vorhandene Werkzeug `f477` gibt es keine Dokumentation in Papierform, wohl aber eine für Unix-ähnliche Systeme übliche *man-page*.[8] Dieses Manual stellt sehr detailliert dar, wie das Werkzeug funktioniert und wo die Grenzen sind. Es werden im Wesentlichen die Syntax angepasst und darüber hinaus sämtliche Stellen markiert, die Probleme verursachen könnten. Die Syntax hat sich im Bereich der *Specification Statements* verändert. So gibt es Erweiterungen, die hier nicht automatisiert eingearbeitet werden können, aber auch kleinere Änderungen, z.B. am `PROGRAM`-Statement, die einfach durch Entfernen umgesetzt werden. Es ist mit FORTRAN 77 nicht mehr möglich, direkt I/O-Kanäle festzulegen. Im Falle der zu portierenden UVC-Implementierung spielten diese Änderungen keine Rolle.

Deutlicher fallen Veränderungen der heute unüblichen 60 Bit Worte ins Gewicht. Alle Stellen im Programm, die von der speziellen 60-Bit-Arithmetik abhängig sein könnten, werden von diesem Werkzeug entsprechend markiert: `MANUAL CHANGE REQUIRED`. Hilfreiche Hinweise finden sich dagegen keine. Das betrifft alle intrinsischen Funktionen zur bitweisen Manipulation und somit nahezu 25% des Codes.

Zur besseren Lesbarkeit werden oktal codierte Konstanten (hexadezimale Konstanten wurden nicht unterstützt) automatisiert konvertiert; aber nur für 32 Bit breite Variablen. Daher findet sich an fast jeder zweiten Zeile im Programm die Anmerkung: `OCTAL CONSTANT TOO LONG; MANUAL CONVERSION REQUIRED`. Mit dieser ganz offensichtlich fehlenden Unterstützung des verfügbaren 64 Bit breiten Ganzzahltyps wird dieses Werkzeug vollends unbrauchbar. Das erzeugte Ergebnis eignet sich daher nur zum schnellen Erlernen der neuen Syntax.

Manuelle Portierung

Nachdem gezeigt wurde, dass diese Portierung nicht automatisiert möglich war, wird der Aufwand dargestellt, der tatsächlich manuell erbracht werden musste. Zunächst wurde das Programm ohne jegliche Kenntnisse des Codes portiert. Da hier der gleiche Programmierer arbeitete, mag das komisch wirken, meint aber nichts anderes als das Belassen der Programmlogik.

So wurde zunächst nur die Syntax angepasst. Dazu gehörten das `PROGRAM`-Statement und weitere Statements, die für das hierarchische Speichermanagement der Cyber notwendig sind und jetzt keine Verwendung mehr finden. Aus `INTEGER` wurde `INTEGER*8`. Zuvor verwendete oktal codierte Ausdrücke wie `3777777B` wurden zu `O"3777777_8` konvertiert. Alle anderen Konstanten erhielten ebenfalls die 64-Bit-Markierung. Dem gingen einige Tests voraus, da die dokumentierte Syntax nicht vollends zum verfügbaren Compiler passt. Für die verwendeten intrinsischen Funktionen wie `AND`, `OR` und `COMPL` finden sich Entsprechungen, die aber zum Teil nicht in der Anleitung stehen. Weitere intensivere Tests folgten, um adäquate Gegenstücke zu finden. Aufwendig ist das Umsetzen der Schiebeoperation `SHIFT`. So wurde an mehreren Stellen ausgenutzt, dass der Linksshift zirkular ist, also die Bits, die links herausgeschoben werden, sich rechts wiederfinden. Bei der Verwendung von 64 statt 60 Bits sind diese potentiellen Stellen zu finden und anzupassen. Tatsächlich sind es nur vier.

Ein etwas aufwendigeres Problem entstand bei den Programmstellen, die das schnelle blockweise Kopieren nutzen, das die Cyber seitens der Hardware unterstützt. Alle Vorkommen der

[8] CYBER 170 FORTRAN 4 to f77 translator, FORTRAN migration aids, Version 1.1, 1991

Funktionen **READEC** und **WRITEC** finden sich recht konzentriert im Code und wurden durch einfache Schleifen ersetzt.

6.3.4. Ergebnisse der Portierung

All diese Anpassungen am Code sind von Programmierern mit Erfahrung im Bereich der Portierung von FORTRAN IV nach FORTRAN 77 zu leisten – ohne das Programm an sich zu kennen, insbesondere ohne Kenntnis der Spezifikation des UVC. Das ist zum einen der Beleg für eine kostengünstige Portierung eines Programms innerhalb einer Sprache in kürzester Zeit. Zum anderen zeigen aber die Zeiten der Programmausführung deutlich das verschwendete Potential der Basismaschine.

Die Cyber 960 braucht zum Decodieren eines 16x16 Pixel großen JPEG-Bildes 28 Minuten. Der hier portierte UVC benötigt 74 Minuten ohne Optimierung. Eine Optimierung kann der Compiler im Anschluss der Codeübersetzung durchführen, um den Programmfluss zu optimieren und unnötige Anweisungen zu entfernen. Leider führten die genutzten Schleifen zur Umsetzung des blockweise Kopierens zu Fehlern bei der Optimierung, was erst auffiel als das „Paging“ (siehe Abschnitt 6.2.3) genutzt wurde, also erst bei größeren Anwendungsfällen. Gefunden wurde die Ursache erst nach drei vollen Tagen intensiver Suche. Abhilfe schaffte nur das Entfernen der Algorithmen zur seitenweisen Speichernutzung. Das erforderte jedoch Kenntnisse der Programmlogik. Hierdurch konnte der Code an sich optimiert werden, da Zugriffe auf das Array direkt möglich wurden. Die Version entspricht der Implementierung ohne ECS, die auf der Cyber das gleiche Bild in nur 14 Minuten decodiert. Durch die zusätzliche Optimierung durch den Compiler decodiert die Implementierung für die CDC 4680 das Bild in „nur“ 29 Minuten.

Der Aufwand der Portierung war auf zwei Tagen verteilt gut zu bewältigen. Ohne Kenntnisse der Spezifikation und der inneren Programmlogik ist eine Portierung möglich, kann aber, wie in diesem Fall, zu unerwarteten und nur äußerst schwer auffindbaren Fehlern führen. Auch die Geschwindigkeit der Programmausführung lässt sehr zu wünschen übrig. Hier stehen 28 Minuten den 74 Minuten gegenüber bzw. 14 gegenüber 29 Minuten, obwohl allein die Hardware eine dreifache Beschleunigung versprach. Auch die Speichernutzung hat sich nicht verändert. Die fest eingearbeitete 4 MB-Speichergrenze lässt sich ohne Kenntnisse der inneren Programmlogik nicht variabel gestalten.

Es bleibt festzuhalten, dass sich eine Portierung nur lohnt, wenn es darum geht, kostengünstig allein die Funktionalität des UVC zu bewahren. Wenn darüber hinaus mit der neuen Hardwaregeneration auch Laufzeitgewinne erzielt werden sollen, sind Portierungen nicht empfehlenswert. Nachträgliche Anpassungen und Optimierungen nehmen schnell so viel Zeit in Anspruch, dass sich eine Reimplementierung lohnt.

6.4. Reimplementierung für CDC 4680

Der vorhergehende Abschnitt motiviert nachhaltig eine Reimplementierung. Zum einen sind die durch Portierungen zu erwartenden Laufzeitgewinne ungenügend, sobald angepasst an die Hardware programmiert wurde, was beim UVC immer der Fall sein wird. Zum anderen

sind bestimmte Einschränkungen bzw. verwendete Algorithmen nicht mehr zeitgemäß und verdienen die Außerdienststellung, um nicht unnötig Rechenzeit und damit Ressourcen zu verschwenden.

Dieser Abschnitt beschreibt kurz die wesentlichen Punkte der Reimplementierung indem zunächst die genutzten Bestandteile und Erweiterungen der Sprache vorgestellt werden und im Anschluss auf die genutzten Strukturen und die Speicherverwaltung eingegangen wird. Die Programmlogik wurde mit dieser Version mehrfach aktualisiert. Während jedoch die Ada-Implementierung den Bitcode der UVC-Anwendungen zunächst in eine eigene Struktur überführt, basiert diese Implementierung auf dem Interpreteransatz. Die einzelnen zu interpretierenden Bits werden ebenso wie in der FORTRAN IV-Implementierung direkt aus dem Speicher eines Segments geladen. Die verteilte Segmentverwaltung, sowie weitere „elegante Tricks“ sind in dieser Implementierung eingeflossen. Sie kann daher zu Recht als Referenzimplementierung für den Interpreteransatz bezeichnet werden. Die entsprechenden Erkenntnisse und die Bewertung der jeweils erreichten Verbesserungen wurden an den Schluss gestellt.

6.4.1. FORTRAN 77 - Genutzte Erweiterungen

Mit der Einführung der Maschine unterstützte die mitgelieferte FORTRAN Version die relevanten Datentypen `Byte`, `Integer*2` und `Integer*4` [CDC, 1991]. Wohl aufgrund der vielen potenziellen Umsteiger wurde der Compiler auch um den Datentyp `Integer*8` erweitert [CDC, 1993b]. Neben dieser direkten Unterstützung von 64-Bit-Integer-Arithmetik bietet der vorhandene Compiler auch Verbunde zur Gestaltung komplexerer Datentypen, etwas unhandliche Zeiger und zudem dynamische Speicherallokierung mittels zweier intrinsischer Funktionen (`MALLOC` und `FREE`). All das gehörte noch nicht zum Sprachumfang von FORTRAN 77[9]. Bereits zur damaligen Zeit konnte jedoch nicht verkannt werden, dass es mit folgenden Versionen zum Standard wird. Mehrere Hersteller wie Cray, Sun oder SGI führten z.T. in Kooperation diese Erweiterungen zeitgleich ein.

Eindrucksvoll lässt sich auch der Einsatz der verfügbaren Blockstrukturen erkennen. In der Reimplementierung finden sich keine Sprungmarken und keine `GOTO`-Anweisungen mehr. Diese sind zwar nicht verantwortlich für langsamen oder gar veralteten Code, Blockstrukturen helfen aber, den Code leichter verständlich zu machen und so Fehler zu vermeiden.

Weder von der Sprache noch vom Compiler werden Rekursionen oder Case-Anweisungen unterstützt. Auch ist das Modul-Konzept erst mit FORTRAN 90 realisiert.

6.4.2. Verbunde und Zeiger als neues Gestaltungsmittel

Komplexe, also aus verschiedenen Datentypen zusammengesetzte Verbundtypen werden über die `STRUCTURE`-Anweisung definiert. Um diesen Strukturen Variablen zuzuweisen, wird das Schlüsselwort `RECORD` genutzt. Obwohl vom Namen her gleich, haben diese nichts mit der Ein- bzw. Ausgabe zu tun. Der folgende Code zeigt die Definition der Variablen `ln` vom Typ eines Listenknotens.

[9] ANSI X3.9-1978

```
STRUCTURE /ListNode/     ! Listenelement
   INTEGER*4 next        ! Zeiger auf nächstes Listenelement
   INTEGER*4 value       ! 32 Bit für den zu speichernden Wert
END STRUCTURE
RECORD /ListNode/ ln     ! Variable ln vom Typ ListNode
```

Variablen sind in FORTRAN 77 nur innerhalb eines Unterprogramms typisiert. Werden sie an andere Unterprogramme übergeben, wird darin zunächst erneut der Typ angegeben und die Variable entsprechend interpretiert. Das eröffnet viele Freiheiten, die helfen, Code und Laufzeit einzusparen. Nachteilig an diesen Freiheiten sind schwerer auffindbare Fehler im Programm und jede Menge doppelte Strukturdefinitionen, die den Quellcode wiederum aufblähen und die Konsistenz gefährden. Dieses Problem ist bereits im kleinen Maßstab von FORTRAN IV bekannt und wurde hier auf die Verbundtypen ausgedehnt.

Die genutzte Kombination aus `STRUCTURE` und `RECORD` gibt es so in folgenden Standards nicht. Stattdessen gibt es die einfachere `TYPE`-Anweisung. Ohne Verbundtypen können jedoch die ebenfalls nicht standardisierten Zeiger, die als *Cray Pointer* bekannt wurden, nicht effizient genutzt werden. Die Reimplementierung strebt die größtmögliche Ausnutzung aller Vorteile des vorhandenen Compilers an und nutzt daher diese sinnvollen Erweiterungen – auch wenn dies erneut Portierungsprobleme nach sich ziehen wird, da auch *Cray Pointer* nicht standardisiert wurden. Eine dynamische Speichernutzung ist im FORTRAN 77-Standard nicht vorgesehen [Smith, 1995].

Die bisherige Speicherverwaltung basierte auf einem starren Array und umfasste Code zur Zuweisung und Freigabe von Speicherzellen. Dieser Code ist jetzt überflüssig. In Kombination mit Zeigern können Speicherblöcke dynamisch allokiert und freigegeben werden. Dazu stehen die Funktion `MALLOC` und das Unterprogramm `FREE` zur Verfügung. Der folgende Code demonstriert den Umgang in aller Kürze:

```
POINTER (pl,ln)    ! Syntax der genutzten Cray Pointer
pl = MALLOC(8)     ! allokiere 8 Byte für die Struktur
ln.next = 0        ! setzt Folgeelement auf 'null'
ln.value = 6       ! setzt den Wert auf 6
CALL FREE(pl)      ! gibt den Speicher wieder frei
```

Über das Schlüsselwort `POINTER` wird zu einem bestehenden Verbund ein Zeigertyp erzeugt. Fortan gehören `pl` und `l` zusammen. Deutlich wird, dass hier hardwarenahes Verständnis erforderlich ist. So ist die Berechnung des zu allokierenden Speicherblocks Aufgabe des Programmierers, ebenso die Freigabe nach Benutzung. Ohne starres Array können sich mehrere UVC-Instanzen den verfügbaren Speicher der Basismaschine flexibel teilen. Besonders bei einer Mehrprozessormaschine, wie der genutzten CDC 4680, ist das vorteilhaft.

6.4.3. Genutzte Strukturen

Viele der aus der Implementierung für die Cyber bekannten Strukturen wurden unverändert übernommen. Unterschiede entstehen dort, wo aufgrund der Verbunde effizientere Strukturen möglich wurden, also der ein oder andere Verweis überflüssig wurde.

Die einfach verkettete Liste besteht nach wie vor aus einem Kopf- und einer wahlfreien Anzahl Folgeelemente. In dem als Programm-Code dargestellten Verbund werden mögliche

Freiheiten bereits erkennbar. *Cray Pointer* sind voll kompatibel zum Type `Integer*4`:

```
STRUCTURE /ListType/     ! Kopfelement
   INTEGER*4 next        ! Zeiger auf erstes Listenelement
   INTEGER*4 last        ! Zeiger auf letztes Listenelement
   INTEGER*4 length      ! Anzahl der Listenelemente
END STRUCTURE
```

Das Kopfelement der Register ist so gestaltet, dass jedes Register auch gleichzeitig eine Liste ist. Ein Register enthält gegenüber einer Liste zusätzlich ein Vorzeichen und die Anzahl der aktuell genutzten Bits. Wenn sich diese zusätzlichen Angaben hinter den anderen, auf gleiche Weise sortierten Einträgen befinden, sind Listenoperationen direkt auf Register anwendbar. Durch dieses übertragene Konzept der Vererbung lässt sich Code wiederverwenden.

```
STRUCTURE /Register/     ! Identisch mit der
   INTEGER*4 next        ! Struktur des
   INTEGER*4 last        ! Listenkopfelementes
   INTEGER*4 length      ! Jeweilige Länge
   INTEGER*4 sign        ! Vorzeichen als Integer
   INTEGER*4 bitlength   ! Anzahl der umfassenden Bits
END STRUCTURE
```

An dem einfachen Verbundtyp der Segmente lässt sich gut die Auswirkung der verteilten Segmentverwaltung beobachten. Lediglich zwei Bäume sind hier zu verwalten:

```
STRUCTURE /Segment/
   INTEGER*4 Registers   ! Zeiger auf Baumstruktur (Register)
   INTEGER*4 MemBlocks   ! Zeiger auf Baumstruktur (Speicherblöcke)
END STRUCTURE
```

Unterprogramme werden spezifikationskonform als Segmente realisiert. Ihr Bitcode wird in Speicherblöcken gespeichert. Die Register dieser Segmente bleiben jedoch ungenutzt. Die Freiheiten der Kompatibilität zum Typ `INTEGER*4` wird dahingehend genutzt, dass im Falle von Unterprogrammen im ersten Eintrag ein Zeiger auf das zum Unterprogramm gehörende Segment 1 abgelegt wird.

Aufgrund der verteilten Segmentverwaltung ist es nur noch notwendig, die *shared* Segmente zentral zu verwalten. Im Code sind sie als Array unter dem Namen `public` zu finden. Neben den Segmenten 1 und 2 sind diese Segmente daher sehr schnell zugreifbar. Es hat sich bewährt auch die Unterprogramme zentral zu verwalten, da mehrere Programmteile darauf zugreifen. Sie werden in einem balancierten Baum verwaltet.

```
STRUCTURE /Segments/
   INTEGER*4 public(1000)    ! Array für "shared" Segmente
   INTEGER*4 sections        ! Zeiger auf Baum (Unterprogramme)
END STRUCTURE
```

Die Struktur eines Speicherblocks ist trivial. Die Blockgröße von 1024 Worten wurde aufgrund der positiven Erfahrung mit der Referenzimplementierung in Ada übernommen. Des Weiteren finden sich zwei Verbunde, die nur innerhalb der Prozessor- und der AVL-Baum-Implementierung genutzt wurden. Auf beides wird später noch detaillierter eingegangen.

6.4.4. Einfluss der Wortbreite auf Algorithmen

Für diese Reimplementierung mussten nahezu alle Algorithmen von Grund auf neu implementiert werden. Zwar änderten sich die Algorithmen nicht, insbesondere die der mehrfachgenauen Arithmetik sind nach wie vor zeitgemäß, aber die interne Logik bedarf aufgrund anderer Datentypen und der verfügbaren Arithmetik der Basismaschine einer Anpassung. Die vielen intrinsischen Funktionen, die alternierende Abarbeitung der Teilwörter bei der Division und der komplizierte Kontrollfluss sind unter anderem Ursache dafür, dass eine einfache Anpassung an die jetzt deutlich effizienteren Strukturen unmöglich wird. Nachdem die Algorithmen bereits vertraut waren, fiel die Neuimplementierung leichter. Das gilt nicht nur für die Algorithmen der mehrfachen Genauigkeit, sondern auch für die des effizienten Speicherzugriffs.

6.4.5. Zeitgemäße Optimierungsmöglichkeiten

Die erste Version der Reimplementierung basierte noch vollständig auf Listen. Insgesamt wurde die Programmlogik nicht verändert. Wobei die effizienten Verbunde bereits genutzt wurden. Insbesondere Register und Speicherblöcke wurden jedoch wie in der Implementierung für die Cyber 960 in Listen verwaltet. Die erste Implementierung ist also bis auf die dynamische Speichernutzung, die jetzt vom Compiler direkt zur Verfügung gestellt wird und nicht selbst über ein starres Array abgebildet werden muss, auf dem Stand des Jahres 1969 (Division nach Knuth ist bereits umgesetzt). Für die Decodierung des kleinen 16x16 Pixel großen JPEG-Bildes braucht diese Version unter gleichen Rahmenbedingungen statt 29 Minuten jetzt nur noch 66 Sekunden. Allein die an die Möglichkeiten der Basismaschine angepassten Algorithmen und die direkt nutzbare dynamische Speicherverwaltung ermöglichen in diesem Fall eine Verbesserung um den Faktor 26.

Im Folgenden werden die im Jahr 1988 verfügbaren Algorithmen und Datenstrukturen aufgezeigt und für diese Implementierung umgesetzt. Die jeweiligen Auswirkungen in Form von zusätzlichen Programmzeilen und Laufzeiten geben Aufschluss über den jeweiligen Aufwand und den Gewinn.

B-Baum

Die Referenzimplementierung in Ada nutzt zur Verwaltung der Register, der Speicherblöcke und der Segmente jeweils einen B-Baum. Diese Datenstruktur ist seit 1970 bekannt und bietet Vorteile, wenn es notwendig ist, Teilbereiche eines Baumes auf andere Datenspeicher auszulagern. Wie auch schon bei der Referenzimplementierung wird auf die Implementierung des Auslagerns verzichtet. Die Implementierung der Algorithmen des B-Baumes der Referenzimplementierung konnten als Vorlage genutzt werden. Die stark abweichende Syntax der *Cray Pointer* und die fehlende Unterstützung rekursiver Funktionen bzw. Unterprogramme seitens FORTRAN 77, verwehrten aber eine einfache Anpassung. Innerhalb eines Tages konnten die notwendigen Algorithmen implementiert werden. Innerhalb eines weiteren Tages wurde die Verwaltung der Register umgestellt. Die Register werden einmalig erzeugt und stehen fortan zur Verfügung. Das Entfernen einzelner Register aus dem Baum findet nicht statt. Folglich wurden auch die sehr aufwendigen Routinen hierfür nicht implementiert.

Die Implementierung umfasst nach der sehr einfach möglichen Einbindung des B-Baums 390 zusätzliche Programmzeilen. Für die Decodierung des gleichen kleinen Bildes werden jetzt durchschnittlich noch 8,914 Sekunden benötigt.

AVL-Baum

AVL-Bäume sind bereits seit 1962 bekannt. Aufgrund der fehlenden Unterstützung einer dynamischen Speichernutzung seitens früheren FORTRAN Versionen, sind balancierte Bäume jedoch erst mit dieser Implementierung effizient umsetzbar. Im Gegensatz zu B-Bäumen können einzelne Knoten im AVL-Baum jedoch immer nur einen Wert aufnehmen, die Knoten der B-Bäume hingegen zwischen n und $2n - 1$. Für jeden B-Baum ist n eine konstante Ganzzahl größer gleich 1. Die im Speicher abgelegte Struktur ändert sich für B-Bäume mit größeren n deutlich seltener als für AVL-Bäume. Daher eignen sich auch einzelne Knoten solcher Bäume gut zum Auslagern. Umso größer jedoch die Knoten werden, desto öfter müssen die Werte innerhalb des Knotens verschoben, also kopiert werden, was sich negativ auf die Laufzeit auswirkt.

Soll ohnehin auf die Funktionalität des Auslagerns verzichtet werden, können auch die etwas schnelleren AVL-Bäume genutzt werden. Deren Struktur belegt im Schnitt weniger Speicher, da es keine nur zur Hälfte befüllten Knoten gibt.

Auch diese Implementierung wurde innerhalb eines Tages abgeschlossen, obwohl keine Vorlage genutzt wurde. Verfügbare Beschreibungen des Algorithmus sind rekursiv formuliert. Daher mussten zunächst aufgrund der fehlenden Unterstützung der Rekursion die über einen Keller zu sichernden Informationen identifiziert werden. Der Aufwand ist gerechtfertigt, denn nicht in jedem Fall sind es alle Parameter und alle lokalen Variablen, die ein rekursives Unterprogramm hat. Die genaue Identifizierung kann helfen, Laufzeiten zu gewinnen und Speicher zu schonen. Im Falle der B-Baum-Implementierung wurde auf die Implementierung von Wirth [1975] zurückgegriffen, der wo immer möglich mit globalen Variablen und sogar parameterlosen Unterprogrammen gearbeitet hat.

Dass sich dieser Aufwand lohnen kann, belegen die erzielten Laufzeiten. Zur Decodierung des kleinen Bildes werden jetzt durchschnittlich nur noch 7,930 Sekunden benötigt. Das entspricht einem Laufzeitgewinn von ca. 11%. Die Implementierung des AVL-Baums umfasst zudem weniger Programmzeilen.

Wenn man sich die Verwendung der Register innerhalb der Prozessorimplementierung genau anschaut, fällt einem auf, dass immer zunächst versucht wird, ein Register aus dem Baum zu holen. Schlägt dieser Versuch fehl, handelt es sich um den ersten Zugriff. Ist dieser gemäß Spezifikation erlaubt, kann ein Register erzeugt und im Baum eingefügt werden. Erlaubt ist die initiale Nutzung eines Registers nur über einen schreibenden Zugriff, also z.B. mittels `LOADC`. Würde man das offener gestalten und alle Register initial als leer und positiv spezifizieren, gäbe es weitere Optimierungsmöglichkeiten.[10] Schon mit dem Prüfen, ob ein Register bereits im Baum gespeichert ist, erreicht der Algorithmus den Knoten, an dem im Falle des Scheiterns im nächsten Schritt der Algorithmus zum Einfügen ansetzen muss. Bereits der erste Algorithmus könnte entscheiden, ob das gefundene Register zurückgegeben werden kann oder

[10] Die gibt es ohnehin, aber derartige „Fehler“ in UVC-Anwendungen, die eigentlich nicht mehr enthalten sein und besser vom Compiler oder Assembler gefunden werden sollten, würden nicht mehr geahndet.

ob direkt an dieser Stelle mit dem Einfügen eines neuen leeren positiven Registers fortgesetzt werden muss.

Auf die Abarbeitung „korrekter“ UVC-Anwendungen hat diese Optimierung keinen Einfluss. Die Implementierung des AVL-Baums verkürzt sich jedoch um 23 Programmzeilen und erzielt eine durchschnittliche Laufzeit von 7,800 Sekunden. Die gleiche Optimierung wäre auch für B-Bäume möglich.

Optimierungen im Prozessor

Dieser Abschnitt beschäftigt sich mit den Möglichkeiten, die Prozessorimplementierung zu optimieren. Die für die Implementierung von virtuellen Maschinen markante Schleife ist das Herzstück eines Prozessors. Mit jeder Iteration wird genau ein Befehl geladen und entsprechend des Opcodes und der Parameter verarbeitet. Dieses fortwährend abgearbeitete Programmstück gehört somit zu den von Knuth identifizierten “10% eines Programms, die 90% der Laufzeit ausmachen“ und verdient daher erhöhte Aufmerksamkeit.

Diese Implementierung des Prozessors geht absichtlich nicht soweit wie die Referenzimplementierung in Ada. In dem durch die CDC 4680 abgedeckten Zeitraum standen die dort genutzten Tricks noch nicht zur Verfügung bzw. wurden nicht im Bereich virtueller Maschinen eingesetzt. An dieser Implementierung werden daher bewusst nur die Möglichkeiten des Interpreteransatzes betrachtet. Dabei wird speziell auf die Eigenschaften des Bitcodes der UVC-Anwendungen geachtet, der bei keiner anderen virtuellen Maschine derart ins Gewicht fällt.

In allen bisher vorgestellten Implementierungen arbeitet der Interpreter wie folgt: Als erstes werden die 8 Bit, die den aktuellen Opcode codieren, beginnend ab der im Befehlszeiger gespeicherten Adresse aus dem Speicher geladen. Hierfür wird ein Unterprogramm bemüht, das die notwendigen Schritte durchführt, darunter das Bestimmen des Speicherblocks und des oder der Indizes, die Bitverschiebung und das Ausmaskieren. Da es ein **CASE**-Konstrukt in FORTRAN 77 noch nicht gibt, werden in einem großen Konstrukt aus **IF**, **ELSE IF**, **ELSE** und **END IF** die zum Opcode passenden Routinen zur Auswertung bestimmt. Diese Routinen laden dann die passende Anzahl an Registern aus dem Speicher. Dazu werden zunächst die 32 Bit geladen, die die Segmentnummer des ersten Registers ausmachen. Hierfür wird wiederum das Unterprogramm für den Speicherzugriff genutzt. Dieser Vorgang wird zum Lesen der folgenden 32 Bits der Registernummer wiederholt. Mit diesen Informationen kann jetzt das Segment bestimmt werden und darin das mittlerweile in einem balancierten Baum abgelegte Register. Dieses wird für alle Operanden wiederholt. Erst danach kann die eigentliche Abarbeitung der Instruktion erfolgen.

Allein die Initialisierung des Algorithmus zum Extrahieren einzelner Bits aus dem in Blöcken organisierten Speicher ist sehr umfangreich. Für eine Division, die drei Operanden verarbeitet, wird diese Initialisierung im vorgestellten Algorithmus siebenmal notwendig. Dabei sind die zu lesenden Bits jeweils auf gleicher Weise verschoben und in den allermeisten Fällen im selben Speicherblock enthalten.

Eine naheliegende Optimierung besteht daher darin, mit nur einem Unterprogrammaufruf die korrekte Anzahl an Bits aus dem Speicher zu laden. Das aufwendige Bestimmen des benötigten Speicherblocks und des passenden Index fällt somit nur zweimal für eine Instruktion an – Speicheroperationen wie **LOAD** und **STORE** ausgenommen. Sollen auch die 8 Bits des Op-

codes in diesem Vorgang geladen werden, so ist der Algorithmus anzupassen. Beginnt man mit diesem Lesen im Block erst mit den Operanden, kann man recht einfach ein Wort mehr lesen. In diesem befindet sich dann der Opcode des folgenden Befehls. Nur dieser ist jetzt noch zu verschieben. Mit diesem, dem *Prefetching* ähnlichen Ansatz ist es möglich, in den meisten Fällen mit nur einem Zugriff pro Instruktion auszukommen. Dabei kann anhand des Opcodes treffsicher bestimmt werden, ob sich dieser Aufwand lohnt. Im Falle von `LOADC` oder `JUMP` wird darauf verzichtet. Für die Instruktionen `CALL` und `BREAK` hingegen lohnt sich das, wenn der Opcode des nächsten Befehls als Bestandteil des aktuellen Prozessorzustands auf dem Keller gesichert wird.

Der Programmcode wird durch diese Optimierung vom Umfang her kürzer und lesbarer:

```
ELSE IF (opcode .EQ. 64) THEN                    ! ADD
   CALL LoadWords(currSection,currIP+8,5,array)  ! lesen in einem Zug
   CALL GetRegister(... array(1), array(2), rR1) ! um Segmentangaben...
   CALL GetRegister(... array(3), array(4), rR2) ! ...gekürzt
   CALL AddRegister(rR1,rR2)                     ! rR1 := rR1+rR2
   currIP = currIP+8+32+32+32+32                 ! IP auf nächste Inst.
   opcode = IAND(ISHFT(array(5),-24),$FF)        ! Opcode zurechtrücken
```

Der so optimierte UVC führt den Decoder auf das gleiche Bild in nur 6,154 Sekunden aus. Das entspricht einer Laufzeitsteigerung von über 20%.

Viele Prozessoren bieten zudem die Möglichkeit, zyklische Schiebeoperationen zu nutzen, so auch die MIPS R6000 Prozessoren der CDC 4680. Der FORTRAN 77 Compiler bietet zur Nutzung die intrinsische Funktion `ISHFTC`. Damit können alle 32 Bits im Wort verschoben werden und befinden sich an der richtigen Position. Die Abbildung 6.8 ist der kleinste Anwendungsfall zu sehen, der bei den wenigen Instruktionen eintritt, die wie `JUMPC` nur ein Register als Operanden erwarten. Hier werden drei 32 Bit Worte aus dem Speicher extrahiert. Dazu sind für jedes der sechs eingefärbten Teilwörter Schiebeoperationen notwendig, mit der zyklischen Schiebeoperation nur noch vier. Da sich der Opcode des nächsten Befehls verschoben im letzten Wort befindet, kommt eine weitere Schiebeoperation dazu. Für alle anderen Instruktionen sieht das Verhältnis noch besser aus: Dreistellige Instruktionen wie die Division laden sieben Worte, benötigen also nur 8 zyklische und eine normale Schiebeoperation, anstatt 12.

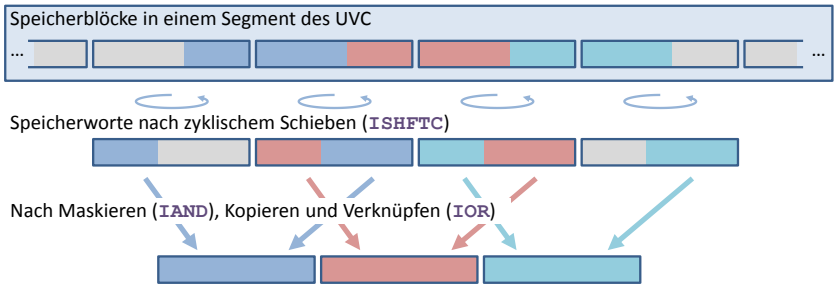


Abbildung 6.8.: Extrahieren der Speicherworte mittels zyklischer Schiebeoperation

Mit dieser im Vergleich kleinen Optimierung lassen sich weitere 4% der Laufzeit einsparen. Die Laufzeit verkürzt sich auf durchschnittliche 5,902 Sekunden. Falls keine zyklische Schiebeoperation verfügbar ist, müssen jeweils zwei Operationen pro Speicherwort durchgeführt

werden. Es lohnt sich jedoch in jedem Fall das vorab Berechnen und Wiederverwenden der Masken.

Nachdem FORTRAN 77 keine `CASE`-Konstrukte und auch keine Sprungtabellen unterstützt, sind die zahlreichen `IF` Anweisungen unabdingbar. Die Komplexität der Bestimmung der passenden Routine aufgrund des aktuellen Opcodes ist somit der des Auffindens eines bestimmten Eintrages in einer unsortierten Liste gleich.

Bei der Beschreibung der Referenzimplementierung wurde eine mögliche Optimierung für diesen Fall bereits vorgestellt (siehe Abschnitt 5.3.4). Diese Optimierung spart im Fall der Referenzimplementierung aufgrund der vielen zusätzlichen Opcodes zur Abbildung auf die interne virtuelle Maschine ganze 10% der Laufzeit ein. Die Implementierung in FORTRAN 77 erfährt immerhin einen Laufzeitgewinn von 2,5%.

Die letzte Optimierung betrifft die verteilte Segmentverwaltung. In diesem speziellen Fall hatte das enorme Auswirkungen. Mit der ersten Version dieser Implementierung basierten alle dynamischen Datenstrukturen noch auf einfach verketteten Listen, um die Vergleichbarkeit mit der Implementierung für die Cyber 960 zu wahren. Die balancierten Bäume wurden im Anschluss ausschließlich zur Verwaltung der Register verwendet, da in keinem Fall ein einzelnes Register aus dem Baum zu löschen ist. Zusammen mit der verteilten Segmentverwaltung wurden auch hier AVL-Bäume eingesetzt. Im Zuge dessen wurde deutlich, dass sich die „geschickte“ Implementierung der Baum-Algorithmen für alle mittels Bäumen verwalteter Datenstrukturen eignet. Register, Speicherblöcke und Segmente können mit dem ersten Zugriff direkt initialisiert werden. Ein Zugriff auf noch nicht initialisierte Instanzen mit Fehlermeldungen zu strafen, macht im Falle des UVC, wie in Abschnitt 5.3.3 dargestellt, keinen Sinn.

Die finale Implementierung decodiert das kleine 16x16 Pixel große JPEG-kodierte Bild in nur 4,646 Sekunden.

6.4.6. Bewertung der finalen Version

Die finale Version ist nicht voll funktionsfähig. Genau wie die Implementierung in FORTRAN IV unterstützt diese Implementierung nicht das initiale Laden von Konstanten. Sowohl die UVC-Anwendungen als auch die zu decodierenden Bilder können ebenfalls nicht als Binärdateien eingelesen werden und bedürfen einer Migration. Abgesehen davon wird die in der Spezifikation geforderte Funktionalität voll unterstützt. Jedoch ist noch nicht alles perfekt optimiert. So nutzt z.B. die Instruktion `LOADC` für sehr große Konstanten noch eine Iteration mit separatem Speicherzugriff. Es fehlen bisher Anwendungsfälle, die eine Optimierung rechtfertigen würden. Die fehlende Funktionalität beeinflusst die zuvor aufgeführten Laufzeiten nicht.

Obwohl die Implementierung aufgrund der zeitgemäßen Optimierungen auf den ersten Blick deutlich anders aussieht, lässt sich nach wie vor der Quelltext klar den identifizierten Bereichen zuordnen. In einigen Bereichen weichen die verwendeten Algorithmen und Datenstrukturen stark ab. Gerade die dynamische Speichernutzung ermöglicht jetzt die effiziente Nutzung dynamischer Strukturen wie balancierte Bäume. Die vielen Quelltextzeilen, die bisher gebraucht wurden, um eine dynamische Speicherverwaltung abzudecken, können jetzt entfallen. An deren Stelle finden sich jetzt Programmzeilen, die die Funktionalität der AVL-Bäume implementieren.

Diese Reimplementierung mit seinen 5103 Programmzeilen wurde innerhalb von 5 Wochen implementiert. Wobei eine erste Version dieser Implementierung in einem Zeitraum von nicht ganz 4 Wochen entstand. Die Einarbeitung der hier dargestellten Optimierungen nahm zusammengenommen nur eine volle Arbeitswoche in Anspruch.

6.5. Portierungen für die SUN Enterprise 10000

Für diesen Abschnitt sind drei Implementierungen interessant. Zum einen existieren die Implementierung für die Cyber 960 und deren portierte Version für die CDC 4680. Zum anderen gibt es die Reimplementierung für die CDC 4680 (siehe Abb. 6.9).

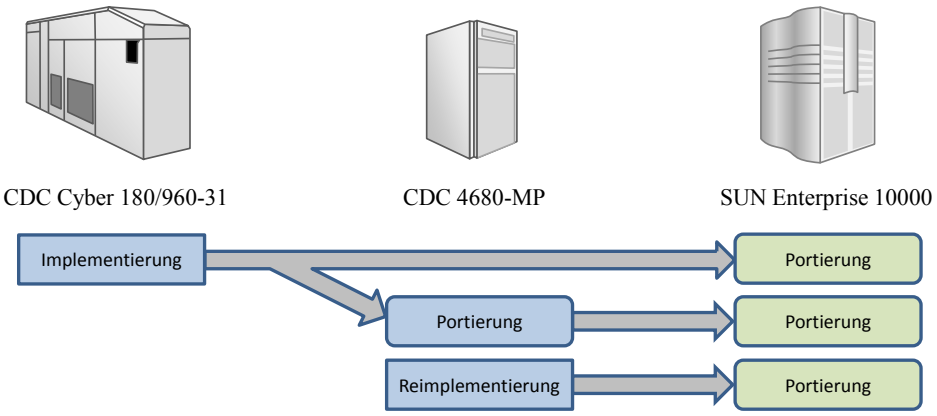


Abbildung 6.9.: Vorhandene Implementierungen und damit mögliche Portierungen

Anhand dieser Gegebenheiten lassen sich bezüglich des Portierungsaufwands folgende Fragestellungen untersuchen:

- Steigt der Aufwand einer Portierung mit der Zeit, die seit der Implementierung vergangen ist?
- Bestätigt sich das extreme Verhältnis der Laufzeiten zwischen dem portierten und dem reimplementierten UVC auch für eine andere Architektur?

Alle vorhandenen Implementierungen müssen für das gleiche Zielsystem portiert werden. Bei Auslieferung der ersten Exemplare der SUN Enterprise 10000 (SUNe10k) war eine Version des SUN Studios verfügbar, die noch einen FORTRAN 77-Compiler enthielt. Mit dem aktuell installierten SUN Studio 11 steht nur noch ein FORTRAN 95-Compiler zur Verfügung, der jedoch über einen Schalter FORTRAN 77-Programme übersetzen kann. Zusammen mit der Erfahrung der Portierung für die CDC 4680 sind das ideale Bedingungen für einen reibungslosen Ablauf.

6.5.1. Portierungen der Cyber 960 Implementierung

Die Implementierung des UVC für die Cyber 960 gibt es in zwei Ausführungen. Portiert für die CDC 4680 wurden zwar jeweils die Versionen mit und ohne Nutzung des ECS. Jedoch

schlug die Nutzung der ECS Variante aufgrund des Compilers fehl, der mit der Optimierung ein fehlerhaftes Programm erzeugte. Betrachtet werden daher hier nur die Varianten ohne ECS.

Die erste Portierung überführt den FORTRAN IV-Quelltext. Die Erfahrungen der Portierung für den FORTRAN 77-Compiler der CDC 4680 helfen dabei enorm. Auch das Zielsystem unterstützt den Datentypen `INTEGER*8`. Damit sind exakt die gleichen Programmstellen anzupassen. Dies betraf erneut die intrinsischen Funktionen, speziell die zyklische Schiebeoperation, und die konstanten Ausdrücke, da oktale Angaben nicht mehr unterstützt werden. Hexadezimalangaben können zudem nur für die direkte Variablen-Zuweisung genutzt werden. Im Programm sind diese gemäß Spezifikation der Sprache FORTRAN 95 nicht vorgesehen. Abhilfe schaffte nur die Konvertierung aller Konstanten in Dezimalschreibweise. FORTRAN 95 kennt weiterhin Sprungmarken und `GOTO`-Anweisungen. Von den wenigen nicht mehr unterstützten, sehr speziellen Sprunganweisungen wurde keine genutzt. Diese Portierung konnte in nur fünf Stunden abgeschlossen werden. Im Vergleich mit der Portierung für die CDC 4680 ist das nahezu exakt der gleiche Aufwand, wenn man berücksichtigt, dass die entsprechenden Erfahrungen bereits vorhanden waren. Die zehn Jahre, die zwischen diesen Portierungen liegen, wirkten sich somit nicht auf den Aufwand aus.

Die zweite Portierung beschränkt sich lediglich auf die Konvertierung der konstanten Ausdrücke. Die zuvor konvertierten Hexadezimalzahlen sind jetzt in die Dezimaldarstellung zu überführen. Mit einem Texteditor, der mit einem Durchlauf mehrere Vorkommen einer Zeichenkette suchen und ersetzen kann, ist diese Portierung in weniger als einer Stunde machbar. Es überrascht wenig, dass beide Portierungen nahezu identisch sind und somit identische Laufzeiten aufweisen. Zum Decodieren des kleinen JPEG-Bildes benötigt die SUNe10k mit beiden UVC Portierungen jeweils durchschnittlich 14,3 Sekunden.

6.5.2. Portierung der Reimplementierung für die CDC 4680

Die Reimplementierung für die CDC 4680 nutzt zusätzlich zu den angenehmen Blockstrukturen des FORTRAN 77-Standards auch zeitgemäße Erweiterungen wie Verbundtypen und Zeiger. Die speziellen und zu dieser Zeit auch weit verbreiteten *Cray Pointer* wurden ebenso wie die genutzte Verbundtypendeklaration nicht zum Standard. Der vom Compiler unterstützte Standard FORTRAN 95 kennt zwar Zeiger und Verbunde, jedoch ist die Syntax und auch die Semantik verschieden.

Der genutzte Compiler der SUNe10k kann allerdings mit dem Schalter `-f77` noch Programme früherer FORTRAN Versionen übersetzen. Mit diesem Schalter werden sowohl *Cray Pointer* als auch `STRUCTURE`-Konstrukte korrekt interpretiert. Mit diesem Schalter sind lediglich kleinere Änderungen nötig.

Da die Reimplementierung im Gegensatz zur Implementierung für die Cyber 960 Funktionen nutzt und deren Syntax sich änderte, sind hier zahlreiche Änderungen vorzunehmen. FORTRAN 95 unterstützt Ganzzahltypen verschiedener Größen, darunter der überwiegend genutzte 32 Bit und der 64 Bit große Ganzzahltyp. Mit dem Schalter ist auch die alte Syntax der Typen möglich, was weder eine Anpassung noch eine Überprüfung auf eventuell falsch arbeitende Schiebeoperatoren nötig macht. Lediglich der Name einer intrinsischen Funktion änderte sich, worauf der Compiler „aufmerksam“ macht.

Etwas schwierig gestaltet sich die Einarbeitung in den vom Compiler unterstützten Sprachumfang. Im Gegensatz zu den bisher genutzten Handbüchern enthält die Anleitung des Compilers der SUNe10k ausschließlich Hinweise zu den Erweiterungen bezogen auf den bereits greifbaren FORTRAN 2003-Standard und den zahlreichen Optionen des Compilers. Bezüglich des unterstützten Sprachumfangs verweist das Handbuch auf den Standard FORTRAN 95. Für den genutzten Schalter zur abwärtskompatiblen Nutzung von FORTRAN 77-Programmen ist kaum Information enthalten. Dies führte dazu, dass erst durch Probieren herausgefunden wurde, welche Teile wie angepasst werden mussten. Trotz des nicht geradlinig möglichen Vorgehens wurde diese Portierung innerhalb eines Tages abgeschlossen.

Compiler vieler anderer Systeme oder folgender Generation unterstützen jedoch weder *Cray Pointer* noch die verwendete Verbundtypdeklaration. Der dadurch eigentlich notwendige Portierungsaufwand wäre enorm, wie eigene Versuche dahingehend zeigten.

Diese portierte Version der Reimplementierung für die CDC 4680 decodiert das bekannte Testbild in durchschnittlich 0,56 Sekunden. Das entspricht einem Verhältnis gegenüber der direkt portierten Version von 1:25. Das auf der CDC 4680 gegebene Verhältnis von 1:374 wird damit etwas relativiert. Beide Implementierungen wurden an sich nicht verändert, haben aber auf unterschiedlichen Architekturen deutlich unterschiedliche Laufzeiten. Beide Ergebnisse für sich lassen aber den Schluss zu, dass sich eine Reimplementierung und die Einarbeitung moderner Algorithmen und Datenstrukturen lohnen.

6.6. Neu-Implementierung für SUN Enterprise 10000

Dieser Abschnitt stellt die Ergebnisse der Bachelorarbeit von Schiller [2012] dar. Bis zur Vergabe der Bachelorarbeit durch den Autor der vorliegenden Arbeit existierten diverse Implementierungen und Portierungen für die genutzten Großrechner der datArena, aber alle aus einer „Feder.“ Zudem fehlte im Puzzle eine Neuimplementierung für den jüngsten genutzten Großrechner der datArena: die SUNe10k. Diese Arbeit wurde daher mit dem Ziel vergeben, eine eigenständig von einem anderen Autor programmierte Implementierung für die SUN e10k in FORTRAN 95 zu erhalten und begleitend dazu Erfahrungen aus anderer Perspektive zu sammeln. So liefert die Arbeit Antworten auf neu aufgekommene Fragen, z.B. zum Nutzen einer verfügbaren Testsammlung, die in einem späteren Kapitel thematisiert wird. Bestätigen konnte die Arbeit auch bisherige, nur einseitig gesammelte Erkenntnisse, z.B. zu den Implementierungszeiten und dem Verhältnis zwischen Portierung und Reimplementierung.

Damit auch die Ergebnisse von Schiller im Kontext der *Zeitreise* im Anschluss bewertet werden können, wird die von ihm erstellte Implementierung kurz umrissen. Zu betrachten sind die genutzten Spracherweiterungen und einige Details seiner Implementierung.

6.6.1. Genutze Erweiterungen von FORTRAN 95

FORTRAN 95 bietet im Vergleich zu FORTRAN 77 einige Neuerungen. Die Wesentlichste ist das Modul-Konzept. Damit lassen sich genutzte Datenstrukturen und Schnittstellen an zentraler Stelle definieren und im gesamten Programm einheitlich nutzen. Im Vergleich zum

FORTRAN 77-Quelltext spart man dadurch viele Zeilen Programmcode ein, gewinnt deutlich an Übersichtlichkeit und vermeidet Inkonsistenzen.

Das Modul-Konzept wurde nicht übertrieben angewendet. So gibt es nur ein Modul, obwohl man auch je ein Modul zu den identifizierten Bereichen wie Speicher, Prozessor usw. hätte nutzen können. Der Übersichtlichkeit schadet die sparsame Anwendung jedoch nicht.

Genutzt wurden jetzt auch die mit FORTRAN 90 standardisierten Zeiger, die die proprietären *Cray Pointer* ablösten.

Ebenfalls neu ist die Möglichkeit, rekursive Funktionen zu schreiben. Dieses Konstrukt ermöglicht es Programmierern, bei zu durchlaufenden rekursiven Datenstrukturen wie einem AVL-Baum übersichtlicheren Programmcode zu erstellen, da nicht zusätzlich ein Keller verwaltet werden muss. Im Fall des AVL-Baums kann auch der Verweis zum Vaterknoten entfallen. Obwohl die Rekursion einen gewissen Verwaltungsmehraufwand verursacht, kann durch die geschickte Verwendung insgesamt Laufzeit gewonnen werden. Die Implementierung für die SUNe10k nutzt dieses Konstrukt vor allem zur Steigerung der Übersichtlichkeit.

Durch die genutzten Erweiterungen stellt sich der erstellte Quelltext auf den ersten Blick völlig anders dar; ein persönlicher Programmierstil wird deutlich. Auf den zweiten Blick jedoch finden sich viele Gemeinsamkeiten, die darauf hindeuten, dass die Implementierung eines UVC immer einem bestimmten Muster folgt. So sind abermals die zuvor identifizierten Bereiche klar zuzuordnen. Auch im Detail werden Gemeinsamkeiten offensichtlich, wie die Division nach Knuth, die Verwaltung mittels balancierter Bäume und die Realisierung der Registerinhalte in verketteten Listen.

6.6.2. Relevante Details zur Implementierung

Die genutzten Datenstrukturen sind zweckmäßig gewählt. Zwar wurde zunächst die Verwaltung mit Listen realisiert. Aber schnell wurde erkannt, dass balancierte Bäume zweckmäßiger sind. Eine Registerverwaltung basierend auf Listen ist aufgrund der Anforderungen an den Registerzugriff aus heutiger Sicht abwegig. Die Verwendung von Listen in einem frühen Prototypen zeugt jedoch davon, dass die bisherige Spezifikation die Anforderungen an den Registerzugriff durch die Zuordnung dieser in unbegrenzter Anzahl jeweils zu den Segmenten in den Hintergrund rückt. Zudem zeigt sich, dass eine Implementierungshilfe begleitend zum UVC durchaus helfen würde, kleine Fehlentscheidungen mit großer Wirkung während der Designphase zu vermeiden.

In diese Kategorie gehört auch die Realisierung des bitadressierten Speichers eines Segments. Die Segmente werden zwar in einem AVL-Baum verwaltet, ebenso die einzelnen Speicherblöcke zu je 1024 Byte, aber der Zugriff auf den Speicher selbst geschieht Byte für Byte. Das heißt, wenn z.B. 64 Bit aus dem Speicher gelesen werden sollen, muss mindestens acht Mal über den AVL-Baum der zugehörige Speicherblock identifiziert und die jeweiligen Bits aus dem zugehörigen Speicherwort ausmaskiert werden. Und das obwohl die genutzte SUNe10k als auch FORTRAN 95 in einem Schritt 64 Bit verarbeiten können.

Für die Speicherung des Bitcodes der UVC-Anwendungen im Speicher der Segmente wurde die Struktur der Segmente optimiert. Der Code findet sich aus Gründen der Effizienz in einem Array. Auf dieses wird aber ebenso byteweise zugegriffen. Diese etwas unglückliche Designentscheidung bewirkt einen recht großen Geschwindigkeitsverlust.

Entfernt erinnert das an die Implementierung von Tamminga (siehe Abschnitt 3.4.1). Durch die Verwendung einer „vernünftigen“ Speicherblockgröße ist jedoch die Auswirkung dieser Fehlentscheidung deutlich geringer. Die Relevanz einer öffentlich zur Verfügung gestellten und regelmäßig aktualisierten Implementierungshilfe begleitend zur Spezifikation des UVC wird aber immer offensichtlicher.

6.6.3. Ergebnis

Insgesamt umfasst der Quelltext der Implementierung 3539 Zeilen, verteilt auf drei Dateien (Implementierung der Funktionalität, Definition der genutzten Datentypen, und Schnittstellenangaben). Die letzte Version musste nochmals überarbeitet werden, da eine fehlende Speicherfreigabe bei längerer Programmausführung zu vorzeitigem Programmabbruch führte. Die aktuelle Version ist nicht mehr in der schriftlichen Arbeit fixiert und umfasst jetzt 4349 Zeilen.

Insgesamt beanspruchte die Implementierung 120 Arbeitsstunden und damit knapp 4 Wochen. In dieser Zeit sind bereits die Einarbeitung und das Testen enthalten.

Seine Implementierung ist um den Faktor 6,25 schneller als die portierte Implementierung für die Cyber 960 ohne ECS. Das ist ein schönes Ergebnis und für einen Bachelorstudenten, der zum ersten Mal Kontakt mit der genutzten Programmiersprache hat und von speziellen Optimierungstechniken noch nichts gehört hat, eine hoch anzuerkennende Leistung.

Die portierte Reimplementierung in FORTRAN 77 ist jedoch um den Faktor 26,09 schneller und damit ungefähr viermal schneller als seine Implementierung, reizt aber viele speziell für diesen Zweck adaptierte Optimierungsmöglichkeiten aus. Das kann man von der Implementierung durch einen Bachelorstudenten nicht erwarten, zumal nicht in der limitierten Bearbeitungszeit.

Durch die konsequente Nutzung des Standards FORTRAN 95 gelingt die Portierung seiner Implementierung auf heutigen PC. Möglich macht das der frei verfügbare Compiler für FORTRAN 95 *g95*, der für Linux- und Windows-Umgebungen existiert.

6.7. Erkenntnisse aus der Vergangenheit

Diese zahlreichen Experimente im Rahmen dieser *Zeitreise* führten zu weitreichenden Erkenntnissen, die in diesem Abschnitt zusammengetragen werden. Alle Implementierungen und Portierungen sind in der Abbildung 6.10 zusammenfassend dargestellt. Die grau unterlegten Implementierungen von IBM wurden bereits vorgestellt (siehe Abschnitt 3.4.1) und werden im Folgenden zur Einordnung der Ergebnisse berücksichtigt. Die in Rottönen unterlegten Felder symbolisieren die Arbeiten von Schiller. Die Erkenntnisse betreffen die Bereiche Implementierungsaufwand, Laufzeiten und die in diesem Kapitel anvisierte Universalität.

6.7.1. Implementierungsaufwand

Der Implementierungsaufwand ist schlecht messbar. Zum einen ist der Aufwand für jeden Programmierer subjektiver Natur und zum anderen hängt der tatsächliche Aufwand sehr stark vom Umfeld, insbesondere von den Fähigkeiten und den Erfahrungen eines Programmierers

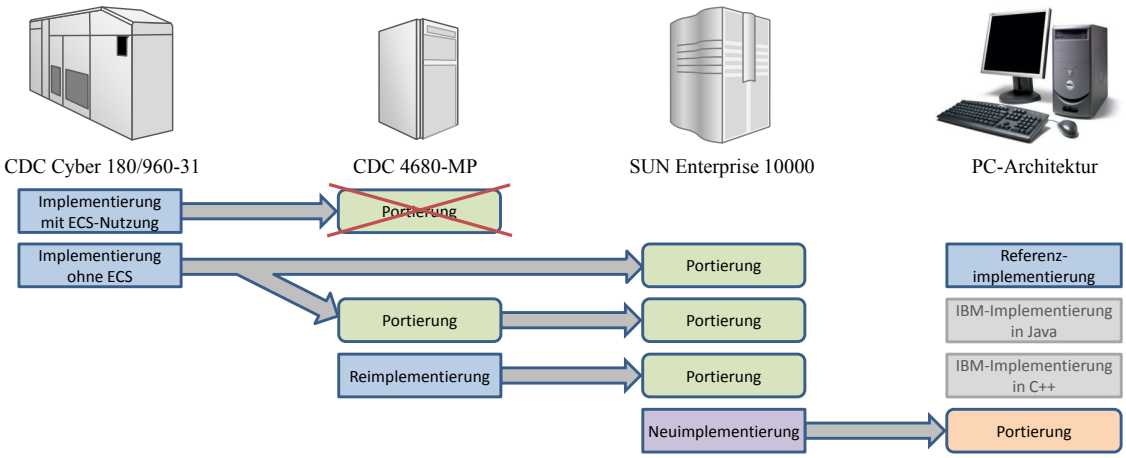


Abbildung 6.10.: Übersicht aller Implementierung und Portierungen, die im Rahmen der Zeitreise erstellt bzw. betrachtet wurden.

ab. Zwei leicht messbare und oft referenzierte Größen mit eingeschränkter Aussagekraft werden herangezogen, um zumindest einen Anhalt und damit erstmals eine nachvollziehbare Kostenabschätzung zu ermöglichen: Die „echten“ Programmzeilen und die Implementierungszeit.

Programmzeilen

Die in Abbildung 6.11 dargestellten Werte sind, im Unterschied zu den bisherigen Angaben, bereits bereinigt, d.h. Kommentar- und Leerzeilen wurden nicht mitgezählt. Ein weiterer Aspekt muss berücksichtigt werden. Die betrachteten Implementierungen sind jeweils die am weitesten entwickelten Versionen; gezählt wurde somit einmalig am Schluss. Die in den Abschnitten 5.3.4 und 6.4.5 dargestellten Optimierungen sind dabei berücksichtig. Das betrifft die Implementierungen in Ada83 und in FORTRAN 77, die dadurch auffallen, dass sie die meisten Programmzeilen umfassen. Zudem ist der jeweilige Schwerpunkt erkennbar: In der Implementierung in FORTRAN 77 wurde stark an der Optimierung der Prozessorimplementierung und des Speichermanagements gearbeitet, bei der Implementierung in Ada83 „beschränkte“ sich die Optimierung auf den Bereich des Prozessors. Durch die Betrachtung der jeweils letzten Implementierung sind die hier dargestellten Ergebnisse genauer als die vorab in Krebs et al. [2011a] veröffentlichten und basieren zudem auf allen vier Implementierungen.

Die Zahlen zeigen, dass auch ein anderer Programmierer eine sehr ähnliche Verteilung erzeugt und sich dabei im durch die Referenzimplementierung vorgegeben Rahmen bewegt.

Eine Auswirkung des Entwicklungszeitpunktes ist auf Basis der durchgeführten Experimente nicht erkennbar. Zu berücksichtigen ist jedoch, dass jeweils nur zeitgemäße Algorithmen und Datenstrukturen umgesetzt wurden. In FORTRAN IV wurden z.B. mit vielen Zeilen die verketteten Listen umgesetzt, dafür keine balancierten Bäume. Auch wenn die Zeit sehr wohl Auswirkungen auf die jeweilige Implementierung hat, gibt es derzeit keinen Grund zur Annahme, dass eine zeitgemäße Implementierung aufgrund der sich anpassenden Entwicklungsumgebungen nicht mit einer beinahe konstanten Zahl an Programmzeilen möglich wäre.

Lines of Code	in Zeilen (ohne Leerzeilen und Kommentare)			
	Ada83	FORTRAN IV	FORTRAN 77	FORTRAN 95
Ein-/Ausgabe	106	158	54	85
Prozessor	1216	403	600	539
Segmentierter Speicher	399	556	527	587
Register	1108	972	1190	1434
Initiales Laden	50	72	61	52
Speichermanagement	535	240	909	398
gesamt	**3414**	**2401**	**3341**	**3095**

Abbildung 6.11.: Programmzeilen zugeordnet zu den Bereichen und den Implementierungen.

Implementierungszeit

Für die Implementierungszeit gibt es durch die vielen Experimente der *Zeitreise* entsprechend viele Daten für verschiedene Architekturen und Programmiersprachen. Dabei ist die Art und Weise der Datenerhebung zu berücksichtigen. Für die Systeme der linken drei Spalten in Abbildung 6.12 wurden die Stunden am Ende eines Tages den bearbeiteten Bereichen zugeordnet. Stunden, die dabei klar der Einarbeitung in die Programmiersprache dienten, wurden nicht berücksichtigt. Allerdings fielen während der Implementierung nur noch wenige an. Der Zeitaufwand für das Testen, inklusive Fehlersuche und Entwicklung der Testfälle, sind dagegen den entsprechenden Bereichen zugeordnet. Nicht dabei sind Zeiten für Laufzeitmessungen.

Entwicklungszeiten	in Stunden			
	Ada83	FORTRAN IV	FORTRAN 77	FORTRAN 95
Ein-/Ausgabe	3	5	6	2
Prozessor	47	10	26	10
Segmentierter Speicher	11	11	14	10
Register	24	95	80	24
Initiales Laden	8	8	8	6
Speichermanagement	49	40	42	6
gesamt	**142**	**169**	**176**	**58**

Abbildung 6.12.: Zu den Bereichen und den Implementierungen zugeordnete Stunden.

Die Zahlen in der vierten Spalte weichen von diesem Muster ab. Sie umfassen tatsächlich nur die Stunden, in denen der Programmtext entwickelt und geschrieben wurde. Die Zeit, die inklusive Einarbeitung und anschließender Testphase gebraucht wurde, gibt Schiller mit 124 Stunden an. Seine Implementierung entstand in einem Zeitraum von drei Monaten im Rahmen einer Bachelorarbeit. Diese wird parallel zum laufenden Studium bearbeitet, weshalb die Arbeitsstunden zwischen allen anderen Verpflichtungen eines Studenten wie Vorlesungen und Prüfungen zu finden sind. Die eigenen Implementierungen wurden dagegen weitestgehend am Stück erstellt. Nur durch kleinere Unterbrechungen verzögert gelang die Implementierung jeweils innerhalb von sechs Wochen. Teilweise war ein voll funktionsfähiger UVC schon nach vier Wochen verfügbar. Die noch mit einberechnete Zeit wurde zur Optimierung verwendet. Auch die Implementierung in FORTRAN IV wurde noch einmal optimiert, als die Freilisten als im Jahr 1964 bereits bekannt identifiziert wurden. Diese Optimierung jedoch benötigte im Vergleich zu den Optimierungen in FORTRAN 77 und Ada83 nur sehr wenig Zeit.

Bewertung

Bisher wurden nur die Gesamtergebnisse betrachtet. Die Gewichtung der einzelnen Bereiche ist jedoch für zwei weitere Betrachtungen wichtig. Die Anteile der Implementierungszeit und der Programmzeilen unterscheiden sich jeweils nicht nur untereinander, sondern auch innerhalb einer Implementierung. Zum Teil ist das abhängig vom Programmierstil und den zeitgemäßen Techniken, aber auch die genutzte Programmiersprache hat einen erheblichen Einfluss. Die jeweils gemittelten Anteile jeweils über alle Implementierungen sind etwas größer ganz rechts in der Abbildung 6.13 dargestellt. Diese beiden Darstellungen ähneln sich sehr stark und können zusammen für Kosten- und Aufwandsabschätzungen herangezogen werden.

Der erste Bereich, für den die Auswertung der Gewichtungen relevant wird, betrifft die Weiterentwicklung der Spezifikation. Mit der Feststellung, dass z.B. das initiale Laden der Konstanten auch problemlos innerhalb einer UVC-Anwendung realisiert werden kann, ist jetzt das Potential einer solchen Einsparung direkt ablesbar.

Der zweite Bereich betrifft potentiell mit der Implementierung des UVC beauftragte Firmen. Und somit auch die Archive, die solche Aufträge ausschreiben bzw. die Kosten dieser Archivierungsmethode abschätzen müssen. Interessant ist, welche Bestandteile mit welchem Aufwand umzusetzen sind. Möglicherweise sind bereits aus anderen Projekten Bestandteile oder Know-How vorhanden, die für eine genauere Kalkulation des tatsächlichen Aufwands herangezogen werden können. Beispielsweise können verfügbare und leicht an die Bedürfnisse des UVC anpassbare Pakete zur Umsetzung der beliebig genauen Arithmetik den zu erbringenden Aufwand extrem verkürzen.

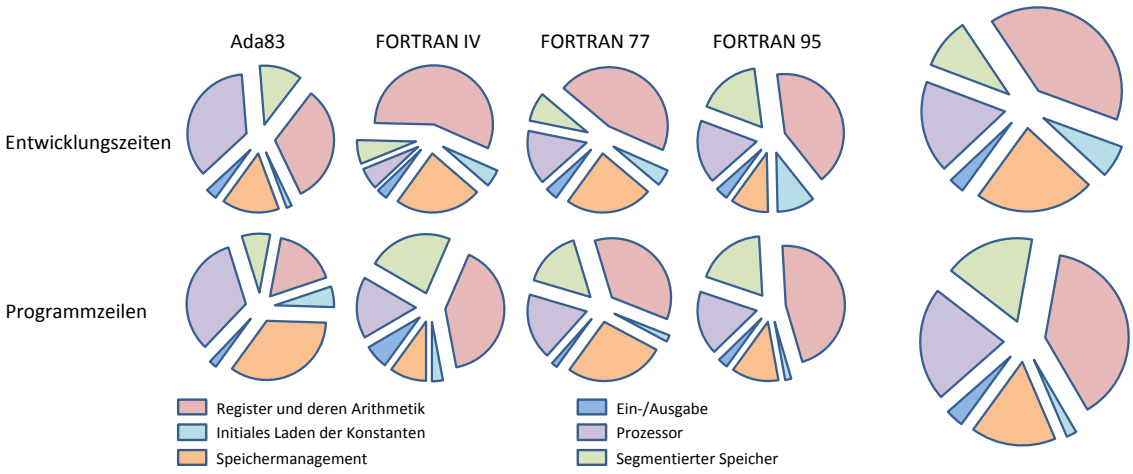


Abbildung 6.13.: Übersicht der anteiligen Zuordnung.

6.7.2. Portierungsaufwand und -nutzen

In der Abbildung 6.10 sind alle Portierungen mit den jeweils abgerundeten Kästchen dargestellt. Alle diese Portierungen ließen sich innerhalb von wenigen Stunden, spätestens nach

zwei Tagen auf den neuen Maschinen ausführen. Allerdings wurde bemerkt, dass sich die erhöhte Rechenleistung der neuen Maschine nicht auf die erzielten Laufzeiten der Portierung auswirkt. Die CDC 4680 sollte im Mittel um den Faktor 2,8 - 4,4 schneller sein, als die genutzte Cyber. Die gelungene Portierung jedoch ist um mehr als den Faktor 2 langsamer. Nicht zu vergessen ist in diesem Zusammenhang, dass nur eine Version der beiden Implementierungen für die Cyber fehlerfrei portiert werden konnte. Die Variante, die den ECS der Cyber nutzt, konnte zwar mit dem Compiler optimiert übersetzt werden, zeigte aber nicht zu umgehende Fehler bei der Ausführung speicherintensiver UVC-Anwendungen. Wäre nur diese Implementierung vorhanden und die ursprünglichen Entwickler nicht mehr greifbar, würde der Versuch einer solchen Portierung bereits hier scheitern. Da die Portierung aber innerhalb der CDC-Familie problemlos bis in die frühen 90'er möglich war, kann man in diesem Fall von 30 Jahren ausgehen, die mittels der Portierung überbrückbar gewesen wären.

Allerdings verzichtet man damit auf die Laufzeitsteigerungen, die sich durch die Neuerungen der Hardware ergeben, ebenso wie die durch Optimierungen mittels neu entwickelter Datenstrukturen und Algorithmen. Im Falle der Portierung für die CDC 4680 ist das Verhältnis gegenüber der optimierten Reimplementierung 1:374. Da sich beide Versionen erneut auf die SUNe10k portieren ließen und dort ein weniger dramatisches Verhältnis von nur 1:25 zeigten, ist das Ergebnis an sich etwas zu relativieren, aber immer noch sehr deutlich.

Durch diese zweite Portierung auf die SUNe10k sollte allein durch die Hardware eine Beschleunigung um den Faktor 320 im Vergleich zur genutzten Cyber zu erwarten sein und damit ungefähr 5,25 Sekunden für die Decodierung des kleinen Bildes. Tatsächlich waren es aber 14,3 Sekunden. Auch hier zeigt sich deutlich, dass sich über Portierungen der durch die Hardware zu erwartende Gewinn nicht in vollem Umfang auf die Laufzeiten auswirkt. Immerhin: diese zweite Variante hätte mittels der Portierung knapp 50 Jahre zur Verfügung gestanden.

Die „zweite" Implementierung für die CDC 4680 in FORTRAN 77 nutzt *Cray Pointer*, die von neueren Standards nicht mehr unterstützt werden. Aufgrund des genutzten Compilers und dessen Kompatibilität mit dieser Variante der Speichernutzung gelang die Portierung für die SUNe10k. Heute verfügbare Compiler leisten sich diese Abwärtskompatibilität nicht mehr. Diese Version hätte damit nur knapp 30 Jahre genutzt werden können.

Die „dritte" Implementierung innerhalb der *Zeitreise* wurde für die SUNe10k erstellt. Die Portierung auf heutigen Systemen ist derzeit noch mühelos möglich, da weiterhin Compiler für FORTRAN 95 verfügbar sind. Diese Implementierung könnte somit seit nunmehr 17 Jahren genutzt werden. Die verwendeten Sprachkonstrukte werden in den aktuellen FORTRAN Standards unterstützt. Weitere Portierungen erscheinen aus derzeitigem Standpunkt daher unkompliziert. Gleiches gilt für die Referenzimplementierung in Ada83, die ohne Portierung bereits knapp 30 Jahre hätte überdauern können. Unter der Annahme, dass beide Systeme nur jeweils einen Kern bzw. Prozessor zur Ausführung der UVC-Laufzeitumgebung nutzen, müsste der gleichermaßen wie zu den Tests im Abschnitt 5.3.2 genutzte Desktoprechner um den Faktor 12 schneller sein als die SUNe10k.[11] Die Laufzeit zur Decodierung des großen Testbildes gibt Schiller mit 22 Minuten und 56 Sekunden an. Für das gleiche Bild benötigt die Portierung 6 Minuten und 17 Sekunden und ist damit nur um den Faktor 3,65 schneller.

[11] Die SUNe10k wird mit 800 MFLOP pro CPU angegeben, Intel gibt für den genutzten Q6600 38,4 GFLOP an, wobei davon 9,6 auf einen Kern entfallen.

Diese Erfahrungen können helfen, den Portierungsaufwand und dessen Nutzen abzuschätzen. Damit gelingt auch eine Schätzung der langfristig auf ein Archiv zukommenden Kosten.

Direkt aus den aufgeführten Werten lässt sich jedoch kein Faktor ableiten, um den die zu beschaffende Hardware leistungsfähiger sein muss, damit anvisierte Verbesserungen der Aushebungszeiten nur durch Portierungen erreichbar werden.

6.7.3. Laufzeiten

Wie im letzten Abschnitt dargestellt gelingt die Portierung der Implementierung in FORTRAN 77 nicht für heutige Systeme. Die genutzten *Cray Pointer* sind zu tief verwurzelt. Das Ersetzen durch standardisierte Zeiger und Speicherallokationen ist nicht ohne erheblichen Aufwand zu leisten. Bei dem Versuch wurde nach einer Woche intensiver Arbeit deutlich, dass diese Überarbeitung einer Reimplementierung gleich kommt. Die *Cray Pointer* müssten durch typisierte Zeiger ersetzt werden. Das führt jedoch zu einem enormen Anpassungsbedarf innerhalb der Datenstrukturen und Algorithmen. Die Versionen wären nicht mehr uneingeschränkt vergleichbar. Dennoch soll das zu erwartende Ergebnis an dieser Stelle mit betrachtet werden. Zum einen lässt es sich, aufgrund der gegebenen Laufzeitverhältnisse auf der SUNe10k, gut für das zur Laufzeitmessung genutzte aktuelle PC-System skalieren. Zum anderen ist dieses Ergebnis auch ein Anhalt für das noch mögliche Potential.

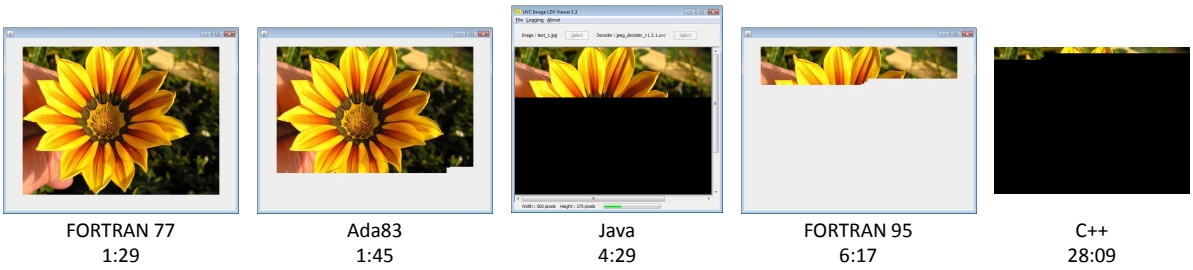


Abbildung 6.14.: Übersicht der Laufzeiten im direkten Vergleich.

Die Abbildung 6.14 zeigt die Laufzeiten und im Vergleich dazu die in Relation zur schnellsten Implementierung anteilig erzeugten Bilddarstellungen. Dabei ist die Laufzeit links im Bild die aus dem Laufzeitverhältnis der SUNe10k skalierte. Die dargestellten Laufzeiten beziehen sich jeweils auf das abgebildete 500x375 Pixel große Testbild, das bei allen Laufzeittests auf der genutzten PC-Architektur zum Einsatz kam.

Bei der Frage nach einem Minimum der Laufzeiten muss man berücksichtigen, dass bei den beiden schnellsten Implementierungen unterschiedliche Strategien verfolgt wurden und Teile davon noch kombiniert werden können. Zudem hat die Implementierung in Ada83 den vollen Umfang der möglichen Optimierung durch die Abbildung auf eine interne virtuelle Maschine noch lange nicht ausgereizt. Unter diesen Gesichtspunkten erscheint eine Decodierung des verwendeten Testbildes in unter einer Minute mit der genutzten Hardware für erreichbar. Diese Information ist bei der Spezifizierung von Aushebungszeiten wichtig. Schnellere, leistungsfähigere Systeme setzen vermehrt auf Parallelität. Davon profitiert der skalare UVC sehr

wenig. Diese Zeit pro Aushebung bleibt damit für die nahe Zukunft recht konstant. Sinn machen massiv parallele Systeme, sobald mit sehr vielen Anfragen gerechnet werden muss oder viele archivierte digitale Objekte gleichzeitig aufzubereiten sind.

Ein sehr wichtiger Punkt in diesem Zusammenhang ist die Effizienz der jeweiligen UVC-Anwendung. Der in jedem Testfall genutzte JPEG-Decoder von IBM scheint auf den ersten Blick zeitgemäße Algorithmen zu nutzen [Nau, 2011]. Bei genauerer Betrachtung fallen jedoch Programmstellen auf, die extrem ineffizient implementiert sind. So wird sehr oft in unzweckmäßiger Weise der indirekte Registerzugriff genutzt. Ebenso finden sich kompliziert ausgedrückte Fallunterscheidungen. Nau identifiziert das Unterprogramm zur inversen diskreten Cosinustransformation (iDCT) als größte „Laufzeitbremse:“ Allerdings wird durch seine Ausführungen deutlich mehr Potential offensichtlich, das auch in anderen Unterprogrammen wie dem *DeZigZag* nutzbar wäre. Erste Versuche dahingehend ließen den Schluss zu, dass sich der Decoder mindestens um den Faktor 2 verbessern lassen müsste. Damit werden im Falle des Testbildes Aushebungszeiten zwischen 25 und 35 Sekunden überaus realistisch. Dabei sind mögliche Effekte einer weiterentwickelten Spezifikation noch unberücksichtigt.

6.7.4. Universalität

Die hier durchgeführte *Zeitreise* ist die erste empirische Evaluation der für die Langzeitarchivierung wichtigsten Eigenschaft: Universalität. Der erste Aspekt der Universalität wurde bereits betrachtet, nämlich der annähernd konstant bleibende Implementierungaufwand.

Der zweite Aspekt betrifft die Implementierung an sich. Gezeigt wurde, dass die Implementierung auf den unterschiedlichsten Hardwarearchitekturen gelingt und dabei keine unüberwindbaren Probleme auftreten. Obwohl allein damit ein sehr starkes Indiz für die Universalität gegeben ist, sollen die erkannten Abhängigkeiten kurz dargestellt und bewertet werden. Die Abbildung 6.15 listet diese Abhängigkeiten grob auf.

Die erkannten Abhängigkeiten lassen sich der Hardware, dem Betriebssystem oder der Programmiersprache zuordnen. Dabei ist erkennbar immer die Wortbreite der Basismaschine zu berücksichtigen; aber jedes Mal aufs Neue. Eine Übervorteilung einer bestimmten Wortbreite ist nicht erkennbar. Obwohl die UVC-Spezifikation sehr häufig 32 Bit große Worte referenziert, sind Algorithmen basierend auf 64 Bit großen Speicherworten nicht im Nachteil. Lediglich die Cyber 960 mit der Wortbreite von 60 Bit bescherte etwas mehr Aufwand. Dies lässt sich aber ausschließlich auf die durch die Ganzzahlarithmetik nicht voll nutzbare Wortbreite zurückführen und hat letztendlich nichts mit der Wortbreite an sich zu tun. Eher ins Gewicht fällt, dass sich auch die 60 Bit gut zur Abbildung des bitweise adressierbaren Speichers und der in den Registern zu speichernden Bitfolgen eignen.

Das Betriebssystem hatte in nur einem Punkt direkten Einfluss. Die Spezifikation definiert das Format der Archivmodule sehr genau. Ein solches Modul enthält die Bitfolge einer archivierten UVC-Anwendung. Wie aber ist diese Bitfolge in Record-orientierten Datensätzen abzulegen? Das Problem wurde schließlich durch Migration der Archivmodule gelöst, wobei nur ein kleiner einmaliger Mehraufwand zu leisten war. Erkennbar an dieser Problematik ist aber, dass eine Bitfolge auf allen wortbasierten Systemen nur eingeschränkt speicherbar ist. Die Spezifikation macht weder eine Aussage darüber, wie die oft notwendigen Füllbits im letzten Speicherwort gesetzt sein sollen, noch ob sie erlaubt sind. Diesem Aspekt wird in der

System	Zeit	Hardware	Betriebssystem	Programmiersprache
CDC 6600 / CDC Cyber 960 NOS V2 FORTRAN IV	1964	• 60 Bit Wortlänge, aber 48 Bit Integer Arithmetik ➡ viele intrinsische Funktionen wirkten Portierung entgegen	• NOS ist gewöhnungsbedürftig • geht nicht ohne Anleitungen • Record orientierte Speicherung ➡ Migration!	• keine dynamische Speichernutzung • viele Labels und GOTOs • keine Pakete, e.g. für Arithmetik großer Zahlen
CDC 4680 EP/IX FORTRAN 77	1988 1990	• 32 Bit Wortlänge, unterstützt nur 64 Bit Ergebnisse (Multiplikation, Spezialregister, aber langsamer Zugriff) ➡ FORTRAN Spracherweiterung ermöglicht leichte, aber langsame Portierung	• komfortables und vertrautes Unix-ähnliches System	• wortkarger Compiler • Bug bei Loop-Optimierung • unterstützt 64 Bit Arithmetik, (langsam per Software) • Keine effiziente Implementierung ohne Erweiterungen • packages wie MPFUN77 sind nicht kompatibel mit den Registern des UVC • kein bitweiser Dateizugriff
Sun e10k Solaris 10 FORTRAN 95	1997	• volle 64 Bit Arithmetik ➡ einfache Portierung • Konzept des UVC sieht Nutzung paralleler Processoren nicht vor ➡ viele Instanzen möglich…	• gut dokumentiertes Unix ➡ einfache Nutzung	• unterstützt Pointer und dynamische Speichernutzung • keine kompatiblen Pakete für Arithmetik großer Zahlen • Unterstützung von Binärdateien nur mit FORTRAN 2003 Erweiter.

Abbildung 6.15.: Übersicht der aufgetretenen Probleme und deren Ursachen.

weiterentwickelten Spezifikation Rechnung getragen.

Die letzte Spalte in Abbildung 6.15 führt die erkannten Programmiersprachenabhängigkeiten auf. Diese sind oft für einen Mehraufwand bei der Implementierung effizienter Algorithmen verantwortlich. Wobei das nur rückblickend auffällt. In der Zeit von FORTRAN III und IV z.B. war es üblich, sich bei einer notwendig dynamischen Speichernutzung selbst um die Organisation zu kümmern. Auch in diese Kategorie fällt die lange Zeit fehlende Unterstützung zur Verarbeitung von Binärdateien durch FORTRAN.

Möchte man einen Trend hinein orakeln, dann folgenden: Mit zunehmender Zeit, die eine Implementierung zurückliegt, ist sie ineffizienter bzw. bedurfte eines höheren Implementierungsaufwands für vergleichbare Ergebnisse, sofern die dazu notwendigen Techniken schon entdeckt wurden. Für die Zukunft kann das Zweierlei bedeuten. Zum einen könnten immer effizientere Implementierungen mit vergleichbarem Aufwand erstellt werden. Neue Algorithmen und Techniken könnten mit der Zeit von Programmiersprachen mit steigendem Abstraktionsgrad immer leichter nutzbar werden. Ein Blick auf die sehr umfangreich nutzbare Funktionalität in Java bestärkt diese Annahme. Weiterentwicklungen wie Scala lassen erahnen, dass neue Programmiersprachen nicht mit weniger Grundfunktionalität ausgestattet werden.

Zum anderen könnte man durch die eher böswillige Interpretierung der Ergebnisse zu der Schlussfolgerung kommen, dass Implementierungen immer ineffizienter und aufwendiger werden, umso mehr wir uns vom heutigen Zeitpunkt und der aktuellen Technik entfernen. Schließlich wurde die Spezifikation in unserer Zeit entwickelt und die Probleme entstehen genau dort, wo wir von den aktuellen Normen abweichen, also z.B. von Record-basierten Dateizugriffen und starren Zuordnungen von Speicher zu abzuarbeitenden Job.

7. Untersuchung des Crosscompiler-Ansatzes

Für die Ausführung archivierter UVC-Anwendungen kann in ferner Zukunft in angemessener Zeit eine UVC-Laufzeitumgebung implementiert werden. Im übertragenem Sinne „baut“ man zunächst die Hardware, um dann die Software auszuführen. Aber das ist nicht der einzige Weg, um Programme ausführen zu können. Ein weiterer Ansatz ist die Crosskompilierung. Hierbei wird das Programm vor der Ausführung in ein Programm überführt, das die Befehle der aktuellen Hardware nutzt. Dabei wird jeder ursprünglich enthaltene Befehl des Programms mithilfe von ein oder mehreren Befehlen der aktuellen Hardware abgebildet.

Die Technik der Crosskompilierung ist nichts Neues, aber die Anwendbarkeit für UVC-Anwendungen wurde bisher noch nicht untersucht. Dieses Kapitel stützt sich im Wesentlichen auf die durch den Autor der vorliegenden Arbeit betreuten Masterarbeit von Müller [2011] ab. Er untersuchte, ob sich mit diesem Ansatz UVC-Anwendungen effizienter ausführen lassen und welcher Aufwand dazu notwendig ist.

Dieses Kapitel stellt im Folgenden den auf den UVC übertragenen Crosscompiler-Ansatz vor und geht auf die damit verbundenen Schwierigkeiten ein. Die Ergebnisse von Müller hinsichtlich der untersuchten Optimierungsmöglichkeiten bilden den zweiten Teil. Anschließend werden diese Ergebnisse mit Hinblick auf die anvisierte Nutzung des UVC im Bereich der Langzeitarchivierung bewertet.

7.1. Crosskompilierung von UVC-Anwendungen

Ein wesentlicher Unterschied zwischen dem bisher betrachteten Interpreteransatz, der erst mit der Ausführung den aktuell abzuarbeitenden Opcode liest und auswertet, und dem Crosskompilieren ist die Unvereinbarkeit mit selbstmodifizierendem Code.

Bevor auf die eigentliche Übersetzung der UVC-Anwendungen und das damit mögliche Optimierungspotential eingegangen wird, muss zunächst geklärt werden, was selbstmodifizierender Code ist und ob er tatsächlich innerhalb von UVC-Anwendungen sinnvoll zum Einsatz kommen kann.

7.1.1. Selbstmodifizierender Code

Die Möglichkeiten des UVC, selbstmodifzierenden Code korrekt auszuführen, wurde bereits im Abschnitt 5.3.3 ergründet. Dabei wurde stillschweigend vorausgesetzt, dass das sich dahinter verbergende Prinzip bekannt ist. Zum Verstehen dieses Abschnitts und der Ableitung

der Ergebnisse ist es jedoch zwingend erforderlich, sich mit der Thematik eingehender zu befassen. Das Schaffen einer gemeinsamen Grundlage ist Aufgabe dieses Abschnitts.

Die Befehle eines jeden Programms sind in irgendeiner Weise codiert und gespeichert. Man spricht hier oft vom Bytecode. Wird dieser Code so im Speicher einer Maschine abgelegt, dass dieser während der Programmausführung durch die Abarbeitung der codierten Befehle manipuliert werden kann, ist selbstmodifizierender Code möglich. Programme können sich dann selbst während der Ausführung umschreiben. Im weitesten Sinne ist das gängige Praxis: Ein Betriebssystem kann zur Laufzeit immer wieder neue Programme laden und starten. Dabei werden diese Programme im Speicher an Positionen geschrieben, an denen zuvor andere Programme standen. Als Ganzes betrachtet gehört das auch zur Selbstmodifikation. Für gewöhnlich sieht man es aber enger und meint die Möglichkeit eines Programms, auch den Opcode des direkt folgenden Befehls ändern zu können.

Nicht in allen Architekturen ist das so vorgesehen. Während Von-Neumann-Architekturen das Ablegen von Daten und Programmen im selben Speicher vorsehen, werden eben diese bei der Harvard-Architektur explizit separiert und nur mit physisch getrennten Bussen zugreifbar [Dal Cin, 1996]. Durch eine solche Trennung wird selbstmodifizierender Code unmöglich.

Der UVC ist aber so spezifiziert, dass der Bitcode explizit im Speicher der Segmente abzulegen ist. Eine Begründung fehlt ebenso wie die durchdachte Anwendung dieses „Features." So ist zwar die Zuordnung des Segments zum Unterprogramm gefordert, das den passenden Bitcode enthält, aber die Umsetzung ist nicht spezifiziert. Zu erwarten wäre eine vereinbarte Segmentnummer, über die der Zugriff möglich wird, z.B. die Nummer 3. Würde man diese Angabe noch ergänzen, könnte ein Unterprogramm auf den eigenen Code Einfluss nehmen. Neue Unterprogramme zu erzeugen und aufzurufen wäre ihm dagegen verwehrt. Vergleichbares wäre vielleicht mit Unterprogrammen möglich, die in den *shared* Segmenten abgelegt sind. Deren Anzahl ist jedoch begrenzt. Zudem ist das nur eine mögliche Interpretation der Spezifikation.

Deutlich wird die unausgereifte Unterstützung der Selbstmodifikation an Müllers umfangreichen Experimenten. Dabei erwies sich von beiden IBM Implementierungen nur die C++ Implementierung (siehe 3.4.1) als sehr eingeschränkt dazu fähig.

Müller stellt in seiner Arbeit selbstmodifizierenden Code als etwas Negatives dar. Auf heutigen Systemen werden oft laufende Programme bzw. Dienste mit diese Technik manipuliert und anschließend missbraucht [Dinaburg et al., 2008]. Malware nutzt die Möglichkeiten des selbstmodifizierenden Codes in den unterschiedlichsten Ausprägungen. Argumentiert man aus Sicht der Computersicherheit, fragt man sich schnell, ob der UVC so etwas braucht.

Aber es gibt auch positive Seiten dieser Technik, die sich behaupten konnte, weil viele alte Computer eine Von-Neumann-Architektur haben. Speicher war früher extrem knapp. Ein gängiger Anwendungsfall ist die Manipulation des Programmflusses [Bala et al., 2000]. Sollte sich ein Programm zu bestimmten Zeiten abweichend verhalten, konnte man das mit bedingten Verzweigungen oder aber durch Überschreiben (*patchen*) des Programmcodes erreichen. Letzteres spart zum einen Speicherplatz – weshalb gern auf die Raumfahrt verwiesen wird[1] – und zum anderen Laufzeit, da die bedingte Verzweigung entfällt. Während beim UVC der

[1] Z.B. in dem Artikel von C.E. Ortiz: On Self-Modifying Code and the Space Shuttle OS
weblog.cenriqueortiz.com/computing/2007/08/18

Codeumfang eine weniger wichtige Rolle spielt, so sind Laufzeitaspekte bedeutend.

Ähnlich wie Bala und Ortiz argumentiert Hapgood [2001], der aus Sicht eines Spieleentwicklers diese Technik nutzt, um Effizienzsteigerungen z.B. bei Grafikanwendungen zu erzielen. Unter seinen Beispielen finden sich einige, die auch für den UVC umsetzbar wären.

Neben der Optimierung gibt es z.B. im Bereich der künstlichen Intelligenz Anwendungsfälle, die diese Technik als unabdingbar erscheinen lassen [Schmidhuber, 2005; Nordin und Banzhaf, 1995]. Hierbei geht es um die Möglichkeit, dass sich Code evolutionär verändern darf. Z.B. können Programme mit zunehmender Laufzeit Informationen sammeln und dabei lernen. Dieses Lernen wurde ihn einmalig vorab implementiert. Selbstmodifizierender Code würde es solchen Programmen auch ermöglichen, die Lernalgorithmen selbst anzupassen.

Wie dargestellt gibt es Für und Wider zu dieser Technik. Nachdem die Spezifikation des UVC im Prinzip erste Voraussetzungen schafft, aber dann wiederum die gezielte Nutzung verwehrt, stellt sich die Frage, ob in einer weiterentwickelten Spezifikation diese Technik berücksichtigt werden sollte. Diese Frage wird am Ende dieses Kapitels noch einmal aufgegriffen.

7.1.2. Voraussetzungen für das Crosskompilieren

Um ein bereits kompiliertes Programm in eine andere Sprache übersetzen zu können, sind bestimmte Voraussetzungen erforderlich. Umso mehr Informationen extrahiert werden können, desto besser kann die Umsetzung für verschiedene Systeme gelingen.

Nehmen wir an, wir wissen, dass das betreffende UVC-Programm lediglich 10 Register nutzt und während der Laufzeit Registerinhalte nie mehr als 32 Bit belegen. Dann können diese Register auf bestimmten Zielmaschinen direkt abgebildet werden. Die wenigen Operationen des UVC lassen sich dann problemlos abbilden und vor allen äußerst schnell ausführen. Ebenso verhält es sich mit Sprungmarken und konstanten Registerinhalten. Kennt man die entsprechenden Stellen im Programm, die angesprungen werden können, und zudem den Programmfluss, lassen sich Verzweigungen und Unterprogramme mit dem Befehlssatz der Zielmaschine direkt abbilden. Gefundene Konstanten können ebenfalls in beinahe jedem Befehlssatz effizient abgebildet werden. Dieser Gedanke war Motivation für Müllers Arbeit.

Verfügbare Techniken, mit denen derartige Informationen aus kompilierten Programmen extrahiert werden können, wurden für die aktuelle Hardware wie die X86-Architektur entwickelt [Cifuentes, 1994; Kinder, 2010]. Dabei spielen die begrenzte Registeranzahl und die überwiegend statischen Sprungziele eine wichtige Rolle. Der UVC dagegen hat unbegrenzt viele Register und kennt ausschließlich potentiell dynamische Sprungziele. Erschwerend kommt hinzu, dass der UVC auf Register auch indirekt zugreifen kann. Diese dem UVC eigene Zugriffsart ermöglicht effiziente UVC-Anwendungen.

Der Algorithmus von Kinder verfolgt einen interessanten Ansatz. Für jede Anweisung im Programm wird für die genutzten Register protokolliert, ob sich deren Inhalt verändert. Die konkrete Änderung ist dabei unerheblich. So werden schnell konstante Register identifiziert. Die veränderlichen Register gelten als unbestimmt. Im crosskompilierten Programm lassen sich so Operationen mit identifizierten konstanten Registerinhalten effizient abbilden und ggf. auch aus Schleifen herausziehen.

Allerdings stört hierbei der eigentlich äußerst sinnvolle indirekte Registerzugriff. Dieser wird verwendet, um auf andere Register des gleichen Segments über die im angegebenen

Register gespeicherte Nummer zuzugreifen. Dabei entfaltet dieses Verfahren erst dann sein volles Potential, wenn sich diese Nummer zur Laufzeit ändert, also z.B. als Listenindex fortschaltet oder als Zeiger zum Traversieren eines Baums dient. Bezieht sich allerdings auch nur ein indirekter Registerzugriff auf ein nicht als konstant identifiziertes Register – was die Regel sein sollte – so gelten damit alle Register in diesem Segment als unbestimmt. Konstante Registerinhalte lassen sich in dem betroffenen Segment nicht mehr finden. Laufzeitvorteile sind dann kaum mehr möglich. Wirklich tragisch ist in diesem Zusammenhang, dass Register des betroffenen Segments Sprungmarken enthalten können. Diese können nicht mehr als konstant identifiziert werden, eine Rekonstruktion des Kontrollflusses wird somit scheitern. Als Folge davon wäre das Unterprogramm lediglich Anweisung für Anweisung umsetzbar und jede Anweisung wäre potentielles Sprungziel. Das fordert von der Zielmaschine eine beinahe vollständige Implementierung des UVC. Was erspart bleibt, ist die Implementierung des Bitcode-Interpreters und auch nur die Laufzeit, die zur Interpretation des aktuellen Opcodes erforderlich ist. Die Arbeit von Müller stellt diesen Zusammenhang sehr anschaulich dar.

Für die weiteren Abwägungen sei bereits hier festgehalten, dass der Ansatz der Crosskompilierung nicht für alle und sehr wahrscheinlich nur für wenige UVC-Anwendungen mit einem echten Laufzeitgewinn umsetzbar ist.

7.2. Ergebnisse des Prototyps

> *It is the case that code in compiled form executes considerably faster than interpreted code, with interpreted code running at one or two orders of magnitude slower than the corresponding compiled form.* [Craig, 2006]

Dieses Zitat ist die ursprüngliche Motivation für Müllers Arbeit. Oben wurde bereits skizziert, warum das vollständige Dekompilieren von UVC-Anwendungen und die Rekonstruktion des Kontrollflusses scheitern können. Eines geht aber fast in jedem Fall: Die Übersetzung der einzelnen Anwendungen, um die Laufzeit zu sparen, die der Interpreter an sich benötigt.

Erste Ergebnisse dahingehend waren überraschend positiv ausgefallen, mussten jedoch „normiert“ werden. Müllers Prototyp basiert auf der Implementierung in C++ von Tamminga. Dabei wurden erhebliche Designfehler ersichtlich, die arg das Ergebnis verzerrten. Der folgende Abschnitt wird diese und deren Behebung durch Müller kurz vorstellen. Im Folgenden werden dann die realistischeren Ergebnisse dargestellt.

7.2.1. Anpassung der Ausgangsbasis

Das gängige Szenario sieht die Implementierung des UVC in einer Hochsprache vor. So lassen sich die jeweils unbegrenzten Strukturen ohne zu großen Zeitaufwand realisieren. Nicht anders verhält es sich mit dem Crosscompiler. Dieser wird keinen Maschinencode für eine spezielle Zielmaschine erzeugen, sondern Programmcode für eine Hochsprache. Zur Wahl standen Ada83 und C++, da hierfür jeweils Implementierungen als Quellcode zur Verfügung standen. Müller entschied sich für die ihm vertraute Sprache C++.

Müller konnte so auf eine bereits vorhandene Basisimplementierung aufsetzen und die Implementierungen für Segmente, Register und Prozessor nutzen. Lediglich der Interpreter war durch den generierten Programmtext zu ersetzen. Um Sprünge im laufenden Programm zu jeder Anweisung zu ermöglichen, wurde pro Anweisung ein Label eingefügt und als Referenz in einem Array gespeichert. So kann die direkte Befehlsabfolge sehr schnell umgesetzt werden, Sprünge dagegen geringfügig langsamer als in der interpretierten Variante.

Die ersten Laufzeitergebnisse ließen eine Verbesserung um den Faktor 7 und höher erkennen. Müller nutzte hierfür unverändert die Segmente und deren Register mit den jeweils implementierten Funktionen. Im Unterschied zur Implementierung von Tamminga wurde jedoch die Segmentverwaltung zweckmäßiger umgesetzt. Tammingas Implementierung zeugt an dieser Stelle von einem äußerst schlechten Design, was sehr wahrscheinlich der irreführenden Verwendung der Begriffe „physisch“ und „logisch“ innerhalb der Spezifikation zuzuschreiben ist. Die als unbefriedigend einzustufende Implementierung wurde in Abschnitt 3.4.1 vorgestellt.

Nicht nur Müller fragte sich, warum ein in C++ implementierter UVC deutlich langsamer ist als die Implementierung in Java. Zu erwarten wäre eigentlich ein Laufzeitvorteil ähnlich der mit Ada erzielten Größenordnung. Zunächst optimierte Müller daher den UVC von Tamminga an den für ihn relevanten Stellen. Er optimierte auch für diese Version die Segmentverwaltung und implementierte eine vernünftige Iteration über dem im Segment gespeicherten Bitcode, basierend auf der effizienten Nachfolgersuche im sortierten ausgewogenen Baum. Den Interpreter optimierte er dadurch, dass er die umfangreiche Switch-Case-Anweisung mit einer Sprungtabelle realisierte, was nach Craig [2006] einer zeitgemäßen Implementierung entspricht. Die Segmente selbst und auch die Register bergen noch erhebliches Potential. Diese Version des UVC verbessert sich bereits um den Faktor 4.

7.2.2. Ergebnisse des Prototyps in Relation

Vergleicht man die Laufzeitergebnisse des Prototyps, der zunächst nur Befehl für Befehl umsetzt, mit denen des jetzt angepassten UVC in C++, so bestätigt sich oben zitierte Aussage von Craig mit einem Faktor von 1,2 bis 2,5 je nach ausgeführter Anwendung (siehe Abb. 7.1). Daraus leitet sich ein erstes Ergebnis ab: Craigs Aussage gilt uneingeschränkt für den UVC.

Durch die für den UVC angepasste Implementierung des Algorithmus von Kinder [2010] lassen sich Kontrollflussgraphen von vorhandenen UVC-Anwendungen rekonstruieren. Dabei kann es zu Problemen kommen. Der Prototyp konnte nur für eine der drei großen verfügbaren UVC-Anwendungen, dem GIF-Decoder, den Kontrollflussgraphen korrekt rekonstruieren. Mit leichten Veränderungen, die allerdings die korrekte Umsetzung gefährden können, konnte auch der Kontrollfluss der SpreadSheet-Applikation rekonstruiert werden. Auch gelang das für den JPEG-Decoder, der jedoch infolgedessen leicht verfälschte Ausgaben produziert.

Ausgehend vom rekonstruierten Kontrollflussgraphen sind weitere Optimierungen möglich, die zum Teil mit dem Prototypen realisiert wurden. Dabei sind die als statisch identifizierten Sprungziele und Unterprogrammaufrufe direkt im erzeugten Programmcode integriert. In einem weiteren Schritt wurden als im relevanten Programmabschnitt statisch identifizierte Operanden direkt im erzeugten Programmcode eingearbeitet. Dazu ein Beispiel:

```
DIV     0,1   0,2   0,3
```

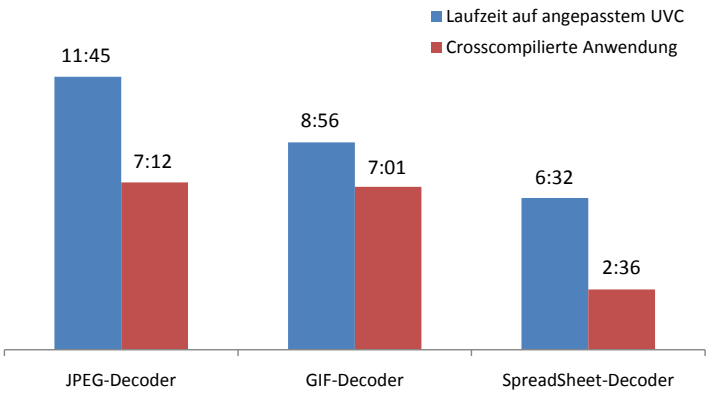


Abbildung 7.1.: Laufzeiten der drei UVC-Anwendungen auf dem durch Müller angepassten UVC von Tamminga im Vergleich zur vorab crosskompilierten Version.

Die auszuführende Division wird vom Prototyp auf eine interne Prozedur abgebildet:

```
DIV(r0_1, r0_2, r0_3);
```

Die hier verwendete Art des Registerzugriffs ist bereits optimiert. Eine solche Optimierung ist möglich für Register von *shared* Segmenten, auf die nicht indirekt zugegriffen wird (siehe hierzu auch Abschnitt 5.3.4).

Ist durch den Algorithmus klar, dass an dieser Stelle im Programm das Register `2` immer den Wert `4` hat, kann die Umsetzung bereits so aussehen:

```
DIV(r0_1, 4, r0_3);
```

Das Einarbeiten der konstanten Operanden spart Laufzeiten durch den wegfallenden Registerzugriff und einer möglichen Vorwegnahme von Fallunterscheidungen. So muss zur Laufzeit nicht mehr geprüft werden, ob der Wert `4` negativ ist.

Ein letzter umgesetzter Schritt ist die Identifizierung von schnelleren Instruktionen, die das gleiche Ziel erreichen. In diesem Fall kann die Division durch 4 durch ein Rechts-Schieben der einzelnen Bits um 2 Stellen erreicht werden. Auch dieses Potential erkennt und nutzt der Prototyp und erzeugt folgenden Code:

```
SHFTR(r0_1, 2, r0_3);
```

Die in diesen Schritten erreichbaren Laufzeiten sind in Abbildung 7.2 aufgeführt.

7.3. Bewertung des Crosscompiler-Ansatzes

Interessant ist zunächst, wie Müller diesen von mir auferlegten Ansatz selbst bewertet, denn er bereitet seine Bewertung intensiv vor. Zunächst erweitert er den vorhandenen UVC um die Möglichkeiten, die durch den Crosscompiler erst nachträglich geschaffen werden. Das heißt konkret, dass er den Bitcode vor der Ausführung analysiert und in eine Struktur überführt, die eine effiziente Interpretierung erlaubt. Das Verfahren wird auch in der Referenzimplementierung genutzt, um die Ausführungsgeschwindigkeit zu erhöhen (siehe 5.3.4), wobei er

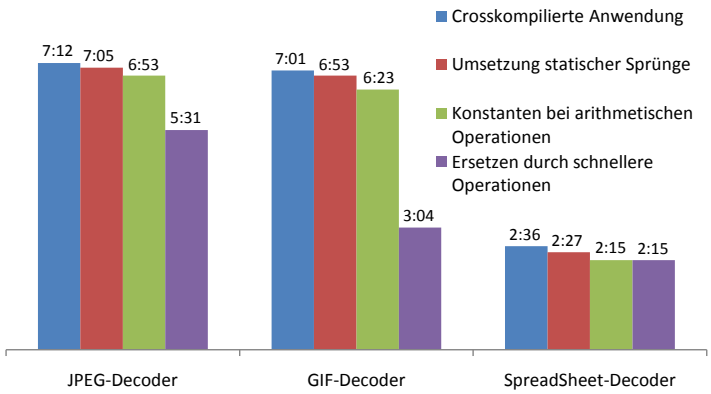


Abbildung 7.2.: Laufzeiten der drei unterschiedlich stark optimierten UVC-Anwendungen

nicht ganz so weit geht. Er erweitert zudem den UVC um weitere Instruktionen, damit dieser mit konstanten Operanden bei den vier arithmetischen Operationen umgehen kann und die zwei zusätzlichen Schiebeoperationen kennt. Den Quellcode der drei vorhandenen komplexen UVC-Anwendungen hat er dazu an allen Stellen, die der Crosscompiler optimiert, per Hand geändert. Einen zu diesem Zweck erweiterten Assembler stellte ich ihm zur Verfügung. Auf diese Weise kann der tatsächliche Gewinn durch den Crosscompiler ermittelt und dem gegenübergestellt werden, was allein durch eine Erweiterung der Spezifikation möglich wäre. Die Ergebnisse sind in Abbildung 7.3 dargestellt.

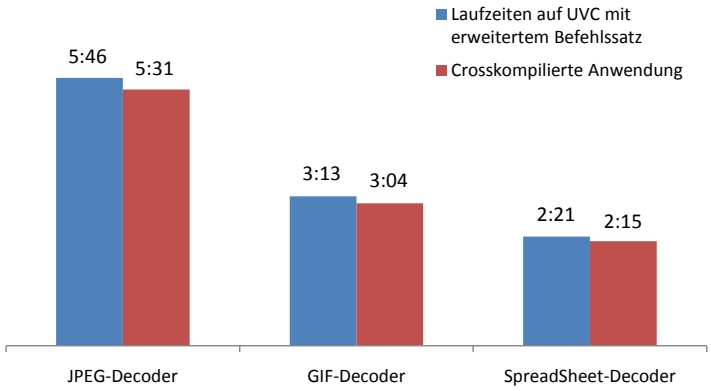


Abbildung 7.3.: Die erzielten Laufzeiten der drei UVC-Anwendungen im direkten Vergleich

Ein Aspekt, den Müller an der Stelle übersieht, ist, dass mit dem zweiten Ansatz selbstmodifizierender Code nicht länger ausgeschlossen ist. So stellen Smith und Nair [2005] fest, das selbstmodifizierender Code kein Ausschlusskriterium, sondern eher eine seit längerem bekannte Herausforderung für die Entwicklung von Emulatoren und virtueller Maschinen ist. So ist es leicht möglich, schreibende Zugriffe auf die Segmente zu erkennen, in denen der Bitcode abgelegt ist. Nach einer solchen Operation müsste der Bitcode bzw. der entsprechende Bereich erneut geparst und die intern aufbereitete Struktur angepasst werden.

Nach wie vor gilt die zitierte Aussage von Craig, auch wenn der jetzt noch erzielte Faktor von knapp 1,05 nicht wirklich überzeugt. Dies allein kann somit kein Grund mehr sein, selbstmodifizierenden Code zwingend auszuschließen. Die weiterentwickelte Spezifikation gestaltet daher die bereits geforderte Speicherung des Unterprogrammcodes im bitadressierbaren Speicher der Segmente zweckmäßig aus, um selbstmodifizierenden Code zuzulassen.

Den Aufwand zur Implementierung des Algorithmus nach Kinder gibt Müller mit über zwei Wochen an. Dazu kommen die einzelnen Optimierungen, die jedoch nur anwendbar sind, wenn der Kontrollfluss rekonstruiert werden kann. Das Arbeiten mit Heuristiken, die nur unzuverlässig richtige Ergebnisse liefern, ist im Bereich der Langzeitarchivierung nicht vorstellbar. Im Vordergrund steht stets das Archivieren korrekter und authentischer Informationen.

Der vollständige Implementierungsaufwand für einen einfachen Crosscompiler beträgt mehr als vier Wochen, die zusätzlich zu dem Implementierungsaufwand des UVC zu leisten sind. Denn der Crosscompiler nutzt bereits fertige Strukturen, die zunächst geschaffen werden müssen: Register und Segmente mitsamt der geforderten Funktionalität und dem unbegrenzt möglichen Wachstum. Deren Implementierungszeit ist bestimmend und der verbleibende Anteil zur Implementierung des Interpreters ist dagegen beinahe vernachlässigbar.

Von Müller nicht betrachtet wurden weitere Codeoptimierungen, die mit dem erfolgreich rekonstruierten Kontrollflussgraphen möglich werden. Es stellt sich aber die Frage, ob es wirklich zweckmäßig ist, heute Programme zu entwickeln, die in ferner Zukunft aufwendig optimiert werden müssen, damit sie sinnvoll nutzbar werden. Diese Frage ist in Bezug zur angedachten Archivierungsmethode überaus berechtigt. So ist der Grundgedanke eben dieser, bereits heute so viel wie möglich des notwendigen Aufwandes zu leisten, damit der Zugriff auf die archivierten Informationen in ferner Zukunft spielend gelingt.

Die Schlussfolgerung kann daher nur sein, bereits heute dafür zu sorgen, dass UVC-Anwendungen optimiert programmiert werden können und keiner nachträglichen Optimierung mehr bedürfen. Die Verwendung konstanter Sprungziele und konstanter Operanden bei den arithmetischen Ausdrücken sind dabei als notwendiges Kriterium von Müller herausgearbeitet worden, ebenso die Notwendigkeit zweier zusätzlicher Schiebeoperationen. Eine weiterentwickelte Spezifikation muss als Konsequenz aber deutlich weiter gehen und konstante Operanden generell vorsehen. Und zwar als 32 Bit große Konstanten ohne Vorzeichen. Diese durchziehen die Spezifikation ohnehin und erscheinen auch für zukünftige Szenarien geeignet.

Ein dennoch sinnvoller Einsatz eines Crosscompilers wäre die Übersetzung spezieller UVC-Anwendungen, die fest in andere Anwendungen integriert werden sollen. Für eine solche Integration wären UVC-Programme als Quellcode der gleichen Programmiersprache überaus hilfreich. Schnittstellen könnten so direkt implementiert werden. Im Falle der Bild-Decoder könnten die errechneten Pixelinformationen direkt von der aktuellen Anwendung verabeitet werden, ohne langsame Schnittstellen per `OUT` und `IN` nutzen zu müssen.

Deutlich einfacher würden sich Crosscompiler entwickeln lassen, wenn nicht vom bereits assemblierten Bitcode ausgegangen würde, sondern der Assembler-Quellcode noch verfügbar wäre. Der Kontrollfluss würde sich leichter und vor allem sicherer rekonstruieren lassen. Auch könnte es helfen, wenn UVC-Programme in einer abstrakteren Form, z.B. in einer speziell für den UVC entwickelten Hochsprache formuliert wären. Der Kontrollfluss wäre so im Idealfall direkt vorgegeben und optimierende Compiler für jeweils zeitgemäße Systeme wären denkbar.

Teil III.

Anwendungsentwicklung für den UVC

8. Implementierung einer Turing-Maschine für den UVC

Turing [1936] stellte in seinem Aufsatz eine sehr einfache, aber abstrakte Maschine vor. Diese Maschine verfügt über ein endloses Band. Mit einem Schreib-/Lesekopf können Zeichen eines vordefinierten Alphabetes auf das Band geschrieben und gelesen werden. Abhängig des gelesenen Zeichens kann die Maschine den Kopf um eine Position bewegen, ein Zeichen schreiben und den Zustand wechseln. Über die Angabe aller möglichen Zustände und deren Übergänge werden Programme für diese Maschine beschrieben. Eingabedaten müssen zum Start der Abarbeitung bereits auf dem Band sein und die erzeugte Ausgabe befindet sich ebenfalls darauf.

Mit der Turing-Maschine gibt es eine Maschine, mit der theoretisch alles berechnet werden kann, was mit modernen Maschinen auch berechenbar ist. Aufgrund ihrer einfachen Struktur eignet sie sich besonders gut zur Untersuchung theoretischer Fragestellungen.

Es gibt verschiedene Arten zu zeigen, dass eine Maschine Turing-vollständig ist, also alles berechnen kann, was heutige Computer auch berechnen können. Eine davon ist die Turing-Maschine für diese Maschine zu implementieren. Es gibt mittlerweile noch einfachere Maschinen mit nur einer Instruktion [Nürnberg et al., 2004] und minimalistisch gehaltene Sprachen [Böhm und Jacopini, 1966], die Turing-vollständig sind und noch einfacher als die Turing-Maschine implementiert werden können.

Mit Blick auf den Funktionsumfang des UVC zweifelt wohl niemand ernsthaft an der Turing-Vollständigkeit. Im Vordergrund stand daher die Frage nach dem Aufwand einer Implementierung für den UVC und mit welchen Problemen dabei zu kämpfen ist. Aus diesem Grund wurde in Form einer Bachelorarbeit die Implementierung einer Turing-Maschine beauftragt. Im Folgenden werden zunächst die Eckpunkte der Realisierung beleuchtet und im Anschluss daran die Ergebnisse zusammengefasst.

8.1. Eckpunkte der Realisierung

Die im Folgenden betrachtete Implementierung stammt von Felix Ritscher. Er entschied sich für eine Einband-Turing-Maschine. Das in beide Richtungen theoretisch unbegrenzte Band stellt den Speicher dieser Maschine dar und kann Symbole eines zuvor festgelegten Alphabets aufnehmen. Das Alphabet wird dabei nicht vorab Zeichen für Zeichen angegeben, sondern lediglich durch die Anzahl der Bits spezifiziert, die zur eindeutigen Darstellung benötigt werden. Bei der Eingabe wird darüber hinaus darauf geachtet, dass die Nachrichten an sich keine Begrenzung der Turing-Maschine implizieren würden.

Vom Programm werden sechs Eingabenachrichten in fester Reihenfolge angenommen. Die erste gibt die Anzahl der zur Codierung des Alphabets genutzten Bits an. Die initiale Position

des Bandes wird in der zweiten Nachricht übermittelt. Die dritte enthält die initiale Belegung des Bandes. Dabei spielt die in der ersten Nachricht übermittelte Anzahl eine tragende Rolle. Die Länge der dritten Nachricht sollte daher einem Vielfachen dieser Anzahl entsprechen. Die vierte Nachricht enthält die Anzahl der für die eindeutige Codierung der Zustandsnummern genutzten Bits, die fünfte gibt die Nummer des Startzustandes an und die sechste enthält schließlich die partiell definierte Zustandsübergangsfunktion in Form einer „Tabelle." Dabei könnte es vorkommen, dass eine dieser Nachrichten für den UVC zu groß ist, da dieser gemäß der Spezifikation die Länge einer Nachricht mit 32 Bit codiert erwartet und damit maximal 2^{32} Bit mit einem `IN` verarbeiten kann. In einem solchen Fall werden noch nicht vollständige Nachrichten mit dem Typ `1` gekennzeichnet, abschließende mit `0`.

Bei der Ausgabe nach Abarbeitung wurde ebenfalls auf die Nachrichtenlänge geachtet. Es werden das linke Teilband, das aktuelle Zeichen an der Kopfposition, gefolgt vom rechten Teilband und einer Gesamtzahl der ausgeführten Schritte in fester Reihenfolge und gleicher Weise ausgegeben. Dabei werden die Nachrichtentypen `3` und `4` genutzt.

Die Verwaltung des beidseitig unendlichen Bandes ist mittels zweier Segmente realisiert, wobei jeweils ein Segment den Teil links bzw. rechts vom Kopf enthält. Bewegt sich der Kopf, so wird jeweils das letzte Zeichen von einem Segment hinter das letzte Zeichen im anderen Segment kopiert. Die einzelnen Symbole auf dem Band sind dabei im bitadressierten Speicher dieser Segmente abgelegt; im zweiten Segment folglich in umgekehrter Reihenfolge. Das Band kann auf diese Weise mühelos mit der Bewegung des Kopfes wachsen. Da die Bewegung des Kopfes mit Unterprogrammen realisiert ist, ergibt sich hier die Notwendigkeit, die beiden Segmente aus dem Pool der *shared* Segmente zu nehmen, da nur jeweils ein privates Segment übergeben werden kann. Sollte auf diese Weise eine Mehrband-Turing-Maschine realisiert werden, wäre die Anzahl der genutzten Bänder auf 499 limitiert. Auch die dynamische Initialisierung von Segmenten ist nicht vorgesehen [Krebs et al., 2011a].

8.2. Bewertung

Das zuletzt dargestellte kleine Problem stellt tatsächlich die einzig erkennbare Herausforderung dar, die mit der Implementierung erkennbar wurde. Die Implementierung der Einband-Turing-Maschine gelingt problemlos und ist mit nur 614 kommentierten Programmzeilen recht überschaubar. Mit diesem Experiment ist erstmals die Behauptung von Gladney und Lorie [2005] zur Turing-Vollständigkeite des UVC nachhaltig belegt.

Die Implementierung gibt Grund zur Annahme, dass ein Programmierer in bestimmten Situationen gern mehr als nur ein privates Segment an ein Unterprogramm übergeben würde. Derzeit ist das unmöglich. Die Erweiterung des UVC dahingehend ist jedoch nicht trivial. Man könnte über eine erweiterte Syntax des Befehls `CALL` oder aber über das temporäre Einblenden privater Segmente in den *shared* Bereich nachdenken. Letzteres könnte leicht auf die Ausführung des aktuellen Unterprogramms begrenzt und mit dem dazugehörigem `BREAK` rückgängig gemacht werden. Sollten weitere Anwendungsfälle bekannt werden, die ohne eine solche Funktionalität des UVC nur sehr aufwendig umzusetzen sind, sollten beide skizzierten Möglichkeiten berücksichtigt und hinsichtlich ihrer Auswirkungen auf die Spezifikation des UVC und den damit verknüpften Implementierungsmehraufwand bewertet werden.

9. Ein Assembler für den UVC

In den Kapiteln 5 und 6 wurde untersucht, ob allein die vorhandene Spezifikation des UVC für die Implementierung des UVC auf heutigen Systemen ausreicht. Für das gesamte Konzept des UVC ist jedoch mehr notwendig als nur die Implementierung.

Zwei wesentliche Punkte sind negativ während der eigenen Implementierung aufgestoßen. Zum einen sind es die nicht vorhandenen Testprogramme, mit denen die eigene Implementierung überprüft werden kann. Die Entwicklung geeigneter Testprogramme kostet Zeit, die bereits heute investiert werden könnte und auch sollte. Zum anderen war es schwierig, eigene Testprogramme zu erzeugen, da kein Assembler für den UVC verfügbar war. Dieser Punkt war besonders schmerzhaft, da ohne einen solchen Assembler auch kein Entwickler auf die Idee kommt, eigene Anwendungen für den UVC zu entwickeln, wenn er zunächst die essentiellsten Tools selbst entwickeln muss.

Die in diesem Kapitel erarbeitete Lösung hat zudem das Potential, den UVC-Ansatz gewinnbringend zu erweitern. Es wird auf einfache Art möglich, UVC-Programme nicht nur als Bitstrom sondern auch im lesbaren Quelltext zu speichern – Kommentare eingeschlossen. Das Kapitel beschreibt die Entwicklung eines Assemblers als UVC-Anwendung. Der Detailgrad der Beschreibung ist absichtlich recht hoch, damit weiterführende Arbeiten sehr leicht darauf aufsatteln können, z.B. wenn zu einer weiterentwickelten Syntax Anpassungen notwendig werden. Die mit der Entwicklung dieses komplexen Programms gesammelten Erfahrungen werden am Schluss zusammengefasst.

9.1. Motivation – Ein Assembler auf Knopfdruck

Während der verschiedenen Implementierungen des UVC nach der gegebenen Spezifikation von Lorie und van Diessen [2005] fiel auf, dass Testprogramme benötigt wurden, um die Funktionsfähigkeit zu überprüfen.

Solche Testprogramme können und sollten in bereits assemblierter Form vorliegen und zusammen mit der Entwicklung der Spezifikation erstellt werden, auch um die Intention der Autoren auszudrücken. Solange sich die Spezifikation nicht ändert, bedürfen diese keinen Anpassungen und können zusammen mit dieser (auch in assemblierter Form) archiviert werden.

Eine jeweils individuelle Anpassung der Testprogramme ist aber im Einzelfall wünschenswert. Ist z.B. die Wortbreite der aktuellen Basismaschine eine andere als zur Zeit der Testprogrammentwicklung, könnten hier andere Testfälle, insbesondere bei der komplexeren Arithmetik notwendig sein. Als Beispiel soll die eigene Implementierung der Speicherverwaltung dienen, die Testprogramme notwendig machte, die unmöglich vor dieser Implementierung hätten erahnt werden können.

Der Speicher soll nach Bedarf beliebig wachsen können und an beliebigen Stellen zugreifbar sein. Daraus resultiert eine dynamische Verwaltung von einzelnen Speicherblöcken. Die Größe solcher Speicherblöcke ist bei der Implementierung frei wählbar und richtet sich nicht zuletzt aus Effizienzgründen nach der zugrunde liegenden Basismaschine. Der Bedarf an Testprogrammen besteht bei der Implementierung der Speicherzugriffsbefehle **LOAD** und **STORE**. Diese Befehle müssen über die jeweils resultierenden Blockgrenzen und über mehrere Blöcke hinweg und zudem bitorientiert an jeder beliebigen „krummen“ Adresse getestet werden (siehe Abb. 9.1). Während der Implementierung sind daher verfügbare Assembler äußerst hilfreich, um Testprogramme speziell für diese Blockgrößen zu schreiben oder anzupassen.

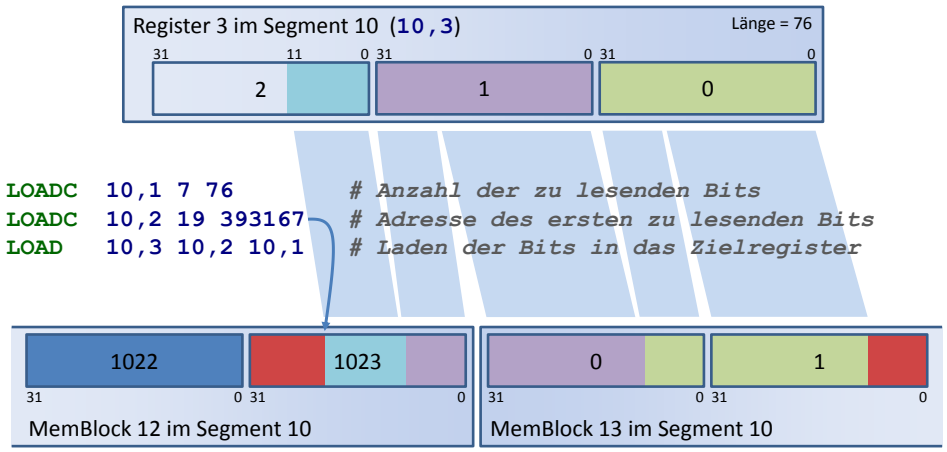


Abbildung 9.1.: Laden von einer Anzahl von Bits über (intern verwendete) Blockgrenzen hinweg. Die dargestellte Befehlssequenz deckt viele Besonderheiten ab: Laden einer nicht zur Wortlänge passenden Anzahl Bits beginnend inmitten eines Speicherwortes (sodass Verschiebungen und Maskierungen an beiden Enden notwendig werden – rote Bereiche) und über die Grenze eines Speicherblocks hinweg.

Der UVC soll darüber hinaus keine Eintagsfliege sein. Das heißt, er wird auch in ferner Zukunft nicht nur dazu benutzt, heutiges Wissen zugreifbar zu erhalten, sondern auch um zukünftiges Wissen langzeit-verfügbar zu archivieren. Damit das gelingt, bedarf es der Möglichkeit, fortwährend neues Wissen ebenfalls durch UVC-Anwendungen zu erhalten. Hierzu ist es unabdingbar, jederzeit UVC-Anwendungen entwickeln zu können. Für heutige Systeme entwickelte Assembler oder Compiler erfüllen diesen Zweck nicht.

Dieser Abschnitt beschreibt die verfügbaren Techniken zur formalen Beschreibung eines Compilers. Formale Beschreibungen wie reguläre Ausdrücke und kontextfreie Grammatiken können mit einem Compiler-Compiler zur automatisierten Erzeugung eines funktionsfähigen Compilers genutzt werden. Neben diesen Techniken wird eine formale Beschreibung des UVC-Compilers vorgestellt, die an sich zeitlos interpretierbar in „Stein gemeißelt“ werden könnte. Jedoch bedarf es jeder Zeit eines Compiler-Compilers.

Ein Grundgedanke im Konzept des UVC ist die Verlagerung des Aufwandes soweit wie möglich in die heutige Zeit. Heutiges Wissen wird durch die Entwicklung aufwendiger Pro-

gramme erhalten, damit später sehr einfach darauf zugegriffen werden kann. Dieser Idee folgend beschreibt dieser Abschnitt, wie durch ein UVC-Programm selbst die Fähigkeit erhalten wird, zukünftig weitere UVC-Programme zu erstellen. Den Schlüssel hierzu stellt ein in UVC-Assembler-Sprache geschriebener UVC-Assembler dar.

9.1.1. Die Techniken der Compiler-Compiler

Bisher wurden die Begriffe Compiler und Assembler genutzt, ohne sie voneinander abzugrenzen. In der Realität können Assembler sehr einfach implementiert werden. Diese müssen die Befehlswörter erkennen, den korrekten Opcode zuweisen, die nachfolgenden Parameter in Binärfolgen codieren und hintereinander hängen. In einem gesonderten Durchlauf werden Sprung- und Speichermarken berechnet und an die Stelle zuvor eingebauter Platzhalter geschrieben. Compiler können dagegen wesentlich mehr. Unter anderem eignen sie sich zur Übersetzung von komplexeren Anweisungen, wie sie in den verschiedensten Hochsprachen vorkommen. Eine Compilerspezifikation ist im Grunde für die UVC-Assemblersprache nicht notwendig, aber zukunftsweisend. So kann sie bei Bedarf erweitert werden, z.B. um Hochsprachenkonstrukte wie Schleifen, Variablen oder Funktionen. Ein „zusammengehackter“ Assembler erfüllt diese Ansprüche nicht.

Die Geschichte der Compiler-Compiler begann bereits mit dem Wunsch, Programmiersprachen einheitlich zu spezifizieren. Das war die Zeit, in der Sprachen wie FORTRAN und ALGOL entstanden. Nur Sprachen, die spezifiziert waren, konnten von einer großen Anzahl von Programmierern erlernt und angewendet werden. Diese Zahl war umso größer, je verbreiteter diese Sprache über die verschiedensten Systeme war. Dazu war es unerlässlich, sich auf gemeinsam geltende Standards zu einigen. Einen solchen legte z.B. John W. Backus mit der Spezifikation der Sprache ALGOL 60 [Backus et al., 1963]. Neben der Sprache selbst wurde vor allem mit der genutzten Metasprache ein noch heute genutzter Standard gesetzt: Die Backus-Naur-Form, kurz BNF.

Ein Compiler besteht zumeist aus insgesamt drei Komponenten: Scanner, Parser und Evaluator (siehe Abb. 9.2). Die lexikalische und syntaktische Struktur einer Programmiersprache, somit auch die der UVC-Assemblersprache, können durch reguläre Ausdrücke und kontextfreie Grammatiken in der Compiler-Definition festgelegt werden. Das Ergebnis, z.B. eine Übersetzung in eine Zielsprache, wird durch Attributierung und anschließender Auswertung bestimmt. Die Verwendung eines Compiler-Compilers erleichtert nicht nur das Erstellen einer Compiler-Definition, sondern auch nachfolgende Modifikationen [Schmitz, 1995].

Die zuvor erwähnte BNF dient der Notation von kontextfreien Grammatiken. Die darin enthaltenen Token (benannte Gruppierung von Zeichenfolgen) werden zuvor von einem Scanner erkannt. Dieser Scanner wird über reguläre Ausdrücke definiert. Nachdem die einzelnen Bestandteile erkannt, also benannt wurden, erzeugt der Parser daraus, basierend auf einer gegebenen kontextfreien Grammatik, einen Syntaxbaum, der schließlich ausgewertet, also evaluiert werden muss. Diese Aufgabe übernimmt der Evaluator (in Abb. 9.2 Attributauswerter). Dieser attributiert in einem ersten Schritt diese Grammatik, d.h. er ordnet den Nichtterminalen Attribute zu. Deren Werte werden während der Evaluierung bestimmt. Schwierigkeiten an dieser Stelle entstehen durch uneinheitliche formale Definitionen für die einzelnen Evaluationsschritte. Attribute können die verschiedensten Typisierungen haben, z.B. Zahlen oder

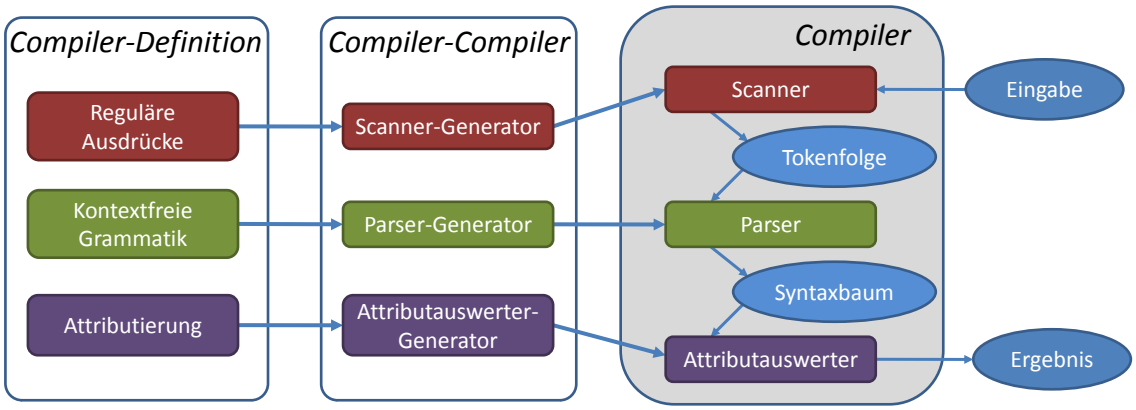


Abbildung 9.2.: Bestandteile eines Compilers: Scanner, Parser und Attributauswerter (Evaluator) und die jeweiligen (Teil-)Ergebnisse.

Zeichenketten beinhalten, und die kompliziertesten Berechnung zu deren Auswertung notwendig machen. Heutige Evaluatoren lassen daher sehr oft für die Definition der Aktionen zur Auswertung geläufige Programmiersprachen zu, die zur Laufzeit übersetzt werden. Entgegen den zeitlosen regulären Ausdrücken und der kontextfreien Grammatik sind diese Definitionen zur Auswertung zeitlich gebunden, da nicht sichergestellt werden kann, dass und wenn ja welche Programmiersprachen zeitlos erhalten bleiben. In dieser Tatsache ist ein weiteres Motiv für die heutige Implementierung eines Compilers für den UVC in UVC-Assemblersprache zu finden.

Mit dem Begriff „Compiler-Compiler“ verband sich zunächst die Vorstellung aus einer gegebenen Sprachspezifikation und einer Spezifikation einer Zielmaschine automatisiert einen Compiler zu erzeugen, der ein Programm in der gegebenen Sprache in ein ausführbares Programm der Zielmaschine übersetzt. Von dieser Vorstellung sind heutige Compiler-Compiler noch weit entfernt.

Frei verfügbare Compiler-Compiler sind z.B. SIC (Smaltalk-basierter, interaktiver Compiler-Compiler) [Schmitz, 1992], JACCIE (Java-basierter Compiler-Compiler mit interaktiver Entwicklungsumgebung Krebs und Schmitz [2012] oder die Kombination aus lex/yacc (Generator for lexical analyzers / Yet Another Compiler-Compiler [Levine et al., 1992].

Die meisten verfügbaren Compiler-Compiler erzeugen Parser für eine gegebene Sprache, die Auswertung des Syntaxbaums obliegt dagegen weiterhin den Programmierern der Compiler. In dieser Arbeit wird JACCIE verwendet, weil hier zu Scannern und Parsern auch Evaluatoren in der geläufigen Sprache Java definiert und erzeugt werden können und darüber hinaus die Evaluation schrittweise gesteuert und grafisch nachvollzogen werden kann. Allerdings ist auch hier keine Hardwarespezifikation die Grundlage. Vielmehr ermöglicht das Tool das Zuordnen von Auswerteregeln zu den jeweiligen Alternativen einer Produktion, also eine strukturierte, am Syntaxbaum orientierte Entwicklung des Evaluators.

Es gibt eine feste Arbeitsaufteilung zwischen den Komponenten eines Compilers. Der Scanner erkennt die Token, die der Parser zu einem Syntaxbaum aufbereitet. Der Evaluator schließlich erkennt die Bedeutung und erzeugt das Ergebnis, z.B. ein Programm, eine Zahl oder ein

Text (siehe Abb. 9.2, rechts). Und dennoch ist es möglich, nach der Spezifikation für den UVC verschiedene Definitionen für Scanner, Parser und Evaluator zu erhalten, je nachdem, welche Komponente mehr „Intelligenz“ erhält. So kann z.B. ein Scanner Zeichenketten erkennen, die erst der Evaluator richtig zuordnen kann, oder der Scanner unterscheidet gleich zwischen Befehlswörtern und anderen Bezeichnern, so dass bereits der Parser sinnvolle Arbeit leistet und nützliche Fehlermeldungen produzieren kann. Die Herausforderung besteht somit darin, die Arbeit über alle drei Komponenten so zu verteilen, dass die einzelnen Komponenten übersichtlich, verständlich und somit für zukünftige Erweiterungen leicht anpassbar bleiben.

In den folgenden drei Abschnitten wird für jede Komponente des Compilers beschrieben, was die allgemeinen Techniken leisten. Zudem wird motiviert, warum sich bestimmte Aspekte für unseren Anwendungsfall gut eignen. Abschließend erfolgt eine skizzierte Darstellung der Implementierung in UVC-Assembler.

9.1.2. Scanner

Der Scanner erkennt Zeichenketten und ordnet sie bestimmten Token zu. Umso genauer hier die Zuordnung erfolgt, desto genauer kann der Parser arbeiten. Zudem können Fehler früher im Ablauf erkannt und somit schneller und detaillierter an den Programmierer weitergegeben werden. Im Folgenden wird die Grundlage des Scanners, die sich aus der UVC-Spezifikation ableitet, diskutiert und sinnvoll erweitert. Ein Scanner soll Token erkennen. Dazu benötigt er deren Namen und eine Beschreibung der jeweiligen Zeichenketten. Wie diese Daten zusammen in einer Tabelle untergebracht werden, wird anhand eines laufenden Beispiels erklärt und schließlich die sehr knappe Implementierung vorgestellt.

Scanner gemäß UVC-Spezifikation

Die Spezifikation des UVC gibt 25 Befehle vor, die eine verschiedene Anzahl und auch verschiedene Typen von Parametern erwarten (siehe Abb. 9.3). In einem Fall ist die Länge eines Parameters von einem anderen abhängig. Die uniforme Gestaltung der Befehle, wie sie in der Spezifikation propagiert wurde (die dunkler gefärbte Zeile in Abb. 9.3), ist praktisch nicht vorhanden. Lediglich 11 der insgesamt 25 Befehle passen in dieses Schema. Der Scanner wird daher nicht nur zwischen Befehl und Bezeichner unterscheiden, sondern gleich gefundene Befehlswörter zu einer Gruppe von Befehlen zuordnen, damit die Grammatik des Parsers darauf aufbauen kann. Unzweckmäßig wäre es, jeden Befehl zu erkennen, da die Grammatik unnötig viele „Wiederholungen“ aufweisen würde. So gibt es Befehle ohne Parameter, solche, die ein, zwei oder drei Register als Parameter erwarten und schließlich Befehle, die sich in keine Schublade stecken lassen: Der Befehl `LOADC` verlangt neben ein Register einen konstanten, maximal 32-Bit großen Wert und schließlich einen *bit string* als die zu ladende Konstante.

Neben den Befehlen selbst, gibt es weitere Bestandteile eines UVC-Quelltextes. So hat jeder Quelltext einen (später nicht mehr relevanten) Namen, eine Nummer der Section und eine Folge der Nummern der genutzten Segmente, die sehr wohl auch für den UVC zur Zeit der Ausführung relevant sind.[1]

[1] Die Spezifikation enthält hier ein Beispielprogramm zur Berechnung der Fakultät, das diesen Aufbau nutzt

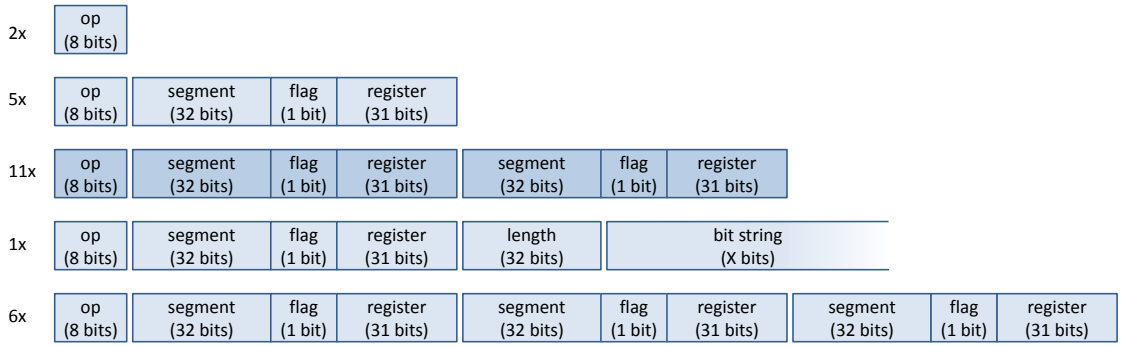


Abbildung 9.3.: Auftretende Befehlsformate in der UVC-Assemblersprache

Sinnvolle Erweiterungen des Scanners Wie an den Beispielen in der Spezifikation ersichtlich, können UVC-Programme hilfreiche Kommentare enthalten, die das Verstehen des Programms erleichtern und deren Weiterentwicklung ermöglichen. Diese sind für das UVC-Programm selbst nicht mehr relevant. Der Scanner kann diese daher erkennen und verwerfen. Hierbei handelt es sich um Zeilenkommentare, die mit einem Doppelkreutz `#` eingeleitet werden und durch das Zeilenende begrenzt sind.

Bei der Implementierung eigener UVC-Programme fiel auf, dass diese schnell schwer lesbar wurden. Viele aneinandergereihte Zahlen lassen schnell Fehler zu, aber nur schwer erkennen. Die Notation der Register, die eigentlich nur aus zwei Zahlen (der Segment- und der Registernummer) besteht, wurde hier so angepasst, dass diese zwei Zahlen durch ein Komma getrennt werden können bzw. durch dieses Komma sichtbar in Beziehung zueinander gesetzt werden:

```
# bisherige Schreibweise
LOADC   0 1   5 0x16   # Register 1 im Segment 0 bekommt Wert 24 (Länge 5)
ADD     0 2   0 1      # Addiere dieses auf Register 2 im Segment 0

# mit Trennung stehen Register- und Segmentnummer sichtbar in Beziehung
LOADC   0,1   5 0x16   # Register 1 im Segment 0 bekommt Wert 24 (Länge 5)
ADD     0,2   0,1      # Addiere dieses auf Register 2 im Segment 0
```

Der Scanner wird dadurch nur unerheblich erweitert. Parser und Evaluator mussten ebenfalls geringfügig erweitert werden, damit beide Notationen parallel möglich sind.[2] Sinnvoller ist es, gleich die Assemblersprache umfassend zu spezifizieren, damit auch außerhalb der in diesem Kapitel beschriebenen UVC-Anwendung kompatible Assembler leicht erstellt werden können. Der passende Ort hierfür sollte die Spezifikation des UVC selbst sein.

Ebenfalls sehr hilfreich ist die Erweiterung des Befehlsumfangs um drei weitere Befehle zur Ausgabe von Speicher- und Registerinhalten sowie die Ausgabe von Texten. Die Befehle **`PRINTM`**, **`PRINTR`** und **`PRINTC`** sind bereits in der Implementierung von IBM enthalten und

und in Kommentaren angibt, dass diese Informationen nur für den Assembler relevant wären. Das ist jedoch für die Java-Implementierung von IBM falsch. Die Information zu den genutzten Segmenten wird im Kopf eines UVC-Programms hinterlegt und beim Laden bzw. Aufruf einer Section dringend benötigt, wie eigene Tests zeigten.

[2] In allen in Seminaren, Praktika, Bachelor- und Masterarbeiten erstellten UVC-Anwendungen wurden Kommata dankend angenommen.

haben im Rahmen von Testprogrammen ihre Nützlichkeit bewiesen. Diese Befehle sind jedoch nicht spezifiziert. Programme, die diese Befehle verwenden, werden auf anderen UVC-Implementierungen nicht ausführbar sein. Trifft ein UVC bei der Programmabarbeitung auf deren Opcodes, ist das weitere Verhalten unbestimmt. Entweder bricht die Abarbeitung an dieser Stelle mit einer Fehlermeldung ab oder es kommt zur Fehlinterpretation, falls diese Opcodes anders vergeben wurden. Die JPEG- und GIF-Decoder von IBM nutzen diese Befehle zur Ausgabe von Fehlermeldungen. Diese Befehlserweiterung sollte daher spezifiziert sein, auch wenn die Spezifikation nur dessen Struktur und die verwendeten Opcodes beinhaltet und die Implementierung darüber hinaus optional gestaltet. Zum einen könnten diese Opcodes ignoriert und zum anderen sicher zur nächsten Anweisung hin übersprungen werden.

Notation Der Scanner-Generator eines Compiler-Compilers ermöglicht die Definition eines Scanners über reguläre Ausdrücke. Der generierte Scanner selbst nutzt zu den regulären Ausdrücken äquivalente deterministische endliche Automaten (DEA). Diese Automaten kennen eine abzählbare Anzahl an Zuständen, die durch Übergänge, sogenannten Transitionen, verbunden sind. Jeder Transition sind Zeichen zugeordnet. Deterministische Automaten zeichnet aus, dass aus jedem Zustand heraus keine zwei Transitionen zu Folgezuständen mit den gleichen Zeichen existieren. In der Abbildung 9.4 sind zwei Automaten zusammen mit den zugehörigen regulären Ausdrücken dargestellt. Hierbei wurden verschiedene Notationen verwendet, die von den genutzten Tools vorgegeben sind. An den Beispielen wird deutlich, dass intuitive Abkürzungen helfen, reguläre Ausdrücke übersichtlich anzugeben. Die native Notation des Ausdrucks `(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*`, der eine Folge von Dezimalzahlen beschreibt, die aus mindestens einer Zahl besteht, kann dadurch zum Ausdruck `(0-9)+` verkürzt werden. Diese verkürzende Notation ist jedoch nicht standardisiert und führt folglich zu verschiedenen Notationen. Im weiteren Verlauf wird die Notation genutzt, die das Tool JACCIE verwendet.[3]

Es gibt Verfahren [Berry und Sethi, 1986] für die Erzeugung von DEA aus regulären Ausdrücken und auch Tools, die diese verwenden.[4] Weder auf die Verfahren noch auf die Nutzung der Tools wird hier näher eingegangen.

Herzstück des Scanners - Eine Tabelle

Einige der bekannten Compiler-Compiler erzeugen Scanner, die während des Scannens alle für die Erkennung der Token erzeugten Automaten parallel starten. Mit jedem zu lesenden Zeichen werden die Automaten jeweils gemäß ihren Transitionen weitergeschaltet. Verfügt ein Automat nicht über eine Transition, die zum gerade gelesenen Zeichen passt, scheidet er aus. Sind alle Automaten ausgeschieden bevor die aktuelle Zeichenfolge vollständig gelesen werden konnte, ist der zu scannende Text fehlerhaft. Sind dagegen nach dem Lesen aller zur Zeichenkette gehörigen Zeichen noch Automaten übrig, werden aus ihnen die Automaten bestimmt, die sich in einem ausgewiesenen Endzustand befinden. Sind weiterhin mehrere

[3] Auch wenn die Notation von JACCIE nicht die intuitivste ist, so bietet das Tool sehr umfangreiche Testmöglichkeiten und wird daher auch für eventuell folgende Erweiterungen empfohlen.

[4] Z.B. JREXX-Lab von Michael Karneim, www.karneim.com/jrexx

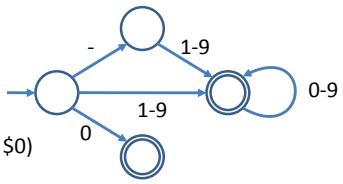
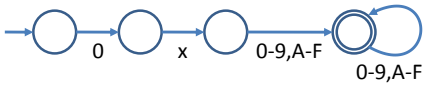
Regulärer Ausdruck	Deterministischer endlicher Automat	Zweck	Beispiele
JREXX: ((-[1-9]|[1-9])[0-9]*)|0 Jaccie: (((${1-$9}|{$1-$9})({$0-$9})[0-*])|$0)		Erkennt ganze Zahlen mit optionalen „-“ ohne führende Nullen	Akzeptiert: 0 12 -27 Akzeptiert nicht: -0 00123
JREXX: 0x[0-9A-F]+ Jaccie: (“0x”({$0-$9}\|{$A-$F})[1-*])		Erkennt Hexadezimal-zahlen mit vorange-stelltem „0x“, führende Nullen sind zugelassen, Großbuchstaben vor-geschrieben	Akzeptiert: 0x0 0x07FE37 0xAFFE Akzeptiert nicht: 0x34cb 04C7

Abbildung 9.4.: Beispiele einfacher regulärer Ausdrücke in verschiedenen Notationen mit jeweils äquivalenten DEA.

Automaten aktiv, entscheidet eine interne Reihenfolge. Das erkannte Token entspricht dem Automaten. Ein Fehler entsteht, wenn sich keiner der verbliebenen Automaten in einem Endzustand befindet. Ein Scanner, der die beiden Automaten in Abbildung 9.4 beinhaltet, wird beide Automaten beim Lesen der Zeichenkette `0x123` zunächst weiter schalten, da beide eine Transition für das Zeichen `0` aus dem Startzustand heraus besitzen. Aber bereits mit dem nächsten Zeichen `x` scheidet der erste Automat aus. Kann während des Lesens der letzte verbliebene Automat nicht weitergeschaltet werden, kommt es nicht zwangsläufig zu einem Fehler. Wurde nämlich bereits ein Token erkannt, wird zu dieser Position zurückgesprungen und erneut angesetzt. Das folgende Beispiel soll dies demonstrieren.

Es könnte sein, dass neben diesen beiden Automaten ein weiterer existiert, der Bezeichner erkennt, die mit Buchstaben, groß oder klein, beginnen und neben folgenden Buchstaben auch Zahlen enthalten können. Nehmen wir weiter an, die zu lesende Eingabe ist `0xn`. Mit dem ersten zu lesenden Zeichen sind die beiden bisherigen Automaten aktiv. Mit dem `x` verbleibt der Automat für die Hexadezimalzahlen. Mit dem `n` schließlich scheidet auch er aus. Scanner, die sich merken, welcher der zuletzt („erfolgreich“) ausgeschiedene Automat war und die zugehörige Position innerhalb der Eingabe, können jetzt dort erneut ansetzen. Diese Scanner erkennen somit eine `0` und den Bezeichner `xn`. Dieses Beispiel ist zwar konstruiert, es gibt aber durchaus Sprachen, bei denen das beschriebene Verfahren obligatorisch ist. Wenn beispielsweise mit Klammern oder Zuweisungen gearbeitet wird, folgen die Zeichen ohne direkt erkennbare Zwischenräume aufeinander. Hier muss der Scanner versuchen, solange zu lesen wie es geht und muss ggf. zum zuletzt im Endzustand befindlichen Automaten zurückspringen. Damit verfolgt er das Prinzip des längsten Präfixes und kann, falls mehrere Präfixe passen, anhand von Prioritäten (z.B. Reihenfolgen) entscheiden.

Auf den ersten Blick erscheint es notwendig, mehrere Automaten zu verwalten und während des Lesens weiterzuschalten. In der Tat genügt es, einen einzigen Automaten zu verwalten, der viele Endzustände enthält und zu jedem dieser Endzustände das zugehörige Token kennt.

Um das gerade dargestellte Problem zu lösen, merkt sich der Scanner den jeweils zuletzt

erreichten Endzustand zusammen mit der Position innerhalb der Eingabe und setzt dort ggf. erneut an. Aber wie können alle diese Automaten in nur einem vereinigt werden?

Es ist möglich aus verschiedenen DEA einen einzigen nichtdeterministischen endlichen Automaten (NEA) zu erzeugen, der einen neuen Startzustand und jeweils ε-Übergänge zu jedem Startzustand der Teilautomaten enthält. Der so erzeugte Automat ist jetzt nicht mehr deterministisch. Die Abbildung 9.5 zeigt das entsprechende Ergebnis für die beiden Automaten in Abbildung 9.4.

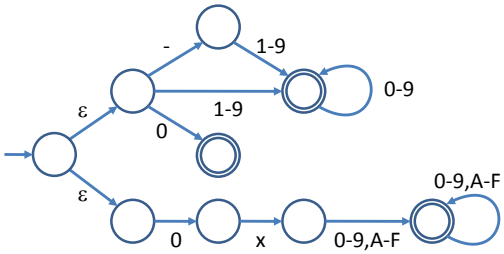


Abbildung 9.5.: Das Ergebnis der Verknüpfung der beiden Automaten aus Abb. 9.4 durch ein neues Startsymbol und ε-Übergangen zu den bisherigen Startzuständen.

In Hopcroft [2007a] wird die Äquivalenz zwischen DEA und NEA bewiesen. Aus diesem konstruktiven Beweis lässt sich ein Algorithmus ableiten, mit dem der NEA in ein DEA überführt werden kann. Mit Hilfe des Myhill-Nerode Theorems lässt sich der erhaltene Automat zudem minimieren. Ein Algorithmus hierzu findet sich z.B. bei Schmitz [1995] oder auch [Hopcroft, 2007b].

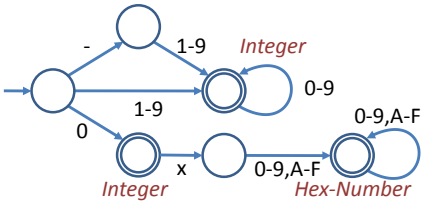


Abbildung 9.6.: Das Ergebnis der Umformung und Minimierung.

Die Abbildung 9.6 zeigt die verwendete Darstellung des minimierten Automaten. Oft wird auf den zusätzlichen Aufwand der Vereinigung und Minimierung der einzelnen Automaten zu Lasten der Laufzeit des erzeugten Scanners verzichtet. In unserem Fall geht es nicht primär um die Ausführungsgeschwindigkeit, sondern vielmehr um den Aufwand der Implementierung des zugehörigen UVC-Programms – auch wenn hierdurch die Erweiterbarkeit zusätzlichen Aufwand erfordert. Die derzeitige Scannerdefinition ist einfach genug, sodass dieser Aufwand auch ohne Hilfsmittel realisierbar bleibt.

Implementierung des Scanners

In Sippu und Soisalon-Soininen [1988] wird ein Automat thematisiert, der mehrere Token erkennt. Dabei werden nach den Token benannte Transitionen von den eigentlichen Endzustän-

den zum Startzustand eingeführt. Das lässt sich hervorragend darstellen, aber nur aufwendig implementieren. Für die eigene Implementierung wird dagegen eine Tabelle verwendet, in der Transitionen durch Folgezustände in der entsprechenden Spalte angegeben werden. Die Zustände werden in Leserichtung zeilenweise durchnummeriert, der Startzustand bekommt die 1. Speicher spielt beim UVC keine wesentliche Rolle. Somit wird für jedes überhaupt lesbare Zeichen (256 bei 8 Bit Zeichencodierung) eine Spalte verwaltet. Die dargestellte Tabelle ist somit verkürzt. Endzustände werden in einer gesonderten Spalte durch Angabe des jeweiligen Tokens vermerkt. Die Abbildung 9.7 enthält die Tabelle für das laufende Beispiel.

Zustand	Folgezustand in Abhängigkeit vom gelesenen Zeichen																	Token
	-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	x	
1	2	4	3	3	3	3	3	3	3	3	3								
2			3	3	3	3	3	3	3	3	3								
3		3	3	3	3	3	3	3	3	3	3								Integer
4																		5	Integer
5		6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6		
6		6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6		Hex-Number

Abbildung 9.7.: Die zum Minalautomaten gehörende Tabelle.

Neben der Organisation des Erkennens, stellt sich die Frage nach dem Format der Ein- und Ausgabe. Hier wird davon ausgegangen, dass der ASCII-Code[5] noch Jahre fortbestehen wird. Von der Eingabe werden daher 8-Bit pro Zeichen erwartet, von denen die unteren 7 Bit gemäß dem ASCII-Code belegt sind. Eine entsprechende Erweiterung auf Unicode[6] kann bei Bedarf durch kleine Anpassungen erreicht werden. Die zu lesenden Zeichen werden im Speicher des Segments `0` erwartet, das Ergebnis wird in Registern des Segments `3` abgelegt. Die Eingabe in das Segment `0` zu schreiben, ist Aufgabe des umgebenden Compilers.

Tabelle des Scanners Wie gelangt das Herzstück des Scanners bestehend aus der Tabelle in den Speicher des UVC? Hier gibt es zwei Möglichkeiten. Zum einen können die Tabellen über entsprechende Eingaben geladen werden. Diese Eingaben müssten in einem festgelegten Format vorliegen und ggf. selbst auf Syntax und Grammatik analysiert werden. Das sprengt den anvisierten Rahmen. Die andere Möglichkeit ergibt sich durch feste Codierung als UVC-Programm. Änderungen am Programm könnten mit dem jeweiligen Vorgängerprogramm problemlos assembliert werden. Zudem fällt für die Archivierung nur eine Datei an, nämlich das UVC-Programm selbst. Diesem Ansatz wird daher gefolgt.

Die Zustandsnummer ergibt sich indirekt aus der Zeilennummer und muss nicht separat gespeichert werden. Für das Ablegen des restlichen Tabelleninhalts gibt es wiederum zwei Möglichkeiten. Zum einen kann sie im Speicher eines Segments abgelegt werden, zum anderen kann der Inhalt in Registern gespeichert werden, davon gibt es beliebig viele und zudem richtet sich ihre Größe nach dem Bedarf. Jedem Speicherzugriff geht außerdem immer ein Laden der Bits in ein Register voraus. Die Register sind somit die schnellere und gleichzeitig bequemere Lösung. Für das laufende Beispiel ergeben sich aufgrund der sechs Zustände

[5] American Standard Code for Information Interchange - ANSI X3.4-1986
[6] ISO 10646, auch als Universal Character Set bezeichnet.

insgesamt $6 \cdot (256 + 1) = 1542$ genutzte Register, in denen Folgezustände bzw. Namen der Token gespeichert werden können. Jedes Register wird mit `0` vorbelegt. Ein Folgezustand `0` steht für einen Konflikt beim Lesen. Der notwendige indirekte Registerzugriff setzt ein Register voraus, in dem die Nummer des Zielregisters gespeichert wird. Dieses Register muss sich im selben Segment befinden. Die Nummerierung der Zustände mit `1` beginnend verschafft damit einen weiteren Vorteil: 256 stets frei nutzbare Register im selben Segment. In der folgenden Sequenz wird die „Implementierung" der Transitionstabelle im Segment `1001` über den indirekten Registerzugriff für das Beispiel angedeutet.

```
LOADC     1003,1        9   256    # 256 möglichen Zeichen
LOADC     1003,2        1     1
COPY      1003,3  1003,1
ADD       1003,3  1003,2          # + eine Spalte für das Token

# Zeile 1
COPY      1003,2  1003,3          # Start der Zeile 1

LOADC     1001,0        8   "-"    # Register indirekt auf Spalte "-"
ADD       1001,0  1003,2
LOADC     1001,0*       8     2    # Folgezustand 2

LOADC     1001,0        8   "0"    # Spalte "0"
ADD       1001,0  1003,2
LOADC     1001,0*       8     4    # Folgezustand 4
...
# Zeile 2
ADD       1003,2  1003,3          # Start der nächsten Zeile

LOADC     1001,0        8   "1"    # Spalte "1"
ADD       1001,0  1003,2
LOADC     1001,0*       8     3    # Folgezustand 3
...
# Zeile 3
...
COPY      1001,0  1003,1          # Spalte "Token"
ADD       1001,0  1003,2
LOADC     1001,0*      56   "Integer"
...
```

Hier können auch Schleifen sehr viel Arbeit abnehmen. Die letzten 3 Zeilen zeigen den Eintrag in der Spalte Token. Dadurch kennzeichnet die Zeile 3 einen Endzustand. Register beinhalten einen Bitstrom, also eine Folge von Nullen und Einsen, die nicht näher typisiert sind. Das Zuweisen einer Zeichenfolge ist nicht „wörtlich" zu sehen, sondern als ein Laden einer Bitfolge in ein Register, die sich durch Umrechnen ergibt. Dieses Umrechnen ist Aufgabe des Compilers, auch des hier implementierten. Um die Effizienz bei der Syntaxanalyse zu verbessern, werden diesen Namen in der finalen Implementierung laufende Nummern zugewiesen (siehe Abschnitt 9.1.3).

Nur eine Schleife Die Vorüberlegungen zahlen sich bei der Implementierung des eigentlichen Scanners aus. Dieser kann nun sehr einfach und zudem effizient in nur einer Schleife

implementiert werden. Das Programm umfasst lediglich 66 Assembler-Befehle und verwaltet sechs interne Zustände, die informell in der Abbildung 9.8 dargestellt sind.

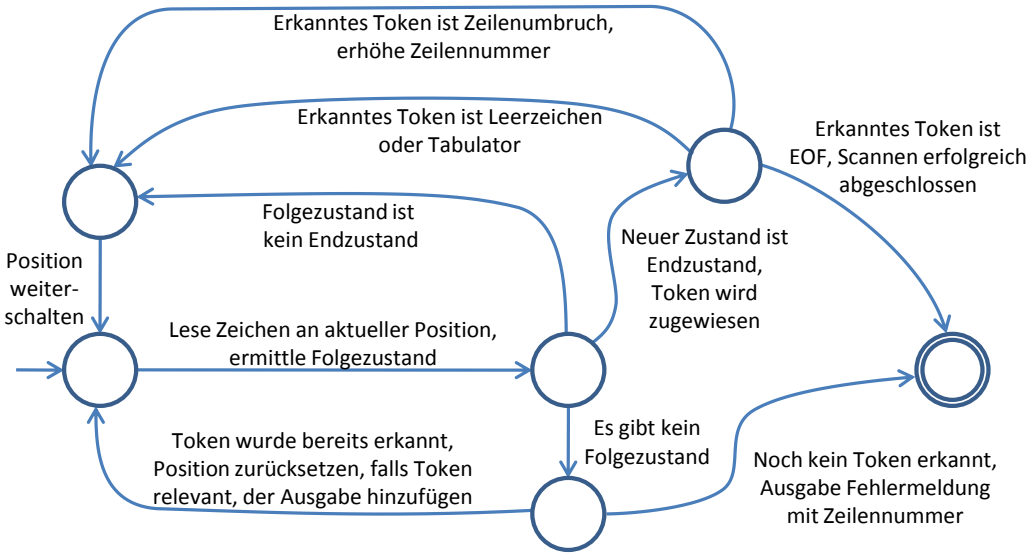


Abbildung 9.8.: Der Scanner als Automat.

Der Scanner erkennt verschiedene Bestandteile eines UVC-Quellprogramms auf Grundlage definierter Token, die wiederum durch reguläre Ausdrücke definiert sind. Die Definition der zum UVC-Assembler passenden Token ist im Anhang A.1.1 zu finden. Die vollständige Scanner-Tabelle für den Automat, der all diese Token erkennt, findet sich ebenfalls im Anhang (A.1.4).

9.1.3. Parser

Die Aufgabe des Parsers ist die Syntaxanalyse und das Erstellen eines Syntaxbaums zu einer Tokenfolge gemäß einer gegebenen Grammatik. In der Literatur werden sehr viele unterschiedliche Parser, d.h. Analyseverfahren behandelt. Im Folgenden wird die Grammatik aus der gegebenen Spezifikation heraus abgeleitet und die Wahl eines Analyseverfahrens motiviert. Anschließend erfolgt wiederum die Skizzierung der Implementierung.

Grammatik gemäß der Spezifikation

In der Spezifikation ist keine Grammatik der Sprache angegeben. Hier kann aber anhand des Beispielprogramms aus dem Anhang und der Befehlsspezifikationen alles Nötige abgeleitet werden. Allerdings ist das nicht trivial. Es gilt zu unterscheiden, welche Informationen direkt aus der Spezifikation entnommen werden können und welche davon aufgrund der angestrebten Kompatibilität mit dem von IBM zur Verfügung gestellten UVC angepasst werden müssen. So kennt das herunterladbare *Factorial* Beispielprogramm von IBM ein neue Variante der Sprungbefehle: `JUMP 1,1 Sprungziel`.

So gelangt man recht schnell, basierend auf den im Scanner definierten Token, zu der folgenden Grammatik für die Beschreibung eines Unterprogramms (Section):

```
Section     -> name number UsedSegs Program
UsedSegs    -> UsedSegs comma number
            -> number
Program     -> Program Instruction
            -> Instruction
Instruction -> op0ins
            -> op1ins Register
            -> op2ins Register Register
            -> op3ins Register Register Register
            -> loadcins Register number Constant
            -> loadcins Register name
            -> printcins number Constant
            -> jumpins Register
            -> jumpins Register name
            -> label name
Register    -> number number
            -> number comma number
            -> number number asterix
            -> number comma number asterix
Constant    -> hexnum
            -> number
            -> string
```

Hierin spiegelt sich bereits das optionale Komma bei der Angabe von Registern als auch die zwei verschiedenen Sprungbefehle, die einen speziellen Befehl zum Laden einer Sprungmarke benötigen, wider. Die Grammatik beschreibt nun vollständig die Syntax der Assemblersprache, ist intuitiv lesbar, aber nicht für alle Parsertechniken geeignet.

Parsertechniken

Parser, auch Syntaxanalysatoren genannt, erzeugen aus einer Tokenfolge aufgrund einer gegebenen Grammatik einen Syntaxbaum. Die oben gegebene Grammatik ist kontextfrei, da auf jeder linken Seite einer Produktion ein Nichtterminal allein steht, es also in keinen Kontext eingebettet ist. Syntaxbäume für solche Grammatiken besitzen eine Wurzel bestehend aus dem Startsymbol und jeweils einem Sohn für jedes Symbol auf der rechten Seite. Falls es sich bei diesen Symbolen wiederum um Nichtterminale handelt, so gibt es hier erneut Teilbäume usw. Die Blätter eines solchen Baums sind daher immer terminale Symbole, die, von links nach rechts gelesen, genau der ursprünglichen Tokenfolge entsprechen. Die im Allgemeinen nicht triviale Aufgabe des Parsers ist es, für ein gegebenes Startsymbol jeweils die zur gegebenen Tokenfolge alternative Produktion zu wählen, so dass der erzeugte Syntaxbaum auch passt.

Für die kontextfreien Grammatiken gibt es sehr viele verschiedene Parsertechniken. Einige davon arbeiten von den Blättern hin zur Wurzel des Syntaxbaums (Bottom-Up-Verfahren), andere dagegen beginnen bei der Wurzel, also mit dem Startsymbol (Top-Down-Verfahren). Um die jeweils richtige Entscheidung zu treffen, bedarf es der Auswertung der vorliegenden Tokenfolge. Alle praktisch relevanten Verfahren arbeiten die Tokenfolge von links nach rechts ab. Unterschieden werden sie an der Zahl der Token, die sie vorweg lesen müssen, um die richti-

ge Entscheidung zu treffen. Sehr anschauliche und umfangreiche Darstellungen verschiedener Analyseverfahren finden sich z.B. bei Schmitz [1995] und Aho und Ullman [1972].

Für den hier zu implementierenden Parser sind zwei Kriterien relevant. Zum einen muss er sehr leicht zu implementieren sein, zum anderen soll er nach Möglichkeit in linearer Zeit arbeiten. Nach Schmitz [1995] leistet ein LL(1)-Parser beides. Das LL(1)-Verfahren wurde erstmals von Foster [1968] vorgestellt und sehr detailliert von Knuth [1971] beschrieben.

Für die Implementierung eines solchen Parsers sind lediglich ein Keller und ein Syntaxbaum zu verwalten. Andere Verfahren benutzen z.T. mehrere Keller oder verwalten ganze Wälder von Teilbäumen. Als Basis für das LL(1)-Verfahren dient eine Parsertabelle, die auf recht einfach zu berechnende Vorschaumengen basiert. Bei der Erstellung dieser Tabelle können zahlreich vorhandene Tools wie JACCIE sehr hilfreich sein. JACCIE ist zudem äußerst hilfreich, wenn es darum geht, die Grammatik analysegerecht zu formulieren. Die zuvor angegebene, intuitiv lesbare Grammatik ist es nicht. Sie enthält Linksrekursionen und darüber hinaus Alternativen mit dem gleichen Präfix. Ein LL-Parser kann an solchen Stellen nicht mit einem Zeichen Vorschau entscheiden, welche Alternative er wählen soll. Die Grammatik muss daher zunächst durch Entfernen der Linksrekursion und durch Faktorisierung umgeformt werden [Schmitz, 1995]. Nicht jede Grammatik ist auf diese Weise umformbar und daher auch für LL(1)-Parser geeignet.[7] Die notwendigen Umformungen, auf die obere Grammatik angewandt, führen nach weiteren geringen kosmetischen Anpassungen zu einer LL(1)-Analysegeeigneten Grammatik, die im Anhang A.1.2 zu finden ist.

Parserimplementierung

Wie auch schon beim Scanner bildet eine Tabelle das Herz der Implementierung des Parsers. Der LL(1)-Parser vergleicht in jedem Schritt das oberste Symbol im Keller mit dem Token der Eingabe. Sind diese gleich, was nur im Falle eines Nichtterminals der Fall sein kann, wird reduziert, also das Zeichen aus dem Keller entfernt. Ist der oberste Eintrag dagegen ein Nichtterminal, so wird gemäß diesem die passende Zeile in der Tabelle gewählt. Anhand des zu lesenden Tokens (das ist das eine Zeichen der Vorschau) wird zudem die Spalte bestimmt. An der so ermittelten Stelle sollten die Symbole der jeweiligen Alternative stehen, durch die jetzt das oberste Symbol im Keller ersetzt wird; es wird expandiert. Begonnen wird mit dem ersten Token von links und dem Startsymbol im Keller. Dieses Verfahren erfordert zudem ein Erweiterung des Sprachschatzes um ein neues Nichtterminal, das als letztes Token der gegebenen Folge rechts angehängt wird: `#`. Der Parser war erfolgreich, wenn der Keller leer und nur dieses angehängte Nichtterminal übrig ist.

Parsertabelle Wie kommt man von der oben angegebenen analysegerechten Grammatik zur Parsertabelle? Für jede Alternative einer Regel müssen die entsprechenden Spalten bestimmt werden, in denen diese stehen soll. Für die obere Grammatik gilt, dass jeweils nur eine Alternative pro Spalte steht. Ein notwendiges Kriterium. Um diese Spalten zu bestimmen, muss die $first_1$-Menge berechnet werden. Diese enthält alle Terminale, die in einer ableit-

[7] Sollte tatsächlich eine Spracherweiterung geplant sein, die eine nicht mehr LL(1)-gerechte Analyse erfordert, muss hier ein anderer Parser implementiert werden.

baren Tokenfolge als erstes stehen können. Für ε-Produktionen bzw. für die Alternativen, die in der $first_1$-Menge ein ε enthalten, muss dagegen zusätzlich die $follow_1$-Menge bestimmt werden. In allen Spalten, die jetzt zu den Symbolen dieser jeweiligen Mengen passen, muss die Alternative bzw. deren Symbole eingetragen werden. Für detailliertere Ausführungen zur Erstellung einer LL(1)-Parsertabelle sei auf Aho et al. [1986] oder Tremblay und Sorenson [1985] verwiesen. Die so erhaltene Tabelle ist in Abbildung 9.9 angegeben.

Nonterminal	Expansion in Abhängigkeit vom gelesenen Terminal														
	asterix	comma	hexnum	loadcins	jumpins	label	name	number	op0ins	op1ins	op2ins	op3ins	printcins	string	#
UVC							name number UsedSegs Program								
UsedSegs								number SegHelp							
SegHelp		comma number SegHelp		ε	ε	ε			ε	ε	ε	ε	ε		ε
Program				loadcins Register LOADCHelp Program	jumpins Register JUMPHelp Program	label name Program			op0ins Program	op1ins Register Program	op2ins Register Register Program	op3ins Register Register Register Program	printcins number Constant Program		ε
LOADCHelp							name	number Constant							
JUMPHelp				ε	ε	ε	name		ε	ε	ε	ε	ε		ε
Register								number RegNo							
RegNo		comma number RegHelp						number RegHelp							
RegHelp	asterix			ε	ε	ε	ε	ε	ε	ε	ε	ε	ε		ε
Constant			hexnum					number						string	

Abbildung 9.9.: Die zur umgeformten Grammatik gehörende LL(1)-Parser-Tabelle.

Namen in Form von Zeichenketten zu handhaben ist recht aufwendig. Um z.B. die Zeile für das Nichtterminal `Register` zu bestimmen, müssten die Zeichenketten der Nichtterminale der Reihe nach verglichen werden. Danach müsste die jeweilige Spalte ebenfalls über mehrere Vergleiche dieser Art bestimmt werden. Dabei haben diese Symbole nur repräsentativen Charakter. Diesen Aufwand kann man deutlich verringern, indem man den Symbolen Nummern zuordnet: Den Symbolen der Spalten, also den Terminalen, die Zahlen `0` bis `14`, den Nichtterminalen die Zahlen `15` und folgende. Somit berechnet sich die entsprechende Zeile $Z = (Symbol - 15 + 1) \cdot 15$. Diese Berechnung ist wesentlich schneller als das fortwährende Durchsuchen von Namenslisten, setzt aber die Lesbarkeit herab. Die passende Spalte wird durch einfache Addition des zu lesenden Tokens bestimmt. Diese Zuordnung der Token zu Nummern wird gleich beim Erkennen der Token im Scanner vorgenommen. In der Scannertabelle werden dazu die Namen in der letzten Spalte durch die entsprechenden Nummern ersetzt. Die Scanner-Tabelle als auch die implementierungsgerechte Tabelle für den Parser enthalten jetzt nur noch Zahlen. Die Spalten und Zeilenbezeichnungen sind implizit enthalten und müssen nicht „implementiert“ werden. An die Parser-Tabelle angefügt sind die jeweiligen Symbole der Alternativen, auf die jeweils durch Angabe der ersten Registernummer „verlinkt“ wird. Die in Abbildung 9.10 jeweils ersten beiden rot unterlegten Felder geben die Knotennummer (fortlaufend nummerierte Alternativen mit 15 beginnend) und die Anzahl der zu ersetzenden Symbole an. Es folgen grün oder blau unterlegte Felder, die entweder Nichtterminale oder Terminale referenzieren.

Nonterminal		Expansion in Abhängigkeit vom gelesenen Terminal														
		*	,	hex	loadc	jump	label	name	num	op0	op1	op2	op3	print	string	#
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
UVC	15							165								
UsedSegs	16								171							
SegHelp	17		175		180	180	180			180	180	180	180	180		180
Program	18				182	188	194			199	203	208	214	221		227
LOADChelp	19							229	232							
JUMPhelp	20				236	236	236	238		236	236	236	236	236		236
Register	21								241							
RegNo	22		245						250							
RegHelp	23	254			257	257	257	257	257	257	257	257	257	257		257
Constant	24			259					262						265	
165 -->		15	4	6	7	16	18	16	2	7	17	17	3	1	7	17
180 -->		18	0	19	4	3	21	19	18	20	4	4	21	20	18	21
195 -->		3	5	6	18	22	2	8	18	23	3	9	21	18	24	4
210 -->		10	21	21	18	25	5	11	21	21	21	18	26	4	12	7
225 -->		24	18	27	0	28	1	6	29	2	7	24	30	0	31	1
240 -->		6	32	2	7	22	33	3	1	7	23	34	2	7	23	35
255 -->		1	0	36	0	37	1	2	38	1	7	39	1	13		

Abbildung 9.10.: Die implementierungsgerechte Tabelle.

Das Speichern einer Tabelle in Registern hat sich bewährt und findet daher auch hier erneut Anwendung. Die jetzige Tabelle ist kleiner und zudem dichter befüllt als die Tabelle des Scanners. Hier eröffnet sich eine neue Methode zum Befüllen der Tabelle mit sehr langen, hexadezimal definierten Konstanten, die Stück für Stück in die Register der Tabelle übertragen werden. Der folgende UVC-Code deutet dieses Verfahren für die ersten Zeilen der Tabelle an, für die insgesamt 17 Register beginnend mit `1003,11` auf diese Art zu verarbeiten sind. Dieses Verfahren ist initial sehr rechenintensiv, demonstriert aber sehr anschaulich die kompakte Programmierweise basierend auf beliebig großen Registern.

```
#--------------- ----  *  ,  hx lo ju la na nu o0 o1 o2 o3 pr st #
LOADC    1003,11 240 0x00000000000000000A500000000000000000000000
LOADC    1003,12 240 0x00000000000000000000000AB000000000000000000
LOADC    1003,13 240 0x0000AF0000B40B40B40000000B40B40B40B40000B4
LOADC    1003,14 240 0x0000000000B60BC0C20000000C70CB0D00D60DD0000E3
LOADC    1003,15 240 0x00000000000000000000E50E800000000000000000000
LOADC    1003,16 240 0x0000000000EC0EC0EC0EE0000EC0EC0EC0EC0EC0000EC
LOADC    1003,17 240 0x000000000000000000000000F100000000000000000000
...

LOADC    1003,0       0      0    # 0
LOADC    1003,1       1      1    # Inkrement
LOADC    1003,2       4     15    # 15 mögliche Terminale/Spalten
LOADC    1003,3       5     17    # 17 einzulesenden Register
LOADC    1003,4       4     11    # Beginne mit Register 11
LOADC    1003,6      13  0x1000   # Schrittweite der Werte (12 Bit)
LOADC    1001,0       0      0    # init
```

```
label: TableLoop
ADD      1001,0  1003,2          # Zur nächsten Zeile springen
ADD      1001,0  1003,2          # Zum Ende der Zeile springen
COPY     1003,5  1003,2          # Anzahl der Spalten (Terminale)
label: TableInnerLoop
SUB      1001,0  1003,1          # Jeweils eins zurücklaufen
DIV      1003,4* 1003,6  1001,0* # Rest ist Wert für finale Tab
SUB      1003,5  1003,1
GRT      1003,5  1003,0
JUMPC    1,1 TableInnerLoop

ADD      1003,4  1003,1          # nächstes Register
SUB      1003,3  1003,1
GRT      1003,3  1003,0
JUMPC    1,1 TableLoop
```

Eingabe Als Eingabe ist die Ausgabe vom Scanner perfekt für die Verarbeitung im Parser geeignet. Sie beinhaltet in je drei Registern im Segment 3 beginnend mit Register 3 die Nummern der Token, die jeweils zugehörige Zeichenkette und die entsprechende Zeilennummer, die für die Ausgabe von Fehlermeldungen sehr hilfreich sein kann. Die Register können nacheinander gelesen werden, ohne dass auf die Längen der entsprechenden Zeichenfolgen geachtet werden muss. Die Zeichenketten wurden bereits vom Scanner als Bitfolge interpretiert und in das jeweils zweite Register geladen. Die Belegung erst mit dem Register 3 ist dem indirekten Registerzugriff geschuldet.

Syntaxbaum Der Syntaxbaum ist die vom Parser erzeugte Ausgabe. Dieser stellt eine rekursive Struktur dar und besteht aus zwei verschiedenen Knotenarten: innere Knoten und Blätter. Die Blätter des Syntaxbaums enthalten nur terminale Symbole und deren Zeichenketten, während die inneren Knoten nur die Verknüpfungen zu ihren Söhnen benötigen. Beide Knotentarten sind in Abb. 9.11 illustriert. Der Wert im Feld Knotentyp entscheidet darüber, ob der Knoten ein Blatt ist oder ein innerer Knoten. Ist der entsprechende Wert kleiner 15, ist es ein Blatt und der Wert entspricht dem gelesenen Token. Ist der Wert dagegen größer, entspricht der Wert einer der durchnummerierten Alternativen. Diese Nummerierung ist für das Traversieren des Baumes und somit für die Implementierung des Evaluators relevant.

Abbildung 9.11.: Die zwei verschiedenen Knotentypen: oben innere Knoten, unten Blätter. Heller unterlegte Felder enthalten Verknüpfungen zu anderen Knoten oder zu den noch nicht definierten Attributen.

Das Feld für die Attribute wird während der Syntaxanalyse nicht genutzt. Auch die Verknüpfung zum Vaterknoten und die Information, der wievielte Sohn der aktuelle Knoten selbst

im Vaterknoten ist, wird zwar erst während der Evaluierung benötigt, aber bereits jetzt im Baum gespeichert, damit dieser nicht erneut angepasst oder gar transformiert werden muss. Die Zeilennummer wird jeweils in den Blättern gespeichert, damit bei der Evaluierung auftretende Fehler mit Zeilennummern versehen ausgegeben werden können.

Jedes Feld in Abbildung 9.11 ist erneut durch ein Register repräsentiert. Die einzelnen Werte in Kombination machen den gesamten Syntaxbaum aus. Diesmal werden die Register des Segments `4` beginnend ab `2` genutzt. Für ein gekürztes UVC-Programm ist der Syntaxbaum in Abbildung 9.12 dargestellt.

Erstes Register	Registerinhalte
2	15 0 -3 4 11 16 21 56 null
11	6 2 1 **Test** 0
16	7 2 2 **1001** 1
21	16 2 3 2 28 33 null
28	7 21 1 **1** 2
33	17 21 2 3 41 46 51 null
41	1 33 1 **,** 2
46	7 33 2 **1001** 2
51	18 33 3 0 null
56	19 2 4 4 65 70 105 116 null
65	3 56 1 **LOADC** 3
70	32 56 2 2 77 82 null
77	7 70 1 **1001** 3
82	33 70 2 3 90 95 100 null
90	1 82 1 **,** 3
95	7 82 2 **3** 3
100	36 82 3 0 null
105	28 56 3 1 111 null
111	6 105 1 **Loop** 3
116	21 56 4 3 124 129 134 null
124	5 116 1 **label:** 4
129	6 116 2 ...

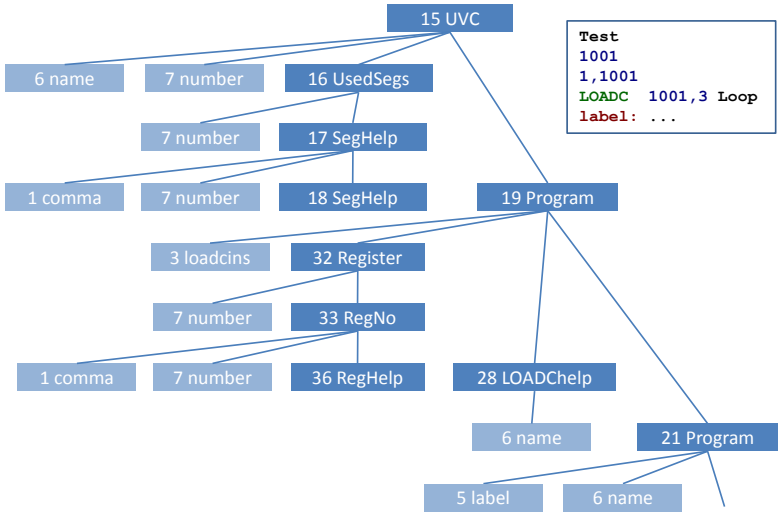


Abbildung 9.12.: Der Syntaxbaum als Registerinhalte und in grafischer Form.

Keller Der Keller kann wiederum durch Register eines privaten Segments dargestellt werden. In der Implementierung wurden die Register beginnend ab `100` des Segments `1003` genutzt. Das Register `99` als aktueller *Stack Pointer* zeigt immer auf das nächste freie Register.

Die folgenden zwei Programmausschnitte zeigen Push- und Pop-Operationen, mit denen der Inhalt des Registers `1003,10` im Keller gesichert und anschließend wieder hergestellt wird:

```
COPY 1003,99* 1003,10 # push          SUB  1003,99  1003,1
ADD  1003,99  1003,1                  COPY 1003,99* 1003,10 # pop
```

Gespeichert werden im Keller für den Algorithmus wichtige Informationen. Für die Syntaxanalyse sind vor allem die expandierten Symbole wichtig, die nach und nach wiederum expandiert oder bei Übereinstimmung mit dem Symbol der Eingabe reduziert werden. Zeitgleich mit der syntaktischen Analyse kann auch der Syntaxbaum aufgebaut werden. Dazu sind weitere Informationen notwendig. Jeder Eintrag im Keller besteht aus drei Registern, die das Symbol, die Nummer des ersten Registers des Vaterknotens und die Nummer des Registers im Vaterknoten, in dem die Startadresse des eigenen Knotens eingetragen werden muss, enthalten. Auf die Nummer des Sohnes im Vaterknoten, den der aktuelle Knoten selbst darstellt, kann verzichtet werden, da sich diese aus der Differenz der beiden anderen Registernummern ergibt.

Der Algorithmus lässt sich so sehr einfach implementieren. Ohne Keller ließe sich das LL(1)-Verfahren auch rekursiv implementieren. Jedoch benötigen Unterprogrammaufrufe im UVC im Vergleich zu Sprüngen viel Zeit. Der genutzte Keller dagegen wächst nur selten über 30 Werte hinaus. Die final implementierte Schleife besteht gerade noch aus 77 Assembler-Anweisungen, die um 17 weitere Anweisungen zur detaillierten Fehlerausgabe ergänzt sind.

9.1.4. Evaluator

Während es bisher um die Syntax des zu kompilierenden UVC-Programms ging, geht es jetzt um die Semantik, also um die Bedeutung der nun syntaktisch zugeordneten Zeichenfolgen der Token. Diese ergeben nach fertiger Auswertung ein kompiliertes UVC-Programm. In einem ersten Schritt wird dieser Syntaxbaum mit Variablen angereichert. Diese Variablen nennt man Attribute, das Verfahren selbst daher Attributierung. Die Attribute sind dann zu evaluieren, also zu berechnen.

Im Folgenden werden verschiedene Techniken vorgestellt, die zur Evaluation eben dieser Attribute genutzt werden können, und im Anschluss werden die hier notwendigen Attribute vorgestellt und deren Funktion während der Programmerstellung skizziert. Schließlich wird die Implementierung an entscheidenden Punkten vorgestellt.

Evaluatortechniken

Ebenso, wie es sehr viele verschiedene Verfahren zur Syntaxanalyse gibt, gibt es auch viele Verfahren zur Auswertung eines attributierten Syntaxbaums. Das liegt zum einen an den verschiedenen Vorgehensweisen, aber vor allem an den zahlreichen Möglichkeiten zur Steigerung der Effizienz durch Integration der Attributauswertung direkt in die Syntaxanalyse. So ist z.B. ein Verfahren zu finden, dass Attribute direkt während einer LL(1)-Analyse auswertet, wenn die verwendete Attributierung gewissen Anforderungen genügt [Schmitz, 1995]. Die hier verwendete Attributierung genügt diesen Ansprüchen und dennoch soll kein allzu spezielles Verfahren gewählt werden, da für zukünftige Erweiterungen nicht sichergestellt werden kann, dass sich diese ebenfalls in vorgegebene Raster pressen lassen.

Neben allen speziellen Verfahren gibt es zwei Verfahren, die für alle zyklenfreien Attributierungen anwendbar sind. Das erste Verfahren durchläuft fortwährend den gesamten Baum und berechnet jeweils die gerade berechenbaren Attribute. Berechnungen einzelner Attribute können von Ergebnissen anderer Berechnungen abhängen. Pro Durchlauf können somit nur diese Attribute berechnet werden, die entweder von keinem anderen Ergebnis abhängen oder nur von bereits berechneten.

Das zweite Verfahren nach Jourdan [1984] wird auch als „bedarfsgetriebene Auswertung“ bezeichnet. Es berechnet prinzipiell nur die Attribute des Wurzelknotens. Sind diese abhängig von anderen Attributen, so werden zunächst diese rekursiv berechnet. Hierbei kann es vorkommen, dass bestimmte Attribute nicht ausgewertet werden, weil sie zur Berechnung der Attribute im Wurzelknoten nicht benötigt werden. Dieses Verfahren ist effizient, wenn von jedem Knoten direkt, also ohne den gesamten Baum erneut zu durchlaufen, in den Vaterknoten bzw. zu den Söhnen gesprungen werden kann. Die dafür notwendigen Verweise zum Vaterknoten werden bereits während der Syntaxanalyse im Baum hinterlegt.

Während das erste Verfahren deutliche Vorteile auf parallelen Systemen aufweist, eignet sich das Verfahren nach Jourdan aufgrund seiner allgemeinen Anwendbarkeit bei angemessener Effizienz für die Implementierung des Evaluators für den UVC.

Evaluatorimplementierung

Nach Jourdan ist die bedarfsgetriebene Auswertung rekursiv. Ebenso wie die LL(1)-Syntaxanalyse auch rekursiven Charakter hat, wird auch hier eine Entrekursivierung vorgenommen. Der Umgang mit dem notwendigen Keller ist bereits erprobt. In diesem Keller werden die notwendigen Attribute abgelegt. Immer wird versucht, das oberste Attribut im Keller zu berechnen. Gelingt dies, wird das Attribut aus dem Keller entfernt. Wenn nicht, werden alle noch nicht evaluierten Attribute, die zur Berechnung notwendig sind, zusätzlich im Keller abgelegt. Dieses Verfahren mündet wiederum in einer recht übersichtlichen Schleife, die diese zwei Fälle unterscheidet. Aber woher weiß das Programm, welche Attribute zur Berechnung eines anderen notwendig sind?

Es werden zwei verschiedene Arten von Attributen unterschieden, zum einen Attribute, deren Information von der Wurzel herab verteilt wird (ererbte Attribute) und zum anderen solche, deren Information jeweils zur Wurzel hin gesammelt wird (synthetisierte Attribute) [Schmitz, 1995].

Attribute können darüber hinaus nur Nichtterminalen zugeordnet werden (Terminale kennen nur ihre zugeordnete Zeichenkette). Sie werden daher direkt im Knoten ihres Nichtterminals gespeichert. Die Berechnung der zwei Arten von Attributen findet jeweils in einem anderen Kontext, also einer anderen Umgebung statt. Der Kontext eines Knotens besteht dabei aus allen Attributen des eigenen Knotens und derer aller direkten Söhne. Während ererbte Attribute im Kontext des Vaterknotens berechnet werden, geschieht dies für synthetisierte Attribute im Kontext des eigenen Knotens.

Für jedes zu evaluierende Attribut gibt es daher einen Kontext, der durch Angabe des Knotens, also einer Registernummer, bestimmt werden kann. Auszuwertende Attribute werden mit zwei Nummern angegeben: die Nummer des Sohnes im Kontext (0 für den Knoten selbst) und die Nummer des Attributes, wobei alle Attribute durchnummeriert werden. Die hier verwendete Attributierung benötigt insgesamt fünf Attribute. Obwohl kein Nichtterminal alle Attribute benötigt, werden dennoch alle fünf jedem Knoten zugewiesen, damit die Attribute direkt per Nummer angesprochen werden können. Nicht genutzte Attribute bedeuten nicht genutzte Register und solche wiederum nicht zugewiesener Speicher. In der Abbildung 9.13 sind die jeweils tatsächlich genutzten gekennzeichnet.

Die kompilierte UVC-Section muss ihre Gesamtlänge kennen. Zudem müssen sämtliche Bestandteile der Section ihre Position innerhalb der kompilierten Section kennen, da z.B. Befehle ihren Opcode und Register- und Segmentnummern ihre Werte selbst im Bitcode eintragen. Daraus folgen die zwei Attribute `pos` und `totalLength`. Letzteres Attribut ergibt sich direkt aus der Position des letzten Programmknotens im Baum. Die Position dagegen wird im Baum herab gereicht und jeweils die eigene benötigte Anzahl an Bits aufsummiert.

Nachdem die Gesamtlänge der Section bestimmt wurde, ist sichergestellt, dass alle Namen und Positionen der Labels gesammelt vorliegen, denn beim Weiterreichen der Position trägt sich jedes Label mit Namen und seiner Position in eine Liste ein. Bei der Evaluation des

	totalLength	pos	codeDone	length	string
UVC	x		x		
UsedSegs	x	x			
SegHelp	x	x			
Program	x	x	x		
LOADChelp	x	x	x		
JUMPhelp					x
Register		x	x		
RegNo		x	x		
RegHelp	x				
Constant		x	x	x	

Abbildung 9.13.: Die Zuordnung der Attribute zu den Nichterminalen.

Attributs **codeDone** wird der eigentliche Code an die Position geschrieben, eventuell benötigte Sprungmarken liegen jetzt in der Liste vor. Bereits beim Eintragen als auch beim Entnehmen können semantische Fehler wie doppelt definierte oder fehlende Bezeichner erkannt werden.

Die Implementierung der Evaluation eines Attributs wird an einem Beispiel verdeutlicht. Gewählt wird hierfür ein Ausschnitt aus einem Baum, der einen Befehl mit einem Register darstellt. Die zugehörige Regel ist: **Program -> op1ins Register Program2**.

Diese Regel gibt einen Kontext vor, in dem vier Attribute auszuwerten sind: die zwei synthetisierten Attribute **totalLength** und **codeDone** des Knotens und jeweils das ererbte Attribut **pos** des zweiten und dritten Sohns. Auszuwerten sind hier zunächst die Attribute **Register.pos** und **Program2.pos**, da **Programm.pos** bereits berechnet vorliegen sollte. Aufgrund des 8 Bit breiten Opcodes ergibt sich: **Register.pos = Program.pos+8**.

Der im Folgenden gegebene Code-Ausschnitt setzt das um:

```
label: S2322 # Program --> op1ins >Register< Program2 (Register.pos)
COPY          4,1     4,0          # Zeiger auf eig. Knoten nach 4,1
ADD           4,1  1003,3
COPY          4,1     4,1*         # Anzahl der Söhne nach 4,1
ADD           4,1  1003,4
ADD           4,1     4,0
COPY       1003,40    4,1*         # 4,1 zeigt auf Attributliste
COPY          4,1  1003,40
ADD           4,1  1003,4          # Program.pos ist zweites Attribut
EQU           4,1* 1003,1          # bereits evaluiert?
JUMPC      1,1 S2322next

COPY          4,1     4,0          # Falls noch nicht evaluiert, muss
ADD           4,1  1003,1          # Attribut samt Kontext in den Keller
COPY       1003,99*   4,1*         # Vaterknoten in den Keller (Kontext)
ADD        1003,99 1003,1
ADD           4,1  1003,1
COPY       1003,99*   4,1*         # eig. Nr. in den Keller
ADD        1003,99 1003,1
LOADC      1003,99*     2     2    # Eval Vater.self.2 -> Program.pos
ADD        1003,99 1003,1
JUMP       1,1 EvaluatorLoop       # Abbruch dieses Versuchs
```

```
label: S2322next                    # Hier startet die Berechnung
ADD          4,1  1003,1
COPY      1003,50    4,1*
ADD       1003,50 1003,8            # Register.pos = Program.pos+8
JUMP      1,1 EvaluatorSaveValue    # speichern und als fertig markieren
```

Falls das Attribut `Program.pos` noch nicht vorliegt, wird der Kontext des Vaterknotens im Keller abgelegt (durch dessen Registernummer) zusammen mit der Nummer des Sohnes, den der eigene Knoten im Vater darstellt, und der Nummer des benötigten Attributs. Falls es bereits berechnet ist, kann die eigene Auswertung beginnen. Die `23` im Label steht für die Knotennummer 23 gemäß der erweiterten Parser-Tabelle, die alle Alternativen ab 15 beginnend durchnummeriert (siehe Abb. 9.10). Die folgende `2` steht für den zweiten Sohn und die letzte `2` steht für das Attribut `pos`. Attribute bestehen immer aus zwei Registern, wobei das erste angibt, ob es bereits evaluiert ist und das zweite den berechneten Wert enthält.

Die Berechnung des Wertes ist schließlich mit 4 Assembler-Befehlen möglich. Aufwendig dagegen erscheint das Prüfen der Abhängigkeiten, wobei Skripte bei der Erstellung des Quellcodes sehr hilfreich sein können. Es wurde zu Gunsten der Laufzeit darauf verzichtet, Unterprogramme für diesen Zweck zu schreiben. Stattdessen ist an dieser Stelle im übertragenen Sinne manuelles „Inlining" zu finden.[8] Es ist sehr wahrscheinlich, dass sehr oft die Abhängigkeiten geprüft werden müssen, bevor es zur eigentlichen Berechnung kommt. Gerade an dieser Stelle waren daher Überlegungen zur Laufzeit entscheidend.

Woher aber weiß der Algorithmus, zu welchem Label er springen muss, wenn z.B. im Keller die Werte `17B,3,2` abgelegt sind? Zunächst muss der Algorithmus den Kontext bestimmen. Da in jedem Knoten als erster Wert die Knotennummer gespeichert ist, die dem Kontext entspricht, wird ein Register mit dem ersten Wert geladen und dann durch indirekten Registerzugriff die Nummer c des Kontexts bestimmt. Die folgenden Nummern entsprechen der Nummer s des Sohns und der Nummer a des Attributes. Schließlich berechnet der Algorithmus eine Nummer $l = (c \cdot 6 + s) \cdot 5 + a$. Bei maximal 5 Söhnen (plus eins für den Knoten selbst) und 5 Attributen ergibt sich so eine eindeutige Zahl. Für alle 63 notwendigen Attributierungsregeln ergeben sich so eindeutige Registernummern, die die jeweiligen Sprungmarken enthalten. Genutzt wird demnach eine Art Sprungtabelle:

```
#          Register  Sprungmarke
LOADC      1001,450  S1500          # (( 15 *6)+ 0 *5)+ 0
LOADC      1001,451  S1501          # (( 15 *6)+ 0 *5)+ 1
LOADC      1001,467  S1532          # (( 15 *6)+ 3 *5)+ 2
LOADC      1001,472  S1542          # (( 15 *6)+ 4 *5)+ 2
LOADC      1001,480  S1600          # (( 16 *6)+ 0 *5)+ 0
...
```

Dieses, bei der Initialisierung des Evaluators einmal durchgeführte Laden aller Sprungmarken, ermöglicht eine sehr effiziente Implementierung einer großen `CASE`-Anweisung, denn der `JUMP`-Befehl kann auch den indirekten Registerzugriff nutzen:

[8] Eine Beschreibung dieser Optimierungstechnik findet sich z.B. in [CRI, 1992].

```
label: EvaluatorEntry
SUB       1003,99 1003,1
COPY      1003,20 1003,99*        # att
SUB       1003,99 1003,1
COPY      1003,21 1003,99*        # num
SUB       1003,99 1003,1
COPY         4,0  1003,99*        # ptr
ADD       1003,99 1003,3
COPY      1003,22    4,0*        # typ
...
COPY      1001,1  1003,22
MULT      1001,1  1003,6
ADD       1001,1  1003,21
MULT      1001,1  1003,5
ADD       1001,1  1003,20

JUMP      1001,1*                 # case ( ( typ *6+ num )*5 + att )
```

Auf Unterprogramme wurde nicht vollständig verzichtet. Die Umrechnung von Zeichenketten in Zahlen oder das Speichern von Werten in die kompilierte Section sind Aufgaben, die erst mit der finalen Berechnung der Attribute anfallen. Diese können der Übersichtlichkeit halber in Unterprogramme ausgelagert werden, ohne größere Einbrüche in der Effizienz. Die Implementierungen der Decoder von IBM verwenden auch einige Unterprogramme. So besteht der JPEG-Decoder aus 21 Sections. Hier wurde jedem Unterprogramm eine eigene Section zugeteilt. Diese Vielzahl führt nicht gerade zu einer verbesserten Lesbarkeit. Der Evaluator benutzt 8 Unterprogramme, die in der eigenen Section über Sprungmarken erreichbar sind. Der folgende Ausschnitt gibt ein Unterprogramm zu Konvertierung einer in einer Zeichenkette gegebenen Ganzzahl.

```
# ------------------------------------------
# Unterprogramm zum Konvertieren einer
# positiven Dezimalzahl
# Eingabe:  1,4 - Zahl als ASCII-Zeichenkette
# Rückgabe: 1,5 - konvertierte Zahl
# ------------------------------------------
label: convertNumber
LOADC     1003,1       8   "0"    # Zahl 0 als Offsset
LOADC     1003,0       0    0
LOADC     1003,8       9   256    # Zeichenbreite
LOADC     1003,10      4    10
LOADC     1003,11      1     1
LOADC        1,5       0     0
label: convertNumberLoop
DIV       1,4     1003,8  1003,2
SUB       1003,2  1003,1
MULT      1003,2  1003,11
MULT      1003,11 1003,10
ADD       1,5     1003,2
GRT       1,4     1003,0
JUMPC     1,1 convertNumberLoop
BREAK
```

Sind für alle Unterprogramme dieser Section die notwendigen Werte `1300` (für die Identifikation der Section selbst) und `2` (für das übergebene Segment) bereits in den Registern `1003,81` und `1003,83`, so besteht der Unterprogrammaufruf aus zwei aussagekräftigen Anweisungen:

```
LOADC     1003,82  convertNumber
CALL      1003,81 1003,82 1003,83
```

Aufgrund der langatmigen Prüfungen der Attributabhängigkeiten, die zugunsten der Effizienz nicht in Unterprogramme ausgelagert wurden, beinhaltet der gesamte Code des Evaluators über 3000 Assembler-Anweisungen.

9.1.5. UVC-Assembler in UVC-Assembler

Die drei vorgestellten Komponenten Scanner, Parser und Evaluator können in dieser Kombination genau eine Section kompilieren. Ein UVC-Programm kann selbst aus mehreren Sections bestehen, die jedoch unabhängig voneinander kompiliert werden können. Der Compiler als Rahmenprogramm lädt über den Befehl `IN` den Quelltext einer Section in den Speicher des Segments 0 und ruft nacheinander die drei Komponenten auf. Das Ergebnis wird dann jeweils an die fortlaufend erhöhte Position im Speicher des Segments 5 gespeichert und mit der nächsten Section fortgesetzt. Über den Messagetyp der Eingabe wird gesteuert, ob weitere Sections folgen oder nicht. Mit dem Ende der Eingabe wird das gesamte UVC-Program per `OUT` ausgegeben. Diese Ausgabe unterscheidet sich im Messagetyp von den Fehlermeldungen der einzelnen Komponenten, die jeweils mit `STOP` das Programm sofort beenden. Anhand dieses Messagetyps kann somit entschieden werden, ob das Ergebnis als UVC-Programm gespeichert oder dem Programmierer als Fehlermeldung ausgegeben wird.

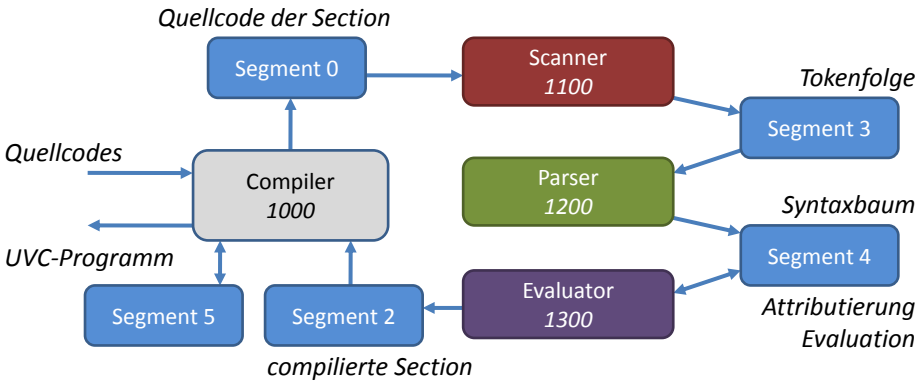


Abbildung 9.14.: Sections und Segmente der vollständigen Implementierung.

Dieses in Abbildung 9.14 grau eingefärbte Rahmenprogramm mit der Sectionnummer 1000 ließ sich mit einer Schleife und nur wenigen Assembler-Anweisungen realisieren.

9.2. Gewonnene Erkenntnisse

Mit der Implementierung des UVC-Compilers in UVC-Assemblersprache ist es ab jetzt jederzeit möglich, neu erstellte oder angepasste UVC-Programme zu kompilieren, sei es zur Anpassung oder Neuerstellung von eigenen Testprogrammen begleitend zur Implementierung eines UVC oder zur Entwicklung weiterer Programme für den UVC zum Zweck der Langzeitarchivierung von verfügbarem Wissen. Daraus folgt unmittelbar, dass die genutzte Syntax zusammen mit der Spezifikation des UVC erhalten bleiben muss. Folgerichtig muss eine jede Spezifikation des UVC auch die Definition der genutzten Assemblersyntax enthalten. Nur so macht das als notwendig erachtete Beispielprogramm in der Spezifikation überhaupt Sinn. Der die Spezifikation begleitende Assembler in Form einer UVC-Anwendung wäre damit auch in ferner Zukunft zur Implementierung neuer UVC-Anwendungen nutzbar.

Vom Autor selbst wurde erhofft, dass mit der vorliegenden Implementierung der Compiler es binnen eines Tages schafft, seinen eigenen Quellcode zu übersetzen. Diese Erwartungen beruhten auf der Erfahrung mit den von IBM vorgestellten Bild-Decodern. Den gesamten Quellcode übersetzt der von JACCIE generierte Compiler in knapp über 5 Minuten. Zur Vorbereitung der Implementierung für den UVC wurden die Komponenten des Compilers erneut in Java implementiert; dieses Mal allerdings sehr „UVC-nah;“ also weitestgehend mit primitiven Datentypen. Dieser Compiler benötigt noch 15 Sekunden. Genau auf dieser Implementierung basiert die in UVC-Assemblersprache implementierte Version. Das UVC-Programm benötigt tatsächlich für die Übersetzung der 118 Zeilen des Compilers eine, der 1044 Zeilen des Scanners vier, der 256 Zeilen des Parser zwei und der 3400 Zeilen des Evaluators 18 Sekunden.[9] Diese insgesamt gerade einmal 25 Sekunden übertrafen die Erwartungen bei Weitem. Zudem zeigen diese Zeiten ein nahezu lineares Verhalten. Diese Implementierung zeigt daher auch, dass der „richtige“ Umgang mit dem UVC durchaus zu applikablen Anwendungen führen kann. Schließlich ist die UVC-Variante etwas schneller als halb so schnell wie die entsprechende Java-Anwendung.

Diese hier vorgestellte Implementierung zählt mit seinen acht Wochen Entwicklungszeit durchaus zu den komplexeren Anwendungen. Im Vergleich ist die bislang größte öffentlich verfügbare Applikation für den UVC der JPEG-Decoder. Dieser ist kompiliert weniger als halb so groß wie der Compiler.

Interessante Erkenntnisse ergaben sich zudem aus Sicht des Softwareentwicklers. So ist die Realisierung eines Kellers mit oder das Speichern ganzer Tabellen in den Registern eines Segments etwas, dass es in dieser Form nur auf dem UVC gibt. Bei anderen virtuellen Maschinen gibt es Sprungtabellen zur Realisierung von größeren `CASE`-Anweisungen. Zwar kennt der UVC nichts Vergleichbares, jedoch lassen sich größere `CASE`-Konstrukte effizient über den indirekten Registerzugriff umsetzen. Eine Beschreibung dieser Technik findet sich erstmals in dieser Arbeit. Dieses Wissen um die gegenüber anderen realen Maschinen vorhandenen Vorzüge kann anderen Programmierern enorm helfen, Algorithmen effizient zu implementieren. Bis zu dem Zeitpunkt, an dem optimierende Compiler für höhere Sprachen entwickelt werden, sollte das Wissen um laufzeiteffiziente Realisierungen nicht verloren gegangen sein.

Um eventuell noch enthaltene Fehler zu erkennen und zu beheben, wurde der entwickelte

[9] Die Zeiten wurden mit der noch nicht optimierten Referenzimplementierung gemessen.

Scanner bereits zum Syntaxhighlighting im zu Implementierungszwecken anfänglich entwickelten Editor erfolgreich genutzt. Dieser Editor nutzte anfangs die mit JACCIE generierten Compilerkomponenten. In der aktuellen Version werden die Komponenten verwendet, die Grundlage für die UVC-Assembler-Implementierung waren. Bislang sind keine Fehler in diesen Komponenten sichtbar geworden. Es ist geplant, den für den UVC implementierten Compiler öffentlich zugänglich zu machen, um durch eine größere Nutzergruppe weitestgehend Fehlerfreiheit sicherzustellen und gefundene Fehler zu beheben.

Mit diesem UVC-Programm wurde ein tatsächlich bereits heute anwendbarer Compiler implementiert, mit dem sich auch komplexere Anwendungen übersetzen lassen. Damit ist für heutige Programmierer und darüber hinaus auch für die ferne Zukunft ein Assembler verfügbar, der zudem aufgrund seiner Modularität und der zugrunde liegenden Definitionen mit den Erweiterungen und Anpassungen der Zukunft mitwachsen kann. Durch die damit geschaffene Möglichkeit, UVC-Anwendungen im Quellcode zu archivieren, bleiben Kommentare erhalten, die Programme bleiben lesbar und UVC-Anwendungen nachfolgender Generationen können Programmteile und Konzepte daraus direkt nutzen. Eine weitere Erkenntnis aus diesem Experiment ist, dass, wann immer es möglich ist, Werkzeuge für den UVC direkt für diesen implementiert werden sollten, damit diese zukünftigen Generationen ohne Aufwand erhalten bleiben.

Von Schaffrath [2012] wurde im Rahmen einer Bachelorarbeit ein Assembler als UVC-Anwendung passend zur weiterentwickelten Spezifikation implementiert. Aus der oben geschilderten Motivation heraus wird dieser im Kontext der Spezifikation verfügbar sein.

10. Testprogramme ergänzend zur Spezifikation

Dieses Kapitel beschäftigt sich mit einem Aspekt der Softwareentwicklung, der schnell in den Hintergrund gerät: dem Testen. Und dabei sprechen viele Gründe dafür, diesem Aspekt mehr Aufmerksamkeit zu schenken.

Im Abschnitt 4.2.3 wurde bereits auf Dinaburg et al. [2008] verwiesen. Die hier relevante Kernaussage ist, dass virtuelle Maschinen bzw. Emulatoren das Original scheinbar nie vollständig fehlerfrei nachbilden können. Und obwohl umfangreiche Spezifikationen der Originale vorliegen, schleichen sich bei der Implementierung kleine Fehler ein. Malware erkennt anhand solcher Abweichungen die mögliche Ausführung in einem Analysetool und gibt sich harmlos. Viele solcher Abweichungen entstehen durch ungenaue Spezifikationen und weniger durch tatsächliche Implementierungsfehler [Martignoni et al., 2009]. Gefunden werden solche Abweichungen oft erst durch die missglückte Programmausführung oder durch abweichende Ergebnisse im Vergleich mit der Ausführung auf der realen Hardware.

Für den UVC ist keine reale Hardware vorhanden, lediglich andere Implementierungen wären vorstellbar. Jedoch ist es eine realistische Annahme, dass in ferner Zukunft keine Implementierung mehr existiert, sondern lediglich die Spezifikation. Wie aber kann man sicherstellen, dass dieser dann neu implementierte UVC korrekt ist? Das Problem ist, dass man sich dahingehend nie sicher sein kann. Eine umfangreiche Testsammlung könnte helfen, zumindest unkorrekte Implementierungen zu erkennen.

Eine solche Testsammlung ist auch für die Programmierer hilfreich, die bestimmte Fragestellungen mithilfe der Spezifikation nicht zweifelsfrei klären können. Finden sie ein archiviertes Testprogramm und wissen zudem, was es leisten soll, können sie sich daraus die korrekte Funktionalität ableiten. Diese Idee lebt davon, dass für jede erdenkliche Fragestellung ein klärender Testfall vorhanden ist. Hier sei angemerkt, dass ebenso wie die Spezifikation selbst auch die Testsammlung nie fertig bzw. vollständig sein wird. Das bedeutet, dass selbst wenn die spezifizierte Funktionalität des UVC in naher Zukunft eine Art Fixpunkt erreicht hat, die Spezifikation und auch die Testfälle weiteren Anpassungen unterliegen können. Z.B. kann sich die Bedeutung verwendeter Begriffe (schleichend) ändern. So hatte das Adjektiv „schlecht“ im 16. Jahrhundert noch eine neutrale Bedeutung im heutigen Sinne von „schlicht.“ Verpasst man die Anpassung einer sonst korrekten Spezifikation oder aber das Hinzufügen klärender Testprogramme über Jahrhunderte hinweg, kann auch eine so sichere Archivierungsmethode wie die den UVC nutzende scheitern.

Jeder Programmierer weiß, dass die Entwicklung von Testprogrammen viel Zeit einnimmt. Eine bereits vorhandene Testsammlung nimmt ihm zwar nicht vollständig diese Arbeit ab, wird aber sicher einige Stunden einsparen helfen.

Diese hier aufgeführten Gründe sind keineswegs vollständig. Auch beziehen sich alle auf-

geführten Gründe auf eine Testsammlung. Der folgende Abschnitt wird begründen, warum aus dem großen Bereich des Testens lediglich die Entwicklung einer solchen Testsammlung für den UVC in Frage kommt. Der Darstellung der Entwicklung dieser Testsammlung ist ein eigener Abschnitt gewidmet. Ebenso den ersten Evaluationsergebnissen, die noch während der Fertigstellung dieser Arbeit gesammelt werden konnten.

10.1. Grundlagen des Testens

Die in diesem Abschnitt gegebene Darstellung der Grundlagen des Testens orientiert sich im Schwerpunkt an den Ausführungen von Myers und Sandler. Gleich zu Beginn ihrer Ausführungen stellen sie klar, was durch Testen nicht zu leisten ist: Tests dienen demnach weder der Demonstration der Abwesenheit von Fehlern, noch sollten Testprogramme zeigen sollen, dass ein Programm korrekt arbeitet. Vielmehr beschreiben sie die Intention des Testens idealisiert wie folgt:

> *Therefore, don't test a program to show that it works; rather, you should start with the assumption that the program contains errors (a valid assumption for almost any program) and then test the program to find as many of the errors as possible.*

Diese Vorstellung untermauern die Autoren noch, indem sie die Testentwicklung als einen destruktiven, ja sogar als sadistischen Prozess bezeichnen, der dem Testentwickler viel Zeit und Kreativität abverlangt. Dalley [1991] geht sogar noch einen Schritt weiter und vergleicht den Softwaretester mit einem lebensmüden Testpiloten, der mit Absicht eine Situation hervorzurufen versucht, in der das Flugzeug abstürzt.

Dalley gesteht Myers die Fähigkeit zu, Fakten über das Testen zu veröffentlichen, für deren Entdeckung andere 20 Jahre brauchen. Die von Myers verfassten Prinzipien sollen daher zunächst vorgestellt werden, bevor auf die einzelnen Techniken näher eingegangen wird.

10.1.1. Prinzipien

Der Prozess der Testentwicklung sollte nach Myers bestimmten Prinzipien folgen. Er formuliert insgesamt 10 solcher Prinzipien, die im Folgenden nur verkürzt und inhaltlich zusammengefasst wiedergegeben werden.

So sollte ein guter Test eine Definition der zu erwartenden Ausgabe enthalten. Dabei müssen Tests zu erwartende Eingaben ebenso abdecken, wie nicht zu erwartende. Zudem sollten diese Tests nicht von Programmierern des zu testenden Programms entwickelt werden, sondern idealerweise von Mitarbeitern einer anderen Institution. Interessant für folgende Betrachtungen ist *Principle 7*:

> *A common practice is to sit at a terminal and invent test cases on the fly, and then send these test cases through the program. The major problem is that test cases represent a valuable investment that, in this environment, disappears after the testing has been completed. Whenever the program has to be tested again (for example, after correcting an error or making an improvement), the test cases must*

> *be reinvented. More often than not, since this reinvention requires a considerable amount of work, people tend to avoid it. Therefore, the retest of the program is rarely as rigorous as the original test, meaning that if the modification causes a previously functional part of the program to fail, this error often goes undetected. Saving test cases and running them again after changes to other components of the program is known as regression testing.*

Ein ebenfalls für die folgenden Betrachtungen relevantes Prinzip ist das mit der Nummer 9:

> *The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.*

10.1.2. Strategien des Testens

Nachdem grundlegende Prinzipien vorhanden sind, stellt sich die Frage nach dem konkreten Vorgehen. Man unterscheidet grundlegend zwei Strategien des Testens: das *Black-Box-Testing* und das *White-Box-Testing*. Im Bereich der virtuellen Maschinen kann eine spezielle Form der Überprüfung zum Einsatz kommen: mittels sogenannter *Traces*. Auf diese Strategien wird im Folgenden näher eingegangen.

Black-Box-Testing

Unter dem *Black-Box-Testing* versteht man das Testen eines fertigen Programms bzw. Teilprogramms ohne Kenntnis der zugrunde liegenden Quellcodes, da der Blick in den Quelltext entweder nicht gewünscht oder nicht möglich ist. Ohne die Kenntnis der inneren Abläufe kann das Programm nur als *Black Box* betrachtet werden, die aufgrund bestimmter Eingaben ganz bestimmte Ausgaben gemäß der Programmspezifikation erzeugen sollte.

Mit dieser Strategie erzeugte Testprogramme müssen daher die zu testenden Eingaben und die zu erwartenden Ausgaben aus der Spezifikation ableiten. Dabei können diese Daten nur stichprobenartig die Funktionalität überprüfen. Theoretisch können zwar alle möglichen Eingabedaten durchprobiert werden. Dazu bedarf es jedoch, schon bei kleineren Wertebereichen seitens der Eingabe, viel Zeit und zudem müssten zu allen möglichen Eingaben auch die Ausgaben erstellt werden. Das wiederum kommt einer doppelten Implementierung gleich. Nur in den allerseltensten Fällen und ausschließlich für kleinste Programme kommt es somit zu dieser als *Exhaustive Input Testing* bezeichneten Alternative.

Als logische Konsequenz daraus leitet sich ab, dass man nur eine begrenzte Auswahl an Testprogrammen bzw. Eingaben überprüfen kann. Diese Auswahl sollte wohl überlegt sein und nach Myers Überlegungen auf Äquivalenzklassen beruhen. Aus der Menge aller Tests, die dazu geeignet sind, den gleichen Fehler zu entdecken, reicht ein Vertreter. Für diese Partitionierung gibt es kein allgemeingültiges Vorgehen, vielmehr sind es Erfahrungswerte und Vorstellungen des Testers hinsichtlich einer möglichen Implementierung.

Neben den gefundenen Vertretern aus den Äquivalenzklassen, die z.B. je einen Vertreter aus dem Bereich der positiven und einen aus dem der negativen Zahlen beinhalten, kommen jetzt weitere Testfälle dazu, die gezielt die Grenzen der Ein- und Ausgabebereiche abdecken

sollen. So z.B. die Eingabe von 0. Testsammlungen, die um solche Tests mittels der auch als *Boundary-Value Analysis* bezeichneten Methode angereichert wurden, sind deutlich effektiver als solche ohne [Myers und Sandler, 2004].

Eine tiefergehende Methode weitere Tests zu entwickeln ist das *Cause-Effect-Graphing*. Hierbei werden aus der Spezifikation direkt Abhängigkeiten von Ursachen und Wirkungen abgeleitet. Vorausgesetzt, diese Abhängigkeiten lassen sich klar aus der Spezifikation ableiten und kleineren Teilbereichen zuordnen, können aus solchen Abhängigkeiten ganze Graphen erzeugt werden, die dann als Ergebnis interessante Knotenpunkte ausweisen. Diese Graphen decken so alle aus der Spezifikation ableitbaren Zusammenhänge ab und sorgen somit dafür, dass keine zweckmäßig zu prüfende Kombination oder Folge von Eingaben übersehen wird.

Die letzte hier betrachtete Technik der Testfallerzeugung ist bekannt unter dem Namen *Error Guessing*. Dabei kommt es stark auf die Erfahrung und Intuition des Testfallentwicklers an. Dieser rät mögliche Kombinationen von Eingabedaten, die seiner Meinung nach dazu neigen könnten, fehlerhafte Ausgaben zu produzieren. Auch fällt unter diese Technik das Erahnen, wie ein Programmierer bei der Umsetzung der spezifizierten Vorgaben vorgegangen sein und welche Fehler er dabei gemacht haben könnte. Hier wird klar, dass die Entwicklung von brauchbaren Tests ebenso, wenn nicht sogar eine komplexere Aufgabe als die eigentliche Entwicklung des Programms sein kann.

White-Box-Testing

Die Strategie des *White-Box-Testing* setzt die Kenntnis des Quelltextes, also der inneren Programmlogik voraus. Dieses Wissen erlaubt ein anderes Vorgehen als dem des *Black-Box-Testing*, macht es jedoch nicht leichter. Die einfache Annahme, es würde genügen, jede Anweisung des Programms wenigstens einmal zur Ausführung zu bringen, reicht nicht. Direkt an diese Forderung geknüpft wäre die Forderung, jeden Programmpfad wenigsten einmal zu durchlaufen. Schon kleinere Schleifen, die bedingte Anweisungen enthalten, können hier schnell Laufzeiten der Tests jenseits des Erwartbaren bedeuten. Auch wenn dieses Vorgehen zum Testen kleinerer Programmabschnitte recht gut geeignet ist, auf komplexere Programme übertragen lässt sich das nicht. Zudem garantiert die Abarbeitung einer jeden Anweisung ohne Programmabstürze kein korrektes Ergebnis. Überdies könnte vergessen worden sein, bestimmte Pfade zu implementieren. Zwei Fehlerursachen, die so nicht gefunden werden.

Mit der Kenntnis des Quellcodes werden zwei effektive Techniken möglich, die sich *Code Inspection* und *Walkthrough* nennen. Beiden gemeinsam ist die Betrachtung des Codes von einer kleinen Gruppe, der auch der Programmierer angehört. Im ersten Fall erklärt der Programmierer sein Programm Zeile für Zeile. Im zweiten Fall dagegen hat ein Mitglied der Gruppe einfache Testfälle vorbereitet, die dann „mental“ von der Gruppe durchgespielt werden. In beiden Fällen steht der Programmierer für Fragen zur Verfügung. Der größte Anteil an Fehlern wird durch die Fragen offenkundig, so dass sich diese im Anschluss schnell beheben lassen. Der Vorteil aller *White-Box-Tests* ist, dass der Fehler mit seiner Entdeckung präzise lokalisiert ist und damit schneller behoben werden kann als ein Fehler, der beim *Black-Box-Testing* gefunden wurde.

10.1.3. Traces

Für die die Emulation nutzende Archivierungsmethode gibt es bereits Überlegungen, wie man die Authentizität bewahren kann, obwohl keine lauffähige Hardware mehr verfügbar ist. Nach Phillips [2010] sollten für die Entwicklung von Emulatoren *Traces* zu Programmen erstellt und zusammen mit den Programmen archiviert werden. Unter solchen *Traces* versteht er z.B. das regelmäßige Speichern des gesamten Maschinenzustands, also Registerinhalte und Speicherauszüge periodisch nach einer bestimmten Anzahl von ausgeführten Instruktionen. Solche *Traces*, aufgezeichnet noch auf den originalen Maschinen, können bei der Entwicklung von Emulatoren sehr hilfreich sein, Fehler zum einen zu finden und zum anderen entweder dem Emulator oder auch einem, möglicherweise fehlerhaft, archivierten Programm zuzuordnen. Das gilt natürlich nicht nur für die Neuentwicklung, sondern auch für die fortwährende Portierung von Emulatoren. Wobei bei der Entwicklung erster Generationen gleich auf entsprechende Schnittstellen geachtet werden sollte, die das Erzeugen eines solchen *Traces* und damit den Vergleich erst möglich machen. Effektiv zum Tragen im Bereich der Langzeitarchivierung kommt dieser Ansatz, wenn in ferner Zukunft die Frage nach dem Grad der erreichten Authentizität zu beantworten ist.

10.2. Umsetzung für den UVC

In den kurzen Anmerkungen innerhalb der Spezifikation geben die Autoren Lorie und van Diessen eine Antwort auf die Frage: *"How can we be sure that an archived program is bug free?"* Dabei unterschieden sie folgerichtig zwei Aspekte: Der erste beschäftigt sich mit der Sicherstellung der Korrektheit der vorhandenen UVC-Implementierung, der zweite mit der Korrektheit der UVC-Anwendung.

Obwohl der vorangegangene Abschnitt deutlich darstellte, dass ein Sicherstellen der Korrektheit nicht durch Testen möglich ist, verweisen die Autoren auf ein *"extensive UVC test program,"* dass sie entwickelt haben und angeblich genau das leisten kann. Dabei soll das Programm fünf Punkte abdecken: Das Laden eines Programms, das Aufrufen von Unterprogrammen, die Bitadressierbarkeit, die Instruktionen und die Ein- und Ausgabe. Mit Blick auf die Darstellungen im vorangegangenen Abschnitt ist es nicht verwunderlich, dass das in Lorie und van Diessen [2005] angepriesene Testprogramm leider noch nicht zugänglich ist, da es noch zahlreichen Anpassungen bedarf.[1]

Wie müsste ein solches Testprogramm aussehen? Nachdem insbesondere zukünftige UVC-Implementierungen getestet werden sollen, kommen nur Testprogramme in Form von UVC-Anwendungen in Frage. Und damit steht fest, dass sich ein in ferner Zukunft implementierter UVC nur von „Innen" heraus testen lassen wird – zumindest, wenn wir bereits heute diese Testphase vorbereiten wollen, um der Idee des UVC zu folgen: so viel wie möglich des Aufwandes bereits heute leisten, damit der Zugriff auf die archivierten Dokumente sehr leicht gelingt.

Im Abschnitt 9.1 wird an konkreten Beispielen die Notwendigkeit eines Assemblers motiviert. Darin wird deutlich, dass ein einzelnes Testprogramm all diese Anforderungen an

[1] Quelle: Direkte Korrespondenz mit van Diessen.

einen umfassenden Test nicht erfüllen kann, insbesondere nicht bezüglich Bitadressierung und Arithmetik.

Als verdeutlichendes Beispiel kann die Implementierung der Division dienen. Prinzipiell gibt es verschiedene Algorithmen zur Umsetzung. Im Wesentlichen wird aber ein iteratives Vorgehen notwendig sein, bei dem pro Iteration eine bestimmte Anzahl an Bits des Ergebnisses aus einer begrenzten Anzahl Bits der Operanden berechnet werden. Die Ergebnisse der einzelnen Iterationsschritte ergeben schließlich das Gesamtergebnis, wobei es zu kleinen „Fehlern“ kommen kann, die aber in folgenden Iterationsschritten zuverlässig entdeckt und behoben werden können [Knuth, 1969; Parthasarathi und Jhunjhunwala, 1995]. Diese Korrekturschritte treten nur mit geringer Wahrscheinlichkeit auf. Bei Knuths Algorithmus ist sie von der Ordnung $2/b$, wobei b die genutzte Basis ist, also der mit der gewählten Bitbreite größtmögliche Wert. Im Falle der Referenzimplementierung sind das weniger als 0,00000005%. Bei einer statistisch gleichmäßigen Verteilung der zu prüfenden Operanden würde das mehr als 2 Milliarden durchzuführende Divisionen bedeuten. Aber was, wenn nun statt 32 Bit 40 oder 64 Bit genutzt werden? Die Implementierung in FORTRAN IV nutzt 20, die in FORTRAN 77 ebenso wie die in Ada83 entwickelte dagegen 32 Bit. Testfälle gezielt auf diese Algorithmen und die möglicherweise genutzte Größe der Basis vorab zu erstellen ist zwar möglich, aber sie wären eventuell schnell obsolet. So könnten zukünftig andere Algorithmen entdeckt und genutzt werden. Eine Vielzahl dann sinnloser Testfälle, die ausschließlich den knapp 0,00000005% großen Bereich abdecken, ist wenig hilfreich und verschwendet absehbar Ressourcen in der Zukunft.

Es steht daher außer Frage, dass hier kein einzelnes Testprogramm entwickelt werden kann. Vielmehr bedarf es einer großen, umfangreichen Testsammlung, die aber nie als vollständig erachteten werden kann. Die Entwicklung und Nutzung einer solchen Testsammlung würde zur Strategie des *Black-Box-Testings* gerechnet werden. Um den zukünftigen Entwickler einen einfachen Zugang zu den Tests zu ermöglichen, wurden auch solche Testprogramme entwickelt, die z.B. zu testende Zahlenkombinationen einlesen. Diese Eingabedaten können flexibel erweitert werden. Damit können neue Szenarien getestet werden, ohne zunächst selbst komplexe UVC-Programme entwickeln zu müssen.

Während das *White-Box-Testing* für die Implementierung eines UVC in ferner Zukunft im Jetzt nicht unterstützt werden kann, gilt das aber nicht für die heutige Entwicklung von UVC-Anwendungen. Mehr als diese Strategie, insbesondere die *Code Inspection* und den *Walkthrough*, für diesen Zweck zu empfehlen, kann die vorliegende Arbeit jedoch nicht. Zweckmäßig ist sicher, dass Archive mit dem Annehmen zu archivierender Dateien neuer Formate nicht nur eine passende UVC-Anwendung fordern, sondern vom Entwickler dieser zu archivierenden UVC-Anwendungen Nachweise einfordern, die zumindest auf eine intensive Testphase schließen lassen.

Für ausschließlich statische Anwendungen wie dem JPEG-Decoder geht das Testen sehr einfach, wie Lorie und van Diessen [2005] richtig feststellen. So empfehlen sie mit jeder neuen UVC-Anwendung das *Exhaustive Checking*, also das Ausprobieren mit allen archivierten Dateien und das automatisierte Überprüfen der Ergebnisse. Allerdings räumen sie ein, dass das eben nur für statische UVC-Anwendungen funktioniert. Dass dieses Vorgehen mitunter Jahre dauern kann und auch nur funktioniert, wenn noch andere korrekte Anwendungen vorhanden sind, mit deren Ergebnissen der Vergleich durchgeführt werden kann, erwähnen sie nicht.

10.2.1. Black-Box-Testing für den UVC

Wie oben festgehalten, kann bereits heute eine umfangreiche Testsammlung erstellt werden, die Programmierern in ferner Zukunft helfen kann. Dieser Abschnitt befasst sich ausführlich mit der Erstellung einer solchen Testsammlung.

Praktikum

Während der Erstellung der Referenzimplementierung und im Rahmen der *Zeitreise* wurden zahlreiche Testfälle erstellt. Einige davon zwar nur wie von Myers und Sandler beschrieben *"on the fly,"* aber viele eben auch in Form von programmierten Testsequenzen. Die mit allgemeingültigem Charakter konnten dabei leicht in Testprogramme für den UVC in Form von UVC-Anwendungen überführt werden. Das waren jedoch nur wenige. Die meisten während der Implementierungen entwickelten Testprogramme decken nur spezielle Facetten ab.

Das Ziel eines vom Autor betreuten Praktikums war es daher, gezielt aus der Spezifikation heraus ableitbare Testszenarien zu identifizieren und in Form von UVC-Anwendungen abzubilden. Ferner sollte der externe Blick aus der „Testperspektive" das eigene subjektive Bild der Spezifikation korrigieren bzw. ergänzen.

Identifizierte Bereiche

Für das Praktikum ließen sich sieben etwas unterschiedlich stark ausgeprägte Bereiche identifizieren, denen anschließend Teilnehmer des Praktikums zugeordnet wurden:

- Registerarithmetik
- Speicherzugriff
- Ein- und Ausgabe
- Sprungbefehle
- Unterprogrammaufrufe
- Aufbau und Interpretation von UVC-Anwendungen
- indirekter Registerzugriff

Entfernt erinnert das an das modulare bzw. auch an das funktionale Testen nach Myers und Sandler [2004]. Diese Art des Testens berücksichtigt zunächst nur kleinere Programmabschnitte. Erst später werden Tests durchgeführt, die die Interaktion solcher Programmabschnitte testen. Das heißt, diese Art des Testens ist bereits während der Erstellung des UVC und im Anschluss daran möglich, jedoch erst wenn der UVC bereits Programme laden und ausführen kann. Bei den eigenen Implementierungen sind ebenfalls modulähnliche Gruppierungen umgesetzt, die weitestgehend zu den identifizierten Bereichen passen. Die Funktionalität des Ladens von UVC-Anwendungen wurde erst sehr spät implementiert. Zukünftig dürfte ein solches Vorgehen wohl abwegig werden.

Anwendbare Prinzipien

Die einleitend zusammengefassten 10 Prinzipien sind nicht alle gleichermaßen bei der Entwicklung der Testsammlung für den UVC umsetzbar. Deren Eignung wird jetzt erörtert.

Das erste Prinzip, nachdem ein Testfall auch konkrete Angaben enthalten muss, was als Ergebnis zu erwarten ist, kann recht gut dadurch gelöst werden, dass das Programm selbst entscheidet, ob der Test ein Fehler gefunden hat, also selbst Kenntnis darüber hat, welches Verhalten bzw. welcher Zustand zu erwarten ist. Abhängig davon kann dann eine klar interpretierbare Ausgabe erzeugt werden. Eine solche klare Ausgabe sollte von einem Tool interpretiert werden, sodass ein Übersehen eines Fehlers unwahrscheinlicher wird und somit dem vierten Prinzip Rechnung getragen werden kann. Nach diesem Prinzip muss eine gründliche Auswertung der Testergebnisse mit jedem Durchlauf gewährleistet sein. Die Einschränkungen, die sich aus einer solchen vom Testprogramm selbst erzeugten Ausgabe ergeben, werden noch detailliert beschrieben.

Schon die angedachte Nutzung der Testsammlung durch Programmierer in ferner Zukunft sorgt für die Einhaltung des zweiten und dritten Prinzips, nach denen, wenn möglich, weder ein Programmierer noch eine Institution eigene Programme selbst testen sollten. Zugute kommt der erstellten Testsammlung, dass kein Praktikumsteilnehmer mit der Implementierung des UVC zu tun hatte, also alle von „außen“ kamen.

Nach dem fünften Prinzip müssen Testprogramme sowohl zu erwartende und gültige Eingaben als auch unerwartete und fehlerhafte Eingaben abdecken. Für UVC-Programme sind das weniger die Eingaben selbst, da diese einem konkret spezifizierten Muster folgen müssen, als vielmehr unerwartete Befehlsfolgen, leere Sections, falsche Parameter, Zugriffe auf nicht initialisierte Segmente bzw. Register usw. Insofern kann dieses Prinzip in „interpretierter“ Form berücksichtigt werden.

Das sechste Prinzip besagt, dass die Arbeit mit dem Auffinden eines Fehlers noch nicht beendet ist. Viel interessanter ist ein konkreter Hinweis darauf, wo sich der Fehler eingeschlichen haben könnte. Diesem Aspekt ist nicht leicht Rechnung zu tragen. Zum einen sind die Möglichkeiten der Ausgabe einer UVC-Anwendung limitiert, zum anderen können nur Fehler bei der Ausführung einzelner Befehlssequenzen erkannt werden. Durch geschicktes Steigern der Komplexität von Test zu Test können jedoch recht zielsicher die fehlerhaft implementierten Befehle identifiziert werden. Konkrete Hinweise auf die eigentlichen Implementierungsfehler können so zwar noch nicht generiert werden, jedoch können Testfälle in ihrer Beschreibung Erfahrungswerte einfließen lassen und damit mögliche Implementierungsfehler aufzeigen. So z.B. können bei einer getesteten Division Vorzeichen und Beträge getrennt überprüft werden. Stimmen die Beträge, aber nicht die Vorzeichen, kann das wohl auf eine fehlerhafte Berücksichtigung der Vorzeichen zurückgeführt werden. Das jedoch erschließt sich nur durch die Interpretation der Testfall-Metadaten, die dazu zusätzlich zur Spezifikation und interpretierbar erhalten werden müssen. Diese Metadaten müssen angeben, wie sich das Testprogramm verhält und welcher Fehlernummer welche möglichen Ursachen zugeordnet wurden.

Dem siebten Prinzip wird schon allein durch die Rahmenbedingungen Rechnung getragen. Sehr oft kommt es demnach vor, dass Tests *“on the fly”* durchgeführt werden und so für spätere Tests verbesserter Versionen nicht mehr zur Verfügung stehen. Zukünftig genutzte Tools zum automatischen Ausführen der Testprogramme sollten so implementiert werden,

dass sie die Testfälle immer vom ersten beginnend ausführen, auch die, die bereits erfolgreich durchlaufen wurden. Die Testprogramme müssen sich daher für eine automatisierte Ausführung und Auswertung eignen. Die Ausgaben aller UVC-Testprogramme müssen dazu einem klaren Muster folgen. Um Verwechslungen, z.B. durch versehentliches oder später eventuell unumgängliches Umbenennen, zu vermeiden, gibt jedes Testprogramm im Messagetyp seine eindeutige Nummer aus, gefolgt von einer Nachricht, die immer 32 Bit umfasst. Die Folge von 32 Nullen entspricht dabei keinem gefunden Fehler.

Das achte Prinzip fordert eine ausreichend geplante Testphase, die auch größere Nachbesserungen vor dem produktiven Einsatz ermöglicht. Die hier entwickelten Testfälle sind dazu geeignet, bereits während der Implementierungsphase zu unterstützen und so das achte Prinzip etwas zu entschärfen. Die Verantwortung zur Einhaltung des siebten und achten Prinzips liegt aber in der Zukunft.

Nach dem neunten Prinzip ist die Wahrscheinlichkeit, einen weiteren Fehler dort zu finden, wo bereits einer entdeckt wurde, deutlich höher als dort, wo bisher keiner entdeckt wurde. Bisher wird diesem Aspekt nicht Rechnung getragen. Sollte die hier vorgestellte Testsammlung im Laufe der Zeit große Anerkennung finden und viele weitere Testprogramme dabei von anderen Programmierern entwickelt werden, könnte man ein abgestuftes Vorgehen in Betracht ziehen. Bestimmte, sehr spezielle Tests würden dann nur durchgeführt, wenn ein „übergeordneter“ Test zuvor einen Fehler aufzeigte. Denkbar wäre, dass Entwickler des UVC selbst zusätzliche Testfälle entwerfen, um einen aufgezeigten Fehler einzugrenzen und diese Testprogramme dann zukünftigen Entwicklern zur Verfügung stellen. Die Abarbeitung der Testreihenfolge darf dann nicht sofort mit dem ersten gefundenen Fehler abbrechen. Vielmehr könnten die Ergebnisse der folgenden spezielleren Tests in Kombination eine Art Fingerabdruck einer bestimmten Fehlerursache sein. Ein so eingegrenzter Fehler kann dann noch schneller im Programm aufgespürt und ausgebessert werden.

Das letzte Prinzip bezeichnet das Testen insgesamt als *“an extremely creativ and intellectually challenging task”* und dient einzig der Motivation. Diesem Aspekt der Motivation sollte dadurch Rechnung getragen werden, dass neue Testfälle sorgfältig dahingehend geprüft werden, ob sie geeignet sind, bisher unentdeckt gebliebene Fehler zu finden oder aber bisher gefundene Fehler effizienter aufzudecken. Eine faire und anerkennende Bewertung neuer Testfälle trägt maßgeblich zur fortwährenden Verbesserung der hier vorgestellten Testsammlung bei. Der Name des jeweiligen Autors und das Jahr der Erstellung sollten obligatorisch die Metadaten der Testprogramme ergänzen.

Anwendbare Techniken

Dieser Abschnitt stellt die aus der Abarbeitung von UVC-Anwendungen ableitbaren Informationen dar. Dabei können dem Programm nur eingeschränkt Informationen entlockt werden. Spezifiziert sind lediglich zwei Befehle zur Ein- und Ausgabe von Bitfolgen. Nur wenn diese zuverlässig implementiert sind, können hierüber verlässliche Informationen gewonnen werden. Alternativ können auch auf anderen Wegen Informationen nach „außen“ gelangen. So können über die Anzahl der ausgeführten Befehle, über Laufzeiten oder gar durch vorzeitige Programmabbrüche, wie bei einer Division durch Null, Rückschlüsse gezogen werden.

Nicht alle Informationen, die einer UVC-Anwendung entlockt werden können, sind in glei-

cher Form noch in ferner Zukunft verfügbar. So könnten zukünftige UVC-Implementierungen die Programme vor der Ausführung intern optimieren und dadurch keine zuverlässigen Angaben bzgl. der ausgeführten Befehle ermöglichen. Auch können Laufzeiten aufgrund weiterentwickelter Hardware-Architekturen so stark abweichen, dass selbst Laufzeitverhältnisse nur schwer auszuwerten sein könnten. Was bleibt sind die spezifizierten Ausgaben, die per `OUT` entstehen. Um auch die Division durch Null testen zu können, müsste die Spezifikation für diesen speziellen Fall genaue Vorgaben machen.

Wie aber stellt man sicher, dass ein durch ein Testprogramm erkannter Fehler auch zu einer entsprechenden Ausgabe führt? Angenommen die Instruktion `OUT` ist bereits fehlerhaft implementiert und liefert als ausgegebene Bitfolge immer `0`. Helfen kann hier eine abgestufte Hierarchie der Testfälle. Zwei der ersten Tests müssen daher die Ausgaben für beide Fälle erzeugen. Gelingt die Interpretation durch das diese Tests ausführende Rahmenprogramm, sind fortan diese Sequenzen durch komplexere Testfälle weitestgehend zuverlässig nutzbar.

Daraus leitet sich eine generelle Vorgehensweise ab, die jedoch nicht perfekt umgesetzt werden kann, wie das folgende Beispiel zeigt. Gemäß dieser Vorgehensweise kann jeder komplexere Testfall voraussetzen, dass andere weniger komplexe Testfälle fehlerfrei abliefen. So benötigt z.B. die Ausgabe einer Bitfolge neben der Instruktion `OUT` auch die Instruktion `LOADC`, wobei nur das fehlerfreie Laden kleinerer Konstanten als Voraussetzung genügt. Erst danach kann auch das Laden größerer Konstanten geprüft werden. Wie aber kann ein Programm selbst entscheiden, ob es fehlerfrei abgearbeitet wurde? Hierzu bedarf es wiederum korrekt arbeitender Vergleichsoperationen usw.

Eine derart aufzustellende Hierarchie von Testfällen ist erkennbar nicht perfekt, aber durchaus dazu geeignet, eine fehlerhaft implementierte Instruktion zusammen mit dem zum Fehler führenden Kontext zu bestimmen. Zwar kann so der Fehler auf eins, zwei Instruktionen zuverlässig eingegrenzt werden, wirklich sicher kann man sich aber nicht sein.

Versuch einer Hierarchie

Wie bereits dargestellt wurden die Tests zu den identifizierten Bereichen im Praktikum von selbständig arbeitenden Programmierern entwickelt. Die über 127 Testprogramme sind dabei so entwickelt worden, dass Abhängigkeiten zueinander aufgrund der steigenden Komplexität dokumentiert und berücksichtigt sind. Durch die selbständige Entwicklung ist das jedoch nicht immer bereichsübergreifend gelungen und bedurfte einer intensiven Nachbearbeitung. Einzelne Testprogramme decken zudem identische Szenarien ab. Ebenso finden sich Testprogramme, die bereits interpretierte Eigenschaften des UVC testen. Ein Beispiel ist das Speichern von Registerinhalten im Speicher. Was geschehen soll, wenn die angegebene Länge nicht der zu speichernden Bitfolge entspricht, ist aus der Spezifikation nicht klar ableitbar. Werden die Bits links oder rechtsbündig im Speicher abgelegt? Testprogramme, die darauf abzielen, sind solange wertlos, wie die Spezifikation hier keine klaren Vorgaben macht. In diesem schwerwiegenden Fall kann nicht von einer Lücke in der Spezifikation ausgegangen werden, die durch ein Testprogramm geschlossen wird. Vielmehr wird der Bedarf zur Korrektur deutlich.

Trotz dieser Umstände sind immerhin noch 118 Testfälle überaus brauchbar. Eine Hierarchie, die sich aus den jeweiligen Abhängigkeiten ergibt, ist in Abbildung 10.1 dargestellt.

Bei dieser Darstellung ist weder Schönheit im Sinne von wenig Platz oder sich nicht über-

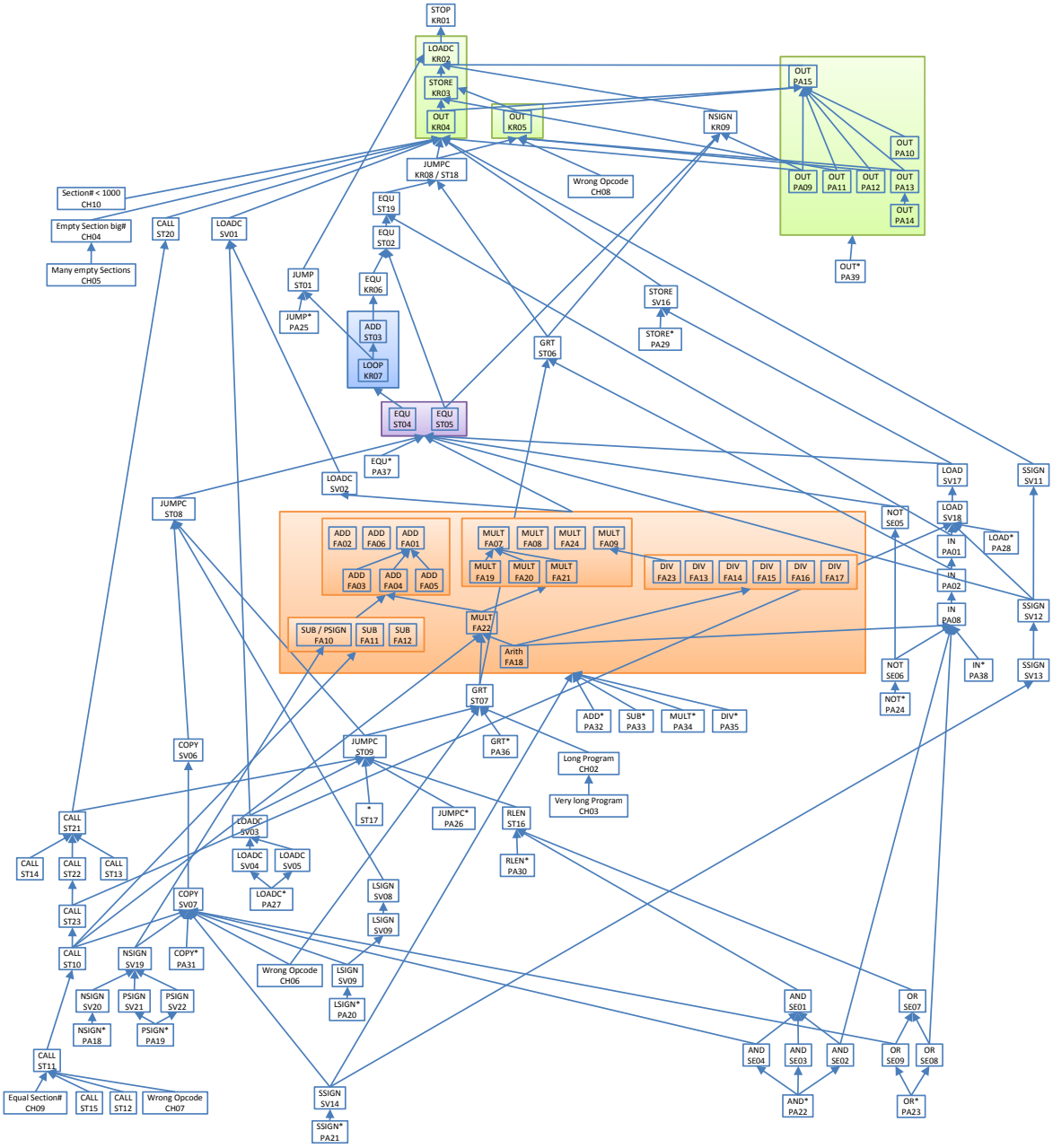


Abbildung 10.1.: Die Testprogramme in hierarchischer Anordnung gemäß ihren Abhängigkeiten zueinander.

schneidenden Linien noch die Übersichtlichkeit ein Kriterium. Das Ziel war zu zeigen, dass sich die Tests zyklenfrei gemäß ihrer Abhängigkeiten ordnen lassen. Alle Pfeile symbolisieren die jeweiligen Abhängigkeiten. Da jeder Pfeil nach oben zeigt, können die Tests von oben nach unten ausgeführt werden. Die Reihenfolge der Tests auf gleicher Höhe ist dabei nicht relevant. Eine solche Reihenfolge soll helfen, den eigentlichen Fehler schnell zu lokalisieren. Die Testprogramme gehen selbst davon aus, dass die zuvor ausgeführten Tests erfolgreich

ausgeführt wurden und können so über die Metadaten Hinweise geben, wo konkret der Fehler stecken könnte. Daraus leitet sich ab, dass mit dem Auftreten eines Fehlers das Testen beendet werden kann. Nachfolgende Tests können zwar noch Fehler aufzeigen, die Metadaten werden aber unbrauchbar, weil sie von dann ungültigen Annahmen ausgehen.

Deutlich an dieser Abbildung wird jedoch das Wirrwarr: Es existieren viele Abhängigkeiten unter den identifizierten Bereichen. Jedes Kästchen symbolisiert ein Testprogramm. Die größeren, farbigen Rahmen symbolisieren Teilbereiche und dienen lediglich der Übersichtlichkeit. In der ersten Zeile eines jeden Kästchens ist der Zweck grob angegeben, in der zweiten der Name des jeweiligen Testprogramms. Anhand identischer Präfixe lassen sich die Tests noch zu den ursprünglich identifizierten Bereichen zuordnen. Erkennbar an der Hierarchie ist jedoch, dass diese Zuordnung nur als Einstieg hilfreich war. Zudem ist Potential erkennbar: Benachbarte Tests könnten in einem Programm zusammengefasst werden. Die vielen Überschneidungen der die Abhängigkeiten symbolisierenden Pfeile sind nur zum Teil umgänglich. Einzelne Tests aus verschiedenen Bereichen müssen aufeinander aufbauen. Zugunsten der Struktur wurden Testprogramme hinzugenommen, die noch vorhandene Lücken schlossen. Diese Tests sind am Präfix „KR“ erkennbar.

Eine solche Struktur ist wichtig, um auch zukünftig entwickelte Tests einzupflegen bzw. um entscheiden zu können, ob ein neu entwickelter Test andere Tests vollständig ersetzen kann. Eine solche Grafik ist nicht unbedingt erforderlich und ab einer bestimmten Anzahl von Testprogrammen auch nicht mehr hilfreich. Letztendlich sollten die spezifizierten Abhängigkeiten z.B. in einer Datenbank verwaltet werden. Über eine solche Verwaltung muss sich schnell die Abarbeitungsreihenfolge bestimmen und die Abhängigkeiten auf Zyklenfreiheit testen lassen.

10.2.2. Das Konzept der Traces übertragen auf den UVC

Nachdem erkennbar ist, dass die hier entwickelte Hierarchie nicht immer allein ausreicht, um einen Fehler einer Instruktion zuordnen zu können, sollte dem zukünftigen Entwickler ein weiteres Hilfsmittel an die Hand gegeben werden: *Traces*.

Auf den UVC lässt sich das Prinzip des *Traces* nicht einfach übertragen. Ein kompletter Speicherauszug oder auch den Inhalt aller Register auszugeben ist aufgrund der fehlenden Begrenzung nicht zweckmäßig. Die eigenen Implementierungen speichern alle initialisierten Register in einer dynamischen Struktur. Hier wäre es möglich, nur die enthaltenen Register samt Inhalt in einem *Trace* auszugeben. Ohne Zweifel wäre so etwas hilfreich. Bei vorschneller Einarbeitung in die Spezifikation einer solchen Möglichkeit, den Inhalt aller initialisierten Register auszugeben, könnte schnell die angestrebte Universalität Schaden nehmen. Man stelle sich vor, der UVC würde auf einem UVC emuliert. Aufgrund seiner in beliebiger Anzahl vorhandenen Register könnten die Register des emulierten UVC einfach durch eigene Register realisiert werden. Der UVC bietet aber einem UVC-Programm selbst keine Möglichkeit zu prüfen, ob Register initialisiert sind. Derartige Befehle gibt es nicht und der Zugriff auf nicht initialisierte Register führt entweder zum Programmabbruch mit Fehlermeldung (Implementierung IBM) oder aber selbst zur Initialisierung (eigene Implementierungen). Wird dagegen zu viel durch die Spezifikation gefordert, wie z.B. das Ausgeben eines vorgegebenen Speicherbereichs oder einer Liste vorgegebener Register, verzögert das eventuell die Implementierung mehr als es dem Programmierer bei der Implementierung hilft.

Nahezu unumgänglich ist die Verwendung eines *CPU do-Instruction Loop* [Phillips, 2010]. Während dieser Schleife, die jeweils genau eine Instruktion abarbeitet, kann sehr einfach auf bestimmte Informationen zugegriffen werden. Zum einen ist das die Anzahl der bisherigen Schleifendurchläufe und zum anderen der jeweils auszuführende Befehl. Ein *Trace* könnte somit ohne größere Probleme alle ausgeführten Befehle der Reihe nach enthalten. Auch möglich ist das Ausgeben der Summe aller ausgeführten Instruktionen, sowie deren Zuordnung zu den jeweiligen Opcodes. Das Ausgeben einer solchen „Statistik“ periodisch nach einer vorgegebenen Anzahl von ausgeführten Instruktionen ist ebenfalls leicht zu realisieren.

Nachdem ein vollständiger Speicherauszug wenig zweckmäßig erscheint, stellt sich die Frage, was stattdessen mit wenig Aufwand in einem solchen *Trace* gespeichert werden kann und dennoch hilfreich ist. In eigenen Tests erwies sich die Angabe der aktuell durch die Instruktion manipulierten Registerinhalte als äußerst hilfreich, wenn zu jeder Instruktion angegeben wird, welche Register beteiligt sind. Zu einem Großteil sind das ein bis zwei Register oder das *Condition Flag*. Bei indirekten Registerzugriffen sollte zudem die resultierende Registernummer angegeben werden. Beim Speicherzugriff über `STORE` erwies sich eine Ausgabe des Speichers als wenig hilfreich, da hier für eine solche Ausgabe die gleichen Programmabschnitte genutzt werden, wie für den Speicherzugriff selbst. Es ist demnach ebenso wenig klar, ob der Fehler beim Lesen oder beim Schreiben auftrat, als wenn das spätere Laden des Speicherinhalts in ein Register abgewartet wird.

Es muss jedoch erwähnt werden, dass sich UVC-Anwendungen auch ohne einen solchen *CPU do-Instruction Loop* ausführen lassen. So zeigt die Arbeit von Müller [2011] die Möglichkeit auf, UVC-Anwendungen einmalig in eine Hochsprache zu überführen, um Laufzeiten zu reduzieren. Bei einem solchen Schritt wird gezielt das erwähnte Schleifenkonstrukt umgangen. Die oben dargestellten Informationen sind dann nicht mehr trivial ausgebbar. Auch die in Abschnitt 5.3.4 vorgestellte Optimierungstechnik wirkt hier entgegen. Es ist aber in beiden Fällen davon auszugehen, dass diese Schritte nicht mit der initialen Implementierung vollzogen werden. Eher vorstellbar ist, dass bereits korrekte Implementierungen vorliegen und optimierte Versionen für bestimmte Anwendungsfälle zusätzlich erstellt werden sollen.

Zu jedem Testprogramm sollte daher ein *Trace* eines fehlerfreien Durchlaufs archiviert werden und eine kleine Statistik, die angibt, welche Instruktionen wie oft genutzt wurden. Ein kleines Beispiel einer Schleife soll die Umsetzung des Gedankens von Phillips verdeutlichen. Als Programm dient ein kleines Schleifenkonstrukt mit drei Durchläufen:

```
TestLoop
1000
LOADC     1,1   2  3          # Anzahl der Schleifendurchläufe
LOADC     1,2   0  0          # Startwert des Schleifenzählers
LOADC     1,3   1  1          # Schleifeninkrement
label: loop
ADD       1,2   1,3            # +1
EQU       1,2   1,1            # fertig?
JUMPC     1,0   fertig         # ja...
JUMP      1,0   loop           # nein, erneut iterieren
label: fertig
STOP
```

Kommt es bei diesem Programm zu einem Fehler, in dessen Folge die Abarbeitung nie

abbricht, kann das mehrere Ursachen haben. Zum Ersten könnte eine der verwendeten Instruktionen zum Laden der konstanten Bitfolgen fehlerhaft arbeiten und so falsche initiale Registerinhalte bewirken. Ebenfalls denkbar wäre ein fehlerhaft arbeitender Additionsbefehl, der derart falsch addiert, dass die `3` als Inhalt des Registers `1,2` nie erzeugt wird. Als dritte Fehlerquelle könnte der implementierte Vergleichsoperator in Frage kommen. Liefert dieser an entscheidender Stelle ebenfalls kein gesetztes *Condition Flag* bricht die Schleife auch nicht ab. Als vierte Ursache kommt ein fehlerhaft implementierter Sprungbefehl in Frage. Führt dieser einen Sprung in einen leeren Bereich des Segments aus, könnte dieser als Folge von Nullen interpretiert werden, was fortwährend `NSIGN 0,0` entspricht. Im Folgenden sind zwei *Traces* gekürzt gegeben; rechts der archivierte:

```
LOADC     1,1   2   3
(00000002) 00000003
LOADC     1,2   0   0
(00000000)
LOADC     1,3   1   1
(00000001) 00000001
ADD       1,2   1,3
(00000001) 00000001
EQU       1,2   1,1
false
LOADC     1,0   32   1003
(00000020) 000003EB
JUMPC     1,0
LOADC     1,0   32   315
(00000020) 0000013B
JUMP      1,0
ADD       1,2   1,3
(00000002) 00000002
EQU       1,2   1,1
false
LOADC     1,0   32   1003
(00000020) 000003EB
JUMPC     1,0
LOADC     1,0   32   315
(00000020) 0000013B
JUMP      1,0
ADD       1,2   1,3
(00000002) 00000003
EQU       1,2   1,1
false
...
```

```
LOADC     1,1   2   3
(00000002) 00000003
LOADC     1,2   0   0
(00000000)
LOADC     1,3   1   1
(00000001) 00000001
ADD       1,2   1,3
(00000001) 00000001
EQU       1,2   1,1
false
LOADC     1,0   32   1003
(00000020) 000003EB
JUMPC     1,0
LOADC     1,0   32   315
(00000020) 0000013B
JUMP      1,0
ADD       1,2   1,3
(00000002) 00000002
EQU       1,2   1,1
false
LOADC     1,0   32   1003
(00000020) 000003EB
JUMPC     1,0
LOADC     1,0   32   315
(00000020) 0000013B
JUMP      1,0
ADD       1,2   1,3
(00000002) 00000003
EQU       1,2   1,1
true
...
```

Im Vergleich mit dem *Trace* des fehlerhaft implementierten UVC wird die Ursache sofort ersichtlich: Die Instruktion `EQU` arbeitet fehlerhaft. Die letzte dargestellte Ausführung sollte `true` als Ergebnis liefern, da die Inhalte der Register `1,2` und `1,1` gleich sind.

Schleifen werden sehr häufig innerhalb der erstellten Testprogramme genutzt. Diese Funktionalität muss daher in der Testreihenfolge weit vorn überprüft werden. Aber alle einzelnen Aspekte durch separate Testprogramme zu prüfen, ist nicht abschließend möglich; es bleiben nicht klar zuzuordnende Fehlerquellen. Mit Hilfe des archivierten *Traces* kann die Fehlerur-

sache deutlich enger eingegrenzt werden.

Dabei ist zu beachten, dass solche zusätzlichen Ausgaben nicht den eigentlichen Ausgabestrom stören, also zu- und abschaltbar sind. Das genaue Format der Ausgaben zu treffen ist dabei weniger wichtig, wie gesammelte Erfahrungen während der *Zeitreise* zeigen. Dabei wurden sehr umfangreiche *Traces* unterschiedlichst formatierter Ausgaben verglichen. Während das oben dargestellte Format die Information recht knapp und damit übersichtlich darstellt, lassen sich mit FORTRAN IV Ausgaben mit exakt dieser Formatierung nicht ohne erheblichen Aufwand erzeugen. So nutzt z.B. die Ausgabe von Zahlen eine feste Anzahl von Zeichen. Einfacher war dagegen die Entwicklung eines Tools, das jeweils die Informationen extrahiert, ggf. konvertiert und diese dann vergleicht. Die folgenden drei Ausschnitte zeigen jeweils die gleiche Information; links die Ausgabe der in Ada implementierten Version, rechts die der FORTRAN 77- und unten die der FORTRAN IV-Version:

```
...                                  ...
GRT       0,20  1,140                 0
false                                 "GRT" 0 20 1 140
LOADC     1,2   32   9200            (0000000020)  00000023F0
(00000020) 00000023F0                 "LOADC" 1 2 32
JUMPC     1,2                         "JUMPC" 1 2
...                                  ...
```

```
...
                    0
 GRT                   0                  20                    1
                   140
(        20)         23F0
 LOADC                      1                   2                    32
 JUMPC                      1                   2
...
```

Traces mit genau diesen Informationen werden daher die Testfälle begleiten. Das Format wird dabei fixiert. *Traces*, die zukünftig entwickelte Testprogramme begleiten, müssen diesem Format exakt folgen, damit die Auswertung ganzer Testreihen nicht durch eine Vielfalt an Formaten unnötig erschwert wird. Dieses Format wiederum muss für die Auswertung der Testergebnisse bekannt sein und sollte in einem die Testsammlung begleitenden Dokument ebenso spezifiziert sein wie die anderen Test-Metadaten.

Aber weder die Ausgabe noch ein bestimmtes Format solcher Informationen sollte durch zukünftige Spezifikationen von den UVC-Implementierungen gefordert werden, selbst die optionale Ausgabe eines *Traces* nicht. Allein das fortwährende Prüfen, ob eine Ausgabe erfolgen muss, verbraucht Rechenzeit. Nicht jede Programmiersprache bzw. nicht jeder Compiler unterstützt die parameterabhängige Übersetzung bzw. Auslassung bestimmter Zeilen.[2]

[2] Frühere Versionen der FORTRAN-Compiler unterstützten Debug-Instruktionen. Das sind mit einem `D` am Anfang gekennzeichnete Programmzeilen, die nur bei gesetztem Compiler-Schalter berücksichtigt werden. Für den genutzten Compiler für FORTRAN 77 ist das die Option `-d_lines`. Somit sorgte dieser Schalter zum einen für eine effiziente Implementierung, da die fortwährende Prüfung entfällt, und zum anderen für die Ausgabe der *Trace*-Informationen.

10.3. Evaluation: Erste Ergebnisse

In diesem Abschnitt gilt es zu bewerten, ob sich eine Testsammlung wie hier anvisiert eignet und damit den Aufwand lohnt. Aber wie kann man das bereits heute bewerten?

Was bedeutet in diesem Zusammenhang Eignung? Geeignet ist ein Test, wenn er mit großer Sicherheit einen im Programm vorhandenen Fehler aufdeckt. Eine Testsammlung begleitend zur Spezifikation des UVC ist dann geeignet, wenn sich durch die Ausführung der Tests Programmierfehler oder unbeabsichtigte Abweichungen von der in der Spezifikation geforderten Funktionalität offenlegen und z.B. aufgrund der Testbeschreibungen auch schnell identifizieren und damit beheben lassen. Auch wurde bereits erarbeitet, dass sich Tests eignen, um eventuelle Ungenauigkeiten in der Spezifikation auszugleichen.

Wenige, aber überaus wertvolle Erfahrungen, die mit der Testsammlung bereits gemacht wurden, prägen diesen Abschnitt. Zum einen wurde direkt nach dem Erstellen der 127 Testprogramme die Implementierung für die SUN Enterprise 10000 erstellt (siehe Abschnitt 6.6) und dabei die Testsammlung bereits genutzt. Zum anderen gibt es bereits mehrere Implementierungen und Portierungen, die „nachträglich“ die Tests ausführten.

Die Erfahrungen im Umgang mit *Traces* wurden bereits dargestellt und der als sinnvoll und bei der Fehlersuche überaus hilfreiche Inhalt daraus abgeleitet. Somit steht fest, dass für eine noch zu erstellende bzw. zu ergänzende Testsammlung für den UVC nach weiterentwickelter Spezifikation *Traces* erstellt werden und wie deren Inhalte zu gestalten sind.

10.3.1. Bewertung der Testergebnisse

Als Abschluss des Praktikums wurden die Tests auf den vorhandenen Implementierungen und Portierungen ausgeführt. Von den insgesamt 11 betrachteten UVC-Laufzeitumgebungen stammen 10 aus der *Zeitreise* (siehe Kapitel 6) und allein 8 davon stammen von nur einem Autor. Ebenfalls betrachtet wurde die Implementierung in Java von IBM (siehe Abschnitt 3.4.1), wobei Änderungen an zahlreichen Testprogrammen notwendig waren, um aussagekräftige Ergebnisse zu erhalten. Der in C++ implementierte UVC entzog sich den Tests durch Inkompatibilität. Die Anpassung der Testprogramme bzw. der UVC-Implementierung in C++ wäre sehr aufwendig und würde ohnehin kein sauber interpretierbares Bild abgeben.

Darstellung der Testergebnisse

Die Ergebnisse der Testprogramme, in noch ursprünglicher Ordnung nach den identifizierten Bereichen, sind in Abbildung 10.2 zu sehen.

Vor der eigentlichen Bewertung muss geklärt werden, was die Farben in der Abbildung bedeuten. In den linken Spalten sind die Testnamen und kurze Beschreibungen gegeben, die im Detail für diesen Abschnitt jedoch weniger relevant sind.

Grau unterlegte Tests sind jetzt nicht mehr in der Testsammlung enthalten, da deren Testergebnisse wenig Aussagekraft haben. So testen sie Dinge, die nur indirekt etwas mit dem UVC zu tun haben, wie die Tests im Bereich der Eingabe. Die Verarbeitung, also die Aufbereitung der ankommenden Nachrichten muss nicht zwingend im UVC selbst geschehen. Das

Test	Maschine Implementierung	Cyber960 FORTRAN IV	Cyber960 FORTRAN IV_ECS	Majestix FORTRAN IV_port	Sun FORTRAN IV_port	Majestix FORTRAN 77	Sun FORTRAN 77_port	Sun FORTRAN 95	PC FORTRAN 95_port	PC Ada 83 regulär	PC Ada 83 optimiert	PC Java (IBM)
Programmaufbautests												
CH01	einf. kurzes Programm	x	x	x	x	x	x	x	x	x	x	x
CH02	eine Section, viele Anweisungen	x	x	x	x	x	x	x	x	x	x	x
CH03	eine Section, sehr viele Anweisungen	x	x	x	x	x	x	x	x	x	x	o
CH04	Prog enth. leere Section großer Nr	x	x	x	x	x	x	x	x	x	x	o
CH05	viele leere Sections	o	o	o	o	x	x	x	x	x	x	o
CH06	falsche Opcodes an unrelevanten Stellen	x	x	x	x	x	x	x	x	o	o	x
CH07	falsche Opcodes in unrelevanten Sections	o	o	o	x	x	x	x	x	o	o	x
CH08	führt falsche Opcodes aus	x	x	x	x	x	x	x	x	x	x	o
CH09	enth. Sections mit gleicher Nummer	x	x	x	o	o	o	x	x	o	o	o
CH10	enth. Sections mit Nr < 1000	x	x	x	x	x	x	x	x	x	x	o
Eingabetests												
PA01	Messagetyp 0	x	x	x	x	x	x	x	x	x	x	o
PA02	Messagetyp größer 0	x	x	x	x	x	o	x	x	x	x	o
PA03	Messagetyp größer 32 Bit	x	x	x	x	x	x	x	x	o	o	o
PA04	MessagLänge = 0	x	x	x	x	x	x	x	x	x	x	o
PA05	Ignorierte Message bei Länge 0	x	x	x	x	x	x	x	x	x	x	o
PA06	MessageLänge größer 32 Bit	o	o	o	o	o	x	x	x	o	o	o
PA07	IN verarbeitet linksbündig?	o	o	o	o	o	o	x	x	o	o	o
PA08	IN übeschreibt keine angrenzenden Daten	x	x	x	x	o	o	x	x	x	x	o
Ausgabetests												
PA09	Negativer Messagetyp	x	x	x	x	x	x	x	x	x	x	o
PA10	Messagetyp mit mehr als 32 Bit	o	o	o	o	x	x	x	x	x	x	o
PA11	Messagelänge 0	x	x	x	x	x	x	x	x	x	x	o
PA12	Messagelänge größer 32 Bit	o	o	o	o	x	x	x	x	x	x	o
PA13	rechtsbündige Ausgabe	x	x	x	x	o	o	x	x	o	o	o
PA14	Messagelänge negativ	x	x	x	x	x	x	x	x	o	o	o
PA15	Null Message	x	x	x	x	x	x	x	x	o	x	o
PA16	k.A. (Ausgabe negativer Registerinhalt)	x	x	x	x	x	x	x	x	o	x	o
PA17	hochfrequentierte Ausgabe von 0	x	x	x	x	x	x	x	x	x	x	o
Indirekte Registeradressierung												
PA18	NSIGN	x	x	x	x	x	x	x	x	x	x	o
PA19	PSIGN	x	x	x	x	x	x	x	x	x	x	o
PA20	LSIGN	x	x	x	x	o	o	x	x	x	x	o
PA21	SSIGN	x	x	x	x	x	x	x	x	x	x	o
PA22	AND	x	x	x	x	x	x	x	x	x	x	o
PA23	OR	x	x	x	x	x	x	x	x	x	x	o
PA24	NOT	x	x	x	x	x	x	x	x	x	x	o
PA25	JUMP	x	x	x	x	x	x	x	x	x	x	o
PA26	JUMPC	x	x	x	x	x	x	x	x	x	x	o
PA27	LOADC	x	x	x	x	x	x	x	x	x	x	o
PA28	LOAD	x	x	x	x	x	x	x	x	x	x	o
PA29	STORE	x	x	x	x	x	x	x	x	x	x	o
PA30	RLEN	x	x	x	x	x	x	x	x	x	x	o
PA31	COPY	x	x	x	x	x	x	x	x	x	x	o
PA32	ADD	x	x	x	x	x	x	x	x	x	x	o
PA33	SUB	x	x	x	x	x	x	x	x	x	x	o
PA34	MULT	x	x	x	x	x	x	x	x	x	x	o
PA35	DIV	x	x	x	x	x	x	x	x	x	x	o
PA36	GRT	x	x	x	x	x	x	x	x	x	x	o
PA37	EQU	x	x	x	x	x	x	x	x	x	x	o
PA38	IN	o	o	o	o	o	o	x	x	x	o	o
PA39	OUT	x	x	x	x	x	x	x	x	x	x	o
Logische Operatoren												
SE01	AND	x	x	x	x	x	x	x	x	x	x	x
SE02	AND manuell	x	x	x	x	x	x	x	x	x	x	x
SE03	AND Stresstest	x	x	x	x	x	x	x	x	x	x	x
SE04	AND Stresstest	x	x	x	x	x	x	x	x	x	x	x
SE05	NOT	x	x	x	x	o	o	x	x	x	x	o
SE06	NOT manuell	x	x	x	x	x	x	x	x	x	x	x
SE07	OR	x	x	x	x	x	x	x	x	x	x	x
SE08	OR manuell	x	x	x	x	x	x	x	x	x	x	o
SE09	OR Stresstest	x	x	x	x	x	x	x	x	x	x	x
Arithmetik												
FA01	ADD mit kleineren Werten	o	o	o	x	x	x	x	x	x	x	x
FA02	ADD mit 0	x	x	x	x	x	x	x	x	x	x	x
FA03	ADD kommutativ	x	x	x	x	x	x	x	x	x	x	x
FA04	ADD assoziativ	x	x	x	x	x	x	x	x	x	x	x
FA05	ADD mit negativen Zahlen	x	x	x	x	x	x	x	x	x	x	x
FA06	ADD korrigiert Länge	o	o	o	x	x	x	x	x	x	x	x
FA07	MULT mit kleineren Werten	x	x	x	x	x	x	x	x	x	x	x
FA08	MULT mit 1	x	x	x	x	x	x	x	x	x	x	x
FA09	MULT mit 0	x	x	x	x	x	x	x	x	x	x	x
FA10	SUB als Teil einer Schleife	x	x	x	x	x	x	x	x	x	x	x
FA11	SUB korrigiert Länge	o	o	o	o	x	x	x	x	x	x	x
FA12	SUB mit 0	x	x	x	x	x	x	x	x	x	x	x
FA13	DIV mit 1	x	x	x	x	x	x	x	x	x	x	x
FA14	DIV mit negativen Zahlen	x	x	x	x	x	x	x	x	x	x	x
FA15	DIV mit 0, also (0/X)	x	x	x	x	x	x	x	x	x	x	x
FA16	DIV mit führenden Nullen	x	x	x	o	x	x	x	x	x	x	x
FA17	DIV mit 0 (X/0)	x	x	x	x	x	x	x	x	x	x	x
FA18	DIV manuell	x	x	x	x	x	x	x	x	x	x	x
FA19	MULT mt negativen Zahlen	x	x	x	x	x	x	x	x	x	x	x
FA20	MULT kommutativ	x	x	x	x	x	x	x	x	x	x	x
FA21	MULT assoziativ	x	x	x	x	x	x	x	x	x	x	x
FA22	MULT distributiv	x	x	x	x	x	x	x	x	x	x	x
FA23	MULT inverses Element	x	x	x	x	x	x	x	x	x	x	x
FA24	MULT korrigiert Länge	x	x	x	x	x	x	x	x	x	x	x
Programmablauf												
ST01	JUMP einfach	x	x	x	x	x	x	x	x	x	x	x
ST02	EQU grundlegend	x	x	x	x	x	x	x	x	x	x	x
ST03	kleine Schleife	x	x	x	x	x	x	x	x	x	x	x
ST04	EQU mit vielen Vergleichen	x	x	x	x	x	x	x	x	x	x	x
ST05	EQU mit speziellen Vergleichen	x	x	x	x	x	x	x	x	x	x	x
ST06	GRT grundlegend	x	x	x	x	x	x	x	x	x	x	x
ST07	GRT aufbauend	x	x	x	x	x	x	x	x	x	x	x
ST08	JUMPC grundlegend	x	x	x	x	x	x	x	x	x	x	x
ST09	Unverändertes Condition flag	x	x	x	x	x	x	x	x	x	x	x
ST10	Rekursion (Exp)	o	o	o	x	x	x	x	x	x	x	x
ST11	Rekursion (Summe)	o	o	o	x	x	x	x	x	x	x	x
ST12	Rekursion (Fibunacci)	o	o	o	x	x	x	x	x	x	x	o
ST13	CALL self + Offset	x	x	x	x	x	x	x	x	x	x	x
ST14	CALL mit Sectionummer	o	o	o	x	x	x	x	x	x	x	x
ST15	Rekursion (Ackermann)	o	o	o	x	x	x	x	x	x	x	x
ST16	RLEN setzt Vorzeichen zurück	x	x	x	x	x	x	x	x	o	o	x
ST17	indirekter Registerzugriff	x	x	x	x	x	x	x	x	x	x	x
ST18	Defaultwert des CF	o	o	o	o	o	o	x	x	x	x	x
ST19	Nicht Zurücksetzen des CF	x	x	x	x	x	x	o	o	x	o	x
ST20	CALL grundlegend	o	o	o	x	x	x	x	x	x	x	x
ST21	CALL mit Parameter	o	o	o	x	x	x	x	x	x	x	x
ST22	CALL mit Parameter intensiv	o	o	o	x	x	x	x	x	x	x	x
ST23	CALL mit Übergabe in Speicher	o	o	o	x	x	x	x	x	x	x	x
Speicherzugriff												
SV01	LOADC für Testnummer	x	x	x	x	x	x	x	x	x	x	o
SV02	LOADC Vergleiche	x	x	x	x	x	x	x	x	x	x	o
SV03	LOADC überschreiben	x	x	x	x	x	x	x	x	x	x	o
SV04	LOADC unpassender Länge	x	x	x	x	x	x	x	x	x	x	o
SV05	LOADC belässt Vorzeichen	o	o	o	o	o	o	o	o	o	o	o
SV06	COPY grundlegend	x	x	x	x	x	x	x	x	x	x	o
SV07	COPY negative Zahlen	x	x	x	x	x	x	x	x	x	x	o
SV08	LSIGN positiv	x	x	x	x	o	o	x	x	x	x	x
SV09	LSIGN negativ	x	x	x	x	o	o	x	x	x	x	x
SV10	Nutzereingabe mit LSIGN vergleichen	x	x	x	x	x	x	x	x	x	x	x
SV11	SSIGN in unitialisierten Speicher	x	x	x	x	x	x	x	x	o	x	o
SV12	SSIGN positiv	x	x	x	x	x	x	x	x	x	x	x
SV13	SSIGN negativ	x	x	x	x	x	x	x	x	x	x	x
SV14	Nutzereingabe mit SSIGN vergleichen	x	x	x	x	x	x	x	x	x	x	x
SV15	STORE grundlegend	x	x	x	x	x	x	x	x	x	x	x
SV16	STORE größerer Wert	x	x	x	x	x	x	x	x	x	x	x
SV17	STORE und LOAD	x	x	x	x	x	x	x	x	x	x	x
SV18	STORE und teilweise LOAD	x	x	x	x	x	x	x	x	x	x	x
SV19	NSIGN und ADD	x	x	x	x	x	x	x	x	x	x	x
SV20	NSIGN und EQU	x	x	x	x	x	x	x	x	x	x	x
SV21	PSIGN und NSIGN	x	x	x	x	x	x	x	x	x	x	x
SV22	PSIGN und EQU	x	x	x	x	x	x	x	x	x	x	x

Abbildung 10.2.: Die Ergebnisse der 127 Testprogramme

Nachrichtenformat ist zudem sehr genau vorgeschrieben. Mit den Tests und den dazugehörigen Testnachrichten wird vielmehr geprüft, wie mit abweichenden Nachrichten umgegangen wird. Sollen heute bereits aussagekräftige Testprogramme bzw. zu testende Eingabedaten entwickelt werden, müssen sich diese innerhalb des spezifizierten Rahmens bewegen, also immer exakt 32 Bit für Typ- und Längenangaben nutzen und exakt so viele Bits als Nachricht senden, wie mittels der Längenangabe codiert wurde. Abweichungen machen keinen Sinn.

Der eine grün unterlegte Test deckt präzise das ab, was ein anderer Test bereits leistet. Er ist jetzt ebenfalls nicht mehr in der Testsammlung enthalten.

Die orange unterlegten Testfälle „interpretieren“ die Spezifikation auf ihre Art. Macht das ein Programmierer auf seine Art anders, kommt es hier zu einem „Fehler.“ Diese Tests verbleiben in der Testsammlung, da die Spezifikation an diesen Stellen konkretisiert wird – und zwar ebenso, wie es die Entwickler der Testprogramme annahmen.

Die Testergebnisse selbst sind intuitiv rot bzw. grün unterlegt. Gelb ist nur in der letzten Spalte zu finden. Viele Tests sind während der Entwicklung ausschließlich auf einer in Ada implementierten Version des UVC getestet worden. Dabei ist nicht aufgefallen, dass bestimmte Fehler der Implementierung von IBM dazu führen, dass ganze Testgruppen immer Fehler anzeigen. Z.B. testen `PA01` und `CH10` Unterprogramme mit Nummern kleiner `1001`. Folgende Tests nutzen dann stets die Nummer `1000`. Diese Tests wurden nachträglich angepasst. Finden diese keinen zusätzlichen Fehler, gibt es die Farbe Gelb.

Bewertung

Wie sind diese Testergebnisse zu bewerten? Hier gibt es gleich mehrere Facetten, die im Folgenden beleuchtet werden.

Zum Ersten ist es die Frage, ob ein Programmierer während der Implementierung des UVC tatsächlich einen Nutzen daraus ziehen kann. Die einzige Implementierung, die bereits während der Erstellung auf die Testsammlung zugreifen konnte, ist die Reimplementierung für die Sun. Deren Ergebnisse sind in den Spalten 7 und 8 zu sehen. Deutlich erkennbar ist, dass im Vergleich zu den anderen Spalten lediglich zwei rote Einträge existieren, also lediglich zwei Fehler gefunden wurden. Einer dieser „Fehler“ ist absichtlich nicht behoben worden, weil sonst die Decoder-Anwendungen von IBM nicht ausführbar sind. Ein weiterer Fehler wird von einem Testprogramm aufgezeigt, das die Spezifikation bereits „interpretierte“ – und zwar anders als der Programmierer. Diese Abweichung wirkt sich jedoch (noch) nicht auf vorhandene UVC-Anwendungen aus und wurde daher so belassen. Insgesamt ließ sich feststellen, dass die UVC-Implementierung gezielt hinsichtlich der Tests entwickelt wurde. Die Implementierung war früh soweit, erste Testprogramme ausführen zu können. Kleinere Fehler konnten auf diese Weise frühzeitig abgestellt werden. Obwohl die äußeren Umstände ähnlich waren (neue Programmiersprache, unbekannte Hardware usw.), ist die Implementierungszeit etwas kürzer. Die in Schiller [2012] aufgelisteten Testzeiten liegen dagegen sehr deutlich unter den eigenen, was klar der vorhandenen Testsammlung zugeschrieben werden kann.

Die anderen Implementierungen und Portierungen weisen auch nach der gedanklichen Streichung aller unrelevanter Zeilen noch genügend „echte“ Fehler auf. Zweifellos wären diese mit einer bereits zur Entwicklungszeit vorhandenen Testsammlung sofort zu beheben gewesen. So aber musste man davon ausgehen, dass die Implementierung bereits korrekt UVC-

Anwendungen ausführen kann. Im Kontext der Langzeitarchivierung könnte das fatale Auswirkungen hinsichtlich der Authentizität haben. Alle bisherigen Anwendungen konnten scheinbar fehlerfrei ausgeführt werden. Auch in diesem Fall wird deutlich, dass eine Testsammlung von großem Nutzen ist – insbesondere für den Nutzer des UVC.

Dass die Testergebnisse sehr präzise Schlüsse zulassen, wo die erkannten Fehler stecken, wird durch die Interpretation der vierten Spalte deutlich. Das ist die letzte Portierung des ursprünglich in FORTRAN IV implementierten UVC. Dieser enthält weniger Fehler als die Vorgänger bzw. das Original. Das liegt daran, dass mit der Portierung und dem weiteren Testen selbst Fehler aufgefallen sind, die dann nur noch in der Portierung behoben wurden. Die aufgezeigten Fehler lassen sich sehr präzise diesen angepassten Stellen zuordnen. Genau anders herum ist es bei den in FORTRAN 77 implementierten Versionen. Hier wurde die originale Version nach der Portierung weiterentwickelt und ist somit aktueller als die Portierung.

Interessant ist auch, dass die verschiedenen Implementierungen Fehler an unterschiedlichen Stellen aufweisen. Und das, obwohl sie fast alle aus derselben Feder stammen. Im Wesentlichen ist das auf die jeweils unterschiedlichen Realisierungsmöglichkeiten der verwendeten Sprachen zurückzuführen und den sich dadurch immer wieder neu auftuenden Fehlerquellen. Offensichtlich muss ein Programm auch dann ausführlich getestet werden, wenn es zum wiederholten Mal vom gleichen Programmierer erstellt wurde.

Eigentlich wurde vermutet, dass sich mit den Testergebnissen ein leicht anderes Bild zeigt. Überraschender Weise wurden aber die Portierungen scheinbar fehlerfrei durchgeführt. Versteckte Fehler wie durch die Umstellung von 60 auf 64 Bit sind keine gefunden worden. Auch sind offensichtlich alle genutzten intrinsischen Funktionen jeweils so abgebildet worden, dass der UVC intern auf gleiche Weise arbeitet. Zu erwarten wäre gewesen, dass sich einige neue Fehler bei den Portierungen finden. In der Tat gibt es in der vierten Spalte zwei neue Fehler, die jedoch nachträglich „hinein optimiert“ wurden. Diese Fehler wären durch die konsequente Nutzung der Testsammlung sofort aufgefallen und hätten leicht behoben werden können.

Fazit

Eine solche Testsammlung ist unbedingt der Spezifikation beizufügen. Zum einen ist dadurch mit einer verkürzten Implementierungszeit zu rechnen, für die es erste Indizien gibt.

Zum anderen sind auch mit einer weiterentwickelten Spezifikation Interpretationsspielräume nicht zu vermeiden. Solche Lücken können durch Testprogramme geschlossen werden, noch bevor diese entdeckt werden. Möglich werden solche Lücken auch durch den Wandel der Zeit. Verändert sich der Kontext zunehmend, z.B. Sprache, Begrifflichkeiten, intuitive Annahmen usw., können die Testprogramme helfen, die Spezifikation im Sinne der ursprünglichen Entwickler so zu überarbeiten, dass auch noch in ferner Zukunft die archivierten UVC-Anwendungen authentische Ergebnisse liefern werden.

Neben diesen Erkenntnissen ist ein Aspekt sehr deutlich geworden: Die Mitwirkung an der Erstellung der Testsammlung von „außen“ lieferte interessante und überaus wertvolle Testfälle und zeigte bisher noch nicht wahrgenommene Problemstellen in der Spezifikation auf. Folgerichtig kann auch eine noch zu erstellende Testsammlung nicht ausschließlich vom Entwickler bzw. Weiterentwickler der Spezifikation erstellt werden. Vielmehr bedarf es der moderierten Mitwirkung von interessierten Programmierern, die den UVC selbst implementieren oder An-

wendungen für diesen entwickeln wollen. Eine regelmäßige Aktualisierung an zentraler Stelle wird dazu zunächst im Rahmen des Webauftritts des Instituts realisiert.

Es bleibt anzumerken, dass diese hier bewertete Testsammlung für den in Lorie und van Diessen [2005] spezifizierten UVC mit den in Abschnitt 3.4.1 aufgeführten Anpassungen erstellt wurde. Die Anpassung an die in der vorliegenden Arbeit weiterentwickelten Spezifikation wird direkt im Anschluss der Veröffentlichung vollzogen. Es ist mit einer deutlich umfangreicheren Sammlung zu rechnen, da neben der erweiterten Funktionalität auch deutlich mehr Möglichkeiten der Operanden getestet werden müssen. Dieser Aufwand ist aber lediglich einmalig im „Jetzt“ zu leisten.

11. Tetris als Beispiel einer interaktiven Anwendung

Bei der Archivierung von komplexen digitalen Objekten geht es nicht allein darum, statische Dokumente zu archivieren. Immer mehr in den Vordergrund rücken auch interaktive Anwendungen. Hedstrom und Lampe [2001] untersuchten anhand eines Computerspiels die Eignung verschiedener Archivierungsstrategien, insbesondere Migration und Emulation. Sie betrachteten ein Computerspiel, weil es Interaktivität und grafische Nutzeroberflächen kombiniert. Beides ist relevant für viele zu archivierende komplexe digitale Objekte. Als Beispiel führen sie Webseiten auf. Computerspiele sind aber nicht nur als Vertreter geeignet. Sie bilden eine eigene Gruppe, die im Rahmen der Langzeitarchivierung zunehmend mehr Aufmerksamkeit erlangt [Hedstrom et al., 2006; Guttenbrunner et al., 2008].

Mit dem Spiel „Chuckie Egg“ untersuchten Hedstrom et al. [2006] was beim Nutzer eines Spiels in Erinnerung bleibt. Dabei fiel auf, dass im Vergleich mit dem Original und der Emulation auch die Reimplementierung als authentisch empfunden wurde. Authentizität ist im Falle eines Computerspiels vielschichtig. Bedienelemente, Grafik und Sound können ebenso ausschlaggebend sein wie der Schwierigkeitsgrad oder die Punktevergabe. In dieser komplexen Betrachtung der Authentizität erwies sich die Reimplementierung als vergleichbar effektiv.

In einem eigenen Experiment wurde untersucht, in wie weit sich der UVC eignet, interaktive grafische Programme als digitale Objekte zu archivieren. Zunächst werden die Eckpunkte des Experiments motiviert, anschließend die Implementierung an ausgewählten Details beschrieben und abschließend der Erfolg bewertet.

11.1. Eckpunkte des Experiments

Ähnlich wie „Chuckie Egg“ bildet auch für dieses Experiment ein Computerspiel die Basis: Tetris. Da es für den UVC noch keine Spiele gibt, wurde mit Tetris ein vergleichsweise einfach zu implementierendes gewählt. Dabei wurden bestimmte Aspekte als relevant angenommen: die Art der aus vier Einzelsteinen zusammengesetzten Spielsteine, das Herunterfallen in verschiedenen Geschwindigkeiten, die Drehbarkeit der Spielsteine, das Entfernen von komplettierten Zeilen und das anschließende Nachrücken der Zeilen darüber sowie das Spielende bei Erreichen des oberen Randes. Während der Implementierung wurde aber schnell klar, dass weitere Aspekte durchaus erheblichen Einfluss auf die wahrgenommene Authentizität haben. Darunter ist die Anzahl der Spalten. Beim Spielen mit einer abweichenden Spaltenanzahl wirkt das „falsch.“ Noch präsente Strategien passen nicht mehr. Selbst wenn die Spiellogik korrekt erhalten bleibt, könnte in ferner Zukunft ein falscher Eindruck vermittelt werden, wenn es aufgrund leichter Veränderungen zu trivial oder unspielbar wird.

Zum Vergleich wurde daher eine authentische Version herangezogen und deren „Vorgaben“ übernommen. Auch bei diesem Experiment unterstützte die datArena maßgeblich durch die Leihgabe eines Gameboys mit dem Spiel Tetris und einer Anleitung. Der Gameboy war eine günstige Spielplattform, die in Deutschland fast ausschließlich mit dem Spiel Tetris vertrieben wurde. Eine zum Abschluss des Experiments selbst durchgeführte Umfrage unter angehenden Informatiktechnikern der Geburtsjahrgänge 1976 bis 1983 reflektiert das sehr gut: Von den 38 Teilnehmern besaßen 22 einen eigenen Gameboy. 35 Teilnehmer kennen das Spiel Tetris und 32 davon hatten es schon einmal auf dem Gameboy gespielt.

Mithilfe des Gameboys konnten recht schnell die relevanten Details geklärt werden. Wie groß ist das Spielfeld? Wie viel verschiedene Steine gibt es? Wie lassen sich diese drehen? Wie fallen sie herunter? Verschiedene Implementierungen haben leicht abweichende Spielfeldhöhen, eingeschränkte Drehrichtungen und ein unterschiedliches Verhalten beim beschleunigten Herunterfallen der Steine. Auch gibt es Unterschiede in der Handhabung der Drehung am Spielfeldrand. Ist z.B. der längliche Spielstein ganz an den Rand geschoben noch drehbar? Hier gibt es Versionen, die den Stein mit der Drehung „zurück“ in das Spielfeld drücken. Die eigene Implementierung richtet sich exakt nach den Vorgaben des Originals.

Das Spiel umfasst drei verschiedene Spieltypen. Eine Endlosversion, in der das Level und damit die Fallgeschwindigkeit der Steine stetig zunimmt, eine Version mit unterschiedlich hoch zufällig vorgefülltem Spielfeld, in der jeweils 25 Zeilen komplettiert werden müssen und ein Zweispielermodus. Sämtliche Spieltypen sind mit Musik unterlegt, einzelne Aktionen zusätzlich durch Soundeffekte. Die Musik wurde als weniger relevant eingeschätzt. Auch diese Annahme bestätigt die angesprochene Umfrage: Lediglich zwei Teilnehmer erwähnten die Musik als ein noch präsentes Detail, beide mit dem Attribut „nervig.“

Die Punktevergabe mutet zunächst sehr komplex und undurchschaubar an. Die Anleitung jedoch erwies sich als überaus hilfreich. Diese enthält eine Tabelle, die eine Berechnung der Punkte für entfernte Zeilen abhängig ihrer Anzahl und dem aktuellem Level ermöglicht. Das Wechseln in höhere Level und damit die ansteigende Fallgeschwindigkeit ist nicht dokumentiert, korreliert jedoch erkennbar mit der Anzahl der bereits komplettierten Zeilen.

Von den drei Spielvarianten wurde nur die erste umgesetzt. Die Punktevergabe und das Wechseln in höhere Level wurden für diese Variante vollständig implementiert. Es wäre nicht schwer, die zweite Variante umzusetzen, jedoch ist mit keiner größeren Aussagekraft des Experiments zu rechnen. Der Zweispielermodus hingegen erfordert durch die notwenige Kommunikation zwischen den Instanzen einen nicht näher untersuchten Mehraufwand.

11.2. Relevante Implementierungsdetails

Dieser Abschnitt geht auf relevante Implementierungsdetails ein. Dabei ist weniger das Spiel selbst im Fokus, als vielmehr die Frage, wie man Interaktivität mit dem UVC umsetzen kann und wie die Schnittstelle definiert wurde, damit eine grafische Anzeige so gelingt, dass das Spiel auch authentisch spielbar wird. Auf beide Aspekte wird im Folgenden näher eingegangen. Für einen angehenden Programmierer von UVC-Anwendungen sei erwähnt, dass die Implementierung ein schönes Beispiel für eine effiziente Nutzung des bitadressierbaren Speichers darstellt.

11.2.1. Implementierung der „nebenläufigen“ Interaktion

Was versteht man unter der Interaktivität im Falle des Spiels Tetris? Der Gameboy besitzt zur Spielsteuerung ein Tastenkreuz und zwei Knöpfe. Mit dem Kreuz lassen sich die Spielsteine horizontal verschieben und schneller nach unten fallen. Die zwei Knöpfe dienen dem Drehen der Spielsteine im und gegen den Uhrzeigersinn jeweils um 90°. Darüber hinaus gibt es einen Knopf, mit dem sich das Spiel starten lässt. Die Ein- und Ausgabefähigkeiten des UVC sind jedoch sehr beschränkt. Er kennt keine Tasten, dafür aber zwei Kanäle. Gedrückte Tasten müssen daher von einer separaten *Restore Application* erfasst und diese Information als Nachricht auf dem Eingabekanal zum UVC gesendet werden.

Wird während des Spiels keine Taste gedrückt, fällt der Stein selbständig weiter herunter. Auch dieses Verhalten erwartet man und ist somit Teil der Interaktion. Aber wie kann der UVC gleichzeitig auf eine Tastennachricht warten und bestimmte andere Aktionen ausführen? Erreicht ein UVC-Programm den Befehl `IN`, stoppt die Abarbeitung bis eine Nachricht eingegangen ist. Nebenläufigkeit kennt der UVC nicht. Zudem besitzt er keine Uhr. Ein UVC-Programm kann daher das Fallen eines Steines nicht so steuern, dass es in allen Umgebungen im Level 0 ungefähr 14 Sekunden dauert. Das Problem wurde erneut durch Auslagern gelöst. Das UVC-Programm erwartet auf dem Eingabekanal eine Art Herzschlag. In regelmäßigen, klar definierten Abständen erwartet er eine Nachricht eines bestimmten Typs. Diese Nachrichten nutzt das Programm intern zum Weiterschalten. Solange die einzelnen Programmabschnitte des UVC-Programms schnell genug ausgeführt werden können, läuft auf verschiedenen Plattformen das Programm scheinbar immer gleich schnell. Sehr oft synchronisieren sich hierfür Computerspiele mit dem Bildaufbau. Nachrichten werden vom UVC gepuffert, so dass während der Abarbeitung von Tasteninformationen kein Schlag verloren geht. Eine gute Skalierung der Fallgeschwindigkeiten wurde mit rund 30 Herzschlägen pro Sekunde möglich.[1] Im Level 0 wird je 28 Schläge gewartet, bis ein Stein von allein eine Zeile nach unten rutscht.

11.2.2. Grafische Darstellung

Ein wichtiger Aspekt ist die Ausgabe. Wie kann auf dem Ausgabekanal die Information des Spielfeldes übertragen werden? Das Spiel Tetris nutzt sehr wahrscheinlich ausschließlich die zeichenbasierte Darstellung. Das heißt, jeweils einem mit 8 Bit codiertem Zeichen wird im Gameboy eine 8x8 Pixel große Grafik zugeordnet. Dazu kann der verwendete Zeichensatz, also diese Zuordnung dynamisch vom Spiel festgelegt werden. Die eigene Implementierung orientiert sich daran, definiert einen Zeichensatz und überlässt die konkrete Darstellung der *Restore Application*. Der Grund hierfür ist zum einen die Ausführungsgeschwindigkeit und zum anderen der Implementierungsaufwand. Es ist wesentlich einfacher und deutlich schneller, nur 8 Bit pro Zeichen verwalten und übertragen zu müssen. Das Display des Gameboys umfasst 20x18 Zeichen und somit müssen auf der Basis von Zeichen nur 2880 Bit pro Bild übertragen werden. Durch die 4 möglichen Graustufen wäre das sonst das 32-Fache. Eine Anpassung der UVC-Anwendung dahingehend würde das authentische Spielen nur auf sehr leistungsfähigen Systemen ermöglichen. Damit die *Restore Application* die Informationen korrekt

[1] Diese technische Beschreibung gibt für den Gameboy eine vertikale Synchronisation mit 59,73 Hz an: http://nocash.emubase.de/pandocs.htm

darstellen kann, muss diese den verwendeten Zeichensatz kennen. Dieser Zeichensatz liegt in Form eines GIF-Bildes vor, das wiederum mit einer bereits verfügbaren UVC-Anwendung interpretiert werden kann. Die alternative Möglichkeit, die Metadaten zum Programmstart durch die UVC-Anwendung selbst zu übermittelten, wurde verworfen. Die Beschreibung, wie diese initialen Nachrichten zu interpretieren sind, ist vermutlich nicht weniger umfangreich als die Beschreibung der angedachten Nutzung des beiliegenden Bildes.

11.2.3. Zufall

Ein Vorteil der UVC-Anwendungen wird hier zum Nachteil: zu gleichen Eingaben werden immer identische Ausgaben erzeugt. Die Realisierung eines zufällig gewählten Spielsteins wird dadurch zu einem Problem. UVC-Anwendungen können selbst sehr einfach einen Zufallsgenerator implementieren, der eine feste Folge von pseudo-zufälligen Werten erzeugt. Die eigene Implementierung nutzt hierfür die oft zitierte Konfiguration von Lewis et al. [1969]. Diese als minimaler Standard bezeichnete Konfiguration besteht auch komplexere Tests mit durchaus akzeptablen Ergebnissen [Park und Miller, 1988]. Zur Umsetzung benötigt man lediglich eine Multiplikation und eine Division mit Rest. Beides ist mit dem UVC leicht nutzbar. Die so erzeugten Werte sind gut für die fortlaufende Wahl folgender Spielsteine geeignet. Um aber mit jedem Spielstart eine abweichende Folge zu erzeugen, ist dem Generator ein wechselnder Initialwert vorzugeben, der die Startposition innerhalb der extrem langen Folge festlegt. Aber woher kommt dieser Initialwert? Der UVC hat weder eine Uhr noch kann er zufällige Werte erzeugen. Entgegen den bisherigen Lösungen, gelingt das überraschend zufriedenstellend UVC-intern. Der recht hoch frequente Herzschlag macht es möglich, die durch das Nutzerverhalten abweichenden Folgen von Zeit- und Tastaturnachrichten zu nutzen. So wird einfach mit jeder Zeitnachricht eine Zufallszahl erzeugt, die Folge also weitergeschaltet. Dem Nutzer gelingt es nicht, immer gleich viel Zeit zwischen sämtlichen Tastendrückvorgängen vergehen zu lassen. Bereits der erste Spielstein erscheint beim gewöhnlichen Spielen zufällig.

11.2.4. Die *Restore Application*

Da bei interaktiven Programmen die Rückkopplung von der Aus- zur Eingabe durch den Nutzer erfolgt, könnten mit Blick auf die Filterprogramme Ein- und Ausgabe getrennt implementiert werden. Im vorliegenden Fall leistet eine Java-Anwendung beides. Eine zentrale Klasse versendet mit jeder gedrückten Taste eine entsprechende Nachricht, sorgt für den Herzschlag und stellt mithilfe des Bildes die Ausgaben des UVC dar. Eine zweite Klasse startet den UVC, verbindet jeweils den Ein- und Ausgabekanal und bereitet die Nachrichten auf. Der Quellcode beider Klassen umfasst nur 150 Programmzeilen. Die Abbildung 11.1 stellt die *Restore Application* links, die exemplarische Nutzung der Kommunikationskanäle zusammen mit der angedachten Interpretation in der Mitte und die UVC-Anwendung rechts dar.

Um die von der UVC-Anwendung auszugebenden Nachrichten einfach zu erzeugen, werden immer ganze Zeilen übertragen. Der Nachrichtentyp codiert dabei die Zeilennummer. Während des Spielens werden immer alle Zeilen mit jeder Änderung am Spielfeld übertragen. Die Java-Anwendung prüft intern, ob an der Anzeige tatsächlich Änderungen vorgenommen werden müssen. Für den Fall, dass die UVC-Anwendung keine Zeichen sondern einzelne Pixel

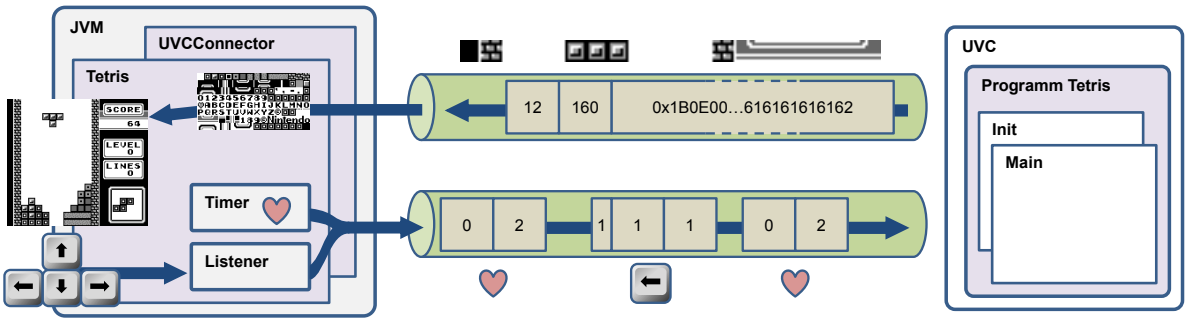


Abbildung 11.1.: Schematische Darstellung am Spiel Tetris beteiligter Komponenten

übertragen soll, könnte eine solche Prüfung bzw. Optimierung seitens der UVC-Anwendung vorgenommen werden, um den Ausgabekanal zu entlasten und Laufzeiten einzusparen.

11.3. Ergebnisse

Aus Sicht der Historie von Computerspielen wäre auch Pong sehr interessant gewesen umzusetzen. Dabei scheitert es aber an der zu erreichenden Vergleichbarkeit. Das originale Pong besteht nicht nur aus Programmlogik. Vielmehr Einfluss auf die tatsächliche Spielwahrnehmung haben die Eingabegeräte. Die traditionellen Pads mit Drehknopf sind von ihrer Reaktionsfähigkeit her am ehesten mit der heutigen Computermaus zu vergleichen. Im traditionellen Zweispielermodus ist eine solche Abbildung nicht zu leisten bzw. nicht ohne handwerkliche Eingriffe möglich; heutige Systeme unterstützen nur jeweils eine Maus. Die eigentlich zu belegende erreichbare Authentizität würde hierbei drohen ins Hintertreffen zu geraten. Vorhandene Emulatoren und Reimplementierungen von Pong belegen die veränderte Wahrnehmung des Spiels gegenüber den in der datArena vorhandenen Originalen sehr deutlich.

Aus den Ergebnissen der erwähnten Umfrage lassen sich noch weitere Schlüsse ziehen. So wurden ebenfalls die noch präsenten Details erfasst. Nur zwei Teilnehmer wussten die Spaltenzahl, keiner die Anzahl der Zeilen. Alle Spielsteine konnten nur drei korrekt aufmalen. An die konkrete Punktevergabe erinnerte sich keiner, ebenso wenig an die Bedingungen zum Wechsel in höhere Level. Immerhin einer wusste von einer Rakete zu berichten. Tatsächlich sind es eine kleine und eine große Rakete, die im ersten Spieltyp abhängig von den erreichten Punkten und beim zweiten Spieltyp abhängig vom Level gezeigt werden. Zuvor tanzen kleine Figuren. Solche Feinheiten wurden mit der eigenen Implementierung nicht berücksichtigt. Das Experiment und die Umfrage bestätigen die Aussage von Hedstrom und Lampe [2001]: Was genau der Nutzer eigentlich erhalten haben will, lässt sich nicht vorhersagen.

Die anschließende Evaluation des Tetris-Spiels mit den daran interessierten Teilnehmern erwies sich als großer Erfolg. Alle bewerteten die getestete Tetrisversion als authentisch.

Die Implementierung des Spiels Tetris gelang innerhalb von 36 Stunden Arbeitszeit. Darin enthalten sind das Spielen mit dem originalen Gameboy zum Eruieren authentischer Vorgaben und die Implementierung der kleinen Java-Anwendung. Der Quellcode der UVC-Anwendung codiert zwei Unterprogramme mit insgesamt 1140 reichlich kommentierten Programmzeilen.

Mit Blick auf die Kosten bewerten Guttenbrunner et al. [2008] das Erhalten interaktiver Programme durch Reimplementierung nur für einfache Spiele erster Generationen für praktikabel. Sie mussten davon ausgehen, dass die Kosten mit jeder Computergeneration erneut anfallen. Das mit der vorliegenden UVC-Anwendung archivierbare Tetris ist für alle zukünftigen Generationen spielbar und bedurfte nur einer einmaligen Reimplementierung. Allerdings fällt Tetris genau in das von Guttenbrunner et al. dargestellte Raster. Der UVC macht eine umfassendere Neubewertung erforderlich. Dafür bedarf es aber mehr als nur eines Experiments.

Die Aussage von Gladney und Lorie [2005] ist hinsichtlich der Ergebnisse ebenfalls umfassend neu zu bewerten:

> *More complex [applications] would need UVC instructions to read a clock, handle interrupts, fork to implement multiprogramming and provide latches for synchronization.*

Das in diesem Kapitel beschriebene Experiment sollte eigentlich diese Aussage untermauern und Anhaltspunkte liefern, wie der UVC dahingehend erweitert werden könnte. Stattdessen setzt die vorgestellte UVC-Anwendung das interaktive Spiel Tetris um, ohne eine Uhr lesen zu müssen. Zudem mussten intern keine Fork-Mechanismen bemüht werden. Vielmehr ist es sehr wahrscheinlich, dass sogar die Implementierung des Zweispielermodus problemlos gelingt. Solange die ausführende Hardware leistungsstark genug ist, wäre auch die Aufbereitung und Übermittlung von Sounddaten möglich. Mit dem vorliegenden Experimentergebnissen allein kann und soll diese Aussage aber nicht widerlegt werden. Vielmehr ist mit der Entwicklung komplexerer UVC-Anwendungen zu bewerten, welcher Aufwand zumutbar ist: Nachdem sich viele Aspekte wie Nebenläufigkeit auch durch getrennt laufende, aber miteinander verzahnte UVC-Instanzen realisieren lassen, stellt sich zurecht die Frage, ob es einfacher ist, in ferner Zukunft einen komplexeren UVC zu implementieren oder basierend auf komplexeren Metadaten eine entsprechend umfangreiche *Restore Application* zu implementieren.

Interessant wäre die Spezifikation einer im UVC verankerten Uhr. In Anbetracht der anvisierten Archivierungszeiträume könnte es passieren, dass die Uhrzeit nicht länger auf dem 1. Januar 1970 basiert. Wie gehen UVC-Anwendungen damit um? Wird tatsächlich eine interne Uhr für den UVC obligatorisch, könnten die Millisekunden mit dem Start beginnend gezählt und über eine Instruktion zugänglich sein. Die Realisierung eines internen Taktgebers sowie die Befähigung zum Prüfen auf anliegende Nachrichten müssten ebenso hinterfragt werden wie eine Erweiterung, um gezielt Unterprogramme bei Eintreten eines bestimmten Ereignisses wie das Eintreffen einer Nachricht oder einer Division durch Null anzusteuern. Letzteres könnte mit vorbelegten Unterprogrammnummern realisiert werden. Eine Abschätzung des damit verbundenen Mehraufwandes bei der Implementierung des UVC sollte bei Bedarf erfolgen. Keine der derzeit verfügbaren UVC-Anwendungen impliziert diesen Bedarf.

Das vorliegende Experiment belegt, dass auch Spiele mittels UVC archiviert und erhalten werden können. Es ist aber immer eine Portierung bzw. Reimplementierung für den UVC erforderlich. Auch wird klar, dass zwar die Spiellogik an sich sehr gut zu erhalten ist, nicht aber die Spielwahrnehmung. Sie hängt sehr deutlich von den umgebenen Komponenten ab. Die grafische Ausgabe muss ebenso wie die Spezifikation der Eingabegeräte in geeigneter Form überliefert werden. Die von Lorie oft vorgeschlagene LDS ist hierfür bei Weitem nicht mächtig genug. Zumal sie nur Ausgaben, aber keinerlei Eingaben berücksichtigt.

Teil IV.

Ergebnisse und Ausblick

12. Ergebnisse und Schlussfolgerungen

Dieses Kapitel fasst zunächst die in den einzelnen Experimenten gesammelten Ergebnisse zusammen. Dabei werden diese Ergebnisse direkt den in der Zielsetzung dieser Arbeit formulierten Fragestellungen zugeordnet (siehe Abschnitt 1.2). Dieser erste Teil des Kapitels liefert somit die Antworten darauf.

Der zweite Teil schließt aus den Ergebnissen auf eine erweiterte Spezifikation des UVC. Dabei wird die Notwendigkeit dieser Weiterentwicklung motiviert, die auch eine Anreicherung umfasst, deren Notwendigkeit mit der Auswertung der Experimente augenfällig wurde.

Ebenfalls Ergebnis dieser Arbeit ist eine Übersicht der evaluierten Strukturen und Algorithmen, die zu einer effizienten Implementierung des UVC führen. Dieses Ergebnis, das sich als Implementierungshilfe verstanden sehen möchte, ist im Anhang A.2 zu finden. Im Wesentlichen handelt es sich dabei um eine geordnete Auflistung der zu leistenden Arbeiten mit jeweils einem Hinweis auf den diesen Aspekt aufgreifenden Abschnitt innerhalb dieser Arbeit.

12.1. Beantwortung der Fragen aus der Zielsetzung

Die Ergebnisse, die sich aus den einzelnen Experimenten ableiten, fließen hier direkt in die Beantwortung der eingangs gestellten Fragen ein. Entsprechend gliedert sich der erste Teil dieses Kapitels.

12.1.1. Integrität der Spezifikation

Die Frage, ob die Spezifikation aus sich heraus verständlich ist, muss verneint werden. Wie kommt es zu dieser Einschätzung?

Das erste in der vorliegenden Arbeit beschriebene Experiment implementierte den UVC. Das Ergebnis (siehe 5.4) allein dieses Experiments beantwortet die Frage. Dabei ist von offensichtlichen Fehlern, unklaren Passagen und von irreführenden Angaben die Rede. Während Fehler offensichtlich beseitigt gehören, unterliegen unklare oder irreführende Angaben subjektiven Einschätzungen. Für das Szenario, in dem der UVC zum Einsatz kommen soll, können ungenaue Angaben fatale Folgen haben. Fehlerhafte Angaben können sehr wahrscheinlich auch in ferner Zukunft erkannt und behoben werden. Darunter fällt z.B. der Fehler in der Spezifikation der Instruktion `AND`. Die hier beschriebene Erweiterung des jeweils kleineren Registerinhalts durch führende Einsen ist relativ leicht als Fehler zu erkennen – zumindest aus heutiger Sicht. Irreführende Angaben, wie die geforderte Speicherung des Bitcodes eines

Unterprogramms im Speicher eines Segments, können zu völlig verschiedenen Implementierungen führen. Denn es wird nicht spezifiziert, wie diese Segmente mit der laufenden UVC-Anwendung verknüpft werden sollen, also ob und wenn ja, wie Anwendungen Zugriff auf den eigenen Code haben. UVC-Anwendungsentwickler könnten heute dazu verleitet werden, selbstmodifizierenden Code zu nutzen, der vielleicht in Zukunft nicht unterstützt wird. Aus Sicht der Langzeitarchivierung ist das ein Szenario mit verheerenden Folgen. Die Abbildung 12.1 stellt die im Zuge der Weiterentwicklung geänderten Passagen wie folgt dar:

- Rot unterlegt sind fehlerhafte Passagen.
- Lila unterlegte Passagen enthalten irreführende Angaben.
- Orange eingefärbte Passagen müssen konkretisiert werden.
- Grüne Passagen werden durch die Weiterentwicklung an sich geändert, z.B. um effizientere Implementierungen bzw. schnellere UVC-Anwendungen zu ermöglichen oder um die Implementierungszeit für den UVC zu reduzieren.
- Grau unterlegte Passagen tragen nicht zur Spezifikation des UVC bei.

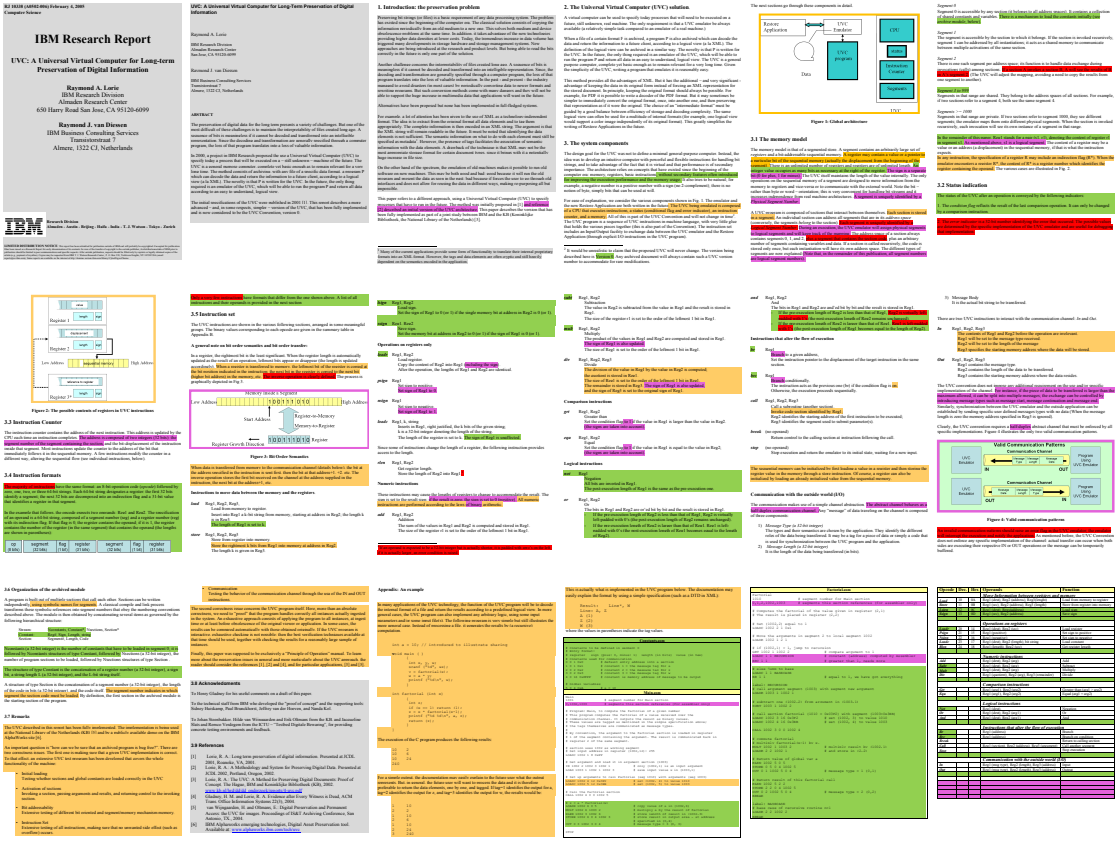


Abbildung 12.1.: Farblich hervorgehobene Stellen in der bisherigen Spezifikation, die im Zuge der Weiterentwicklung angepasst wurden

Das Erstellen der Referenzimplementierung allein führte jedoch nicht zu dieser Einschätzung. Da diese wie zuvor angedeutet subjektiver Natur ist, flossen hier neben den Ergebnissen der in dieser Arbeit vorgestellten Experimente auch alle im Rahmen von durch den Autor der vorliegenden Arbeit betreuten Seminaren, Praktika, Bachelor- und Masterarbeiten gesammelten Erfahrungen ein.

Dabei wurde deutlich, dass verschiedene Passagen unterschiedlich relevant sind. So spielt die verwirrende Benutzung von physischen Segmentnummern innerhalb der Spezifikation keinerlei Rolle für das Gelingen der Implementierung an sich, jedoch lässt sie einen Programmierer bei der Erstellung einer eigenen UVC-Laufzeitumgebung schnell unglückliche Entscheidungen treffen.

Ungenau spezifizierte Instruktionen hingegen fallen eher dem UVC-Anwendungsentwickler auf, wenn er die Möglichkeiten ausreizt. Aus Sicht eines Programmierers, der den UVC implementiert, ist mit der Spezifikation der Instruktion `DIV` klar, was zu tun ist. Sehr knapp und dennoch präzise wird hier auch der *remainder* spezifiziert. Jedoch bleibt folgende Frage offen, die sich der UVC-Anwendungsentwickler stellt: Was steht nach Ausführung der folgenden Zeilen im Register `1,0`?

```
LOADC  1,0     1     1
DIV    1,0   1,0   1,0
```

Alle verfügbaren Implementierungen legen zunächst das Ergebnis und anschließend den Rest in das Register ab. Folglich ist der Wert des Registers nach Ausführung `0`. Aber diese Reihenfolge ist nicht spezifiziert, es könnte ebenso gut der Wert `1` sein. Diese Frage betrifft alle Instruktionen, die mehr als ein Register manipulieren. In der bisherigen Spezifikation sind das nur `DIV` und `IN`.

Eine klare Spezifikation ist jedoch an der aufgezeigten Stelle nicht einfach, denn es kommt ein weiterer Aspekt hinzu: der indirekte Registerzugriff.

```
LOADC  1,1     2     2
LOADC  1,2     2     3
DIV    1,1   1,1   1,1*
```

Was steht wohl nach Ausführung im Register `1,2`? Durch das vorangegangene Beispiel können wir annehmen, dass der Rest der Division erst am Schluss abgelegt wird. In diesem Fall wird der Rest in das Register gelegt, dessen Nummer im Register `1,1` gespeichert ist. Vor Ausführung ist das die `2`, während der Ausführung wird das die `1`. Wann genau findet die Auslösung der Indirektion statt? Auch wenn sich einmal mehr alle Implementierung gleich verhalten und den Rest in Register `1,2` ablegen, so gilt das nicht für die Instruktion `IN`, für die sich leicht ein ähnlicher Fall konstruieren lässt.

12.1.2. Universalität

Die Frage nach der Universalität ist zweigeteilt. Zum einen geht es um den Aspekt der Implementierbarkeit auf allen denkbaren Hardwarearchitekturen, zum anderen um den der Implementierungszeit bzw. des -aufwands.

Der erste Aspekt lässt sich nicht abschließend beantworten, da wir heute noch nicht ahnen können, welche Hardwarearchitekturen in ferner Zukunft genutzt werden. Ein sehr überzeu-

gendes Indiz für die Universalität liefert aber die *Zeitreise*. Bis auf die Speicherung der UVC-Applikationen als Bitstrom wurden keine weiteren Hardwareabhängigkeiten entdeckt. Ein weiteres Indiz liefert der geglückte Versuch, UVC-Anwendungen über einen Cross-Compiler für verschiedene Architekturen verfügbar zu machen (siehe Kapitel 7). Auch wenn der Aufwand sich umfangreicher als bei der Entwicklung einer UVC-Laufzeitumgebung darstellte, erwies sich damit eine alternative Herangehensweise als praktikabel.

Auch der zweite Aspekt konnte im Schwerpunkt innerhalb der *Zeitreise* beleuchtet werden. Es zeigte sich – etwas überraschend – bei allen Implementierungen in etwa der gleiche Aufwand. Die Implementierungszeiten betragen im Maximum sechs Wochen und weichen auch nicht zu stark voneinander ab. Es muss aber erwähnt werden, dass innerhalb der *Zeitreise* nur solche Algorithmen und Strukturen genutzt wurden, die zum jeweiligen Zeitpunkt schon verfügbar gewesen wären. Die aktuellste Implementierung in Ada (siehe Kapitel 5) genauso in FORTRAN IV für die CDC 6600 aus dem Jahr 1964 umzusetzen, dürfte weit mehr als sechs Wochen beanspruchen. Gleiches wird für zukünftig entwickelte Techniken sicher auch gelten, würde man diese mit heutigen Mitteln umzusetzen versuchen. Es ist aber zu erwarten, dass sich in gleicher Weise die Programmiersprachen und deren Unterstützung moderner Techniken weiterentwickeln werden, sodass sich wiederum die Implementierungszeit für einen effizienten UVC nicht deutlich verändern sollte. Die in der *Zeitreise* beobachteten System- und Sprachabhängigkeiten wirkten sich in der Summe nur unwesentlich auf die Implementierungszeit aus. Stärkeren Einfluss hatten sie auf die Performanz des implementierten UVC.

12.1.3. Vollständige Programmierbarkeit

Die Implementierung der Turing-Maschine hat gezeigt, dass der UVC Turing-vollständig ist (siehe Kapitel 8). Die bereits vorhandenen Anwendungsfälle zeigen, dass das angedachte Einsatzgebiet zum Zugriff auf statische digitale Objekte möglich ist. Die eigene komplexe Implementierung des Assemblers auf der Basis gängiger Compilertechniken zeigt zudem, dass die Erstellung komplexerer UVC-Applikationen mit akzeptablem Zeitaufwand möglich ist.

Im Prinzip kann der UVC auch zur Archivierung von interaktiven Anwendungen genutzt werden, wie das Beispiel Tetris zeigt (siehe Kapitel 11). Für die Archivierung von sehr komplexen interaktiven Anwendungen wie Spielen erscheint der UVC jedoch nur bedingt tauglich. Zu viele Informationen müssen bisher „außerhalb“ von UVC-Anwendungen erhalten und fehlerfrei interpretiert werden können. Schon das mit Absicht sehr klein gehaltene Beispiel der Tetris-Anwendung zeigt einen aus Sicht der Langzeitarchivierung nicht zu unterschätzenden Aufwand. Zwar lassen sich aus den Ergebnissen erste Lösungsansätze ableiten, aber diese werden (noch) nicht in die Weiterentwicklung einfließen. Die zugrunde liegenden Erkenntnisse basieren nur auf einem einzigen Anwendungsfall. Diese Arbeit versteht sich an dieser Stelle als Wegbereiter für eine umfassendere Evaluation, deren Ergebnisse eine fundierte Weiterentwicklung erlauben werden.

Es bleibt festzustellen, dass es – außer bei zeitkritischen oder rechenzeitintensiven Anwendungen – keinen erkennbaren Grund gibt, warum eine Anwendung nicht für den UVC entwickelt werden könnte.

12.1.4. Schnelle Programmierung

Dieser Aspekt beinhaltet zwei erstrebenswerte Ziele. Zum einen möchte man ein Programm mit möglichst wenigen intuitiven Instruktionen möglichst knapp formulieren können, zum anderen ist es als Programmierer frustrierend, wenn ein Programm trotz eleganten Designs nicht schnell ausgeführt wird. Der UVC bietet bisher in beiden Punkten enorme Verbesserungsmöglichkeiten. Im Wesentlichen sind das die bisher fehlenden konstanten Operanden (siehe Kapitel 7), bisher fehlende, aber überaus zweckmäßig und zeitschonend implementierbare Instruktionen im Bereich der Arithmetik, der Vergleichsoperatoren und der Sprunganweisungen und die eindeutig spezifizierten Registerinhalte, also der generelle Verzicht auf führende Nullen und eine eindeutige Darstellung der Null. Es soll aber nicht verschwiegen werden, dass deutlich erkennbar die verfügbaren UVC-Anwendungen erheblichen Anteil an den hohen Laufzeiten haben und das Potential des bisher spezifizierten UVC bei Weitem nicht ausnutzen. Wie die Ergebnisse in Kapitel 9 belegen, sind Laufzeit-effiziente Anwendungen möglich.

Bei den im Folgenden dargestellten Erweiterungen wurde im Vorfeld bereits genau darauf geachtet, dass sie sich mit nur sehr geringem Aufwand umsetzen lassen.

12.2. Weiterentwicklung der Spezifikation

Aus den im vorangegangenen Teil angesprochenen Defiziten der Spezifikation leitet sich die Weiterentwicklung ab, die sich vollständig im Anhang B findet. Alle Erweiterungen, Präzisierungen und Berichtigungen sind durch die Gesamtheit aller Ergebnisse der vorliegenden Arbeit begründet. Nicht alle kann dieser Abschnitt erneut aufgreifen. Vielmehr ist es das Ziel, unter den Neuerungen im Schwerpunkt diese vorzustellen und zu motivieren, deren Notwendigkeit zwar begründet wurde, nicht jedoch die genaue Ausgestaltung.

Die vorliegende Dissertation erarbeitet auch die Erkenntnis, dass eine vollständig unmissverständliche und fehlerfreie Spezifikation nicht mit dem Ziel vereinbar ist, eine möglichst knappe und unkomplizierte Spezifikation an zukünftige Generation weiterzureichen. Zudem ist diese Weiterentwicklung noch nicht vollständig. So wurde im Teil I erarbeitet, dass z.B. das Problem der Interpretation der Ausgabe des UVC noch nicht vollständig gelöst ist und ebenfalls spezifiziert werden muss. Ebenso lässt das Kapitel 11 erkennen, dass für einen umfassenderen Einsatz innerhalb der Langzeitarchivierung weitere Erweiterungen notwendig werden würden.

Diese Weiterentwicklung versteht sich daher als Arbeitsgrundlage, die aber auf jeden Fall zu einem produktiven Einsatz innerhalb der Langzeitarchivierung einladen möchte. Fortführende Entwicklungen des UVC müssen in Anbetracht der hier erarbeiteten Ergebnisse auf diese Weiterentwicklung aufbauen. UVC-Anwendungen, die für den hier spezifizierten UVC entwickelt werden, müssen auch von abermals weiterentwickelten UVC unterstützt werden bzw. für diesen anzupassen sein. Es kann nicht oft genug betont werden, dass dazu der Quellcode zu allen UVC-Anwendungen zwingend erforderlich ist und folgerichtig mit archiviert werden muss.

12.2.1. Konkretisierungen und Berichtigung von Fehlern

Alle erkannten Fehler wurden berichtigt. Darunter sind die offensichtlichen Fehler wie die Beschreibungen der Instruktion `AND` und des Parameter-Segments. Die irreführenden Angaben bzgl. der physischen Segmentnummern wurden komplett entfernt. Stattdessen findet sich jetzt ein Hinweis auf die auch lückenhafte Nutzung des sequentiellen Speichers, der davon abhalten soll, den Speicher als zusammenhängendes „Stück“ im Speicher der Basismaschine abzubilden.

Als wichtig wurde auch erachtet, die Spezifizierung des Archivmoduls zu konkretisieren (siehe Abschnitt 6.7.4). Die bisherige Darstellung wird jetzt mittels der Backus-Naur-Form (BNF) [Backus et al., 1963] deutlich formaler beschrieben. Bei diesem Schritt wurden die mit dem Modul vorab zu definierenden konstanten Werte gestrichen, da sie auch auf anderem Wege in die Register des Segments 0 geladen werden können und nur unnötig Implementierungszeit kosten. Das Segment 0 verliert damit zwar sein letztes Alleinstellungsmerkmal gegenüber den *shared* Segmenten, wird aber dennoch als solches aufgrund der Abwärtskompatibilität erhalten bleiben. Des Weiteren werden Füllbits direkt mit aufgenommen und deren Werte spezifiziert. Damit ist implizit eine Anleitung für die Migration der Archivmodule auf nicht byteweise speichernde Dateisysteme gegeben.

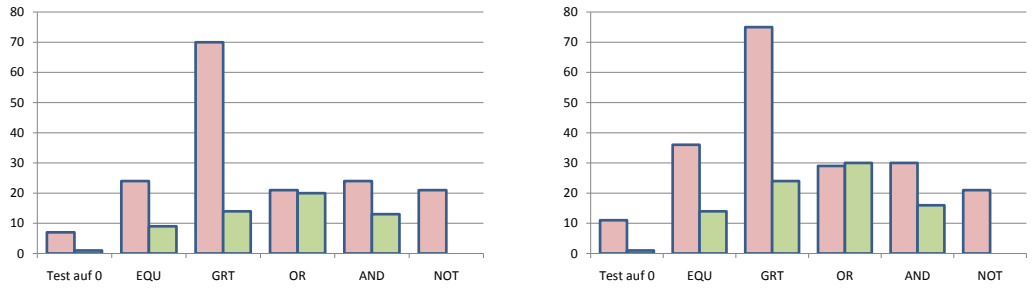
Die BNF kommt ein weiteres Mal bei der formalen Angabe der Assembler-Syntax zum Einsatz. Diese war bislang nicht spezifiziert, obwohl ein Beispielprogramm in der Spezifikation enthalten ist. Die im Beispiel genutzte Syntax ist irreführend, da sie scheinbar Befehle zulässt, die nicht spezifiziert wurden. Um Friktionen dieser Art zu vermeiden, wird jetzt die Syntax direkt mit aufgenommen und das Beispielprogramm darauf basierend überarbeitet. Die spezifizierte und damit ebenso langzeit-verfügbare Syntax erlaubt zudem das Archivieren von UVC-Anwendungen im Quellcode. Dadurch sind diese bei eventuell folgenden Weiterentwicklungen der Spezifikation mit nur wenig Aufwand anpassbar.

Eine wichtige Konkretisierung ist die der Abarbeitung der Register innerhalb einer Operation. Die Reihenfolge und auch die Auswertung der Indirektion werden jetzt für alle Operationen vorgeschrieben. Dabei ist die Forderung der vorangestellten Auswertung der Indirektion einerseits kurz und eindeutig spezifizierbar und andererseits leicht umzusetzen.

Eine noch zu erwähnende Konkretisierung fand im Bereich der Speicherbefehle statt. Jetzt wird eindeutig angegeben, was im Falle abweichender Längen zu erfolgen hat. Dabei wurden mögliche Anwendungsszenarien ebenso berücksichtigt, wie die zeitgemäße Erwartungshaltung. Soll z.B. der Registerwert 7 mit einer Länge von 8 Bit geschrieben werden, dann erwarten wir beim anschließenden Lesevorgang mit gleicher Länge und Adresse den gleichen Wert. Um eine Beeinflussung durch bereits im Speicher vorhandene Werte zu verhindern, müssen fehlende Bits durch Nullen ersetzt werden. Die gewählte Rechtsausrichtung der zu speichernden Bits entspricht den heute üblichen Normen und erleichtert damit den Umgang mit wortweise organisierten Dateiformaten. Das Abschneiden links überstehender Bits bei zu großen Registerinhalten ist eine Folge dieser Entscheidungen. Die bisherige Grafik wird daher durch 3 Grafiken ersetzt, die jeweils einen dieser Fälle visualisieren. Im Zuge der angepassten Beschreibung der Speicherinstruktionen wird auch die implizite Zuordnung des Segments (siehe Abschnitt 5.3.3) explizit erwähnt.

12.2.2. Eindeutige Darstellungen der in Registern gespeicherten Werte

Im Abschnitt 5.3.2 wurde dargestellt, dass die mehrdeutige Zahlendarstellung bei der Speicherung als Registerinhalt zu einer erhöhten Komplexität bei der Implementierung führt. Das lässt sich in Form von Zahlen ausdrücken. Die Ergebnisse dieser Auswertung finden sich in gekürzter Form in [Krebs et al., 2011b]. Die Abbildung 12.2 stellt die benötigten Programmzeilen zweier Implementierungen dem gegenüber, was ohne führende Nullen bzw. Nullbits ausreichen würde. Betrachtet werden dabei nur die Programmabschnitte zur Umsetzung der relevantesten Instruktionen. Mit dem „Test auf 0“ ist ein Beispiel gegeben, dass auch programmweite Auswirkungen verdeutlicht. Dieser ausgelagerte Test wird z.B. bei der Registermultiplikation oder dem Speicherzugriff verwendet. Mit der eindeutigen Darstellung wird dieser Test mit nur einer Anweisung realisierbar; ein Auslagern ist nicht länger erforderlich.



(a) Ada83 (b) FORTRAN 77

Abbildung 12.2.: Vergleich der benötigten Programmzeilen. Der jeweils linke Balken zeigt die führende Nullen berücksichtigende Implementierung.

Allein der Umstand, dass die Problematik der führenden Nullen nicht sofort aus der Spezifikation ersichtlich ist und übersehen werden könnte, birgt eine Fehlerquelle in sich. Die Berücksichtigung wiederum benötigt angepasste Algorithmen, lässt also kein intuitives Vorgehen zu. Dieses wiederum eröffnet weitere Fehlerquellen. Auch wenn die eingesparten Programmzeilen im Vergleich zur Gesamtzahl auf den ersten Blick wenig beeindrucken mögen, auf den zweiten Blick ist das Potential einer eindeutigen Zahlendarstellung enorm. Dies erstreckt sich nicht nur auf Programmzeilen und längere Testphasen, sondern auch auf Laufzeiten und auf das Interpretieren der Spezifikation. Diese enthält bisher explizite Vorgaben wie Registerinhalte zu organisieren sind: zu der Bitfolge ein separates Vorzeichen und eine separat zu speichernde Länge. Ohne führende Nullen wäre aber das Ermitteln der Länge anhand der Bitfolge direkt möglich. Solche expliziten Angaben, die sich eventuell sogar negativ auf die Laufzeit auswirken können, finden sich jetzt nicht mehr.

Ohne führende Nullen ist die Instruktion **NOT** nicht länger sinnvoll nutzbar (siehe Abschnitt 5.3.2). Auf die Erweiterung der Befehlssyntax wird verzichtet und stattdessen die Instruktion vollständig aus der Spezifikation gestrichen. Zum einen nutzt diese Instruktion bisher keine der verfügbaren Anwendungen, zum anderen lässt sich die wohl anvisierte Funktionalität leicht über die Instruktion **SUB** realisieren. Minimal längere Laufzeiten sind erst bei sehr großen Registerinhalten zu erwarten.

12.2.3. Ergänzungen zugunsten der Laufzeit

Im Kapitel 7 wurden bereits konstante Operanden als Voraussetzung für effiziente UVC-Programme motiviert und teilweise durch Laufzeitanalysen von Müller [2011] belegt. Solche Laufzeitanalysen wurden unabhängig davon bereits bei der Implementierung in FORTRAN 77 durchgeführt. Gleichzeitig wurden nicht nur die Laufzeiten gemessen, sondern jeweils auch der damit verbundene Mehraufwand bei der Umsetzung dieser Funktionalität in der UVC-Implementierung erfasst. Dabei spielten nicht nur konstante Operanden, sondern auch zusätzliche, aber „billige“ Instruktionen eine entscheidende Rolle. Die Ergebnisse werden im Folgenden vorgestellt.

Als verdeutlichendes Beispiel soll folgendes intuitiv geschriebenes UVC-Programm dienen, das in einer Schleife sehr oft einen Wert ausgibt:

```
LOADC   1,1    0  0x0     # i=0
label: Loop

LOADC   1,5   16  0xCAFE  # auszugebender Wert
LOADC   1,6    0  0x0     # Adresse im Speicher
LOADC   1,7    5  0x10    # Länge der Message
STORE   1,5  1,6  1,7     # Speichern des Wertes
LOADC   1,8    4  0xD     # der Messagetyp
OUT     1,8  1,7  1,6     # die Ausgabe

LOADC   1,2    1  0x1     # das Schleifeninkrement
ADD     1,1  1,2          # i = i + 1
LOADC   1,3   12  0xFFF   # maximale Schleifendurchläufe
GRT     1,3  1,1          # if i<max
LOADC   1,4   32  Loop    # Laden der Sprungmarke
JUMPC   1,4               # then jump Loop
STOP
```

Zur Optimierung der Laufzeit könnte man das Laden der Konstanten außerhalb der Schleife bewerkstelligen. Das so optimierte Programm ist im Folgenden gegeben:

```
LOADC   1,1    0  0x0     # i=0
LOADC   1,2    1  0x1     # das Schleifeninkrement
LOADC   1,3   12  0xFFF   # maximale Schleifendurchläufe
LOADC   1,4   32  Loop    # Laden der Sprungmarke
LOADC   1,5   16  0xCAFE  # auszugebender Wert
LOADC   1,6    0  0x0     # Adresse im Speicher
LOADC   1,7    5  0x10    # Länge der Message
LOADC   1,8    4  0xD     # der Messagetyp
label: Loop

STORE   1,5  1,6  1,7     # Speichern des Wertes
OUT     1,8  1,7  1,6     # die Ausgabe

ADD     1,1  1,2          # i = i + 1
GRT     1,3  1,1          # if i<max
JUMPC   1,4               # then jump Loop
STOP
```

Der Schleifenrumpf ist auf das Wesentliche reduziert, aber nur noch schwer ohne Kommentare nachvollziehbar. Eine solche Optimierung ist mit der bisherigen Spezifikation möglich.[1] Das so optimierte Programm benötigt gegenüber der vorhergehenden Variante nur noch 43 anstatt 65 Sekunden.

Die zu erzeugende Ausgabe muss zuvor im bitadressierbaren Speicher eines Segments abgelegt werden. Offensichtlich muss aber genau diese Bitfolge bei der Verarbeitung der Instruktion `OUT` erneut geladen werden. Der unumgängliche Weg über den Speicher kostet enorm Laufzeit und zusätzliche verwirrende Programmzeilen. Intuitiver ließe sich das Programm schreiben, wenn Registerinhalte direkt ausgebbar wären, z.B. mit einer Instruktion `OUTR`:

```
LOADC   1,1    0  0x0     # i=0
LOADC   1,2    1  0x1     # das Schleifeninkrement
LOADC   1,3   12  0xFFF   # maximale Schleifendurchläufe
LOADC   1,4   32  Loop    # Laden der Sprungmarke
LOADC   1,5   16  0xCAFE  # auszugebender Wert
LOADC   1,7    5  0x10    # Länge der Message
LOADC   1,8    4  0xD     # der Messagetyp
label: Loop

OUTR    1,8  1,7  1,5     # die Ausgabe

ADD     1,1  1,2          # i = i + 1
GRT     1,3  1,1          # if i<max
JUMPC   1,4               # then jump Loop
STOP
```

Anstatt der 45 Sekunden benötigt diese Variante nur noch 33 Sekunden. Die Umsetzung dieser Funktionalität ist äußerst einfach. Lediglich der Programmabschnitt zur Realisierung der Instruktion `OUT` ist zu kopieren und die entsprechenden Programmzeilen zum Laden der Bitfolge aus dem bitadressierbaren Speicher sind zu löschen. Der extrahierte Wert, der zuvor die Adresse codierte, entspricht bereits dem auszugebenen Wert. Die Umsetzung erzeugt zwar viele neue Programmzeilen, gelingt aber in nur zwei Minuten sehr schnell.

Sind darüber hinaus konstante Operanden erlaubt, kann man das Programm abermals anpassen:

```
LOADC   1,1      0  0x0     # i=0
LOADC   1,5     16  0xCAFE  # auszugebender Wert
label: Loop

OUTR    0xD   0x10  1,5     # die Ausgabe

ADD     1,1   0x1           # i = i + 1
GRT     0xFFF 1,1           # if i<max
JUMPC   Loop                # then jump Loop
STOP
```

[1] In diesem Fall wurde nicht auf das Speichern des konstanten Wertes verzichtet, obwohl immer der gleiche Wert ausgegeben wird. Für die Betrachtung der einsparbaren Laufzeit wird angenommen, dass sich ändernde Werte in Registern befinden, die fortwährend ausgegeben werden. Diese müssten vor jeder Ausgabe gespeichert werden.

Die Sprungbefehle und viele andere Instruktionen lassen sich ähnlich schnell wie die Intruktion `OUTR` umsetzen. Lediglich einige arithmetische Instruktionen bedurften einer „längeren“ Anpassung. Für alle notwendigen Anpassungen bzgl. der Unterstützung konstanter Operanden wurden 12 Stunden benötigt. Das Ergebnis aber kann sich sehen lassen: anstatt der 33 Sekunden benötigt diese Variante nur noch 22 Sekunden, ist intuitiver und deutlich kürzer.

Die bisher möglichen Operanden werden aber im vollen Umfang in der Spezifikation gefordert bleiben. Zum einen basiert die schnelle Anpassung auf diesen bereits umgesetzten Funktionalitäten, zum anderen sind diese auch effizient nutzbar, wie folgendes Programm mit einer eleganten Möglichkeit zum Aufrufen von „Unterprogrammen“ zeigt:

```
JumpAround
1001

LOADC  1,2 Back1     # Rücksprungadresse merken
JUMP   SubP          # Unterprogramm aufrufen
label: Back1

LOADC  1,2 Back2     # neue Rücksprungadresse merken
JUMP   1,1           # Unterprogramm erneut aufrufen
label: Back2

STOP

label: SubP          # Unterprogramm im gleichen Kontext
JUMP   1,2
```

Diese Möglichkeit bietet sich besonders zur Auslagerung kleinerer Programmabschnitte an, die auf den gleichen Kontext (Register, Segmente) zugreifen. Durch die Verwendung eines sehr leicht zu realisierenden Kellers mithilfe des indirekten Registerzugriffs (siehe Abschnitt 9.1.3) ließen sich auch Rekursionen und tiefere Schachtelungen leicht abbilden. Sehr schön sind auch Sprungtabellen auf diese Weise realisierbar (siehe Abschnitt 9.1.4).

Neben diesen Erweiterungen sind auch Schiebebefehle, Vergleichsbefehle und ein weiterer Sprungbefehl aufgenommen worden. Beides dient der Laufzeit und der intuitiveren Programmierung. Wobei Letzteres eigentlich kein Entscheidungsgrund ist. Ein Compiler oder auch ein intelligenter Assembler kann hier sehr hilfreiche Arbeit leisten und so Aufwand von zukünftigen Programmierern fernhalten. Alles, was wir heute an Aufwand investieren, müssen zukünftige Generationen nicht mehr aufbringen.

Ob und wenn ja, welche Vergleichsoperationen bzw. Sprunganweisungen fehlen, soll anhand nachfolgender Betrachtung geklärt werden. Der Code liefert die entscheidenden Argumente. Im ersten Kommentar ist jeweils die „Aufgabe“ angegeben. Für alle Programmteile gilt, dass die Variable `a` mit dem Register `1,1` und `b` mit `1,2` abgebildet wird:

```
# if (a > b) a++;  # Standardfall
GRT     1,1   1,2
JUMPC   ja
JUMP    weiter
label:  ja
ADD     1,1     1
label:  weiter
```

```
# if (a < b) a++;  # geht durch
GRT     1,2   1,1  # vertauschte
JUMPC   ja         # Operanden
JUMP    weiter
label:  ja
ADD     1,1     1
label:  weiter
```

```
# if (a >= b) a++; # geht durch
GRT     1,2   1,1  # Negation
JUMPC   weiter
ADD     1,1     1
label:  weiter

# if (a = b) a++;  # Standardfall
EQU     1,1   1,2
JUMPC   ja
JUMP    weiter
label:  ja
ADD     1,1     1
label:  weiter

# if (a = 0) a++;  # geht mit
EQU     1,1     0  # konstantem
JUMPC   ja         # Operanden
JUMP    weiter
label:  ja
ADD     1,1     1
label:  weiter
```

```
# if (a <= b) a++; # geht auch
GRT     1,1   1,2  # durch
JUMPC   weiter     # Negation
ADD     1,1     1
label:  weiter

# if (a != b) a++; # geht erneut
EQU     1,1   1,2  # mit
JUMPC   weiter     # Negation
ADD     1,1     1
label:  weiter

# if (a < 0) a++;  # geht durch
GRT       0   1,1  # Vertauschen
JUMPC   ja
JUMP    weiter
label:  ja
ADD     1,1     1
label:  weiter
```

Was hierbei auffällt ist, dass sich alle denkbaren Vergleiche gut abbilden lassen. Umfangreicher sind jeweils die „passenden“ Vergleiche, die mit `Standardfall` gekennzeichnet sind. Das erscheint paradox. Man würde erwarten, dass die Umsetzung der anderen, nicht direkt unterstützten Fälle mehr Programmzeilen benötigt. Jedoch sind gerade diese sehr günstig zu realisieren. Offensichtlich fehlt eine Sprunganweisung, die den Sprung nur ausführt, wenn das *Condition Flag* nicht gesetzt ist: **`JUMPNC`**. Das Prüfen auf 0 und kleiner 0 ist ebenfalls komplexer als es sein muss. Durch die nunmehr eindeutige Darstellung der Registerwerte gelingen diese Tests sehr einfach. Lediglich das Vorzeichen bzw. die Existenz des Bitstrings ist zu prüfen. Da **`GRT`** und **`EQU`** komplexere Algorithmen bemühen, kann mittels zusätzlicher Vergleichsoperationen Laufzeit eingespart werden: **`TESTZ`** und **`TESTN`**.

12.2.4. Ergänzungen zugunsten einer dynamischen Speichernutzung

In Abschnitt 5.3.3 wurde dargestellt, dass die bisherige Spezifikation bzgl. der Initialisierung nur schwer umzusetzen ist. Insbesondere das Überwachen, welche Bits im sequentiellen Speicher bereits beschrieben wurden und welche nicht, ist unverträglich mit einer effizienten Implementierung. Auch wenn das mit Registern leichter umzusetzen ist, erscheint es wenig intuitiv, warum der lesende Zugriff auf ein bisher nicht genutztes Register zu einem Programmabsturz führen sollte. Intuitiver und auch effizienter umsetzbar ist die Initialisierung mit dem ersten Zugriff. Ist dies ein lesender, müssen Initialwerte festgelegt werden, damit alle zukünftig ausgeführten UVC-Anwendungen gleiche Ergebnisse liefern. Diese Initialwerte sind jetzt mit aufgenommen.

Mit den bisherigen Instruktionen ist es nur mit einem „Trick“ möglich, neben den direkt im Programmcode referenzierten Segmenten dynamisch weitere zu nutzen. Dieser „Trick“

übergibt die Nummer eines bisher nicht referenzierten Segments als Parametersegment an ein Unterprogramm. Das so neu erzeugte Segment wird als Segment mit der Nummer 2 im Unterprogramm sichtbar und kann bearbeitet werden. Allerdings ist der Inhalt im aufrufenden Programm nicht sichtbar. Der zweite Teil des „Tricks“ besteht darin, ein weiteres Unterprogramm zu bemühen, das jetzt bestimmte Inhalte des übergebenen Segments in eines der *shared* Segmente kopiert. Das ist aufwendig, aber machbar.

Insgesamt passt diese Restriktion auf die direkt im Programmtext referenzierten Segmente nicht zum Konzept des UVC, der eigentlich in allen „Dimensionen“ unbegrenzt erscheint. Vielmehr mutet das wie ein Fehler an, der zu beseitigen ist. Besonders einfach ist das jedoch nicht. Nachdem mit den konstanten Operanden neue Operandentypen eingeführt wurden, wird gleich ein weiterer notwendig, um das Problem nachhaltig zu lösen: der variable Registerzugriff (VarReg). Diese Zugriffsart gestattet eine indirekte Segmentangabe und erlaubt somit, beliebige Register anzusprechen. Mit diesen Operanden wird durch die implizite Segmentangabe bei den Schreib- und Leseinstruktionen auch die flexible Nutzung des bitadressierbaren Speichers möglich. Das Fehlen der expliziten Segmentangabe wird damit ausgeglichen, ohne auf Abwärtskompatibilität mit den bisherigen Anwendungen verzichten zu müssen. Konsequenter Weise findet sich dieser Operandentyp bei allen Adressangaben und bei der Instruktion `COPY`. Intensive Berechnungen z.B. sollten zugunsten der Laufzeit über ein Unterprogramm realisiert werden. Eine stark ausgeprägte Nutzung dieser Operanden ist nicht zu erwarten. Basierend auf dem indirekten Registerzugriff ist der variable Registerzugriff leicht realisierbar. Eine Herausforderung könnten die nunmehr fünf Operandentypen dagegen für den Assembler darstellen.

12.2.5. Selbstmodifzierender Code

Es gibt gute Gründe sowohl für als auch gegen die Unterstützung selbstmodifizierenden Codes (siehe Abschnitt 7.1.1). Die spezifizierte Speicherung des Programmcodes in Segmenten lässt vermuten, dass selbstmodifizierender Code grundsätzlich unterstützt werden sollte. Eine Implementierung von IBM unterstützt dies sogar in Teilaspekten. Im Abschnitt 7.3 wurde zudem festgestellt, dass keine Gründe bzgl. der Laufzeiten entgegenstehen.

Die Weiterentwicklung trägt dem Rechnung und gibt der Segmentnummer 3 eine neue Funktion. Während mit der Segmentnummer 2 immer das Parametersegment eingeblendet ist, soll mit der Segmentnummer 3 der jeweils eigene Programmcode zugänglich werden. Damit kann ein Unterprogramm den eigenen Code modifizieren.

Wie verhält es sich jedoch mit der Erstellung und Ausführung neuen Codes? Im Prinzip ist es möglich, durch Kopieren und geschicktes Beschreiben des Speichers immer wieder neue Unterprogramme zu erzeugen und auszuführen. Wünschenswert wäre es aber, auch auf den Code anderer Unterprogramme Einfluss zu nehmen und sogar ganz neue Unterprogramme zur Laufzeit zu erschaffen, die fortan zugreifbar blieben. Erst das ermöglicht es, die von Hapgood [2001] aufgeführten Techniken tatsächlich zu nutzen.

Der Zugriff auf den Code von Unterprogrammen erfolgt über die Unterprogrammnummer, die jedem Unterprogramm schon beim Erstellen zugewiesen wird und über die ohnehin die Unterprogrammaufrufe durchgeführt werden. Mit einer zusätzlichen Instruktion soll das Einblenden der Segmente beliebiger Unterprogramme möglich werden. Erneut bietet sich

das Segment mit der Nummer 3 an, das somit nur initial auf das Segment mit dem eigenen Programmcode verweist. Die Manipulation dieser Abbildung wird mittels einer einstelligen Operation möglich: `MAP`. Dieser eine Parameter gibt die Unterprogrammnummer an. Dabei sind Unterprogrammnummern nicht zu verwechseln mit Segmentnummern. Unterprogramme werden zentral vom UVC mittels der Unterprogrammnummern verwaltet. Die Verwaltung der Segmente findet dagegen aufgrund der abgestuften Zuordnung zu den Unterprogrammen überwiegend dezentral statt. Durch diese strikte Trennung liegt der Programmcode des Unterprogramms 0 nicht im *shared* Segment mit der Nummer 0 und wäre erst mit der erfolgten Abbildung über das Segment mit der Nummer 3 zugreifbar. Die Vorteile liegen auf der Hand. Zum einen ist eine solche Trennung klar und unmissverständlich zu formulieren und zum anderen ist es wesentlich leichter, Änderungen am Programmcode nachzuvollziehen und die betroffenen Abschnitte vor der erneuten Ausführung gegebenenfalls zu optimieren (siehe Abschnitt 7.3).

An dieser Stelle muss erwähnt werden, dass selbstmodifizierender Code darauf basiert, dass dem erzeugenden Programm die jeweils gültigen Opcodes bekannt sind. Ändern diese sich im Laufe der Zeit aufgrund weiterer Entwicklungen, funktionieren diese Programme unter Umständen nicht mehr, auch wenn der Quelltext noch vorhanden ist. Es könnte hilfreich sein, über bestimmte Assembleranweisungen den jeweils gültigen Opcode zu einer angegebenen Instruktion zu referenzieren:

```
MAP      1001              # Bildet Section 1001 auf Segment 3 ab
STORE     3;0   8   @BREAK  # Speichert Opcode für BREAK an Adr. 0
CALL     1001   0   0       # Ruft das neue Unterprogramm auf
```

Verwenden selbstmodifizierenden Code nutzende UVC-Anwendungen strikt solche Platzhalter, ist die Wahrscheinlichkeit größer, dass diese Anwendungen weitere Entwicklungsschritte ohne aufwendige Rekonstruktionen überstehen.

12.3. Erste Evaluation der weiterentwickelten Spezifikation

Weiterführende Arbeiten müssen die weiterentwickelte Spezifikation ähnlich intensiv evaluieren, wie es die vorliegende Arbeit mit der von Lorie und van Diessen erstellten gemacht hat. Getreu nach dem siebten Prinzip von Myers sind alle bisher verfügbaren Tests mit jeder Verbesserung oder Fehlerbehebung erneut auszuführen. Für den vorliegenden, sehr speziellen Fall erscheint die Wiederholung der Experimente der *Zeitreise* nicht zwingend erforderlich, solange sich an der bitorientierten Ausrichtung nichts ändert. Wohl aber sollte geprüft werden, ob sich angedachte Änderungen der Spezifikation erheblich auf die Umsetzung in verschiedenen Programmiersprachen auswirken. Unbedingt notwendig erscheint es aber, zumindest eine komplexere UVC-Anwendung zu entwickeln, um die Praxistauglichkeit zu erproben. Einen Assembler als UVC-Anwendung sollte es in jedem Fall geben. Bei zukünftigen Weiterentwicklungen muss auch darauf geachtet werden, dass bereits vorhandene Anwendungen ausführbar bleiben oder aber portiert werden können, idealerweise automatisiert. Mit den letzten beiden Aspekten beschäftigten sich bereits zwei durch den Autor betreute Bachelorarbeiten.

In der ersten Arbeit bildeten nicht die Spezifikation an sich, sondern ein nach dieser Spezifikation implementierter UVC zusammen mit einer Befehlsübersicht und einer Auflistung aller Änderungen im Detail die Basis. Auf dieser Grundlage wurde ein Assembler für diesen UVC als UVC-Anwendung entwickelt, der den neuen Befehlsumfang unterstützt. Die zweite hier betrachtete Arbeit ging der Frage nach, warum der JPEG-Decoder von IBM so langsam arbeitet und ob sich dieser durch Portierung für den weiterentwickelten UVC beschleunigen lässt. Die Ergebnisse beider Arbeiten werden im Folgenden vorgestellt.

12.3.1. Ein neuer Assembler als UVC-Anwendung

Der in Kapitel 9 beschriebene Assembler nutzt komplexe Compilertechniken, die insbesondere im Falle des Evaluators überdimensioniert erscheinen. Das Assemblieren eines UVC-Quellprogramms mit der gewählten Syntax lässt sich auch deutlich einfacher realisieren. Nachdem es in erster Linie darum geht, zu einer Spezifikation auch einen passenden Assembler als UVC-Anwendung bereitzustellen, verfolgte Schaffrath [2012] einen einfacheren Ansatz. Der Quellcode seines Assemblers umfasst jedoch wesentlich mehr Programmzeilen. Obwohl die erweiterte Syntax einen nicht unerheblichen Anteil daran trägt, ist sie nicht allein dafür verantwortlich. Die Assemblierzeiten geben eine Hinweis darauf, wie effizient der Assembler wirklich arbeitet. Die Zeiten zum Assemblieren einer Anwendung sind eigentlich zweitrangig: Jede im Archiv im Quelltext aufbewahrte Anwendung muss nur einmalig initial übersetzt werden. Dennoch lassen die deutlich längeren Laufzeiten erkennen, dass Effizienz nicht oberste Priorität hatte. Im Vordergrund stand die Lesbarkeit und die Wartbarkeit, damit zukünftige Anpassungen leicht möglich sind.

Eine Neuerung der erweiterten Spezifikation schlägt sich bis auf die Ebene des Assemblers nieder: Der Verzicht auf führende Nullen. Zum einen wird verlangt, dass die Bitfolgen, die mittels `LOADC` in Register geladen werden, keine führenden Nullen haben. Daraus lässt sich ableiten, dass der Assembler die notwendige Längenangabe selbstständig berechnet. Während das für gewöhnliche Bitfolgen eine leichte Aufgabe ist, sieht es mit Sprungmarken ganz anders aus. Wie viel Bits zur Codierung der Adresse einer Sprungmarke notwendig sind, steht erst nach der Assemblierung fest. Zwischendurch muss mit Näherungen gearbeitet werden. Der hierfür notwenige Algorithmus ist nicht trivial. Der gegenüber der bisherigen Spezifikation zu erbringende Mehraufwand hält sich jedoch in Grenzen.

Ein Blick in die Implementierungsdetails gibt Aufschluss über den Nutzen der weiterentwickelten Spezifikation. Durch die nunmehr nutzbaren konstanten Operanden verliert die Instruktion `LOADC` etwas an Bedeutung. Deutlich in den Vordergrund drängt jetzt die Instruktion `ADD` mit konstantem Operanden, was durch die häufige Verwendung in Schleifen begründet ist. Hoch frequentiert genutzt wurden Sprünge zu konstanten Sprungmarken. Bei der Realisierung eines Automaten finden sich aber auch Sprungmarken in Registern. Insgesamt bewertet Schaffrath den erweiterten Befehlsumfang als sehr passend. Beinahe alle Befehle wurden genutzt, jedoch nur 40% der möglichen Variationen, die durch die verschiedenen Operandentypen möglich werden. Wie zu erwarten mussten keine VarRegs bemüht werden. Durch die Verwendung von Konstanten ist sein Quelltext gut lesbar und leicht nachvollziehbar.

12.3.2. Adaption des JPEG-Decoders

Zwei Fragen begleiteten die vorliegende Arbeit, die immer wieder den JPEG-Decoder von IBM als Benchmark nutzt: Wie groß wäre der Gewinn, wenn diese Anwendung gleich für den weiterentwickelten UVC implementiert worden wäre und wie effizient haben die Entwickler von IBM eigentlich implementiert?

Diesen Fragen ging Nau [2011] in seiner Bachelorarbeit nach. Zugleich beantwortete er damit auch die für die Langzeitarchivierung wichtige Frage nach der Anpassbarkeit vorhandener UVC-Anwendungen. Die Anpassung an die weiterentwickelte Spezifikation gelang mühelos. Aufgrund der im Segment 0 abgelegten konstanten Werte und des einheitlich genutzten Musters für Sprunganweisungen konnten zudem alle Sprungmarken und viele Konstanten direkt als Operanden genutzt werden. Allein diese Anpassung ermöglichte eine Laufzeitverbesserung von 44%. Die aufgrund des erweiterten Befehlsumfangs nachträglich mögliche Anpassung des Kontrollflusses erzielte einen zusätzlichen Laufzeitgewinn von 2%. Auch die Modifizierung der Ausgabe hat einen deutlich messbaren Einfluss. Allein durch diese leicht möglichen Anpassungen, die so auch für andere archivierte Anwendungen möglich wären, konnte fast ein Drittel der benötigten Laufzeit eingespart werden.

Die Frage nach der Effizienz versuchte Nau zunächst über die Auswertung der verwendeten Algorithmen zu beantworten. Diese stellten sich jedoch als zeitgemäß heraus. Auf den ersten Blick bedurften diese nicht unbedingt einer Generalüberholung. Der Blick ins Detail lässt aber erhebliches Potential erkennen. Wohl bedingt durch die Nutzung verschiedener Programmierhilfen, sind erkennbar unsinnige indirekte Registerzugriffe zu finden. Die Eliminierung der offensichtlich unnötigen Zugriffe bringt einen Laufzeitvorteil von 2,5%. Ein detaillierter Blick in die Umsetzung eines der am meisten genutzten Programmabschnitte lässt erahnen, welches Verbesserungspotential darüber hinaus noch im Decoder steckt. Nau stellt eine angepasste Realisierung der inversen diskreten Cosinustransformation vor, die anstatt der 6544 ursprünglich genutzten Additionen nur noch 3488 benötigt. Bei genauerer Betrachtung der vorgestellten Lösung findet man weiteres Potential und schafft mühelos eine Implementierung mit nur 1023 Additionen durch „Ausrollen“ sämtlicher Schleifen. Der originale Decoder benötigt insgesamt weit über 177 Millionen Instruktionen zur Decodierung des großen Testbildes, die Anpassung von Nau nur noch knapp 91 Millionen. Die von mir nachträgliche Anpassung benötigt lediglich noch knapp über 68 Millionen Instruktionen und benötigt zur Decodierung nur noch 52% der Laufzeit. Diese Anpassungen beschränkten sich auf nur zwei Unterprogramme, die überdies nicht die besten Algorithmen umsetzen. Es ist daher Raum für die spekulative Annahme, dass ein hoch optimierter Decoder nur noch ein Viertel der ursprünglichen Laufzeit benötigen wird. Vom erweiterten Befehlsumfang profitieren neue UVC-Anwendung bereits in der Entwicklungsphase und sollten keiner nachträglichen Optimierung bedürfen.

12.3.3. Destillat der ersten Evaluationsergebnisse

Zusammenfassend lässt sich feststellen, dass der erweiterte Befehlsumfang zugleich lesbareren Quelltext als auch deutlich effizientere Programme ermöglicht. Die zahlreichen Varianten der Befehle zur Nutzung der verschiedenen Operandentypen können ohne permanentes Referenzieren der Opcodetabelle verwendet werden, da die Syntax weiterhin intuitv gehaltet ist.

13. Ausblick

Dieser Ausblick dient der Sammlung der Gedanken, die es meines Erachtens wert sind, genauer betrachtet zu werden, für deren Ausgestaltung jedoch innerhalb dieser Arbeit kein Platz mehr war. Im Wesentlichen sind diese Gedanken von dem Wunsch geprägt, den UVC noch mehr in der Langzeitarchivierung zu verorten.

13.1. Bewusstseinsförderung

Es lässt sich nicht leugnen, dass der UVC unlängst in der Forschergemeinschaft der Langzeitarchivierung wahrgenommen wird. Einige wichtige Belege dafür gibt es bereits. So ist dem UVC ein eigener Anhang in Gladneys [2007] Buch gewidmet.

Andere Projekte wie Dioscouri [van der Hoeven et al., 2007] machen aber den Ansatz des UVC nahezu unsichtbar. Archive wiegen sich dadurch in falscher Sicherheit. Die über drei Jahre andauernde Entwicklungszeit für diesen Emulator steht in keinem Vergleich zu den wenigen Wochen für die Entwicklung des UVC. Das Bewusstsein, für den UVC bereits jetzt Anwendungen zu entwickeln, ist noch nicht vorhanden. Der Bedarf ist noch nicht sichtbar. Dieser Aspekt ist aus meiner Sicht leider der größte Nachteil dieser Archivierungsmethode. Die Anwendungen, die man heute schreibt, werden in absehbarer Zukunft gar nicht zum Einsatz kommen, sondern erst viel, viel später.

Hier könnte man sich ein Tool vorstellen, das zentral den UVC beinhaltet und real einen Bildkonverter darstellt. Davon gibt es bereits sehr viele, aber die wenigsten können frei programmiert werden. Man stelle sich also ein Tool vor, das bereits eine Datei in den Speicher des UVC laden kann. Es können zudem aus einer Liste von UVC-Programmen Filter gewählt werden und zusätzlich stehen danach noch Möglichkeiten zu weiteren Verarbeitung zur Verfügung. Solche Filter können von einer Community im Sinne von „UVC 2.0“ selbst erstellt und fortentwickelt werden. Auf diese Weise dürfte ein schneller Zuwachs an UVC-Anwendungen erreicht werden – ohne öffentliche Gelder zu beanspruchen. Der intern verwendete UVC müsste dafür aber äußerst effizient implementiert sein. Die Erfahrung in solchen Gebieten zeigt, dass nicht nur der Funktionsumfang durch eine solche Community zunehmen kann, sondern auch regelmäßig Verbesserungen in der bereits bestehenden Funktionalität vorgenommen werden. So wären von selbst – allein durch den Antrieb der Mitglieder – immer effizientere UVC-Anwendungen zu erwarten, sobald dieses Tool eine breitere Verwendung findet.

13.2. Wettbewerbsgedanke

Es gibt zahlreiche Programming-Contests. Warum nicht einen für den UVC, der verschiedenste Plattformen umfasst? Die jeweils schnellste Implementierung würde öffentlich zugänglich. Neben universitären Einrichtungen gäbe das zum Beispiel auch Schulen Möglichkeiten zur sinnvollen und motivierenden Gestaltung von Praktika. Wie in der vorliegenden Arbeit gezeigt, werden sehr viele Bereiche der Informatik bei der Implementierung eines UVC berührt, gerade wenn eine effiziente Implementierung angestrebt wird.

Etabliert sich ein solcher Wettbewerb, wären in kürzester Zeit für die verschiedensten Plattformen UVC-Laufzeitumgebungen verfügbar, die zudem fortlaufend verbessert würden. Bei konstantem Interesse ließen sich dann über die Zeit fortlaufend neue Plattformen ausschreiben. So wären zu jeder Zeit für die relevanten Plattformen UVC-Implementierungen verfügbar.

Um die geforderte Funktionalität zu prüfen, könnten die in dieser Arbeit erarbeiteten Testprogramme an die Hand gegeben werden. Allzu fehlerhafte Einreichungen wären damit ausgeschlossen. Damit nicht nur ausgewählte Aspekte des UVC effizient implementiert werden, muss eine geeignete UVC-Anwendung als Benchmark implementiert und fortwährend weiterentwickelt werden. Absehbar notwendig wird es werden, mehr als einen Wert zum Vergleich heranzuziehen. Zu unterschiedlich sind die Ausprägungen einer intensiven Nutzung des UVC.

13.3. Zukünftige UVC-Anwendungen

Aufgrund knapper Ressourcen stellt sich die Frage nach der Priorisierung zukünftig zu entwickelnder UVC-Anwendungen. Viele Formate werden bereits bei der Archivierung digitaler Objekte genutzt. Aber zwei Dateiformate erscheinen aus fachfremder Sicht besonders interessant: ZIP und PDF/A.

UNZIP

Bitströme kommen für gewöhnlich selten allein. Sie sind in einer Datei gespeichert. Diese wiederum hat einen Namen und wird von anderen Metadaten wie Größe und Datum der Erstellung begleitet. Sehr oft ist eine Datei in einem Kontext gebunden, der aus vielen Dateien besteht. So ist z.B. eine UVC-Anwendung allein nicht verständlich. Eine Anleitung, die die zu erwartenden Eingaben und das Format und die Bedeutung der Ausgaben angibt, wird ebenfalls benötigt. Die vorliegende Arbeit schlägt darüber hinaus auch die Archivierung des Quelltextes vor. Für jedes Unterprogramm gibt es dann eine zusätzliche Datei. Zur Bündelung dieser Dateien eignen sich komprimierte Ordner sehr gut. Bei der derzeit bequemen Nutzung von komprimierten Dateiarchiven gibt es jedoch zu beachten, dass jederzeit Software zur Dekompression bereitsteht und folglich der Migration bedarf [Borghoff et al., 2006]. An dieser Stelle drängt sich förmlich die Entwicklung einer solchen Software als UVC-Anwendung auf.

PDF/A

Das hoch gesteckte Ziel, einen PDF-Betrachter als UVC-Anwendung zu entwickeln, wurde bisher nicht erreicht. Als zu komplex wurde sowohl das Format als auch die zu implementie-

rende Funktionalität eingestuft [Lorie, 2002b]. Wohl aus der Not heraus wurde stattdessen die vorangehende Transformation in statische Bilddateien vorgeschlagen.

Aufgeben sollte man dieses Vorhaben insgesamt aber nicht. Allein Textextraktoren wären schon von beachtlichem Nutzen angesichts stets wachsender Akzeptanz von PDF/A-Dateien. Dies erscheint eine realistische Aufgabe z.B. für eine Masterarbeit. Mit der vollständigen Implementierung eines PDF-Betrachters sollte gewartet werden, bis effiziente und handliche Programmierwerkzeuge und Unterprogrammbibliotheken für den UVC zur Verfügung stehen.

13.4. Weiterführende Arbeiten

Neben der intensiven Evaluation der erarbeiteten weiterentwickelten Spezifikation lässt sich die vorliegende Arbeit in drei Richtungen fortführen. Diese werden im Folgendem motiviert und erste Pfade geebnet.

Compiler

Die vorliegende Arbeit ist mit der Feststellung nicht allein, dass sich komplexere Anwendungen wesentlich leichter mit zunehmender Abstraktion realisieren lassen. Werden dabei gut entwickelte Programmierwerkzeuge genutzt, muss das nicht zwingend mit einem Laufzeitverlust einhergehen. Solche Programmierwerkzeuge können der Akzeptanz der den UVC nutzenden Archivierungsmethode dienen und zudem das Verhältnis zwischen Entwicklungsaufwand und Grad an Authentizität deutlich verbessern.

Höhere Abstraktion erlangt man durch Hochsprachen und entsprechende Compiler. Aber welche Sprache sollte unterstützt werden? Dass sich Java scheinbar nicht eignet, haben Kol et al. [2006] implizit bereits gezeigt. Lässt sich das verallgemeinern?

Craig [2006] entwickelt exemplarisch für eine Sprache eine virtuelle Maschine, setzt also für die Entwicklung einer Maschine eine damit leicht abzubildende Hochsprache voraus. Wenn das Ziel also ein UVC für einen MiniJava-Dialekt sein sollte, wäre das Design vom UVC gründlich daneben gegangen. Dieser Weg eignet sich jedoch nicht für den UVC, der aufgrund der speziellen Ausrichtung auf die Langzeitarchivierung bestimmte Eigenschaften besitzt. Ziel sollte es sein, eine entsprechende, tatsächlich abgestimmte Hochsprache für den UVC zu entwickeln. Das ist deutlich keine triviale Aufgabe. Zu sehr müsste diese Sprache von vorhandenen Beispielen abweichen, um alle Vorzüge des UVC trotz der Abstraktion an den Programmierer durchzureichen. Primitive wie auch komplexere Datentypen müssten nutzbar sein, ebenso Methoden mit Übergabeparametern. Wer aber Felder im bitadressierbarem Speicher oder Fließkommazahlen mit nur einem Register abbildet, begeht bereits strukturelle Fehler. Die Parameterübergabe schließlich wird extrem komplex, da alle aufzurufenden Unterprogramme nur exakt ein Parametersegment übergeben bekommen. Das sind nur wenige Aspekte, die sofort ins Auge springen.

Sicher nicht auszuschließen ist in diesem Zusammenhang, dass erneut eine Weiterentwicklung der Spezifikation evident wird. Noch gravierender, aber durchaus zweckmäßig, wäre das Ziel einer für die LZA designten Hochsprache mit passendem UVC. Gelingt ein solcher um-

fassender Schritt, ohne die überprüften Eigenschaften des UVC wie die Universalität zu verlieren, müsste die Neuentwicklung bisheriger Anwendungen akzeptiert werden.

Craig [2006] „spendiert“ z.B. einer VM für eine objektorientierte Sprache unter anderem einen Maschinenbefehl für den Methodenaufruf. Hierbei „wühlt“ dieser sich selbst durch die Klassenhierarchie und findet in den Tabellen den richtigen Eintrag. Derart komplexe Vorgänge mit nur einem Befehl der Basismaschine abzudecken, ist überaus sinnvoll. Es spart enorm Laufzeit, da diese Schritte nicht in der Sprache selbst formuliert und anschließend „interpretiert“ werden müssen. Diese Aspekte gilt es dann zu berücksichtigen. Die Entwicklung einer solchen Hochsprache stelle ich mir daher als Dissertations- oder sogar als Habilitationsprojekt vor.

Die Arbeit von Rothe [2011] ebnet einen ersten Pfad in dieses komplexe Gebiet. Er untersuchte, ob sich die Assemblersyntax direkt um ausgewählte Hochsprachenkonstrukte erweitern lässt. So wurden kombinierte boolesche Ausdrücke, bedingte Anweisungen und Schleifen realisiert. Bislang ungeklärt ist, ob auch komplexere Konstrukte wie Unterprogrammaufrufe allgemeingültig und zugleich effizient umzusetzen sind. Zu untersuchen wären in einem ersten Schritt die Teilbereiche Programmfluss inklusive Unterprogrammtechnik, Datentypen und -strukturen und die Realisierung und Nutzung ausgelagerter Bibliotheken. Neben dem imperativen Ansatz sollte auch die Abbildung eines deklarativen Paradigmas untersucht werden.

LDS und LDV / Ontologie

Das Problem wurde bereits in Abschnitt 3.1.2 beschrieben. Es geht im Schwerpunkt um die Frage, wie das Wissen über die Anwendung einer UVC-Anwendung erhalten bleibt. Die Forschung in diesem Bereich berührt viele externe Bereiche und wird damit schnell sehr komplex.

Das Ziel sollte es nicht nur sein, die von einer UVC-Anwendung erzeugten Ausgaben und deren Bedeutung knapp und intuitiv zu beschreiben, sondern auch die Eingaben. Im Idealfall wird es damit auch möglich, zukunftssicher zu beschreiben, wie UVC-Instanzen auf den Ergebnissen anderer aufbauen bzw. wie verschiedene Instanzen miteinander interagieren.

Erschließen weiterer Anwendungsfälle

Mit der Implementierung des Spiels Tetris wurde die Tür in den Bereich interaktiver Anwendungen für den UVC geöffnet. Die ersten Ergebnisse deuten zwar eine mögliche Fortentwicklung des UVC an, aber bedürfen noch konkreter Evaluation. Neben den interaktiven Anwendungen könnte auch ganz allgemein untersucht werden, inwiefern sich der UVC zur Abbildung paralleler Prozesse eignet.

Bislang sieht die Nutzung des UVC vor, dass eine *Restore Application* den UVC mit der UVC-Anwendung startet, anschließend mit Daten füttert und die vom UVC erzeugten Ausgaben aufbereitet. Was wäre, wenn eine UVC-Anwendung auf eine komplexe Datenbank zugreifen müsste. Die Logik kann ohne Zweifel mit einer UVC-Anwendung realisiert werden. Aber wo und wie wird der Datenbestand so archiviert, dass dieser zugreifbar bleibt? Wie spezifiziert man das notwendige Zugriffsprotokoll?

Anhang

A. Anhang

Dieser erste Teil des Anhangs nimmt vor allem diejenigen Anteile auf, für die zur umfassenden Darstellung im laufenden Text kein Platz war. Dazu gehört die Spezifikation der UVC-Assemblersprache sowie die vollständige tabellarische Darstellung des Scannerautomaten.

Im zweiten Teil findet sich ein für Programmierer relevantes Ergebnis der vorliegenden Arbeit. Die gesammelten Hinweise für eine effiziente Implementierung des UVC sind nicht aus Platzgründen im Anhang platziert, sondern möchten sich gern als kompakte und somit schnell greifbare Referenz verstanden wissen.

A.1. Spezifikation des UVC-Assemblers

Bisher fehlt der UVC-Spezifikation von Lorie und van Diessen [2005] eine Spezifikation der verwendeten Assembler-Sprache. Und das obwohl diese ein Beispielprogramm enthält. Da in der vorliegenden Arbeit ebenfalls viel UVC-Assemblercode zu finden ist, wird hier die Syntax und die Grammatik angegeben. Zudem sind diese Basis der eigenen Assemblerimplementierung aus Kapiel 9. Diese spezifizierenden Teile können daher auch referenzierend genutzt werden, wenn auf Basis der bisherigen Implementierung eine Erweiterung oder Anpassung durchzuführen ist.

Der durch die eigene Implementierung unterstützte Sprachumfang sieht ein optionales Komma bei den Registerangaben zwischen den Segment- und den Registernummern vor. Dadurch werden Quellprogramme intuitiver lesbar.

Das Beispiel in der Spezifikation von Lorie und van Diessen gibt keine Auskunft über die Position des `*` zur Kennzeichnung des indirekten Registerzugriffs. Die Spezifikation nutzt `R*`, weshalb diese Darstellung abgeleitet wurde:

```
NSIGN 1,1*
```

Erst nach der Fertigstellung des eigenen Assemblers wurde der von IBM entwickelte Assembler veröffentlicht. Dieser sieht eine andere Position vor. Ebenfalls von der Spezifikation weichen einige Schlüsselwörter ab. Diese sind in Anlehnung an die intuitiveren Namen, die in den ebenfalls veröffentlichten neueren Beispiel-Versionen auftauchen, angepasst worden.

Die folgenden Angaben sind in der von JACCIE unterstützen Form angegeben. Nachdem diese ebenso wie andere Formen intuitiv lesbar sind, dient dies vorrangig als Einladung zur Nutzung unseres Tools. Die Nutzung von JACCIE am Beispiel der UVC-Assemblersprache ist sehr schön aufbereitet in [Krebs und Schmitz, 2012] zu finden.

A.1.1. Syntax – Definition des Scanners

```
<asterix>   := #042
<comma>     := $,
<hexnum>    := $0$x(({$0-$9}) | ({$A-$F}))[1-*]
<loadcins>  := "LOADC"
<jumpins>   := ("JUMP"|"JUMPC")
<label>     := "label:"
<name>      := ({$a-$z}|{$A-$Z}) ({$a-$z}|{$A-$Z}|{$0-$9})[0-*]
<number>    := ({$0-$9})[1-*]
<op0ins>    := ("BREAK"|"STOP")
<op1ins>    := ("NSIGN"|"PSIGN"|"NOT"|"PRINTR")
<op2ins>    := ("LSIGN"|"SSIGN"|"AND"|"OR"|"RLEN"|"COPY"|
                "ADD"|"SUB"|"MULT"|"GRT"|"EQU"|"PRINTM")
<op3ins>    := ("CALL"|"LOAD"|"STORE"|"DIV"|"IN"|"OUT")
<printcins> := "PRINTC"
<string>    := $"( ({#032-#033}) | ({#035-#127}) )[0-*]$"
```

A.1.2. Grammatik – Definition des Parsers

```
Section   -> name number UsedSegs Program
UsedSegs  -> number SegHelp
SegHelp   -> comma number SegHelp
          ->
Program   -> op0ins Program
          -> op1ins Register Program
          -> op2ins Register Register Program
          -> op3ins Register Register Register Program
          -> loadcins Register LOADChelp Program
          -> printcins number Constant Program
          -> jumpins Register JUMPhelp Program
          -> label name Program
          ->
LOADChelp -> number Constant
          -> name
JUMPhelp  -> name
          ->
Register  -> number RegNo
RegNo     -> comma number RegHelp
          -> number RegHelp
RegHelp   -> asterix
          ->
Constant  -> hexnum
          -> number
          -> string
```

A.1.3. Opcode-Tabelle für den UVC 1.3.2

Instruktion	Parameter			Opcodes		Beschreibung
NSIGN	Register			0	00	setzt Vorzeichen negativ
PSIGN	Register			1	01	setzt Vorzeichen positiv
LSIGN	Register (Ziel)	Register (Adresse)		2	02	lädt Bit aus Speicher als Vorzeichen
SSIGN	Register (Quelle)	Register (Adresse)		3	03	speichert Vorzeichen als Bit
AND	Register (Ziel)	Register (Argument)		16	10	bitweise Und
OR	Register (Ziel)	Register (Argument)		17	11	bitweise Oder
NOT	Register (Ziel)			18	12	bitweise Negation
JUMP	Register (Adresse)			32	20	unbedingter Sprung
JUMPC	Register (Adresse)			33	21	bedingter Sprung
CALL	Register (Section ID)	Register (Adresse)	Register (Segment Nr.)	34	22	Aufruf einer Section
BREAK				35	23	Rücksprung
LOADC	Register (Ziel)	Länge	bit string…	48	30	Laden einer Konstante
LOAD	Register (Ziel)	Register (Adresse)	Register (Länge)	49	31	lädt Bitfolge aus dem Speicher
STORE	Register (Quelle)	Register (Adresse)	Register (Länge)	50	32	speichert Bitfolge aus dem Speicher
REGLENGTH	Register (Ziel)	Register (Quelle)		51	33	lädt Länge eines Registers als Wert
COPY	Register (Ziel)	Register (Quelle)		52	34	kopiert den Wert eines Registers
ADD	Register (Ziel)	Register (Argument)		64	40	addiert Register auf
SUB	Register (Ziel)	Register (Argument)		65	41	subtrahiert Argument von Ziel
MULT	Register (Ziel)	Register (Argument)		66	42	multipliziert Ziel mit Argument
DIV	Register (Quotient)	Register (Argument)	Register (Rest)	67	43	dividiert Ziel mit Argument, füllt Rest
GRT	Register1 (Argument)	Register2 (Argument)		80	50	setzt Bedingung auf Reg1 > Reg2
EQU	Register1 (Argument)	Register2 (Argument)		81	51	setzt Bedingung auf Reg1 == Reg2
IN	Register (Messagetyp)	Register (Länge)	Register (Adresse)	96	60	Input
OUT	Register (Messagetyp)	Register (Länge)	Register (Adresse)	97	61	Output
STOP				112	70	stoppt Programmausführung
PRINTR	Register (Quelle)			128	80	Print Register
PRINTM	Register (Adresse)	Register (Länge)		129	81	Print Memory
PRINTC	Länge	bit string… (ASCII codiert)		130	82	Print String/Konstante

Abbildung A.1.: Opcode-Tabelle zum UVC V1.3.2

Sowohl die Opcodes als auch einige Namen haben sich im Vergleich zur Spezifikation nach Lorie und van Diessen [2005] geändert. Von der Funktionalität aber weichen nur die Instruktionen **LOADC** und **AND** ab (siehe 3.4.1).

Die letzten drei Instruktionen müssen nur soweit implementiert werden, dass der Prozessor bei Abarbeitung diese Instruktionen ignorieren, also ohne Fehler überspringen kann. Die systemunabhängige Implementierung des **PRINTC** erscheint zudem fragwürdig, denn nicht alle Systeme verwenden die ASCII-Codierung. Die Spezifikation müsste hierfür die genutzte Codierung des verwendeten Zeichensatzes beinhalten – ähnlich wie bei Knuth [1969b] zu sehen – oder aber darauf verweisen, dass diese Ausgaben nur für die jeweilige Epoche relevant sein dürfen und keinen Einfluss auf das Gelingen der Interpretation der erzeugten Ausgaben haben darf.

A.1.4. Tabellarische Darstellung des Scannerautomaten

state	#0	TA	EF	LF	CR	SP	,	*	#	:	"	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f	g	h	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	Token
1	79	79	82	80	81	79	71	72	73		83	74	75	75	75	75	75	75	75	75	75	8	11	16	22	24	7	26	7	28	29	7	34	39	41	47	49	7	55	63	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	2	7	7	7	7	7	7	7	7	7	7	7	7	7	7	
2												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	3	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
3												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	4	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
4												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	5	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
5												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
6										78		7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
7												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
8												7	7	7	7	7	7	7	7	7	7	7	7	7	9	7	7	7	7	7	7	7	7	7	9	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
9												7	7	7	7	7	7	7	7	7	7	7	7	7	10	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
10												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	op2ins
11												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	12	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
12												7	7	7	7	7	7	7	7	7	7	7	7	7	7	13	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
13												7	7	7	7	7	7	7	7	7	7	14	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
14												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	15	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
15												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	op0ins
16												7	7	7	7	7	7	7	7	7	7	17	7	7	7	7	7	7	7	7	7	7	7	7	7	20	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
17												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	18	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
18												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	21	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
19												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	20	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
20												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	10	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
21												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	op3ins
22												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	23	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
23												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	21	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
24												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	25	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
25												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	10	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
26												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	27	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
27												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	10	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
28												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	21	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
29												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	30	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
30												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	31	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
31												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	32	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
32												7	7	7	7	7	7	7	7	7	7	7	7	33	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	jumpins
33												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	jumpins
34												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	35	7	7	7	64	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
35												7	7	7	7	7	7	7	7	7	7	36	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
36												7	7	7	7	7	7	7	7	7	7	7	7	7	37	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
37												7	7	7	7	7	7	7	7	7	7	7	7	38	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	op3ins
38												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	loadcins
39												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	40	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
40												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	27	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
41												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	42	7	7	7	44	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
42												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	43	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
43												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	op1ins
44												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	45	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
45												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	46	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
46												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	43	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
47												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	10	7	7	48	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
48												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	21	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
49												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	50	44	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
50												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	51	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
51												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	52	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
52												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	53	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
53												7	7	7	7	7	7	7	7	7	7	7	7	54	7	7	7	7	7	7	7	7	7	10	7	7	7	7	43	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
54												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	printcins
55												7	7	7	7	7	7	7	7	7	7	7	7	7	7	56	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
56												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	57	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
57												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	58	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
58												7	7	7	7	7	7	7	7	7	7	7	7	7	7	59	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
59												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	60	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
60												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	61	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
61												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	62	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
62												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	10	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
63												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	64	67	70	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
64												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	65	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
65												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	66	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
66												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	10	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
67												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	68	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
68												7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	15	7	69	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
69												7	7	7	7	7	7	7	7	7	7	7	7	7	7	21	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
70												7	7	7	7	7	7	7	7	7	7	7	10	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	name
71																																																																										comma
72																																																																										asterix
73	73	73				73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	73	comment
74												75	75	75	75	75	75	75	75	75	75																																																	76			number
75												75	75	75	75	75	75	75	75	75	75																																																				number
76												77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77																																																		
77												77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77																																																	hexnum
78																																																																										label
79	79	79				79																																																																			separator
80																																																																										newLine
81				80																																																																					
82																																																																										EOF
83	83	83				83	83	83	83	83	84	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	83	
84																																																																										string

Abbildung A.2.: Gekürzte Tabelle für den Scanner.

Der dem Scanner zugrunde liegende Automat ist in tabellarischer Form angegeben. Die Tabelle umfasst für jedes lesbare Zeichen eine Spalte und für jeden Zustand eine Zeile. Eine zusätzliche Spalte enthält das erkannte Token, falls die Zeile einen Endzustand codiert.

Die gekürzte Tabelle umfasst $(72 + 1) \cdot 84 = 6132$ Einträge. Gekürzt wurden leere Spalten oder solche, die nur die Werte 73 und 83 enthalten. Wird die Tabelle mit 0 initialisiert, gelingt das Befüllen mit „nur“ 141 UVC-Intruktionen. Die grau unterlegten Einträge lassen sich durch geschicktes Nutzen von verschachtelten Schleifen befüllen.

A.2. Hinweise zur effizienten Implementierung eines UVC

Die hier zusammengetragenen Hinweise referenzieren Abschnitte innerhalb der gesamten Arbeit. Eine Auflistung mit dem jeweiligen Bezug zum entsprechenden Abschnitt soll helfen, sich die vorliegende Arbeit mit einem starr auf die Implementierung eines UVC gerichteten Blick zu eigen zu machen.

- **Speicherverwaltung**
  - **Balancierte Suchbäume** eignen sich gut zur Realisierung der unter Umständen stark dynamischen Speichernutzung durch UVC-Anwendungen (3.4.1, 5.3.3).
    Das Löschen einzelner Blätter oder Knoten ist an keiner Stelle notwendig (6.4.5).
    Das Löschen kompletter Bäume samt referenziertem Inhalt dagegen schon (5.3.6).
    Schlüssel in Wortgröße, aber mind. 32 Bit; größer kann sinnvoll sein (3.2).
  - **Speicherbereinigung** ist notwendig; falls nicht bereitgestellt, sollte Speicher selbst freigegeben werden (5.3.6).
  - **Die verteilte Segmentverwaltung** ist deutlich schneller (5.3.3, 6.4.3, 6.4.5).
    Ein Array für *shared* Segmente.
    Ein balancierter Baum für Segmente ab 1000. Je einer pro Section.
    Segment 2 als Attribut des zu rettenden Prozessorzustandes (6.2.3).
    Segment 1 als Attribut einer Section (6.2.3).
    Zentrale Verwaltung der Sections (6.4.3, 6.2.3).
  - **Bitadressierbarer Speicher** im Segment
    Speicherblöcke pro Segment als B-Baum (5.3.3), wenn Auslagerung angedacht ist.
    Sonst sind andere Baumstrukturen wie AVL-Bäume schneller (6.4.5).
    Blockgröße zwingend größer als ein Speicherwort (3.4.1, 5.3.3). Aber zu große Speicherblöcke führen zu Speicherverschwendung und Laufzeitverlusten bei Vorabinitialisierung mit festgelegtem Wert.
    Beim Zugriff Wortbreite der Basismaschine ausnutzen (6.6.2) und eventuelle Unterstützung beim Aufbereiten größerer Abschnitte berücksichtigen (6.4.5).
  - **Register** im Segment
    Falls möglich, sollte eine Programmiersprache gewählt werden, für die es verfügbare Bibliotheken zur Arithmetik beliebiger Genauigkeit gibt (5.3.2, 5.4.1). Falls nicht, gibt es zahlreiche Quellen, die sich diesem nicht trivialen Thema widmen (5.3.2).
    Von der Hardware unterstützte Wortbreite sollte genutzt werden. Genutzte (Hoch-) Sprache muss noch den Überlauf erkennen können (5.3.1, 6.4.1, 6.2.3, [Schiller, 2012]). Aufwendige Maskierungen sollten nach Möglichkeit vermieden werden (6.2.3, 6.2.3).
    Sofern nicht mit sehr großen Divisionen gerechnet wird, eignen sich einfach verkettete Listen zur Speicherung der Worte im Register mit dem am wenigsten signifikanten Wort voran (5.3.1, 6.6.2). Falls doch, bietet die Normalisierung bzw. die

Denormalisierung im Verfahren nach Knuth eine laufzeitneutrale Möglichkeit, die Reihenfolge umzudrehen (5.3.1).
Register selbst sollten in einer Baumstruktur gehalten werden (5.3.3).
Effiziente Vergleiche sind aufgrund führender `1` möglich (12.2.2).
Realisierung der VarRegs zentral mit indirektem Registerzugriff (12.2.4).

- **Prozessor**
  - **Auswertung der Opcodes** mit Sprungtabellen (7.2.1, [Müller, 2011]). Alternativ können über die Reihenfolge häufige Instruktionen bevorzugt werden (6.4.5).
  - **Interpreter** des Bitcodes, modifizierbarer Code im Segment möglich (6.4.5).
    Bits der Operanden in einem Zug laden und dabei Hardwareunterstützung ausnutzen (6.4.5, insb. Abb. 6.8).
    Neben den Operanden kann auch in vielen Fällen der nächste Opcode mit geladen werden. Eventuell folgende Shift-Operationen können entfallen, wenn die Fallunterscheidung entsprechend angepasst wird (6.4.5).
    Speichern des Bitcodes in einem Array passender Größe zusätzlich zur Baumstruktur (6.6.2).
  - **Aufbereiteter Bitcode** – parsende Interpreter, Instruktionen vorab erkennen und in eigener Struktur speichern (5.3.4, [Müller, 2011]).
    Struktur sollte das Ablegen eines Registerverweises ermöglichen (5.3.4).
    Direkte Register können zusätzlich zur Verwaltung im Baum direkt verlinkt werden (5.3.4, [Müller, 2011]).
    Sprungadressen lassen sich in einem Array verwalten (7.2.1), Suche der Sprungmarken nach Bisektion ist dann möglich, direkter Bezug bei konstanten Sprüngen (5.3.4). Das gilt auch für lediglich entzerrten Bitcode, der auf gerade Adressen gerückt wurde.
    Abbilden der Instruktionen auf interne 3-Register-Maschine (5.3.4). Dadurch werden weitere Optimierungen möglich.
  - **Selbstmodifizierender Code** ist durch dynamische Anpassung nach schreibendem Zugriff auf Segment 3 möglich (7.3, 12.2.5).
  - **Unterprogrammaufrufe** mit Stack (6.2.3) oder Rekursion ([Müller, 2011]).
  - Das Implementieren eines optionalen Loggings (***Traces***) kann Laufzeitverlust bedeuten; ggf. sind zwei separate Kompilierungen zu erstellen (10.2.2).
- **Ein- und Ausgabe**
  - Die Ein- und Ausgabeformate sollten mit den zu implementierenden *Restore Applications* abgestimmt und dabei durch das OS bereitgestellte effiziente Wege genutzt werden (5.3.5, 11.2.4).
  - Zur Realisierung auf heutigen Systemen bieten sich die Standardein- und -ausgabe an (5.3.5).

Um die Testsammlung bereits während der Entwicklung nutzen zu können, muss die UVC-Implementierung das Laden eines Programms möglicht früh umsetzen (10.2.1).

B. Extended Specification of the Universal Virtual Computer

This specification is based upon an earlier specification written by Raymond A. Lorie and Raymond J. van Diessen. The Universal Virtual Computer (UVC) was first introduced in Lorie [2001]. Lorie [2002a] added more detail and outlined the UVC architecture. As part of a joint proof-of-concept study [Lorie, 2002b], IBM and the KB (Koninklijke Bibliotheek, the National Library of the Netherlands) fully implemented a considerably different and more advanced UVC version. Some three years later, a complete specification of this version was published by Lorie and van Diessen [2005] and named the UVC Convention, version 0. With great foresight they noted that "it would be unrealistic to claim that the proposed UVC will never change."

Consequently, the specification given here will not be final, either. It is named the UVC Convention, version 0.5. A few aspects are still unresolved and should be included as soon as a working solution is found. One of these open issues concerns the presentation of information to future generations: The UVC produces bit streams containing the archived information. How do we specify the process of converting such bit strings into a format that is fit for human consumption? Obviously, this belongs into a truly usable specification.

In a long series of experiments, the UVC and the Convention, version 0 were tested thoroughly. The greater part of these experiments belongs to a "virtual time trip" undertaken with machines from our computer museum datArena – providing a sampler of outdated, but still fully operational supercomputers – and covering the last 50 years of computer history. The remaining experiments were aimed at investigating performance issues. The new specification reflects experience from all these experiments.

To assist readers who are familiar with the previous specification, new or modified text passages are shown in blue.

B.1. Introduction: the preservation problem

Preserving bit strings (or files) is a basic requirement of any data processing system. The problem has existed since the beginning of the computer era. The classical solution consists of copying the information periodically from an old medium to a new one. This solves both medium and device obsolescence problems at the same time. In addition, it takes advantage of the new technologies providing higher data densities at lower costs. Today, the tremendous increase in data volume has triggered many developments in storage hardware and storage management systems. New approaches are being introduced at the research and product levels. But being able to read the bits correctly in the future is only one part of the solution.

Another challenge concerns the interpretability of files created long ago. A sequence of bits is meaningless if it cannot be decoded and transformed into an intelligible representation. Since, the decoding and transformation are generally specified through a computer program, the loss of that program translates into the loss of valuable information. In the past – and present – the industry managed to avoid disasters (in most cases) by periodically converting data to newer formats and rewriting programs. But such conversion methods come with many dangers and they will not be able to support the huge increase in multimedia data that applications will want in the future.

Alternatives have been proposed but none has been implemented in full-fledged systems.

For example, a lot of attention has been given to the use of XML as a technology-independent format. The idea is to extract from the original format all data elements and to tag them appropriately. The complete information is then encoded in an XML string. The argument is that the XML string will remain readable in the future. It must be noted that identifying the data elements is not sufficient. The semantic information on what to do with each element must still be specified as metadata.[1] However, the presence of tags facilitates the association of semantic information with the data elements. A drawback of the technique is that XML may not be the most appropriate storage format for certain document types, since it brings with it a potentially huge increase in file size.

On the other hand of the spectrum, the emulation of old machines makes it possible to run old software on new machines. This may be both good and bad: good because it will run the old program and present the data as seen in the past; bad because if forces the user to go through old interfaces and does not allow for reusing the data in different ways, making re-purposing all but impossible.

This paper refers to a different approach, using a Universal Virtual Computer (UVC) to specify processes that have to run in the future. The UVC and its architecture were initially described in Lorie [2001] and Lorie [2002a]. More recently, Lorie and van Diessen [2005] published the first complete (and much improved) UVC specification and named it the UVC Convention, version 0. That version had been fully implemented in an earlier joint study between IBM and the KB, as described in Lorie [2002b]. The extended UVC Convention, version 0.5, developed here was evaluated using a UVC implementation, too.

[1] Many of the current applications provide some form of functionality to translate their internal proprietary formats into an XML format. However, the tags and data elements are often cryptic and still heavily dependent on the semantics encoded in the application.

B.2. The Universal Virtual Computer (UVC) solution.

A virtual computer can be used to specify today processes that will need to be executed on a future, still unknown, real machine. The only requirement is that a UVC emulator be always available (a relatively simple task compared to an emulator of a real machine.)

When a file of a certain format F is archived, a program P is also archived which can decode the data and return the information to a future client, according to a logical view (a la XML). The definition of the logical view can be archived in a similar way. The novelty is that P is written for the UVC. In the future, the only thing required is an emulator of the UVC, which will be able to run the program P and return all data in an easy to understand, logical view. The UVC is a general purpose computer, complete yet basic enough as to remain relevant for a very long time. Given the simplicity of the UVC, writing a program that emulates it is reasonably easy.

This method provides all the advantages of XML. But it has the additional - and very significant - advantage of keeping the data in its original form instead of forcing an XML representation for the stored document. In principle, keeping the original format should always be possible. For example, for PDF it is possible to write a decoder of the PDF format. But it may sometimes be simpler to immediately convert the original format, once, into another one, and then preserving that representation as if it were the original. The choice of an "intermediate format" must be guided by a good balance between efficiency of storage and decoding complexity. The same logical view can often be used for a multitude of internal formats (for example, one logical view would support a color image independently of its original format). This greatly simplifies the writing of Restore Applications in the future.

B.3. System components

The design goal for the UVC was not to define a minimal general-purpose computer. Instead, the idea was to develop an intuitive computer with powerful and flexible instructions for handling bit strings, and to take advantage of the fact that it is virtual and that performance is of secondary importance for future access on machines which (according to Moore's law) will probably be much more powerful than contemporary ones. Still, performance issues are a big concerns now, because all UVC applications have to be implemented (and tested thoroughly!) while the format to be supported is still in use. Therefore, all parts of the specification must allow for efficient implementation. The architecture relies on concepts that have existed since the beginning of the computer era: memory, registers, basic instructions. It also tries to be natural; for example, a negative number is a positive number with a sign (no 2-complement); there is no notion of byte, simply bits that can be used at will.

For ease of explanation, we consider the various components shown in Fig. B.1. The UVC emulator and the new Restore Application are both written in the future. The UVC being emulated is composed of a CPU that executes instructions, a status (a conditional flag and an instruction pointer), and a segmented memory. All of this is part of the UVC Convention and will not change in time.[2] The UVC program is a sequence of UVC instructions in machine

[2]It would be unrealistic to claim that the proposed UVC will never change. The version being described here

language, with very little glue that holds the various pieces together (this is also part of the Convention). The instruction set includes an Input/Output facility to exchange data between the UVC emulator and the Restore Application (through explicit I/O instructions in the UVC program).

The next sections go through these components in detail.

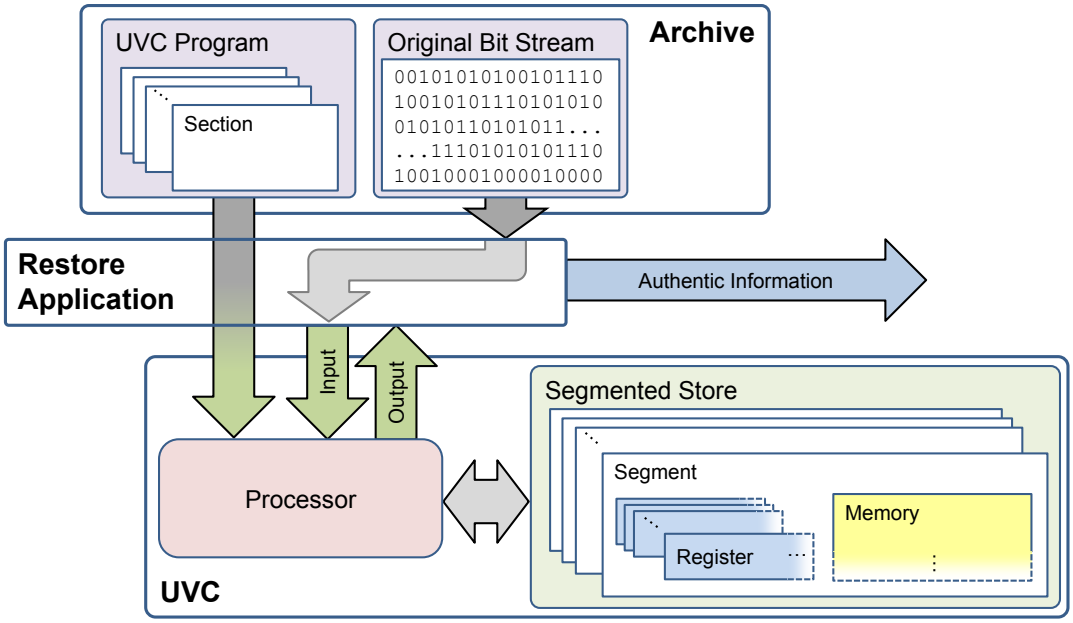


Figure B.1.: Global Architecture

B.3.1. Memory model

The memory model is that of a segmented store. A segment contains an arbitrarily large set of registers and a bit-addressable sequential memory. A register contains a value represented by a bit string. That bit string may be interpreted either as a value or as a pointer to a particular bit of the sequential memory (actually the displacement from the beginning of the segment). There is an unlimited number of registers and the bit string of a register can be of unlimited length. An integer value stored in a register always only occupies as many bits as necessary. That implies the leftmost bit to be 1; the value zero is represented through an empty bit string and is positive. To allow for signed values, registers have to maintain both the sign and the length internally. The only operations on the sequential memory of a segment are designed to move information from the memory to registers and vice-versa or to communicate with the external world. Note the bit – rather than byte or word – orientation; this is very convenient for handling bit streams and it increases independence from real machine architectures. A segment is uniquely identified by a segment number and its bits can be accessed at random.

is Version 0.5. Any archived document will always contain such a UVC version number to accommodate for rare modifications.

Segments and Sections

A UVC program is composed of sections that interact between themselves. Each section is stored in a segment. An individual section can address all segments that are in its address space by a segment number (conversely, the segments belong to the section). During an execution, the UVC emulator will assign these numbers to segments and will keep track of the mapping. The address space of each section always shares some segments with other sections. An arbitrary number of segments containing variables and data can be part of the address space, too. If a section is called recursively, the code is stored only once, but each instantiation will have its own address space. The different types of segments are now explained.

Segment 0 Segment 0 is accessible by any section (it belongs to all address spaces). Its purpose is to store a collection of shared constants and variables.

Segment 1 For each section, a unique segment is accessible by the number 1. If the section is invoked recursively, segment 1 can be addressed by all instantiations in the same way. Multiple activations of the same section can share information via this segment.

Segment 2 There is one such segment per address space; its function is to handle data exchange during invocations (calls) among sections. If a section A invokes a section B, one segment of A's address space will be used to transfer parameters and results. B will see the parameters and store the results in segment 2. (The UVC will adjust the mapping, avoiding a need to copy the results from one segment to another).

Segment 3 The code of each section is stored in a segment. Initially, a section can access the segment that stores its own code by the segment number 3. One instruction allows for adjusting the mapping during program execution.

Segments 4 to 999 Segments in that range are shared. They belong to the address spaces of all sections. For example, if two sections refer to a segment 4, both see the same segment 4.

Segments >= 1000 Segments in that range are private. If two sections refer to segment 1000, they see different segments. When a section is invoked multiple times, each invocation will see its own instance of a segment in that range. This implies that all information stored in assigned private segments is lost as soon as the invocation has returned.

Initial values

The sequential memory will be initialized at the first access. All bits are initially set to 0. A register will be initialized by the first access too. Its value is initially set to zero.

B.3.2. Status

The status of the UVC is represented by a condition flag and an instruction pointer. If the execution stops, an error number identifies the cause.

The condition flag reflects the result of the last comparison operation and is initially cleared. Other instructions, i.e. section invocation, must not change that flag.

The instruction pointer refers to the next instruction. This pointer is updated by the CPU each time an instruction completes. The pointer is composed of the segment containing the section, and the bit-displacement of the instruction inside that segment. Most instructions update the pointer to the address of the bit that immediately follows it in the sequential memory. A few instructions modify the pointer in a different way, altering the sequential flow.

The error number is represented by a bit string identifying the cause of program termination. The possible values are defined as follows:

error number	cause	description
0	Program finished	Program was stopped by a stop instruction.
1	Program abort	Program was stopped by a break instruction.
2	Program load error	An error occurs during interpretation of the bit stream of the UVC application.
3	Out of memory	The UVC implementation fails executing an instruction by reason of a lack of memory.
4	Input error	The execution of an input instruction failed.
5	Division by zero	Division with 0 as divisor.
6	Wrong opcode	The opcode of the current instruction is not a legal opcode.
7	Wrong section number	The invocation of the section with the given number failed.

B.3.3. Instruction formats and operand types

Every instruction is identified by an 8-bit operation code (opcode) followed by up to three operands. The opcode defines which combination of operands will be accepted. Five different kinds of operands are supported:

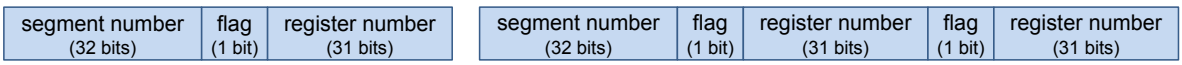
Constants A constant occupies 32 bits and is always unsigned. Some instructions will consider constants to be negative, i.e., a negative sign is inferred from the opcode.

Registers A 64-bit string designates a register: the first 32 bits identify a segment; the next 32 bits consist of an indirection flag and a 31-bit value identifying a register in that segment. If the indirection flag is set, the value of the identified register is interpreted as a register number again. The register thus identified belongs to the same segment and will be used as an operand.

VarRegs A VarReg allows register to be accessed dynamically; it is designated by a 96-bit string: the first 64 bits identify a register as described above. The contents of that (possibly indirectly accessed) register identifies a segment *S* of the current address space. The next 32 bits also consist of an indirection flag and a value that identifies a register in segment *S*. Again, indirection can be used.

Addresses An address allows memory to be accessed statically; it is designated by a 64-bit string: the first 32 bits identify a segment; the next 32 bits are taken as an offset. Together they address one bit position within a segment that belongs to the current address space.

Bit strings Some instructions handle bit strings with an individual, fixed length. The length is always defined by a constant operand preceding the operand which defines the bit string.



(a) Register (b) VarReg

Figure B.2.: Bit Encodings for Register and VarReg Operands

B.3.4. Instruction set

The UVC instructions are shown in the following sections, arranged in meaningful groups. The binary values corresponding to each opcode are shown in the summary table in Appendix B.7.

A general note on bit order semantics and operand specifications

In a register, the rightmost bit is the least significant. When the register length is automatically updated as the result of an operation, leftmost bits appear or disappear (the length is updated accordingly).

The contents of registers must obey to the following rules: The leftmost bit is always set. A zero is represented by an empty bit string. The register contents representing a zero is positive.

The possible kinds of operands are denoted in the following paragraphs as follows: `F` stands for a 32-bit constant, `C` stands for a variable length bit string, `R` stands for a register, `V` stands for a VarReg, and `A` stands for an address. The Letter `S` followed by an `f` or a `c` means that the constant values may have a leading sign. Since constants are unsigned, the opcode implicitly includes the sign information.

For example: `op2RSf` indicates the second operand of an instruction may be a register or a signed constant.

The instruction definitions below use mnemonic opcodes which have to be expanded into concrete opcodes indicating the types and the signs of the operands. For details, see the opcode table in Appendix B.7.

Instructions to move data between the memory and the registers

In all Load and Store operations, the sign of all operands are ignored. By construction, the results of Load instructions are non-negative.

- **Load from memory to register**

LOAD	op1R op2RVA op3RF
	Let *Seg* be the segment op2 belongs to. Starting at address in op2 extract from *Seg* a bit string *S* of length *k* (defined by op3). Discard all leading 0s from *S* and insert result in op1.

As indicated in (Fig. B.3) below, the rule regarding register contents is respected.

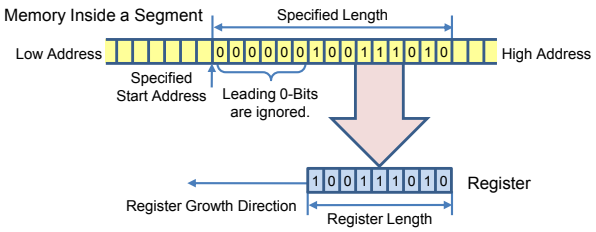


Figure B.3.: Loading Bits from Memory into Registers.

- **Load sign information from memory to register**

LSIGN	op1R op2RVA
	Let *Seg* be the segment op2 belongs to. If the bit at address in op2 in *Seg* is set to 1 and the value in op1 is not zero, op1 will be negative, otherwise positive.

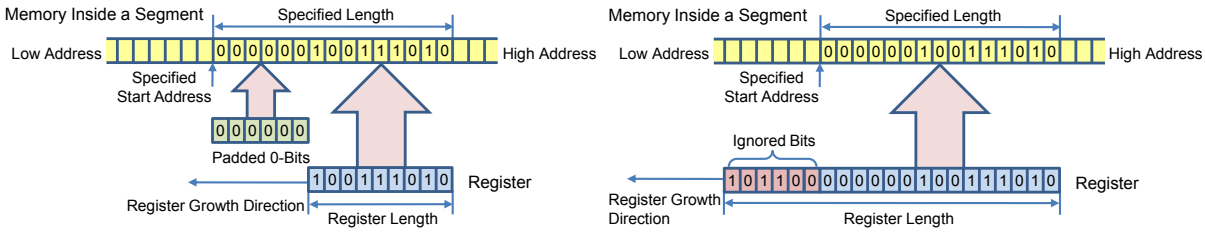
- **Store bits from register into memory**

STORE	op1R op2RVA op3RF
	Let *Seg* be the segment op2 belongs to. Store the rightmost *k* bits from op1 into *Seg* at address in op2; the length *k* is in op3.

If *k* is greater than the amount of bits in op1, leading 0-bits are stored (Fig. B.4(a)).

- **Store bit strings into memory**

STORE	op1RVA op2F op3C
	Let *Seg* be the segment op1 belongs to. Store the bit string in op3 with the length in op2 into *Seg*, starting at address in op1. The bit string can have leading 0-bits.



(a) Register length < specified length

(b) Register length > specified length

Figure B.4.: Storing Bits from Registers into Memory.

- **Store sign information into memory**

SSIGN	op1R op2RVA If op1 is negative, the bit at address in op2 is set to 1, otherwise to 0. The accessed segment is the same as the register or address given in op2 belongs to.

Operations on registers only

- **Copy register contents**

COPY	op1RV op2RV Copy the contents of op2 into op1. After the operation, the contents of the registers are identical.

- **Manipulating the sign**

PSIGN	op1R Set sign to positive.
NSIGN	op1R Set sign to negative, if the value stored in op1 is not zero.

- **Loading bit strings into registers**

LOADC	op1RV op2F op3Sc Inserts into op1 the bit string in op3 with the length k in op2. The length of the register is set to k. Empty bit strings are allowed. Since the rule of register contents must not be violated, bit strings starting with 0 are illegal in op3.

- **Access to the length of registers**

RLEN	op1R op2R Store the length of op2 into op1. After operation op1 is not negative.

Numeric instructions

These instructions may cause the lengths of registers to change to accommodate the result. The sign is set to the result sign; if the result is zero, the sign is positive. All numeric instructions are performed according to the laws of signed integer arithmetic.

- **Addition**

ADD	op1R op2RF The sum of the values in op1 and op2 is computed and stored in op1.

- **Subtraction**

SUB	op1R op2RF The value in op2 is subtracted from the value in op1 and the result is stored in op1.

- **Multiply**

MUL	op1R op2RSf The product of the values in op1 and op2 is computed and stored in op1.

- **Divide**

DIV	op1R op2RSf op3R The division of the value in op1 by the value in op2 is computed; the quotient is stored in op1. The remainder is stored in op3 and the sign of op3 is set to the original sign of op1. If both op1 and op3 denote the same register *R*, *R* will contain the remainder. A division by zero aborts program execution.

- **Shifts**

  All instructions ignore the sign of the first operand.

SHFTR	op1R op2RF Let *k* be the number value of op2. If $k \geq 0$, then the rightmost *k* bits in op1 are removed; empty bit strings remain empty. If $k \leq$, and the bit string in op1 is not empty, *k* 0-bits are attached to the right hand side of op1.
	op1R op2RF op3R Let *k* be the number value of op2. If $k \geq 0$, then the rightmost *k* bits in op1 are removed; empty bit strings remain empty. Removed bits are stored in original order in op3. If $k \leq$, and the bit string in op1 is not empty, *k* 0-bits are attached to the right hand side of op1. The value of op3 is zero.
SHFTL	op1R op2RF SHFTL op *k* has the same effect as SHFTR op *-k*.

Comparison instructions

- **Greater than**

GRT	op1RSf op2RSf Set the condition flag if the value in op1 is larger than the value in op2, respecting the signs. Otherwise the flag is cleared. At least one operand must be a register.

- **Equal to**

EQU	op1RSf op2RSf Set the condition flag if the value in op1 is equal to the value in op2, respecting the signs. Otherwise the flag is cleared. At least one operand must be a register.

- **Test if zero**

TESTZ	op1R Set the condition flag if the value in op1 is zero. Otherwise the flag is cleared.

- **Test if negative**

TESTN	op1R Set the condition flag if the value in op1 is negative. Otherwise the flag is cleared.

Logical instructions

According to the rule of register contents the results will not have leading 0-bits. Signs are not affected at all.

- **Or**

OR	op1R op2RF The bits in op1 and op2 are or'ed bit by bit and the result is stored in op1.

- **And**

OR	op1R op2RF The bits in op1 and op2 are and'ed bit by bit and the result is stored in op1.

Instructions that alter the flow of execution

- **Jump**

JUMP	op1RF Set the instruction pointer to the address in op1 in the same section.

- **Jump conditionally**

JUMPC	op1RF If the condition flag is set, a JUMP op1RF is performed; otherwise, execution proceeds with the next instruction.
JUMPNC	op1RF If the condition flag is not set, a JUMP op1RF is performed; otherwise, execution proceeds with the next instruction.

- **Call a section (subroutine)**

CALL	op1RF op2RF op3RF Invoke code of the section identified by op1at the starting address given in op2 with parameters provided in the segment given in op3. The invoked and the current section may be the same. Initially, sections will be assigned to its numbers by loading the UVC application. The use of unassigned numbers will abort program execution.

- **Return to calling section**

BREAK	(no operand) Return control to the calling section at instruction following the call. If there is no calling section, program execution aborts.

- **Stop execution**

STOP	(no operand) Stops program execution with error number 0 to signal normal program termination.

Accessing the section code

Segment 3 contains the code of a section. Initially, the section's own code is mapped onto that segment. If the code of another or of a new section is to be accessed, the mapping must be adjusted. Therefore, the section numbers provided by the application's bit stream can be used. Note, that section numbers are not segment numbers: E.g., the code of section number 4 is never found in segment number 4.

- **Mapping the code of a section**

MAP	op1RF Maps the code of a section to segment number 3. The section number in op1 refers to the section numbers provided by the application's bit stream. If there is no corresponding section, a new segment will be created and, henceforth, will be accessible via Call instructions.

Communication with the outside world (I/O)

The communication makes use of a simple channel abstraction. Any "message" of data traveling on a channel consists of three components:

1. Message Type (a 32-bit unsigned integer)
   The types and their semantics are chosen by the application. They identify the different roles of the data being transferred. It may be a tag for a piece of data or simply a code that is used for synchronization between the UVC program and the application.

2. Message Length (a 32-bit unsigned integer)
   It is the length of the data being transferred (in bits).

3. Message Body
   It is the actual bit string to be transferred. Its length is given by message length. If the message length is zero, the message body is empty.

There are five UVC instructions to use the communication channel:

- **Receive input**

IN	op1R op2R op3RVA op1 will be set to the message type received. op2 will be set to the length of the message. op3 specifies the starting memory address where the data will be stored. The accessed segment is the same as the register or the address given in op3 belongs to. The sign of op3 is ignored.
INR	op1R op2R op3R op1 will be set to the message type received. op2 will be set to the length of the message. op3 will contain the bit string of the message without leading 0-bits.

After operation, all results stored in registers will be non-negative. If the message length is zero, memory will not be accessed; in case of an INR the register contents of op3 will be set to zero.

- **Send output**

OUT	op1RF op2RF op3RVA op1 contains the message type. op2 contains the length of the data to be transferred. op3 contains the starting memory address where the data resides. The accessed segment is the same as the one the register or the address given in op3 belong to. If the message length is zero, the specified memory address is ignored.
OUTR	op1RF op2RF op3R op1 contains the message type. op2 contains the length *k* of the data to be transferred. op3 contains the bit string. Leading 0-bits will be transferred first if the bit string stored in op3 is not as long as specified in op2. If the bit string is longer, only the *k* rightmost bits are transferred.
OUTC	op1RF op2RF op3F op1 contains the message type. op2 contains the length *k* of the data to be transferred. op3 contains the bit string. Leading 0-bits will be transferred first if the bit string stored in op3 is not as long as specified in op2. If the bit string is longer, only the *k* rightmost bits are transferred.

The UVC convention does not impose any additional requirement on the use and/or specific implementation of the channel. For instance, if the piece of data to be transferred is larger than the maximum allowed, it can be split into multiple messages; the exchange can be controlled by introducing message types such as message start, message continuation and message end. Similarly, synchronization between the UVC emulator and the outside application can be established by sending specific user defined message types with no data. Figure B.5 illustrates the only two valid communication patterns.

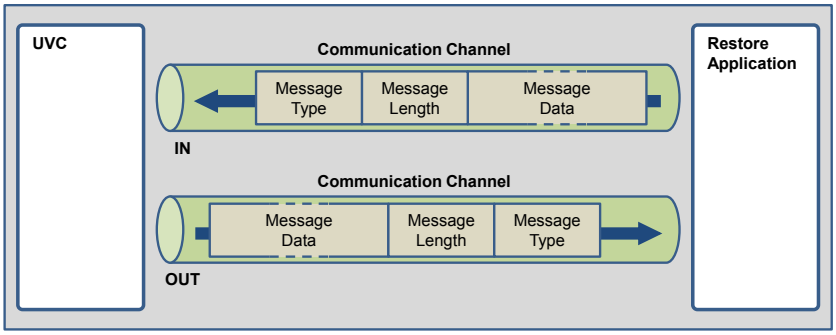


Figure B.5.: Valid Communication Patterns

An invalid communication pattern will lead to program abortion. As mentioned before, the UVC specification does not enforce any specific implementation of the channel: actual transfer can occur when both sides are executing their respective In or Out operations or the message can be temporarily buffered.

Output instructions for debugging

There are three debug instructions that every UVC must be able to handle, because such instructions can be part of the code. This implies knowledge of how to step over these instructions. The full implementation of the functionality is not enforced. If implemented the resulting outputs must not disturb the output channel.

PRINTR	op1RV Prints out the length, the sign and the stored bit string of op1.
PRINTM	op1RVA op2RF Prints out the bit string stored at address in op1 with the length in op2. The accessed segment is the same as the register or the address given in op1 belong to. Signs are ignored.
PRINTC	op1F op2C Prints out the bit string in op2 with the lenght in op1. If the encoding used by the application is known, the bits can be interpreted accordingly.

B.3.5. Organization of the archived module

A program is built out of multiple sections that call each other. A Call instruction refer to a section by its section number. It is illegal to use the same section number for different sections. Though the code of every section is stored in a segment, section numbers are not typed like segment numbers. This means that the bit code of section number 4 is never stored in a shared segment. The bit code of a section is always stored in private segments. Sections can access this segments by previously mapping them into their own address space.

An archived module has the following hierarchical structure:

```
<Module>      ::= <Nsections><Sections><Padding>
<Nsections>   ::= <32bit>
<Sections>    ::= <Section><Sections>
<Section>     ::= <SecNumber><Length><Code>
<SecNumber>   ::= <32bit>
<Code>        ::= <Bitstring>
<Length>      ::= <32bit>

<Bitstring>   ::= <bit><Bitstring> |
<32bit>       ::= <bit><bit><bit><bit><bit><bit><bit><bit>
                  <bit><bit><bit><bit><bit><bit><bit><bit>
                  <bit><bit><bit><bit><bit><bit><bit><bit>
                  <bit><bit><bit><bit><bit><bit><bit><bit>
<bit>         ::= 0 | 1
<Padding>     ::= 0<Padding> |
```

Nsections (a 32-bit integer) is the number k of program sections to be loaded, followed by k structures of type **Section**.

A structure of type **Section** is the concatenation of a section number (a 32-bit integer), the length of the code in bits (a 32-bit integer), and the code itself. By definition, the first section in the archived module is the starting section of the program. If the sum of all given

lengths exceeds the length of the module, a program load error occurs. For compatibility with different file systems any attachment of 0-bits must disturb neither the program behavior nor the loading process. Such a padding can be used to place the last bits left-justified into a word, regardless of its length.

B.4. Proposed syntax for UVC assembly language

This section specifies the proposed syntax of the UVC assembly language. The syntax is implemented in an assembler application for the UVC. Therefore, it is possible to assemble newly written applications as well as example and test programs that accompany this specification.

The assembler is designed to relieve programmers from mundane tasks, like the calculation of the bit string length for a LOADC instruction in order to prevent leading zeros. Also, the assembler enables signs for instructions without matching opcodes, e.g. ADD 1,0 -5. Obviously, opcodes of inverse instructions will be used in that case.

```
<Section>          ::= <Name><Delim><SecNumber><Delim><Code>
<Name>             ::= <Letter><LetterorNumbers>
<SecNumber>        ::= <Number>
<Code>             ::= <Instruction><Delim><Code> |
<Instruction>      ::= LOADC<S><opRV><opS> |
                       ADD<S><opR><S><opRS> |
                       SUB<S><opR><S><opRS> |
                       MUL<S><opR><S><opRS> |
                       DIV<S><opR><S><opRS><S><opR> |
                       SHFTL<S><opR><S><opRS> |
                       SHFTR<S><opR><S><opRS> |
                       SHFTR<S><opR><S><opRS><S><opR> |
                       AND<S><opR><S><opRC> |
                       OR<S><opR><S><opRC> |
                       GRT<S><opR><S><opRS> |
                       GRT<S><opS><S><opR> |
                       EQU<S><opR><S><opRS> |
                       EQU<S><opS><S><opR> |
                       TESTZ<S><opR> |
                       TESTN<S><opR> |
                       JUMP<S><opRN> |
                       JUMPC<S><opRN> |
                       JUMPNC<S><opRN> |
                       CALL<S><opRC><S><opRC><S><opRC> |
                       BREAK |
                       MAP<S><opRC><S> |
                       STOP |
                       COPY<S><opRV><S><opRV> |
                       LOAD<S><opR><S><opRVA><S><opRC> |
                       STORE<S><opR><S><opRVA><S><opRC> |
                       STORE<S><opRVA><S><opC><S><opC> |
                       LSIGN<S><opR><S><opRVA><S> |
                       SSIGN<S><opR><S><opRVA><S> |
                       RLEN<S><opR><S><opR> |
                       IN<S><opR><S><opR><S><opRVA> |
```

```
                    OUT<S><opRC><S><opRC><S><opRVA> |
                    INR<S><opR><S><opR><S><opR> |
                    OUTR<S><opRC><S><opRC><S><opR> |
                    OUTC<S><opRC><S><opRC><S><opC> |
                    NSIGN<S><opR> |
                    PSIGN<S><opR> |
                    MAP<S><opRC> |
                    PRINTR<S><opRV> |
                    PRINTM<S><opRVA><S><opRC> |
                    PRINTC<S>"<AnyChars>" |
                    label:<S><Name>

<opR>              ::= <Number>,<Number> | <Number>,<Number>*
<opA>              ::= <Number>;<Number> | <Number>;0x<Hexnumber>
<opC>              ::= <Number> | 0x<Hexnumber>
<opS>              ::= <opC> | -<opC>
<opV>              ::= (<opR>),<Number> | (<opR>),<Number>*
<opRS>             ::= <opR> | <opS>
<opRC>             ::= <opR> | <opC>
<opRV>             ::= <opR> | <opV>
<opRVA>            ::= <opR> | <opV> | <opA>
<opRN>             ::= <opR> | <Name>

<LetterorNumbers>  ::= <LetterorNumber><LetterorNumbers> |
<LetterorNumber>   ::= <Letter> | <Digit>
<Letter>           ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
                       N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
                       a | b | c | d | e | f | g | h | i | j | k | l | m |
                       n | o | p | q | r | s | t | u | v | w | x | y | z
<Number>           ::= <Digit><Number> | <Digit>
<Digit>            ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Hexnumber>        ::= <Hexdigit><Hexnumber> | <Hexdigit>
<Hexdigit>         ::= <Digit> | A | B | C | D | E | F
<Delim>            ::= <S><Delim> | <NewLine><Delim> |
<NewLine>          ::= [newLine] | <Comment>[newLine]
<Comment>          ::= #<AnyChars>
<AnyChars>         ::= <AnyChar><AnyChars> |
<AnyChar>          ::= <LetterorNumber> | . | , | ; | ! | ? | <S> | - | +
<S>                ::= [space]<S> | [tabulator]<S> |
```

B.5. An example program

In many UVC applications, the function of the UVC program will be to decode the internal format of an input and return the results according to a predefined logical view. In more general cases, the UVC program can also implement any arbitrary logic, using the input. The following program is very simple but still illustrates the more general case. Instead of decoding a format, it generates the factorial of the input by (recursive) computation. The input is treated as an unsigned number. The output will be an unsigned number, too.

```
Main
1000                # section number for Main section
#############################################################################
# Program: Main, to compute the factorial of a given number.
# This program computes the factorial of a value received over the
# communication channel. It outputs the result as binary value.
#
# By convention, the argument to the factorial section is loaded in
# register 1 of the segment containing the argument. The result is
# communicated back in register 2 of the same segment.
#
# The section uses segment 1001 as parameter segment.
#############################################################################

# Get argument, ignoring type and size
INR    1001,2   1001,3   1001,1    # in 1001,1 is the input argument

# Call the factorial section
CALL     1010        0     1001

RLEN   1001,3   1001,2             # store length of result in 1001,3
OUTR        3   1001,3   1001,2    # message type = 3

STOP


Factorial
1010                # section number for Factorial section
#############################################################################
# Computes the factorial of the value given in register 2,1
# The result is placed in register 2,2
#############################################################################

GRT       2,1        1           # if the given value is > 1
JUMPC  calculate                 # do calculation

LOADC     2,2        1           # otherwise return 1 in 2,2
BREAK

# Calculate the factorial using the result of the decremented argument
label: calculate
COPY   1001,1      2,1           # copy for independent use
COPY   1002,1      2,1           # 2 times
SUB    1001,1        1           # decrement
CALL     1010        0     1001  # let calculate n-1

MUL    1002,1   1001,2           # calculate n using n-1

COPY      2,2   1002,1           # return the result in 2,2
BREAK
```

B.6. Test suite

The aim of this section is to make sure that the test suite that accompanies this specification is not missed. Besides a lot of test cases, the test suite comprises a manual describing in detail how to use the test cases and how to interpret the attached metadata.

The test suite is geared towards to a programmer who wants to implement a UVC. Nevertheless, the suite is helpful for engineering new UVC applications, too: The source code contains best practice patterns that have been tested and, therefore are guaranteed to work. In many cases, software engineers will find implicit guidance on how to implement efficient applications.

B.6.1. Design goals

The overall design goal is to avoid programmer effort and to quickly detect errors. The test cases are arranged in a predefined order. Every test assumes that all previous tests did complete without error.

All test cases are designed to be executed automatically. Therefore, a simple test program can execute all test applications and analyze the results based on the provided metadata.

Finally, the test applications together with their expected results may help to clarify the meaning of not sufficiently precise wordings in the specification.

B.6.2. Effectively use

To use the test suite effectively, all tests have to be executed in the predefined order, up to the first error. After correcting an error or some other modification, all tests have to be rerun to ensure that the modification has not introduced any new errors. To use the test suite effectively, the steps of the IVC implementation should start as follows: First, ensure that a single-section application can be loaded; then add a few basic instructions to the processor; then add the rest.

Every test case consists of a UVC application and metadata which describe the expected input and the output. A lot of test cases are accompanied by traces. These traces are best described as log files, which contain for every executed instruction data, like opcodes, operands and the contents of every register involved. Together with the predefined order and the attached metadata, these traces will help to detect the cause of the error easily.

B.6.3. Ongoing effort

The test suite cannot guarantee the correctness of a UVC implementation. Nevertheless, it helps to detect errors. The quality of the test suite depends on the number and the conclusiveness of the test cases and their attached metadata. In order to detect errors which currently are not uncovered, software engineers are invited to contribute more test cases.

B.7. Opcode table

Opcode	No.	Operands				Function
LOADC	1	Reg (Dest.)	Length	bit string...		Load constant
	2	Reg (Dest.)	Length	bit string...		Load negative constant
	3	VarReg (Dest.)		Length	bit string...	Load constant
	4	VarReg (Dest.)		Length	bit string...	Load negative constant
ADD	10	Reg (Dest.)	Reg (Arg.)			Add
	11	Reg (Dest.)	Const.			Add
SUB	12	Reg (Dest.)	Reg (Arg.)			Subtract
	13	Reg (Dest.)	Const.			Subtract
MUL	14	Reg (Dest.)	Reg (Arg.)			Multiply
	15	Reg (Dest.)	Const.			Multiply
	16	Reg (Dest.)	Const.			Multiply (negative)
DIV	17	Reg (Dest.)	Reg (Arg.)	Reg (Remainder)		Divide
	15	Reg (Dest.)	Const.	Reg (Remainder)		Divide
	16	Reg (Dest.)	Const.	Reg (Remainder)		Divide (negative)
SHFTL	20	Reg (Dest.)	Reg (Arg.)			Shift left
	21	Reg (Dest.)	Const.			Shift left
SHFTR	22	Reg (Dest.)	Reg (Arg.)	Reg (Remainder)		Shift right
	23	Reg (Dest.)	Const.	Reg (Remainder)		Shift right
	24	Reg (Dest.)	Reg (Arg.)			Shift right
	25	Reg (Dest.)	Const.			Shift right
GRT	30	Reg (Arg.)	Reg (Arg.)			Greater than
	31	Reg (Arg.)	Const.			Greater than
	32	Reg (Arg.)	Const.			Greater than (negative)
	33	Const.	Reg (Arg.)			Greater than
	34	Const.	Reg (Arg.)			Greater than (negative)
EQU	35	Reg (Arg.)	Reg (Arg.)			Equal
	36	Reg (Arg.)	Const.			Equal
	37	Reg (Arg.)	Const.			Equal (negative)
	38	Const.	Reg (Arg.)			Equal
	39	Const.	Reg (Arg.)			Equal (negative)
TESTZ	40	Reg (Arg.)				Test if zero
TESTN	41	Reg (Arg.)				Test if negative
JUMP	50	Reg (Arg.)				Jump
	51	Const.				Jump
JUMPC	52	Reg (Arg.)				Jump on condition
	53	Const.				Jump on condition
JUMPNC	54	Reg (Arg.)				Jump on inverse condition
	55	Const.				Jump on inverse condition
CALL	56	Reg (Section)	Reg (Address)	Reg (Param.)		Call a section
	57	Reg (Section)	Reg (Address)	Param		Call a section
	58	Reg (Section)	Address	Reg (Param.)		Call a section
	59	Reg (Section)	Address	Param.		Call a section
	60	Section	Reg (Address)	Reg (Param.)		Call a section
	61	Section	Reg (Address)	Param		Call a section
	62	Section	Address	Reg (Param.)		Call a section
	63	Section	Address	Param.		Call a section
BREAK	64					Return to calling section
MAP	65	Reg (Arg.)				Map section code
	66	Const.				Map section code
AND	70	Reg (Dest.)	Reg (Arg.)			And
	71	Reg (Dest.)	Const.			And
OR	72	Reg (Dest.)	Reg (Arg.)			Or
	73	Reg (Dest.)	Const.			Or
COPY	80	Reg (Dest.)	Reg (Arg.)			Copy register
	81	VarReg (Dest.)		VarReg (Src.)		Copy register
	82	VarReg (Dest.)		Reg (Src.)		Copy register
	83	Reg (Dest.)	VarReg (Src.)			Copy register

Opcode	No.	Operands	Function
NSIGN	84	Reg (Dest.)	Set sign to negative
PSIGN	85	Reg (Dest.)	Set sign to positive
RLEN	86	Reg (Dest.) | Reg (Src.)	Load register length
LOAD	90	Reg (Dest.) | Reg (Address) | Reg (Length)	Load from memory
	91	Reg (Dest.) | Address | Reg (Length)	Load from memory
	92	Reg (Dest.) | VarReg (Address) | Reg (Length)	Load from memory
	93	Reg (Dest.) | Reg (Address) | Length	Load from memory
	94	Reg (Dest.) | Address | Length	Load from memory
	95	Reg (Dest.) | VarReg (Address) | Length	Load from memory
STORE	96	Reg (Src.) | Reg (Address) | Reg (Length)	Store into memory
	97	Reg (Src.) | Address | Reg (Length)	Store into memory
	98	Reg (Src.) | VarReg (Address) | Reg (Length)	Store into memory
	99	Reg (Src.) | Reg (Address) | Length	Store into memory
	100	Reg (Src.) | Address | Length	Store into memory
	101	Reg (Src.) | VarReg (Address) | Length	Store into memory
	102	Reg (Address) | Length | bit string...	Store into memory
	103	Address | Length | bit string...	Store into memory
	104	VarReg (Address) | Length | bit string...	Store into memory
LSIGN	105	Reg (Dest.) | Reg (Address)	Load sign from memory
	106	Reg (Dest.) | Address	Load sign from memory
	107	Reg (Dest.) | VarReg (Address)	Load sign from memory
SSIGN	108	Reg (Src.) | Reg (Address)	Store sign into memory
	109	Reg (Src.) | Address	Store sign into memory
	110	Reg (Src.) | VarReg (Address)	Store sign into memory
IN	100	Reg (Message type) | Reg (Length) | Reg (Address)	Input into memory
	101	Reg (Message type) | Reg (Length) | Address	Input into memory
	102	Reg (Message type) | Reg (Length) | VarReg (Address)	Input into memory
OUT	103	Reg (Message type) | Reg (Length) | Reg (Address)	Output from memory
	104	Type | Reg (Length) | Reg (Address)	Output from memory
	105	Reg (Message type) | Length | Reg (Address)	Output from memory
	106	Reg (Message type) | Reg (Length) | Address	Output from memory
	107	Reg (Message type) | Reg (Length) | VarReg (Address)	Output from memory
	108	Type | Length | Reg (Address)	Output from memory
	109	Type | Reg (Length) | Address	Output from memory
	110	Type | Reg (Length) | VarReg (Address)	Output from memory
	111	Reg (Message type) | Length | Address	Output from memory
	112	Reg (Message type) | Length | VarReg (Address)	Output from memory
	113	Type | Length | Address	Output from memory
	114	Type | Length | VarReg (Address)	Output from memory
INR	115	Reg (Message type) | Reg (Length) | Reg (Dest.)	Input in register
OUTR	116	Reg (Message type) | Reg (Length) | Reg (Src.)	Output from register
	117	Type | Reg (Length) | Reg (Src.)	Output from register
	118	Reg (Message type) | Length | Reg (Src.)	Output from register
	119	Type | Length | Reg (Src.)	Output from register
OUTC	120	Reg (Message type) | Reg (Length) | Const.	Output of constant value
	121	Reg (Message type) | Length | Const.	Output of constant value
	122	Type | Reg (Length) | Const.	Output of constant value
	123	Type | Length | Const.	Output of constant value
PRINTR	140	Reg (Src.)	Print register
	141	VarReg (Src.)	Print register
PRINTM	142	Reg (Address) | Reg (Length)	Print from memory
	143	Reg (Address) | Length	Print from memory
	144	Address | Reg (Length)	Print from memory
	145	Address | Length	Print from memory
	146	VarReg (Address) | Reg (Length)	Print from memory
	147	VarReg (Address) | Length	Print from memory
PRINTC	148	Length | bit string...	Print constant value
STOP	255		Stop execution

Literaturverzeichnis

(1968). *Control Data 6400/6500/6600 Extended Core Storage Systems, Reference Manual*. Control Data Corporation, St. Paul, Minnesota, USA, 60225100, Rev. A edition.

(1980). *Hardware Reference Manual: Cyber 170 Computer Systems*. Control Data Corporation, 4201 North Lexington Avenue, St. Pual, Minnesota, USA, 60420000, Rev. M edition.

(1983). *FORTRAN Extended Version 4, Reference Manual*. Control Data Corporation, P. O. Box 3492, Sunnyvale, California, USA, 60497800, Rev. J edition.

(1986). *CFT77 Fortran compiler for Cray supercomputers*. Cray Research, Inc., Minneapolis, MN 55402, USA, MP-1009 edition.

(1986). *FORTRAN Version 5, Reference Manual*. Control Data Corporation, P. O. Box 3492, Sunnyvale, California, USA, 60481300, Rev. J edition.

(1987). *Hardware Maintenance Manual: Central Processor and Central Memory*. Control Data Corporation, 4201 North Lexington Avenue, St. Pual, Minnesota, USA, 60458170, Rev. F edition.

(1988). *NOS Version 2 Reference Set, System Commands*. Control Data Corporation, St. Paul, Minnesota, USA, 60459680, Rev. K edition.

(1990). *Control Data 4000 Series, 4680 Computer Technical Reference*. Control Data Corporation, Minneaplois, MN, USA, 62940686, Rev. A edition.

(1991). *Control Data 4000 Series, FORTRAN Programmer's Guide and Language Reference Manual, Part 2*. Control Data Corporation, Minneaplois, MN, USA, 62940786, Rev. B edition.

(1992). *CF77 & SCC Features and Optimization*. Cray Research Inc., USA, TR-OPT, Rev. D edition.

(1993a). *Assembly Language Programmer's Guide*. Control Data Corporation, Santa Clara, CA, USA, 62940764, Rev. B edition.

(1993b). *FORTRAN Language Reference*. Control Data Corporation, Santa Clara, CA, USA, 62940786, Rev. C edition.

Adelson-Velsky, G. M. and Landis, E. M. (1962). An algorithm for the organization of information. In *Doklady Akademii Nauk SSSR 146*, pages 263–266.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Aho, A. V. and Ullman, J. D. (1972). *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Appel, A. and Palsberg, J. (2002). *Modern compiler implementation in Java*. Cambridge University Press.

Babbage, C. (1982). On the Mathematical Powers of the Calculating Engine. In *The Origins of Digital Computers , 3rd ed.*, pages 19–54, Berlin. Springer-Verlag.

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., and Woodger, M. (1963). Revised report on the algorithm language algol 60. *Commun. ACM*, 6(1):1–17.

Bala, V., Duesterwald, E., and Banerjia, S. (2000). Dynamo: A Transparent Dynamic Optimiziation System. In *PLDI'00 Proceedings of the ACM SIPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA. ACM.

Bayer, R. and McCreight, E. (1970). Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA. ACM.

Berry, G. and Sethi, R. (1986). From regular expressions to deterministic automata. *Theor. Comput. Sci.*, 48(1):117–126.

Böhm, C. and Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371.

Borghoff, U. M., Gasteiger, T., Schmalz, A., Weigele, P., and Siegert, H. (1987). MI - Eine Maschine für die Informatikausbildung. Technical report, Technische Universität München.

Borghoff, U. M., Rödig, P., Scheffczyk, J., and Schmitz, L. (2006). *Long-Term Preservation of Digital Documents: Principles and Practices*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Bowles, K. L. (1980). *Beginner's Manual for the UCSD Pascal System*. McGraw-Hill, Inc., New York, NY, USA.

Byers, F. R. (2003). *Care and handling of CDs and DVDs : A Guide for Librarians and Archivists*. Council on Library and Information Resources ; National Institute of Standards and Technology, Washington, D.C., USA.

CCSDS (2002). *Reference Model for an Open Archival Information System (OAIS). Blue book*. Consultative Committee for Space Data Systems (CCSDS).

Ceruzzi, P. E. (2003). *A History of Modern Computing*. MIT Press, Cambridge, MA, USA.

Chen, Z. (2000). *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, Boston, MA, USA.

Cifuentes, C. (1994). *Reverse Compilation Techniques.* PhD thesis, Queensland University of Technology.

Cook, S. A. (1966). *On the Minimum Computation Time of Functions. Doctoral Thesis*. PhD thesis, Harvard Univ., Cambridge, Mass.

Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*, chapter 10.2. McGraw-Hill Higher Education, 2nd edition.

Craig, I. D. (2006). *Virtual Machines*. Springer-Verlag, London, GB.

Dal Cin, M. (1996). *Rechnerarchitektur, Grundzüge des Aufbaus und der Organisation von Rechnerhardware*. Teubner.

Dalheimer, M. K. (1997). *Java Virtual Machine*. O'Reilly, Köln.

Dalley, J. (1991). The art of software testing. In *Aerospace and Electronics Conference, 1991. NAECON 1991., Proceedings of the IEEE 1991 National*, volume 2, pages 757–760.

Dinaburg, A., Royal, P., Sharif, M., and Lee, W. (2008). Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 51–62, New York, NY, USA. ACM.

Drexler, M., Sawyer, D., and Smith, G. (1990). CCSDS - An approach to the definition of common standards for understanding space-related data. In *AIAA and NASA, 2nd International Symposium on Space Information Systems*.

Folliot, B., Piumarta, I., and Riccardi, F. (1998). A dynamically configurable, multi-language execution platform. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, EW 8, pages 175–181, New York, NY, USA. ACM.

Folliot, B., Piumarta, I., Seinturier, L., Baillarguet, C., Khoury, C., Leger, A., and Ogel, F. (2002). Beyond Flexibility and Reflection: The Virtual Virtual Machine Approach. In *Proceedings of the NATO Advanced Research Workshop on Advanced Environments, Tools, and Applications for Cluster Computing-Revised Papers*, IWCC '01, pages 16–25, London, UK. Springer-Verlag.

Foster, J. M. (1968). A syntax improving program. *The Computer Journal*, 11(1):31–34.

Fuegi, J. and Francis, J. (2003). Lovelace Babbage and the creation of the 1843 'notes'. *Annals of the History of Computing, IEEE*, 25(4):16–26.

Funk, S. E. (2010). Emulation. In Neuroth, H., Oßwald, A., Scheffel, R., Strathmann, S., and Huth, K., editors, *nestor Handbuch: Eine kleine Enzyklopädie der digitalen Langzeitarchivierung*, pages 8:16–8:23, Göttingen. nestor - Kompetenznetzwerk Langzeitarchivierung.

Gladney, H. (2007). *Preserving Digital Information*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Gladney, H. M. and Lorie, R. A. (2005). Trustworthy 100-Year Digital Objects: Durable Encoding for When It's Too Late to Ask. *ACM Trans. Inf. Syst.*, 23:299–324.

Granger, S. (2000). Emulation as a Digital Preservation Strategy. *D-lib Magazine*, 6.

Guttenbrunner, M., Becker, C., Rauber, A., and Heister, C. (2008). Evaluating strategies for the preservation of console video games. In *Proceedings of The Fifth International Conference on Preservation of Digital Objects*, pages 115–121. Vortrag: iPRES 2008: The Fifth International Conference on Preservation of Digital Objects, The British Library, London, UK; 2008-09-29 – 2008-09-30.

Hapgood, B. (2001). Self-Modifying Code. In DeLoura, M., editor, *Game programming gems two*, Lecture Notes in Computer Science, pages 91–99. Charles River Media, Massachusetts.

Hedstrom, M. L. and Lampe, C. A. (2001). Emulation vs. Migration: Do Users Care? *RLG DigiNews*, 5.

Hedstrom, M. L., Lee, C. A., Olson, J. S., and Lampe, C. A. (2006). "The Old Version Flickers More": Digital Preservation from the User's Perspective. *American Archivist*, 69(1):159–187.

Heminger, A. R. and Robertson, S. (2000). The digital rosetta stone: a model for maintaining long-term access to static digital documents. *Commun. AIS*, 3.

Hoeven, J. V. D. and Wijngaarden, H. V. (2005). Modular emulation as a long-term preservation strategy for digital objects. In *5th International Web Archiving Workshop (IWAW05)*.

Holdsworth, D. (2001). C-ing ahead for digital longevity. In *CAMiLEON Project*. University of Leeds, UK.

Hopcroft, J. E. (2007a). *Introduction to Automata Theory, Languages, and Computation*, chapter 2.3.5. Pearson Addison Wesley.

Hopcroft, J. E. (2007b). *Introduction to Automata Theory, Languages, and Computation*, chapter 4.4.3. Pearson Addison Wesley.

Huth, K. (2010). Computermuseum. In Neuroth, H., Oßwald, A., Scheffel, R., Strathmann, S., and Huth, K., editors, *nestor Handbuch: Eine kleine Enzyklopädie der digitalen Langzeitarchivierung*, pages 8:24–8:31, Göttingen. nestor - Kompetenznetzwerk Langzeitarchivierung.

Janée, G., Mathena, J., and Frew, J. (2008). A data model and architecture for long-term preservation. In *JCDL '08: Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*, pages 134–144, New York, NY, USA. ACM.

Jebelean, T. (1997). Practical integer division with Karatsuba complexity. In *ISSAC '97: Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 339–341, New York, NY, USA. ACM.

Jourdan, M. (1984). Strongly non-circular attribute grammars and their recursive evaluation. *SIGPLAN Not.*, 19(6):81–93.

Karatsuba, A. and Ofman, Y. (1963). Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, 7:595–+.

Kinder, J. (2010). *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt.

Knuth, D. E. (1969). *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, chapter 4.3.1. Addison-Wesley.

Knuth, D. E. (1969b). *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, chapter 1.3. Addison-Wesley.

Knuth, D. E. (1971). Top-down syntax analysis. *Acta Informatica*, 1:79–110.

Knuth, D. E. (1999). *MMIXware - A RISC Computer for the Third Millennium*. Springer.

Knuth, D. E. (2001). *Arithmetik*, chapter 4.3.3, page 111. Springer.

Kol, N., van Diessen, R., and van der Meer, K. (2006). An improved Universal Virtual Computer approach for long-term preservation of digital objects. *Information Services and Use*, 26(4):283–291.

Krebs, N. and Borghoff, U. M. (2010). State-of-the-Art Survey of Long-Term Archiving – Strategies in the Context of Geo-Data / Cartographic Heritage. In Cartwright, W., Gartner, G., Meng, L., Peterson, M. P., and Jobst, M., editors, *Preservation in Digital Cartography*, Lecture Notes in Geoinformation and Cartography, pages 101–127. Springer Berlin Heidelberg.

Krebs, N., Rönnau, S., and Borghoff, U. M. (2010). The Universal Virtual Computer: Practices and Experiences. In Moreno-Días, M., Pichler, F., and Quesada-Arencibia, A., editors, *Extended Abstracts of the 13th International Conference on Computer Aided Systems Theory*, pages 95–96. IUCTC Universidad de Las Palmas de Gran Canaria.

Krebs, N., Rönnau, S., and Borghoff, U. M. (2011a). Fostering the Universal Virtual Computer as Long-Term Preservation Platform. In *Engineering of Computer Based Systems (ECBS), 2011 18th IEEE International Conference and Workshops*, pages 105–110.

Krebs, N. and Schmitz, L. (2012). JACCIE: A Java-based compiler–compiler for generating, visualizing and debugging compiler components. *Science of Computer Programming*.

Krebs, N., Schmitz, L., and Borghoff, U. M. (2011b). Implementing the Universal Virtual Computer. In Moreno-Días, M., Pichler, F., and Quesada-Arencibia, A., editors, *Proceedings of the 13th International Conference on Computer Aided Systems Theory: Part I*, LNCS 6927, pages 153–160. Springer Berlin Heidelberg.

Levine, J., Mason, T., and Brown, D. (1992). *Lex & yacc, 2nd edition*. O'Reilly.

Levis, P. and Culler, D. (2002). Mate: a tiny virtual machine for sensor networks. *SIGPLAN Not.*, 37:85–95.

Lewis, P. A. W., Goodman, A. S., and Miller, J. M. (1969). A pseudo-random number generator for the system/360. *IBM Syst. J.*, 8(2):136–146.

Lindholm, T. and Yellin, F. (1999). *Java Virtual Machine Specification*. Addison-Wesley, Boston, MA, USA.

Lorie, R. A. (2001). Long Term Preservation of Digital Information. In *Proceedings of the 1st ACM/IEEE-CS joint conference on Digital libraries*, JCDL '01, pages 346–352, New York, NY, USA. ACM.

Lorie, R. A. (2002a). A methodology and system for preserving digital data. In *JCDL '02: Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pages 312–319, New York, NY, USA. ACM.

Lorie, R. A. (2002b). The UVC: A Method for Preserving Digital Documents: Proof of Concept. Technical report, IBM and Koninklijke Bibliotheek (KB).

Lorie, R. A. and van Diessen, R. J. (2005). UVC: A universal virtual computer for long-term preservation of digital information. Technical report, IBM Res. rep. RJ 10338. IBM, Yarktown Heights, NY.

Lovelace, A. A. (1953). Notes on Manabrea's Sketch of the Analytical Engine Invented by Charles Babbage. In Bowden, B., editor, *Faster Than Thought*, pages 362–408, London.

Martignoni, L., Paleari, R., Roglia, G. F., and Bruschi, D. (2009). Testing cpu emulators. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 261–272, New York, NY, USA. ACM.

Metcalf, M., Reid, J., and Cohen, M. (2011). *Modern Fortran explained*. Oxford Univ. Press, Oxford, GB.

Müller, A. (2011). Universal Virtual Computer: Bewertung der Crosscompilierung des Bitcodes als Möglichkeit zur Laufzeitgewinnung. Master's thesis, Universität der Bundeswehr München.

Myers, G. J. (1979). *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA.

Myers, G. J. and Sandler, C. (2004). *The Art of Software Testing*. John Wiley & Sons, New York, NY, USA.

Nau, C. (2011). Optimierung eines JPEG-Decoders für den UVC. Bachelor's thesis, Universität der Bundeswehr München.

Newell, A. (1964). *Information Processing Language-V Manual*. Prentice Hall, Englewood N. J., 2nd edition.

Newman, R. and Dennis, C. (2009). JPC: An x86 PC Emulator in Pure Java. In Spinellis, D. and Gousios, G., editors, *Beautiful Architecture*, pages 199–234. O'Reilly, Sebastopol, CA, USA.

Nordin, P. and Banzhaf, W. (1995). Evolving Turing-Complete Programs for a Register Machine with Self-modifying Code. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 318–327, San Francisco, CA, USA. Morgan Kaufmann.

Nürnberg, P., Wiil, U., and Hicks, D. (2004). A grand unified theory for structural computing. In Hicks, D., editor, *Metainformatics*, volume 3002 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg.

Oehme, M. (2012). Re-Implementierung der MI in Java mit Fokussierung auf die Lehre an der Universität der Bundeswehr München. Bachelor's thesis, Universität der Bundeswehr München.

Olson, J. E. (2008). *Database Archiving: How to Keep Lots of Data for a Very Long Time*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Oltmans, E. and Kol, N. (2005). A comparison between migration and emulation in terms of costs. *RLG DigiNews*, 9(2).

Paleari, R., Martignoni, L., Roglia, G. F., and Bruschi, D. (2009). A fistful of red-pills: how to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, WOOT'09, Berkeley, CA, USA. USENIX Association.

Park, S. K. and Miller, K. W. (1988). Random number generators: good ones are hard to find. *Commun. ACM*, 31(10):1192–1201.

Parthasarathi, R. and Jhunjhunwala, A. (1995). Modified straight division: a computer implementation of multiple-precision division. *Microprocess. Microprogram.*, 41:193–209.

Petzold, H. (2000). Z3 und Z4 von Konrad Zuse. In Frieß, P. and Trischler, H., editors, *Meisterwerke aus dem Deutschen Museum III*, pages 52–55. Deutsches Museum, München.

Phillips, P. (2010). Enhanced debugging with traces. *Queue*, 8(3):40–45.

Righi, M. (2008). Cd and dvd preservation issues. In *AXMEDIS 2008: Proceedings of the 4th International Conference on Automated Solutions for Cross Media Content and Multi-Channel Distribution*, pages 46–50, Florence, Italy. Firenze University Press.

Rothe, S. (2011). Implementierung eines Assemblers basierend auf Compilertechniken zur Realisierung ausgewählter Hochsprachenkonstrukte für den UVC. Bachelor's thesis, Universität der Bundeswehr München.

Rothenberg, J. (1999). *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation*. Council on Library & Information Resources.

Schaffrath, P. (2012). Assembler für UVC-Sprache nach erweiterter Spezifikation geschrieben in UVC-Sprache. Bachelor's thesis, Universität der Bundeswehr München.

Schiller, S. (2012). Implementierung der Universellen Virtuellen Computers für einen Großrechner der datArena. Bachelor's thesis, Universität der Bundeswehr München.

Schmidhuber, J. (2005). Completely Self-referential Optimal Reinforcement Learners. In Duch, W., Kacprzyk, J., Oja, E., and Zadrozny, S., editors, *Artificial Neural Networks: Formal Models and Their Applications – ICANN 2005*, volume 3697 of *Lecture Notes in Computer Science*, pages 753–753. Springer Berlin / Heidelberg. 10.1007/11550907_36.

Schmitz, L. (1992). The visual compiler-compiler SIC (abstract). In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, page 236, New York, NY, USA. ACM.

Schmitz, L. (1995). *Syntaxbasierte Programmierwerkzeuge*. B. G. Teubner, Stuttgart, D.

Schönhage, A. and Strassen, V. (1971). Schnelle Multiplikation großer Zahlen. In *Computing 7*, pages 281–292.

Shaylor, N., Simon, D. N., and Bush, W. R. (2003). A java virtual machine architecture for very small devices. *SIGPLAN Not.*, 38(7):34–41.

Sippu, S. and Soisalon-Soininen, E. (1988). *Parsing theory. Vol. 1: languages and parsing*, chapter 3.6. Springer-Verlag New York, Inc., New York, NY, USA.

Sisson, E. H. (2008). Monuments, Memorials, and Spacecraft: A Test-Case in the Treatment of a Spacecraft as a Semiotic Artifact. *SSRN*. Available at SSRN: http://ssrn.com/abstract=1319376.

Slattery, O., Lu, R., Zheng, J., Byers, F., and Tang, X. (2004). Stability Comparison of Recordable Optical Discs–A Study of Error Rates in Harsh Conditions. *Journal of Research of the National Institute of Standards and Technology*, 109:517–524.

Smith, A. G. (1995). Using MPI 2 One Sided Communications on Cray T3D. Technical report, EPCC, The University of Edinburgh.

Smith, J. and Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Staab, S. and Studer, R. (2009). *Handbook on Ontologies*. Springer Publishing Company, Incorporated, 2nd edition.

Sturm, E. (2009). *The New PL/I ... for PC, Workstation and Mainframe*. Vieweg-Teubner, Wiesbaden.

Suchodoletz, D. V., Rechert, K., Schröder, J., and van der Hoeven, J. (2010). Seven Steps for Reliable Emulation Strategies Solved Problems and Open Issues. *iPres*.

Susemihl, F. (1857). Übersetzung von Platons Kritias. In *Platons Werke: Die Platonische Kosmik*, pages 891–916, Stuttgart. Verlag der J.B. Meßler'schen Buchhandlung.

Suter, B. (2010). Archivability of Electronic Literature in Context. In Schäfer, J. and Gendolla, P., editors, *Beyond the Screen: Transformations of Literary Structures, Interfaces and Genre*, pages 443–464. transcript Verlag, Bielefeld.

Tremblay, J.-P. and Sorenson, P. G. (1985). *Theory and Practice of Compiler Writing*. McGraw-Hill, Inc., New York, NY, USA.

Triebsees, T. and Borghoff, U. M. (2006). Preservation-centric and constraint-based migration of digital documents. In *Proceedings of the 2006 ACM symposium on Document engineering*, DocEng '06, pages 59–61, New York, NY, USA. ACM.

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265.

van der Hoeven, J., Lohman, B., and Verdegem, R. (2007). Emulation for Digital Preservation in Practice: The Results. *International Journal of Digital Curation*, 2(2).

van der Hoeven, J., van Wijngaarden, H., Verdegem, R., and Slats, J. (2005). Emulation – a viable preservation strategy. Projektstudie 1.4, The National Library of the Netherlands/-Koninklijke Bibliotheek, Nationaal Archief of the Netherlands.

van der Hoeven, J. R., van Diessen, R. J., and van der Meer, K. (2005). Development of a universal virtual computer (uvc) for long-term preservation of digital objects. *Journal of Information Science*, 31, Nr. 3:196–208.

Welschenbach, M. (2005). *Cryptography in C and C++*. Apress, second edition.

Wendel, K. (2006). Eine ARCHE zur Rettung digitalen Archivguts. *Archivnachrichten / Landesarchiv Baden-Württemberg*, 32:18–19.

Wijngaarden, H. and Oltmans, E. (2004). Digital preservation and permanent access: The uvc for images. In *Proceedings of the Imaging Science & Technology Archiving Conference*, pages 254–258.

Wirth, N. (1975). *Algorithmen und Datenstrukturen*, chapter 4.5. Teubner.

Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. (2010). Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53:91–99.