FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Evaluierung und Weiterentwicklung des Micro16 für die Lehre

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

David KRYWULT
Matrikelnummer 1141760

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dr. Wolfgang KASTNER
Mitwirkung: Senior Lecturer Mag. Dipl.-Ing. Johannes MATIASCH, BSc

Wien, 7. Februar 2017

David KRYWULT

Wolfgang KASTNER

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluation and further development of the Micro16 for teaching

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

David KRYWULT
Registration Number 1141760

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dr. Wolfgang KASTNER
Assistance: Senior Lecturer Mag. Dipl.-Ing. Johannes MATIASCH, BSc

Vienna, 7th February, 2017

David KRYWULT | Wolfgang KASTNER

Erklärung zur Verfassung der Arbeit

David KRYWULT
Heinrich-Lefler-Gasse 14/2/14

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. Februar 2017

David KRYWULT

Danksagung

Die Arbeit an dieser Bachelorarbeit hat lange gedauert. Im Laufe dieser Zeit wurde ich von mehreren Personen tatkräftig unterstützt und ich möchte die Gelegenheit nutzen, um mich bei diesen Personen zu bedanken.

Allen voran möchte ich mich bei meinen Betreuern am Institut für Rechnergestützte Automation der TU Wien bedanken, Dr. Wolfgang Kastner und Johannes Matiasch. Sie haben mich ermutigt und mich immer wieder in die richtige Richtung gewiesen. Ohne sie wäre von all dem hier nichts entstanden.

Ich möchte mich auch bei Jannis Meixner bedanken, der die Arbeit am Micro16-SDK übernommen hat und Weiterentwicklungen durchführt.

Einen großen Dank möchte ich auch Elke Krywult, meiner Frau, aussprechen, ohne die diese Arbeit wohl nie fertig geworden wäre. Danke dir für deine Ausdauer, deine Ermutigungen und das Korrekturlesen!

Acknowledgements

This Bachelor paper was in progress for quite a while. Many people have supported me during this process and I would like to take the opportunity to thank them.

Above all I would like to thank my advisors at the Institute of Computer Aided Automation at TU Wien, Dr. Wolfgang Kastner and Johannes Matiasch. They always encouraged me and guided me in the right direction. Without them this bachelor thesis would never have been written.

I would also like to thank Jannis Meixner who continues working on the Micro16-SDK.

A special thanks goes to my wife, Elke Krywult. Without her this work would never have been finished. Thank you for your endurance, your encouragement and proofreading this document.

Kurzfassung

In der Erstsemester-Lehrveranstaltung „Technische Grundlagen der Informatik“ an der Technischen Universität Wien wird den Studierenden das technische Grundwissen zum Fachgebiet der Informatik vermittelt. Ein essentieller Teil dieser Lehrveranstaltung behandelt das Thema Prozessoren, die den wichtigsten Bauteil in modernen Rechnersystemen darstellen. Gegenwärtig wird im Lehrbetrieb der Beispielprozessor Micro16 eingesetzt. Dieser Prozessor wird im Lehrbuch „Einführung in die Technische Informatik“ vorgestellt. In den vergangenen Jahren wurde die Lehrveranstaltung inhaltlich durch höhere Konzepte erweitert, die vom Micro16 nicht abgedeckt wurden. Deswegen entstanden Überlegungen, den Micro16 entweder zu erweitern oder durch einen anderen Prozessor im Lehrbetrieb zu ersetzen.

Anhand einer Vergleichsstudie wird in dieser Arbeit zunächst ermittelt, ob der Micro16 noch zeitgemäß für den Einsatz in der Lehre ist oder ob sich ein anderer Prozessor besser eignet. Konkret wird die Architektur des Micro16 jener des Atmel AVR, des Intel Skylake sowie der MIPS-Architektur gegenüber gestellt.
Im Zuge der Vergleichsstudie wurde der Micro16 aufgrund seiner einfachen und leicht vermittelbaren Architektur als Mittel der Wahl für diese Erstemester-Lehrveranstaltung bestätigt. In weiterer Folge wurde der Micro16 dahingehend erweitert, dass die Konzepte Pipelining, Hazard Detection und Caching ebenfalls abdeckt werden.
Im praktischen Teil dieser Arbeit wurden ein Simulator inklusive Entwicklungsumgebung sowie eine Implementierung als IP-Core auf einem FPGA erstellt. Durch diese Implementierungen kann der Micro16 im Übungsbetrieb der Lehrveranstaltung optimal eingesetzt werden. Auf dem Simulator können Studierende ihre Programme schrittweise ausführen, wodurch der Mikrocode sowie seine Funktionsweise besser nachvollzogen werden können. Die Entwicklungsumgebung hilft bei der Arbeit mit dem Simulator und erlaubt durch eine detaillierte Visualisierung einen tiefen Einblick in die Arbeitsweise des Micro16. Durch ein Case-Testing-System in der Entwicklungsumgebung können die Übungsleiterinnen und Übungsleiter die Abgaben der Studierenden automatisiert testen.
Der Simulator sowie das Case-Testing-System werden inzwischen seit mehreren Jahren erfolgreich in der Lehrveranstaltung „Technische Grundlagen der Informatik“ eingesetzt. Im Zuge der jährlichen Lehrveranstaltungsbewertungen gab es durchwegs äußerst positive Rückmeldungen über den Einsatz des Simulators.

Abstract

The introduction course “Fundamentals of Computer Engineering“ at the Technical University of Vienna intends to give the students an understanding of the basic technical concepts of computers. A main part of the course covers the topic of processors, the most important component of modern computers. Currently, the Micro16 is being used as an example processor as it is presented in the book “Einführung in die Technische Informatik”.
Over the last years, higher concepts were added to the course which were not covered by the Micro16. Thus it was considered to either extend the Micro16 or to replace it with a different processor.
This work evaluates if the Micro16 is still state of the art for this teaching course or if a different processor would be a better fit. Therefore, the architecture of the Micro16 is compared to the architecture of Atmel AVR, Intel Skyplake and the MIPS-architecture. The evaluation confirmed that the Micro16 is still the best choice. In the next step, the Micro16 was extended to include the concepts of pipelining, hazard detection and caching.
Within the practical part of this work, a simulator including an integrated development environment (IDE) was developed. Additionally, the Micro16 was implemented as IP-Core on an FPGA. Through these implementations, the Micro16 is an optimal fit for the practical part of the course. The students can execute programs on the simulator which helps them to better understand the microcode and its functionality. The IDE simplifies the work with the simulator. Its detailed visualization provides a deep look into the Micro16’s mode of operation. A case-testing-system helps the course instructors to automatically test the students’ assignments in the practical sessions.
During the last few years, the simulator and the case testing system have been successfully used in the course “Fundamentals of Computer Engineering”. Throughout the last years, the students’ course feedback has been extremely positive on the simulator.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Problemstellung

An der Technischen Universität Wien wird angehenden Informatikerinnen und Informatikern in der Erstsemester-Lehrveranstaltung „Technische Grundlagen der Informatik“, kurz TGI, unter anderem die Funktionsweise von Prozessoren vermittelt. Für eine fundierte Informatik-Ausbildung ist es nötig, die generelle Funktionsweise eines Prozessors zu verstehen, um effiziente Programme schreiben zu können.

Um dieses Wissen praxisnah vermitteln zu können, ist es notwendig, einen geeigneten Prozessor für die Wissenvermittlung zu wählen. Anhand des gewählten Prozessors sollten sich sämtliche technische Konzepte darstellen lassen. Gleichzeitig sollte die Architektur des Prozessors einfach genug sein, um im Rahmen dieser erstsemestrigen Lehrveranstaltung schnell vermittelt werden zu können.
Gegenwärtig wird hierfür eine Eigenentwicklung, der Micro16, verwendet. Dieser Prozessor wurde von Gerhard Schildt et al. im Lehrbuch „Einführung in die Technische Informatik“ eingeführt [Sch05b]. Kritikpunkte an diesem Prozessor waren, dass dieser nie in realen Anwendungsfällen eingesetzt wurde, bis dato keine technische Implementierung davon existierte, und der Einsatz in der Lehre seit Jahren nicht mehr auf Eignung und Aktualität überprüft worden war.

1.2 Methodik

Zunächst wird im Zuge einer vergleichenden Studie analysiert, welcher Prozessor sich für die Lehrveranstaltung „Technische Grundlagen der Informatik“ am besten eignet. Hierfür werden mehrere Prozessoren, die entweder historisch wichtig waren, aktuell verwendet und/oder von anderen Universitäten in ähnlichen Lehrveranstaltungen eingesetzt werden, einander und dem Micro16 gegenübergestellt. Vor- und Nachteile in Bezug auf den Einsatz

in der Lehre werden erörtert. Dabei liegt der Fokus auf den nachfolgenden Inhalten der Lehrveranstaltung:

- Arithmetic Logic Unit (ALU)
- Mikroinstruktionen
- Speicherzugriff
- Sequencing Logic
- Mikroprogramme und Assembler
- Register und Stacks
- Interrupts
- Funktionsaufrufe
- Pipelining
- Caching
- Peripherie
- Firmware und Betriebssystem
- Speicherverwaltung

Aufbauend auf den Ergebnissen der Vergleichsstudie werden Konzepte erarbeitet und umgesetzt, mit dem Ziel, die Vermittlung der Inhalte der Lehrveranstaltung „Technische Grundlagen der Informatik“ nachhaltig zu verbessern.

1.3 Erwartetes Resultat

Ziel dieser Arbeit ist es, eine fundierte Aussage dahingehend zu treffen, ob der Micro16 weiterhin die beste Wahl zur Vermittlung der Inhalte der Lehrveranstaltung „Technische Grundlagen der Informatik“ ist.

In weiterer Folge sollen Software- und Hardware-basierte Lehrmaterialen entwickelt werden, um die Wissensvermittlung bestmöglich zu unterstützen. Bei der Entwicklung dieser Lehrmaterialen ist darauf zu achten, den organisatorischen Anforderungen einer Lehrveranstaltung mit über 1000 Studierenden pro Jahr gerecht zu werden.

Im praktischen Teil dieser Arbeit wurden in weiterer Folge ein Software-basierter Simulator für den Micro16 sowie eine Realisierung des Selbigen als IP-Core auf einem FPGA erstellt. Seit dem Studienjahr 2012/13 werden diese erfolgreich sowohl im Vorlesung- als auch im Übungsteil der Lehrveranstaltung eingesetzt.

KAPITEL 2

Übersicht der Prozessoren

2.1 Micro16

Aktuell wird in der Lehrveranstaltung „Technische Grundlagen der Informatik“ der Micro16 als Anschauungsbeispiel für einen Mikroprozessor verwendet.

Der Micro16 wurde im Lehrbuch „Einführung in die Technische Informatik“ eingeführt, das für die Lehrveranstaltung als primäre Literaturquelle dient. Die Architektur des Micro16 ist stark angelehnt an jene des MIC-1, der in der dritten Edition des Buches „Structured Computer Organization“ vorgestellt wurde. In späteren Editionen wurde der MIC-1 stark verändert, so dass er nur noch entfernte Ähnlichkeiten mit dem Micro16 aufweist. [RR98]

Ziel bei der Entwicklung des Micro16 war es, einen möglichst einfachen Prozessor zu schaffen, der die meisten Basiskonzepte des Prozessordesigns in sich vereint. Er wird sowohl im Rahmen des Buches als auch der Lehrveranstaltung von den Schaltwerken kommend Stück für Stück aufgebaut. Damit ist es möglich, den Studierenden einen beinahe nahtlosen Übergang von einzelnen Logikgattern über Schaltnetze und Schaltwerke zu einem vollen Prozessor nahezubringen. Der überschaubare Befehlssatz des Micro16 (siehe Abschnitt 7.1 im Anhang) ist gleichzeitig seine Stärke sowie seine Schwäche: Er ist dadurch sehr klein und leicht nachvollziehbar, wodurch ihn Studierende verhältnismäßig schnell erlernen können. Auf der anderen Seite werden höhere Konzepte, wie zum Beispiel Pipelining, nicht abgedeckt. [Sch05b, 125ff]

Im Rahmen dieser Arbeit wurde für den Micro16 eine simple Programmierumgebung entwickelt, die Syntax-Überprüfung, Assemblierung und Simulation des Micro16 sowie Unit-Tests für selbst geschriebene Programme bietet. Diese Programmierumgebung wird seit mehreren Jahren hauptsächlich im Übungsteil der Lehrveranstaltung von den Studierenden verwendet. Desweiteren wurde der Micro16 auch als Softcore für den Cyclone II-FPGA von Altera implementiert.

2.2 Atmel AVR

Die Atmel AVR-Architektur ist eine 8-Bit-RISC-Architektur, die hauptsächlich in den Mikrocontroller-Serien ATMega und ATTiny verwendet wird. Diese Mikrocontroller werden häufig im kommerziellen sowie nicht-kommerziellen Umfeld verwendet, um elektronische Anlagen zu steuern. So werden zum Beispiel verschiedene AVR-Mikrocontroller als Basis für die Arduino-Plattform verwendet.

Der Atmel AVR ist deutlich weniger komplex als die nachfolgenden Prozessoren, da er nur für die Verwendung in eingebetteten Systemen gedacht ist, für die weniger Leistung benötigt wird. Durch diese geringe Komplexität ist der Umgang mit Atmel AVR-Prozessoren leicht zu erlernen, weswegen er bei vielen Hobby-Entwicklern beliebt ist.

Für diese Arbeit wird hierbei der ATMega328p als Beispiel genommen.

2.3 MIPS

Die MIPS-Architektur wurde ab 1981 von John L. Hennessy an der Stanford Universität entwickelt. Seit seinen Anfängen wurde der MIPS in verschiedensten Anwendungsfällen verwendet. So wurden MIPS-basierte Prozessoren bis Ende der 90er-Jahre in Workstations eingesetzt. Der Nintendo 64 sowie die Playstation, die Playstation 2 und die Playstation Portable verwendeten diese Architektur. Heutzutage werden MIPS-basierte Prozessoren hauptsächlich in eingebetteten Systemen verwendet, da sie im Hochleistungssegment immer mehr von ARM und x86-Prozessoren verdrängt werden.

Da die MIPS-Architektur relativ alt ist, ist sie immer noch vergleichsweise simpel. So wird beispielsweise Pipelining grundsätzlich unterstützt, allerdings ignoriert der MIPS absichtlich alle Hazards und verlangt von der Programmiererin / dem Programmierer oder dem Compiler, sich selbst um die Vermeidung dieser zu kümmern.

2.4 Intel Skylake

Der Intel Skylake ist das andere Ende des Spektrums. Hierbei handelt es sich um eine Mikroarchitektur von Intel, die 2015 veröffentlicht wurde. Diese Architektur ist aktuell eine der leistungsstärksten Architekturen am Markt. Es handelt sich um eine RISC-Architektur, die mithilfe einer Mikrocode-Zwischenschicht x86/x64-Maschinencode ausführen kann.

Diese Prozessoren sind sehr fortgeschritten und komplex. Sie bieten alle Features, die im Rahmen dieser Lehrveranstaltung durchgenommen werden und darüber hinaus noch um vieles mehr. Allerdings sind einige dieser Features vom Endnutzer nicht direkt verwendbar – z.B. lässt sich der Mikrocode des Intel Skylake nur mit von Intel signierten Updates verändern. Auch sind viele Design-Details dieser Architektur nicht öffentlich einsehbar, da es sich dabei um Geschäftsgeheimnisse handelt.

Der Hauptvorteil des Intel Skylake im Kontext dieser Lehrveranstaltung besteht darin, dass viele der Studierenden einen solchen Prozessor, oder zumindest einen dazu Bytecode-kompatiblen x86/x64-Prozessor besitzen.

KAPITEL 3

Konzepte von TGI

3.1 Arithmetic Logic Unit (ALU)

Die Aritmetic Logic Unit (ALU) ist das Herzstück eines Prozessors. Sie ist dafür zuständig, Logik- und Rechenoperationen durchzuführen. Damit ist sie eines der wichtigsten befehlsausführenden Elemente auf einem Prozessor. [PH05, 286] [HH10, 242]

Jeder der Prozessoren, die im Rahmen dieser Arbeit behandelt werden, verfügt über eine ALU. Der Micro16 hat hier den geringsten Funktionsumfang. Seine ALU kann nur addieren, sowie die binären Operationen NOT und AND durchführen. Darüber hinaus kann sie gleichzeitig das Ergebnis einer Operation mit einer Bitfolge vergleichen, oder überprüfen, ob das höchstwertige Bit gesetzt ist und es sich somit – in Zweierkomplementdarstellung interpretiert – um eine negative Zahl handelt. Der ALU nachgeschaltet befindet sich ein Shifter, der das Ergebnis der Rechnung um eine Position bitweise nach links oder rechts verschieben kann.
In der Vorlesung wird die ALU komplett aus logischen Bausteinen aufgebaut. Es ist für die Studierenden somit leicht möglich, die Funktionsweise jedes einzelnen ALU-Befehls zu verstehen. [Sch05b, 126ff]

Der Atmel AVR besitzt 28 verschiedene arithmetische und logische Funktionen. Alle Funktionen beziehen sich auf 8-Bit-Befehle und sind relativ leicht verständlich. Viele dieser Befehle sind Varianten von einander. So gibt es z.B. den Befehl SUB, der den Inhalt eines Registers vom Inhalt eines anderen Registers abzieht. Zusätzlich gibt es den Befehl SUBI, der einen konstanten Wert vom Inhalt eines Registers abzieht. Desweiteren gibt es noch die Befehle SBC und SBCI, die sich verhalten wie SUB und SUBI, lediglich mit dem Unterschied, dass zusätzlich der Übertrag der Subtraktion berücksichtigt wird. Für manche Befehle gibt es vorzeichenbehaftete und nicht vorzeichenbehaftete Varianten. [Cor15, 616]
Im Rahmen dieser Lehrveranstaltung wäre es aus zeitlichen Gründen nicht möglich,

jede einzelne dieser Funktionen aus logischen Bausteinen aufzubauen. Allerdings wäre es durchaus vorstellbar, einige wenige Funktionen detailliert vorzuzeigen, während der Rest der Befehle nur überflogen wird.

Der MIPS hat mit 53 arithmetischen und logischen Funktionen einen deutlich höheren Funktionsumfang. Wie beim Atmel AVR sind wieder viele Befehle Varianten derselben einfachen Befehle. [MT09, 34ff]
So wie der Atmel AVR geht auch die Komplexität des MIPS über den Rahmen dieser Vorlesung hinaus.

Beim Intel Skylake gibt es 73 arithmetische und logische Operationen. Zusätzlich dazu gibt es noch etliche andere Befehle, die die ALU verwenden (z.B. Sprungbefehle oder Move-Befehle), sowie viele weitere mathematische Befehle, die nicht von der ALU verarbeitet werden, wie z.B. die Float-Operationen. Die Anzahl der Operatoren würde den Rahmen der Vorlesung bei weitem übersteigen. Wie beim MIPS und dem Atmel AVR könnte man auch hier ein paar der Wichtigsten im Detail vorstellen und einige weitere Operationen kurz aufzeigen. [Cor16, 5-4ff]

3.2 Mikrocode, Mikroprogramme und Assembler

Mikrocode ist eine Technik, die erlaubt, komplexere Befehle auf einem simpleren Prozessor zu implementieren. Dabei werden sogenannte Mikroinstruktionen, also Befehle, die der Prozessor direkt in Hardware abarbeiten kann, verwendet, um komplexere Befehle zu implementieren. So könnte z.B. auf einem Prozessor, der in Hardware nur addieren kann, die Multiplikation in Mikrocode implementiert werden. [Sch05b, 139ff]

Der Micro16 erlaubt das direkte Programmieren des Mikrocodes. Der Befehlssatz für den Mikrocode ist sehr simpel und verständlich. Der Funktionsumfang ist stark eingeschränkt und lässt sich verhältnismäßig einfach im Rahmen der Lehrveranstaltung erlernen. [Sch05b, 139ff]

Bei MIPS und Atmel AVR würde dieses Themengebiet vollständig entfallen, da der vollständige Instruktionssatz dieser beiden Prozessoren direkt in Hardware implementiert ist und es somit keinen Mikrocode am MIPS bzw. Atmel AVR gibt.

Der Intel Skylake verwendet zwar Mikrocode, allerdings ist dieser proprietär und durch eine Signatur vor unautorisierten Manipulationen geschützt. Modifikationen durch den Nutzer sind somit nicht möglich. [Cor16, 9-36]

Da in der Vorlesung Mikrocode allerdings nicht verwendet wird, um einen höheren Instruktionssatz zu implementieren, sondern nur um darin Programme zu schreiben, könnte man stattdessen Atmel AVR-, MIPS- bzw. x86-Assembler verwenden. Darin lassen sich, so wie aktuell in der Vorlesung umgesetzt, kleine Programme hardwarenahe schreiben. Sowohl in Atmel AVR-, MIPS- als auch in x86-Assemblern könnte man einen limitierten Befehlssatz vermitteln, der im Umfang in etwa jenem des Micro16 entspricht. Dadurch könnte gewährleistet werden, dass dieser von Studierenden ähnlich schnell erlernt

werden kann. Der volle Instruktionssatz wäre hingegen bei jedem der drei Prozessoren deutlich zu umfangreich für die Vorlesung. Das könnte in der Übung Probleme bereiten, falls Studierende in ihren Lösungen Befehle verwenden, die von der Übungsleitung nicht vorgesehen sind und die somit den ÜbungsleiterInnen, TutorInnen und anderen Studierenden nicht bekannt sind. Speziell bei dem verhältnismäßig großen Umfang des Befehlssatzes des Intel Skylake wäre dies naheliegend.

3.3 Speicherzugriff

In den Registern eines Prozessors lassen sich nur sehr wenige Daten abspeichern, üblicherweise weit unter einem Kilobyte. Dazu kommt, dass Register üblicherweise nicht persistent sind. Verliert der Prozessor die Stromzufuhr, dann bleiben üblicherweise keine Daten im Prozessor erhalten. Deswegen ist es notwendig, auf weitere Speicher zuzugreifen, die größer und/oder persistent sind. [Hof13, 318ff]

Der Micro16 besitzt ein sehr simples Speicherinterface. Um auf den Speicher zugreifen zu können, sind zwei Register (MAR und MBR) sowie zwei Signalleitungen (MS und RD/WR) vorhanden. Das Register MAR wird mit der zu lesenden oder schreibenden Hauptspeicheradresse beladen. Soll auf den Haupspeicher geschrieben werden, dann wird der zu schreibende Wert in das Register MBR geschrieben. Dann wird MS für zwei Takte auf hoch gezogen. Die Signalleitung RD/WR zeigt mit ihrem Zustand währenddessen dem Speichermodul an, ob gelesen oder geschrieben wird. Nach diesen zwei Takten ist entweder der Wert aus MBR in den Hauptspeicher geschrieben worden oder umgekehrt (je nachdem, auf welchen Wert RD/WR gesetzt wurde). Wurde der Wert vom Hauptspeicher ins MBR geschrieben, so kann er dann vom Prozessor aus diesem Register geladen werden. [Sch05b, 132ff]

Der Atmel AVR besitzt drei verschiedene Speicherbereiche, die verschieden adressierbar sind. Der einfachste Block ist hierbei der SRAM. Dieser Speicherblock ist sehr schnell, allerdings nicht persistent und vergleichsweise klein. Er kann direkt adressiert werden. Der zweite Block ist das EEPROM. Im Gegensatz zum SRAM ist das EEPROM sehr langsam und persistent. Es ist allerdings auch ähnlich klein wie der SRAM. Es ist nur mithilfe der I/O-Register adressierbar. Der Vorgang dafür ist ähnlich wie beim Micro16. Der dritte Block ist der Programmspeicher. Da der Atmel AVR eine Harvard-Architektur verwendet, ist dieser Speicherblock vom Rest des Speichers getrennt. Er ist nur indirekt auslesbar und nicht vom Prozessor beschreibbar. Dafür ist er sehr groß und schnell. [Cor15, 17ff]

Um mit einem Speicher zu kommunizieren, besitzt der MIPS die Befehle LoadMemory und StoreMemory. Diese Befehle laden den gewünschten Wert aus dem Speicher und kümmern sich gleichzeitig darum, dass der Cache valide gehalten wird.

Der Speicherzugriff beim Intel Skylake ist deutlich komplizierter. Da dieser Prozessor virtuellen Speicher sowie mehrere Caching-Stufen besitzt, passiert hier sehr viel intern ohne Einblick für den Programmierer / die Programmiererin. Zuerst wird der Wert,

der geladen oder geschrieben werden soll, durch die Caches geschickt. Sofern es einen Cache Hit gab, wird dieser Wert zurückgegeben. Gibt es einen Cache Miss, wird der Wert an den Data Translation Lookaside Buffer geschickt, der bestimmt, ob sich die passende Speicherseite im Puffer befindet. Wenn nicht, dann muss bestimmt werden, ob sich diese Speicherseite überhaupt im Speicher befindet, oder ob sie nachgeladen werden muss. Muss sie nachgeladen werden, so wird dies durchgeführt, der Data Translation Lookaside Buffer und die Caches werden passend befüllt und der Wert wird an den Prozessor weitergegeben. [Cor16, 2-39ff]

3.4 Sequencing Logic

Die Sequencing Logic ist dafür zuständig, dass die Befehle in der richtigen Reihenfolge ausgeführt werden. Zusätzlich zum normalen sequentiellen Abarbeiten können mit der Sequencing Logic Sprünge und bedingte Sprünge ausgeführt werden. So kann ein Codesegment wiederholt ausgeführt werden (Schleifen), oder nur, wenn eine bestimmte Vorbedingung eingetreten ist. [Sch05b, 135ff]

Der Micro16 hat eine sehr simple Sequencing Logic, die neben der normalen Ausführung nur drei Sprungbefehle unterstützt. So gibt es den unbedingten Sprung (GOTO), einen bedingten Sprung mit der Bedingung, dass das Ergebnis der Rechnung gleich 0 ist (IF Z GOTO) sowie einen bedingten Sprung mit der Bedingung, dass das höchstwertige Bit des Ausgangsregisters gesetzt ist (IF N GOTO).
Mit den bedingten Sprüngen lassen sich leicht alle üblichen Vergleichsoperatoren darstellen. Subtrahiert man zwei Werte, so kann man mit IF Z GOTO feststellen, ob diese Werte gleich oder ungleich sind. Mit IF N GOTO kann festgestellt werden, ob einer der Werte größer ist als der andere. Kombiniert man beide Befehle, kann man auch größergleich und in weiterer Folge auch kleinergleich darstellen. [Sch05b, 135ff]

Der Atmel AVR besitzt 36 verschiedene Sprungbefehle. Zusätzlich zu den Befehlen, die der Micro16 beherrscht, kann der Atmel AVR auch das Carry-Flag für Sprünge verwenden, sowie verschiedene Werte miteinander vergleichen, statt nur Werte mit 0 zu vergleichen. Es gibt Sprungbefehle, die auf Werte in Statusregistern oder auf Interrupts reagieren. Jeden Befehl gibt es auch in einer invertierten Variante, z.B. gibt es zu dem Befehl BREQ (Branch if Equal) das Gegenstück BRNE (Branch if Not Equal). Im Gegensatz vom Micro16 können diese Befehle allerdings nicht mit ALU-Befehlen kombiniert werden. Das heißt, man muss vor dem Sprungbefehl die benötigte Sprungbedingung fertig berechnet haben, anstatt sie im gleichen Takt zu berechnen. [Cor15, 616]

Der MIPS besitzt 16 Sprungbefehle. Diese sind eingeteilt in Branches, Jumps und Subroutine Calls. Die sieben Branch-Befehle vergleichen zwei Register, die als Parameter übergeben werden, und führen dann einen Sprung aus, sofern der Vergleich zutrifft. So führt der Befehl BEQ einen Sprung aus, wenn der Inhalt der beiden Register, die als Parameter übergeben wurden, gleich ist. Die beiden Jump-Befehle führen unbedingte Sprünge aus, wobei bei dem Befehl J das Sprungziel im Programmcode definiert wird, während bei dem Befehl JR als Sprungziel der Inhalt des angegebenen Registers verwendet

wird. Der Subroutine Call-Befehl JAL kopiert vor dem Sprung die aktuelle Position im Code in das Register $ra (= „Return Address“), welches dann mit dem Befehl JR als Rücksprungadresse aus einer Subroutine verwendet werden kann. [MT09, 35]

Was Sprungbefehle angeht, ist der Intel Skylake dem Micro16 ähnlicher als dem MIPS. Um einen bedingten Sprung auszuführen, muss man den CMP-Befehl verwenden, der zwei Werte vergleichen kann und je nach Ergebnis entsprechende Flags setzt. Diese Flags werden dann in einem der Jcc-Befehl verwendet, um zu bestimmen, ob der Branch ausgeführt wird oder nicht. Das cc im Jcc steht für den genauen condition code, der bestimmt, welches der Flags überprüft werden soll. So führt z.B. der JE/JZ-Befehl (Jump Equal/Jump Zero) den Branch aus, wenn das ZF (Zero Flag) gesetzt ist. Das ist dann der Fall, wenn mit dem CMP-Befehl zwei gleiche Werte verglichen wurden. [Cor11, 7-23ff] Insgesamt besitzt der Intel Skylake 20 bedingte und elf unbedingte Sprungbefehle. [Cor11, 5-7f]

3.5 Register und Stacks

Register und Stacks sind zwei Varianten, wie Prozessoren Daten verwalten können. Register sind fest vorgegebene Speicherbereiche, die direkt mit den Recheneinheiten des Prozessors verbunden sind. Ein registerbasierter Prozessor hat üblicherweise eine feste Anzahl an Registern, deren Größe vorgegeben ist. Diese Register werden als sehr schnelle Zwischenspeicher für Eingabe- und Ausgabewerte der ALU und anderen Bauteilen des Prozessors verwendet. Sie sind üblicherweise der schnellste und kleinste Speicher, den ein Computer zur Verfügung hat. [Sch05b, 129ff] [PH07, C-3] [PBLS16, 66]

Je nach Implementierung gibt es Datenregister sowie Register mit speziellen Funktionen. So hat der Micro16 13 Datenregister, drei Konstantenregister und zwei Spezialregister, die für den Speicherzugriff verwendet werden. Eines dieser beiden Spezialregister beinhaltet eine Speicheradresse (Speicher-Adressregister), das andere einen Wert, der entweder an diese Speicheradresse geschrieben werden kann, oder von dort ausgelesen wurde (Speicher-Datenregister). Jedes dieser Register beinhaltet 16 Bit Speicher. Die 13 Datenregister sowie die Konstantenregister und das Speicher-Datenregister können als Eingabewerte für die ALU verwendet werden. Das Ergebnis einer Rechnung lässt sich entweder in die Datenregister oder in das Speicher-Datenregister schreiben oder aber auch verwerfen. Die Konstantenregister sind nicht beschreibbar und beinhalten immer die Werte 0 (=0x0000), 1 (=0x0001) und -1 (=0xFFFF). Sie können verwendet werden, um mithilfe der Datenregister und der ALU beliebige Werte in die Register zu schreiben. [Sch05b, 129ff] [Sch05a, 49ff]

Der Atmel AVR besitzt 32 Datenregister zu je 8 Bit. Sechs dieser Register können jeweils paarweise zusammengefasst als drei 16-Bit-Register verwendet werden. Diese Register können ansonsten beliebig ausgetauscht werden. Dazu gibt es noch einige Statusregister, die Informationen über den inneren Zustand des Prozessors geben können, sowie einige Kontrollregister, mit denen Hardware-Einstellungen sowie der Zustand der IO-Pins gesteuert werden können. [Cor15, 9ff]

Der MIPS besitzt auch 32 Datenregister, sowie zwei spezielle Register, genannt Hi und Lo, die die Resultate von Multiplikationen und Divisionen enthalten. Im Gegensatz zum Micro16 haben diese Datenregister alle noch besondere Funktionen. Register 0 ist ein Konstantenregister, das immer den Wert 0 beinhaltet. Die anderen Register werden als Funktionsparameter, Rückgabewerte, temporäre Werte, als Adresszeiger für den Stack bzw. Funktionsrücksprungadressen oder Ähnliches verwendet. Das sorgt für erhöhte Funktionalität, führt gleichzeitig aber zu einer gegenüber dem Micro16 deutlich erhöhten Komplexität. [MT08, 6]

Die Register des Intel Skylake sind deutlich komplexer. Es gibt acht normale Datenregister, die allerdings unterschiedliche Sonderfunktionen haben. Diese bestehen darin, dass die Prozessor-Befehle z.B. immer das erste Register als Akkumulator verwenden. Jedes dieser Register kann als 64-Bit-, 32-Bit- oder 16-Bit-Register verwendet werden. Die ersten fünf Register können zusätzlich auch als je zwei 8-Bit-Register angesprochen werden.
Darüber hinaus gibt es sechs Segmentregister. Eines der Register zeigt auf das Segment, in dem sich der Stack befindet, eines auf das Segment, in dem sich die aktuelle Code-Zeile befindet. Die restlichen vier Register zeigen auf Segmente, in denen sich aktuell verwendete Daten befinden. Zusätzlich dazu gibt es noch einige andere Register, die für die diversen Subsysteme auf dem Prozessor verwendet werden: Beispielsweise die Register der Floating Point Unit, die für das Rechnen mit Gleitkommazahlen zuständig ist. [Cor11, 7-1ff]

Der Stack, auch Stapelspeicher genannt, ist als Komplement zum Registersatz gedacht. Register sind zwar sehr schnell, bieten aber nur sehr wenig Speicherplatz. Der Stack hingegen liegt im Hauptspeicher und bietet verhältnismäßig sehr viel Speicherplatz. Der Name Stapelspeicher beschreibt sehr gut, wie er funktioniert: Der Prozessor kann Daten aus den Registern oben auf den Stack legen. Will er Daten vom Stack lesen, so kann er immer nur den obersten Wert lesen. Diesen Wert kann er wahlweise während dem Lesen vom Stack entfernen und somit den nächsten Wert darunter lesbar machen. Dazu gibt es oft ein spezielles Register, den Stackpointer, der auf den aktuell obersten Wert zeigt. Der Stack ist somit eine sehr nützliche Datenstruktur, da sie (theoretisch) beliebig groß werden kann und gleichzeitig einfach zu verwalten ist. [Sch05b, 151ff]

Im Micro16 gibt es keinen dedizierten Stack, allerdings lässt sich ein solcher sehr leicht in Mikrocode implementieren. Dafür verwendet man ein beliebiges Datenregister als Stackpointer. Diesen Wert initialisiert man z.B. mit dem Wert 0xFFFF (= Start bei höchster verfügbarer Adresse). Wenn man ein Element hinzufügen will, verringert man den Stackpointer um 1 und schreibt den Wert, den man hinzufügen will, an die Position im Speicher, die durch den Stackpointer angegeben wird. Möchte man den aktuellen Wert am Stack auslesen, muss nur der Wert an der Stelle im Speicher ausgelesen werden. Möchte man den Wert vom Stack entfernen, muss der Stackpointer um 1 inkrementiert werden. [Sch05b, 153ff]

Ähnlich ist es beim MIPS. Auch hier gibt es keine Stack-Befehle und auch kein technisch festgelegtes Register für den Stackpointer, allerdings wird dafür üblicherweise das Register 29 verwendet. Auch hier müssen die Stack-Befehle also manuell implementiert werden.

Der Intel Skylake besitzt drei explizite Stack-Befehle: Push, um einen Wert auf den Stack zu legen; Peek, um den letzten Wert vom Stack auszulesen; Pop, um den Wert auszulesen und vom Stack zu entfernen. Da das Stackpointer-Register (das fünfte Register, u.a. auch RSP genannt) als normales Register ausgelesen und beschrieben werden kann, können diese Befehle auch manuell implementiert werden. [Cor16, 2-36]

Beim Atmel AVR gibt es vier Stack-Befehle. Zusätzlich zu den bekannten Befehlen PUSH und POP gibt es noch die Befehle CALL und RET, die für Funktionsaufrufe verwendet werden. Diese Befehle kombinieren einen Sprungbefehl (Sprung in eine Funktion oder davon wieder zurück) damit, dass die Rücksprungadresse auf den Stack gelegt oder davon gelesen wird. [Cor15, 13]

3.6 Interrupts

Interrupts sind eine Möglichkeit, asynchron auf Ereignisse zu reagieren, während der Prozessor mit einer Aufgabe beschäftigt ist. Wenn ein Interrupt ausgelöst wird, wird der aktuell laufende Prozess pausiert und stattdessen eine Funktion ausgeführt, die diesen Interrupt abarbeitet. Sobald diese Funktion fertig ist, wird der unterbrochene Prozess fortgeführt. Eine häufige Anwendung von Interrupts ist das sofortige Reagieren auf Nutzereingaben. Die Alternative zu Interrupts ist Polling: Bei diesem Verfahren muss der Prozess neben seinen eigentlichen Aufgaben regelmäßig den Status der Eingaben überprüfen, was allerdings zu vielen unnötigen Abfragen führt. Außerdem ist beim Polling die Reaktionszeit unter Umständen deutlich höher. [Sch05b, 150f] [Shi08, 359]

Der Micro16 unterstützt keine Art von Interrupts. Hier muss in jedem Fall auf Polling ausgewichen werden.

Der MIPS hingegen unterstützt nicht nur Interrupts, sondern auch Exceptions. Exceptions funktionieren hier im Grunde gleich wie Interrupts, mit dem Unterschied, dass sie nicht von außerhalb des Prozessors kommen, sondern vom Prozessor selbst – beispielsweise wenn unerwartete Fehler auftreten, wie z.B. ein Überlauf in einer Rechenoperation oder wenn versucht wird, einen unbekannten Opcode auszuführen. Für Interrupts von außen gibt es zwei Schnittstellen, IRQ und IACK. IRQ, kurz für Interrupt ReQuest, wird von externen Geräten verwendet, um dem Prozessor mitzuteilen, dass ein Interrupt stattfinden sollte. Da der Prozessor eventuell mehrere Taktzyklen benötigt, um auf diesen Interrupt Request zu reagieren, gibt es die Schnittstelle IACK, kurz für Interrupt ACKnowledge. Der Prozessor verwendet IACK, um dem Interruptauslöser zu signalisieren, dass der Interrupt angenommen wurde und bearbeitet wird. An dieser Stelle springt der Prozessor dann an die dafür im Code vorgesehene Interrupt-Routine, führt diese aus und springt wieder an die Stelle im Code, von der aus er in die Interrupt-Routine gesprungen war. [MT08, 7]

Der Atmel AVR unterstützt eine Vielzahl möglicher Interrupt-Quellen: So kann ein Interrupt durch externe Quellen, verschiedene Timer (regelmäßig auftretende Events)

und Watchdog-Timeouts ausgelöst werden. Es gibt spezifische Interrupts für verschiedene serielle Übertragungsprotokolle sowie analoge und digitale Eingangswerte. [Cor15, 57f]

Auch der Intel Skylake besitzt viele verschiedene Interrupt-Quellen. Ähnlich wie beim MIPS sind sie beim Intel Skylake auch in Interrupts und Exceptions aufgeteilt, wobei Exceptions hauptsächlich bei Fehlern auftreten. Interrupts können beim Intel Skylake nur durch Input-Änderungen an dafür vorgesehenen Input-Pins sowie durch spezielle Funktionsaufrufe ausgelöst werden. [Cor11, 6-14ff]

3.7 Funktionen und Subroutinen

Funktionen und Subroutinen ermöglichen es, Programme in kleinere wiederverwendbare Teile aufzuspalten. Subroutinen sind die einfachere Variante der beiden: Hierbei springt die Code-Ausführung wie bei einem normalen goto an die angegebene Stelle im Programm. Im Gegensatz zu einem reinen goto merkt sich das Programm, von wo der Sprung ausgeführt wurde und kann wieder dorthin zurückspringen. [Sch05b, 148ff]

In MIPS-Assembler lässt sich leicht darstellen, wie Subroutinen funktionieren: Zuerst wird die Subroutine mit dem Statement jal (steht für „jump and link“) aufgerufen. jal speichert die Adresse der aktuellen Codezeile in das Register $ra (steht für „return address“) und springt dann an die angegebene Code-Stelle. Um dann wieder zurückzuspringen, verwendet man jr (steht für „jump register“), wodurch an die im gegebenen Register angegebene Programmzeile gesprungen wird.
Ein Beispiel dafür könnte wie folgt aussehen [MT09, 130,135]:

```
jal target_label      # schreibt die aktuelle Adresse nach $ra
                      # und springt nach target_label

target_label:         # Ziel-Label des Sprungs
                      # beliebiger Code in der Subroutine

jr $ra                # Rücksprung in die Zeile nach jal
```

Subroutinen ermöglichen es somit Code wiederverwendbar zu machen, indem der gleiche Code-Teil an mehreren Stellen im Programm verwendet werden kann.

Allerdings gibt es mit reinen Subroutinen immer noch zwei Probleme: Erstens, die Subroutine und das Hauptprogramm teilen sich die gleiche Ausführungsumgebung sowie die gleichen Register. Sollte die Subroutine eines der Register modifizieren, das das Hauptprogramm benötigt, dann könnte das zu Datenverlust und somit zu falschen Berechnungen oder sogar zu Programmabstürzen führen. Zweitens überschreibt jal (oder das Äquivalent in anderen Prozessorarchitekturen) jedes Mal das gleiche Register (im Falle des MIPS das Register $ra). Damit ist es nicht möglich, von einer Subroutine aus in eine andere Subroutine zu springen, da sonst die Information verloren geht, wo der Sprung ursprünglich hergekommen ist.

Um diese Probleme zu umgehen gibt es Funktionen, die das Konzept der Subroutine erweitern. Im Gegensatz zu Subroutinen kapseln Funktionen den Code der Funktion. Das bedeutet allgemein gesprochen, dass eine Funktion ausgeführt werden kann, unabhängig vom Zustand des Programms außerhalb des Funktionsaufrufs. Das erlaubt einerseits – abhängig vom verfügbaren Hauptspeicher – annähernd beliebig tief verschachtelte Funktionsaufrufe. Andererseits macht es den Programmcode leichter wartbar, da die einzelnen Funktionen unabhängig voneinander programmiert und getestet werden können. Durch die Kapselung gibt es weniger unvorhergesehene Wechselwirkungen von Code-Teilen, die nicht zusammen gehören, die sich aber intern die gleichen Ressourcen (z.B. Register) teilen müssen.

Als Schnittstelle zur Funktion dienen Funktionsparameter und Rückgabewerte. Einer Funktion können beliebig viele Parameter übergeben werden, die verwendet werden, um Werte innerhalb der Funktion zu initialisieren. Damit kann die aufrufende Funktion – trotz Datenkapselung – Werte an die aufgerufene Funktion übergeben. Ist die Ausführung der aufgerufenen Funktion beendet, so kann sie einen oder mehrere Rückgabewerte zurückgeben. Als Beispiel dient folgender mathematischer Funktionsaufruf:

$\sin(30°) \rightarrow 0.5$

Diese Funktion bekommt einen Parameter übergeben, nämlich 30°. Der Rückgabewert ist 0.5. [Sch05b, 148ff]

Im Gegensatz zur Subroutine gibt es für Funktionen bei MIPS, Atmel AVR und x86 keinen Prozessorbefehl für Funktionsaufrufe. Stattdessen werden Funktionen in Software implementiert. Dafür gibt es je nach Betriebssystem, Compiler bzw. Programmiersprache oft verschiedene Konventionen, wie genau ein Funktionsaufruf vonstatten geht – das generelle Prinzip ist bei all diesen Konventionen dasselbe.

Für Funktionsaufrufe wird immer ein Stack verwendet. Wird eine Funktion aufgerufen, wird der aktuelle Zustand inklusive aller relevanter Variablen sowie der aktuellen Rücksprungadresse auf den Stack geschrieben. Damit ist der aktuelle Status für den Rücksprung gesichert. Dieser Datensatz am Stapel wird „Call Stack Frame“ genannt. Im Anschluss wird ein neuer Call Stack Frame erstellt, in den alle Funktionsparameter geschrieben werden, damit die Funktion diese Werte zur Verfügung hat. Hiermit sind die Vorbereitungen erledigt und der eigentliche Aufruf kann durchgeführt werden. Dazu wird ein normaler Subroutinen-Aufruf durchgeführt. Dadurch springt das Programm an die zur Ausführung der Funktion angegebene Stelle im Programmcode und die Rücksprungadresse wird abgespeichert. Wird die Funktion beendet, dann springt die Ausführung an die letzte Rücksprungadresse, verwirft den aktuellen Call Stack Frame und lädt die Daten des letzten Call Stack Frames vom Stack. Dadurch wird die Umgebung der aufrufenden Funktion wiederhergestellt. Diese kann nun den Rückgabewert der Funktion verwenden und weiterverarbeiten.

Wie schon erwähnt, unterstützen sowohl die MIPS-, die Atmel AVR- als auch die x86-Architektur nur Subroutinen direkt. Funktionen werden in diesen drei Architekturen softwareseitig implementiert.

Der Micro16 unterstützt weder Funktionen noch Subroutinen, da sich der Wert des Micro Instruction Counters, in dem gespeichert wird, an welcher Programmzeile sich die Ausführung befindet, nicht in ein Register gespeichert oder aus einem Register geladen werden kann. Allerdings lässt sich leicht Mikrocode definieren, der Programme, die im Hauptspeicher liegen, interpretieren kann. Da für solche Programme der Program Counter in einem Register abgespeichert werden muss, lassen sich auf diese Weise Subroutinenaufrufe implementieren. Damit ist es in weiterer Folge auch möglich, Funktionsaufrufe zu implementieren.

3.8 Pipelining

Pipelining ist eine Möglichkeit, die Ausführungsgeschwindigkeit auf einem Prozessor zu erhöhen, indem die Instruktion in mehrere Teile zerlegt wird, die parallel ausgeführt werden. Diese Teile werden Stufen genannt. Pro Stufe kann im Idealfall ein Befehl abgearbeitet werden. Das heißt, dass eine fünfstufige Pipeline bis zu fünf Befehle gleichzeitig abarbeiten kann, was die Ausführung erheblich beschleunigen kann. [PH07, A-2f]
Dabei gibt es allerdings Ressourcenkonflikte zu beachten, die dadurch entstehen können, dass mehrere Befehle in der Pipeline die gleiche Ressource verwenden. Diese sogenannten Hazards müssen gegebenenfalls erkannt und abgearbeitet werden. Genaueres zu Pipeline-Hazards und ihrer Vermeidung findet sich unter Punkt 5.1.2. [PH07, A-11f]

Der Micro16 unterstützt Pipelining nicht, allerdings wird im Kapitel 5.1 beschrieben, wie der Micro16 um Pipelining erweitert werden kann.

Die Pipeline des Atmel AVR ist die einfachste Pipeline von den Prozessoren, die hier verglichen werden. Sie besteht aus nur zwei Stufen: pre-fetch und execute. Das heißt, während der erste Befehl noch ausgeführt wird, wird der nächste Befehl schon geladen. Auf diese Weise muss der Prozessor mit der Ausführung nicht darauf warten, dass der nächste Befehl geladen wird. Durch diese kurze Pipeline kommt es nur bei Sprungbefehlen zu Hazards, die sich leicht verhindern lassen. [Cor15, 9]

Der MIPS hat eine etwas komplexere Pipeline. Hier gibt es fünf Stufen: Instruction Fetch, Instruction Decode, Execute, Memory Access und Write Back. Diese etwas komplexere Pipeline erlaubt eine deutliche Beschleunigung der Ausführung, sorgt aber dafür, dass es mehr Möglichkeiten für Pipeline-Hazards gibt. Um keine Hazard-Erkennung implementieren zu müssen, wurde diese Arbeit der Programmiererin / dem Programmierer überlassen. Es muss selbstständig erkannt werden, wo im Programm mögliche Pipeline-Hazards auftreten können. An diesen Stellen muss das Programm mithilfe von No-Operation-Befehlen – also Befehlen, die keine Auswirkung haben – kurzzeitig pausiert werden, bis der Hazard abgewendet ist. [MT08, 3]

Die Pipeline des Intel Skylake ist wesentlich komplexer. Im Gegensatz zu den anderen Prozessoren, die in dieser Arbeit verglichen werden, besitzt der Intel Skylake keine einfache mehrstufige Pipeline, sondern eine Pipeline mit verschiedenen Ausführungseinheiten. Je nach Befehl werden verschiedene Ausführungseinheiten benötigt. Somit benötigen

verschiedene Befehle verschieden viele Taktzyklen auf ihrem Weg durch die Pipeline. Dazu unterstützt der Intel Skylake Out-of-Order-Execution: Das bedeutet, dass der Prozessor die Ausführungsreihenfolge von voneinander unabhängigen Befehlen ändern kann, so dass die Befehle möglichst effizient von der Pipeline abgearbeitet werden können. Das erhöht die Effizienz, aber auch die Komplexität. Während die Pipeline der anderen Prozessoren innerhalb kurzer Zeit vermittelt werden kann, müsste für die Pipeline des Intel Skylake deutlich mehr Zeit veranschlagt werden. [Cor16, 2-2ff]

3.9 Caching

Idealerweise möchte man einen unendlich großen Speicher haben, auf den man sofort zugreifen kann. Leider sind Zugriffsgeschwindigkeit und Speicherkapazität in der Regel einander widersprechende Ziele. Speicher mit hoher Kapazität sind meist langsam, schnelle Speicher hingegen brauchen meist viel Platz bzw. Ressourcen und sind verhältnismäßig teuer. Deswegen besitzen die meisten Computer eine Hierarchie von Speichern, beginnend mit Registern, die extrem schnell (Zugriff innerhalb von einem Prozessortakt) aber sehr klein sind (üblicherweise unter einem Kilobyte) bis hin zur Festplatte, die sehr groß (bis zu mehreren Terabyte) aber auch sehr langsam ist (Zugriff benötigt mehrere Millisekunden). [BGvN46, 94]

Caching ist eine Möglichkeit, um dieses Problem zu verringern: Hierbei wird der Speicher in eine Hierarchie aus verschieden schnellen und verschieden großen Speichern aufgeteilt, wobei höhere Stufen schneller aber kleiner sind. Jede Schicht beinhaltet dabei einen Teil der Daten aus der nächsten (tieferen) Schicht, so dass diese Daten nicht aus dieser langsameren Schicht geladen werden müssen. Dabei gibt es verschiedene Strategien, wie bestimmt wird, welche Daten in der schnelleren Schicht gehalten werden. [PH05, 468ff] [Cle06, 344ff]

Prozessor-Caches sind dabei eine Caching-Schicht, die üblicherweise zwischen dem Prozessorkern und dem Hauptspeicher angeordnet ist. Nicht jeder Prozessor hat einen eigenen Cache. Manche Prozessoren haben sogar mehrere Prozessor-Cache-Schichten.

Der Micro16 besitzt keinen Prozessor-Cache, allerdings wird im Kapitel 5.2 beschrieben, welche Änderungen am Micro16 vorgenommen werden müssen, damit Prozessor-Caches unterstützt werden.

Auch der Atmel AVR unterstützt keinen Cache. Im Gegensatz zum Micro16 ist es allerdings nicht möglich, Caching am Atmel AVR zu implementieren. Da der Zugriff auf seinen Hauptspeicher innerhalb eines Taktes erfolgt, gibt es bei dem Atmel AVR auch keinen Grund, Caching zu implementieren, da es keinen Performance-Gewinn bringen würde.

Die MIPS-Architektur unterstützt Caches, allerdings wird der verwendete Cache nicht exakt spezifiziert und ist somit implementierungsabhängig. Allerdings sind die Befehle LoadMemory und StoreMemory so spezifiziert, dass sie korrekt mit Caches umgehen können. [MT09, 25]

Der Intel Skylake unterstützt drei Prozessor-Cache-Schichten. Die genaue Art und Größe der verwendeten Caches unterscheidet sich je nach Implementierung. [Cor16, 2-5f]

3.10 Direktvergleich der Prozessoren

Nachdem die einzelnen Prozessoren im Detail besprochen wurden, werden in diesem Abschnitt die Prozessoren sowie ihre Eignung für das Vermitteln der einzelnen Konzepte in Tabelle 3.1 zusammengefasst. Die Einschätzungen hinsichtlich Komplexität und Aufwand für die Wissensvermittlung wurden in Rücksprache mit dem Lehrveranstaltungsteam, allen voran mit dem verantwortlichen Senior Lecturer Hrn. Johannes Matiasch, getroffen.

Aus dieser Tabelle geht hervor, dass der Intel Skylake deutlich zu komplex für einen zielführenden Einsatz in einer erstsemestrigen Lehrveranstaltung wäre. Die hohe Anzahl an Befehlen und Bestandteilen des Intel Skylake bedeuten, dass – um die Hardware und die Assemblersprache dieses Prozessors hinreichend erklären zu können – viel mehr Zeit bzw. mehrere aufeinander aufbauende Lehrveranstaltungen benötigt werden würden. Der größte Vorteil, der sich aus der Verwendung des Intel Skylake ergeben würde, ist, dass einige Studierende bereits über so einen Prozessor verfügen würden.

Der MIPS ist deutlich älter als der Intel Skylake und somit auch deutlich weniger komplex. Allerdings bewegt sich seine Komplexität noch immer auf einem zu hohen Niveau, um die vollständige Architektur – ohne entsprechendes Vorwissen – in einer erstsemestrigen Lehrveranstaltung zielführend behandeln zu können. Im Gegensatz zum unmodifizierten Micro16 beherrscht der MIPS Pipelining und – je nach Implementierung – auch Caching. Es wäre also möglich, einen limitierten Funktionsumfang des MIPS für die Vorlesung zu verwenden. Allerdings ist die MIPS-Programmierung deutlich komplexer als die Programmierung des Micro16 und die Software, die dafür vorhanden ist, richtet sich hauptsächlich an die produktive Verwendung des MIPS, ist also deutlich komplexer als die Entwicklungsumgebung des Micro16.

Der Atmel AVR ist ein gutes Stück simpler als der MIPS, da er hauptsächlich für kleine Embedded-Projekte und den Hobbygebrauch gedacht ist. Als einziger Prozessor in diesem Vergleich basiert er auf einer 8-Bit-Architektur. Er lässt sich leicht in C und Assembler programmieren. Aufgrund seines niedrigen Preises (bei großen Stückzahlen rund €1 pro Stück) könnten interessierte Studierende sich selbst Atmel AVR-Prozessoren kaufen. Allerdings fehlt diesem Prozessor Caching und seine Pipeline ist zu simpel (nur zwei Stufen) um sie als sinnvolle Erklärungshilfe für das Konzept Pipelining zu verwenden. Das einzige Konzept, das der Atmel AVR zusätzlich vom Micro16 abdeckt, sind Interrupts.

Beim Micro16 zeigt sich, dass er für die Verwendung in einer einführenden Lehrveranstaltung konzipiert wurde. Der Funktionsumfang des Micro16 ist viel zu gering, um tatsächlich praxisrelevante Verwendung zu ermöglichen. Dafür sorgt diese Simplizität dafür, dass er in der Lehrveranstaltung schnell vermittelt und gut eingesetzt werden kann. Durch die Erweiterungen, die in dieser Arbeit vorgestellt werden, erfüllt der Micro16 in weiterer Folge die Anforderungen der Lehrveranstaltung und deckt die Themen Caching

Tabelle 3.1: Vergleich der Prozessoren

	Micro16	**Atmel AVR**	**MIPS**	**Intel Skylake**
ALU	Simpel, auf Gate-Ebene erklärbar	Eher komplex, einzelne Befehle erklärbar	Eher komplex, einzelne Befehle erklärbar	Sehr komplex
Mikrocode, Mikroprogramme	Vorhanden und modifizierbar	Nicht vorhanden	Nicht vorhanden	Vorhanden, aber nur durch Intel modifizierbar
Assembler	Implementierbar	Vorhanden	Vorhanden	Vorhanden
Speicherzugriff	Sehr simpel	Eher komplex aufgrund versch. Speicherarten	Eher komplex	Sehr komplex
Sequencing Logic	Sehr simpel	Eher komplex	Eher simpel	Sehr komplex
Register	Sehr simpel	Sehr simpel	Eher komplex durch special purpose registers	Sehr komplex
Interrupts	Nicht vorhanden	Vorhanden, eher simpel	Vorhanden, eher komplex	Vorhanden, eher komplex
Funktionen, Subroutinen	Nicht vorhanden, aber in Assembler implementierbar	Vorhanden, eher simpel	Vorhanden, eher simpel	Vorhanden, eher simpel
Pipelining	Durch Erweiterung aus 5.1: vier Stufen, mit Hazard Detection	Vorhanden, zwei Stufen, keine Information über Hazard Detection	Vorhanden, fünf Stufen, ohne Hazard Detection	Vorhanden, 14 Stufen, mit Hazard Detection
Caching	Durch Erweiterung aus 5.2 vorhanden	Nicht vorhanden	Implementierungs-abhängig	Mehrstufiges Caching
Implementierungen	Emulator, FPGA	Emulator, Hardware-Implementierung	Emulator, Hardware-Implementierung	Emulator, Hardware-Implementierung

und Pipelining ab. Aufgrund seiner Simplizität und Verständlichkeit ist der Micro16 zusammen mit den Erweiterungen aus dieser Arbeit, die im folgenden Kapitel vorgestellt werden, der für diese Lehrveranstaltung vorgeschlagene Prozessor.

KAPITEL 4

Erweiterungen für den Micro16

Da der Micro16 in seiner ursprünglichen Form nicht alle Inhalte der Lehrveranstaltung abdeckt, werden in diesem Kapitel einige Erweiterungen vorgeschlagen und umgesetzt. Das Konzept einer Erweiterung passt dabei auch relativ gut zum Inhalt der Lehrveranstaltung, da der Micro16 bislang im Rahmen der Vorlesung Schritt für Schritt aufgebaut wird.

4.1 Pipelining

4.1.1 Vorbereitung

Pipelining ist nach den Ausführungen der Lehrveranstaltungsleitung eines der wichtigsten Konzepte, das der Micro16 gegenwärtig misst. Es ist allerdings verhältnismäßig leicht, Pipelining am Micro16 zu implementieren, da sich die Architektur des Micro16 aufgrund der folgenden Eigenschaften gut dafür eignet – dies gilt im Allgemeinen für einen Großteil der gängigen RISC-Architekturen [PH07, A-4]:

- Alle Datenoperationen wirken nur auf Daten in Registern und verändern üblicherweise das gesamte Register.
- Die einzigen Befehle, die Zugriff auf den Hauptspeicher haben, sind die Operationen rd (Load) und wr (Store).
- Es gibt nur ein Befehlsformat und alle Befehle besitzen dieselbe Instruktionslänge (32 Bit).

Aktuell besitzt der Micro16 bereits vier verschiedene Berechnungsphasen, die allerdings aktuell nicht unabhängig von einander ausgeführt werden können. Diese vier Phasen sind:

1. MIR: Lade die aktuelle Befehlszeile vom Programmspeicher in das MIR.

2. REG: Lade die passenden Werte vom Registerspeicher oder MBR in die Arbeitsregister A und B.

3. ADR: Falls gewünscht, schreibe Werte von Arbeitsregister B in das Speicher-Adressregister MAR und, falls gewünscht, vom Ergebnisregister S nach MBR.

4. WB (Write Back): Schreibe den Wert des Ergebnisregisters S in den Registerspeicher, werte die Flags N und Z aus und passe den Wert des MIC an.

Möchte man diese Phasen in aufeinander aufbauende Pipeline-Stufen trennen, so müssen sie leicht anders geordnet werden. Eine bessere Aufteilung wäre beispielsweise die Folgende:

1. MIR: lade die aktuelle Befehlszeile vom Programmspeicher in das MIR, im Falle eines unbedingten Sprunges passe den Wert des MIC an.

2. LOAD: lade die passenden Werte vom Registerspeicher und ggf. MBR in die Arbeitsregister A und B.

3. EXE: führe Berechnungen in der ALU und dem Shifter aus.

4. WB: Schreibe die Werte des Ergebnisregisters S in den Registerspeicher und, falls gewünscht, nach MBR. Schreibe, falls gewünscht, den Inhalt des Registers B nach MAR. Im Falle eines bedingten Sprunges, werte die Flags N und Z aus und passe den Wert des MIC an.

Damit würde der Micro16 dem gleichen Schema wie viele RISC-Prozessoren folgen (Instruction Fetch, Register Fetch, Execution, Memory Access, Write-Back), allerdings mit dem Hauptunterschied, dass die Stufen Memory Access und Write-Back beim Micro16 in der Stufe Write-Back vereinigt werden [PH07, A-4]. Dieser Unterschied kommt daher, dass bei der MIPS-Architektur die Befehle LOAD und STORE jeweils einen vollen Taktzyklus benötigen und gleichzeitig nichts anderes ausgeführt werden kann. Beim Micro16 hingegen benötigen die Befehle rd und wr, die die gleiche Aufgabe haben, per Definition zwei Taktzyklen. Dafür könnten parallel andere Aufgaben ausgeführt werden.

4.1.2 Pipelining ohne Hazard-Erkennung

Im nächsten Schritt kann man ein simples Pipelining implementieren, das Hazards ignoriert, so wie dies bei der MIPS-Architektur umgesetzt wurde. Hierbei muss der Compiler bzw. die Programmiererin / der Programmierer darauf achten, gegebenenfalls selbst NOPs einzufügen, um Hazards zu vermeiden. Das Äquivalent des Micro16 zu einer No-Operation (NOP) ist die Codezeile 0x0000.

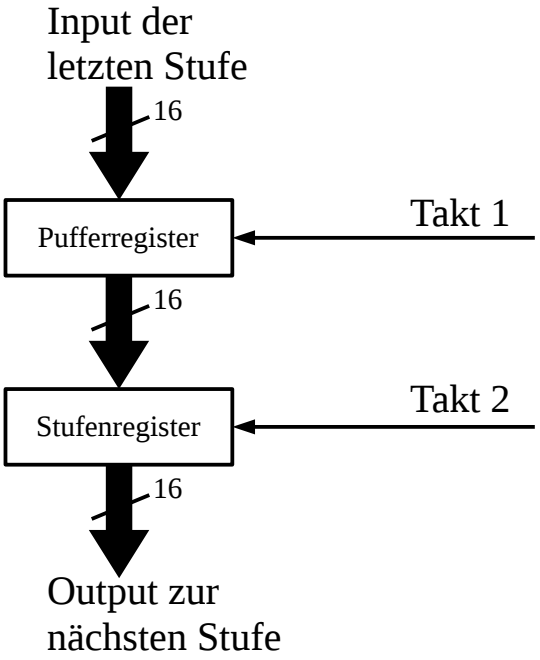

Abbildung 4.1: Doppelte Pufferung zwischen Pipeline-Stufen

Damit ein derartiges Pipelining, funktioniert muss jede der Stufen über zwei Register gepuffert werden. Das bedeutet, dass an den Grenzen zwischen den einzelnen Stufen Register eingebaut werden, die am Ende jeder Phase den Wert aus der vorigen Stufe annehmen und für die nächste Phase an die nächste Stufe abgeben.
Werte, die über mehrere Phasen hinweg erhalten werden müssen, müssen auf mehrere Register – eines für jede Stufe – aufgeteilt werden.
Um konfliktfreie Datenübergaben von einer Stufe an die nächste zu gewährleisten, sind diese Register doppelt ausgeführt. Beim ersten Taktschlag nehmen die Pufferregister die Daten der letzten Stufe auf, beim zweiten Taktschlag geben sie die Daten an das nachfolgende Stufenregister weiter. Diese Daten werden dann der nachfolgenden Stufe zur Verarbeitung zur Verfügung gestellt. Auf diese Weise wird sichergestellt, dass der Inhalt eines Stufenregisters erst dann überschrieben wird, wenn der alte Inhalt aus der letzten Phase nicht mehr benötigt wird.

Um diese Vorgaben umsetzen zu können, sind folgende Änderungen notwendig:

- Das MIR wird auf drei Stufenregister aufgeteilt, MIR2, MIR3 und MIR4. Jedes dieser Register enthält die Befehlszeile, die von dem Prozessor in der aktuellen Stufe bearbeitet wird. Dabei beginnt jeder Befehl damit, dass er zu Beginn der ersten Phase aus dem Programmspeicher in das MIR2 geladen wird. Am Ende jeder Phase wird dieser Wert erst in das Pufferregister der aktuellen Stufe geschrieben und dann in das Stufenregister der nächsten Stufe geladen.

- Die Arbeitsregister A und B werden nur in Stufe 2 (LOAD) und 3 (EXE) benötigt. Das Register B wird zusätzlich noch in Stufe 4 (WB) benötigt, um gegebenenfalls seinen Wert ins MAR zu schreiben. Diese Register werden somit A2 und A3 sowie B2, B3 und B4 genannt. Diese Register werden in Phase 2 beladen und reichen wie MIR2-4 ihre Werte am Ende jeder Phase an ihr Nachfolgeregister weiter. B wird hierbei aus dem Registerspeicher beladen, während A aus dem Ergebnis des A-MUX – also entweder aus dem Registerspeicher oder aus dem MBR – beladen wird.

- Das Arbeitsregister S wird nur für die Phase 4 (WB) benötigt. Es existiert somit nur als S4. Am Ende der Phase 3 wird es mit dem Ergebnis der ALU und des Shifters beladen. In Phase 4 wird der Inhalt des Registers S – je nach Inhalt des MIR – wahlweise in das Register MBR und/oder in den Registerspeicher geschrieben.

- Ähnliches gilt für die Werte N und Z. Diese beiden Werte werden im originalen Micro16 nicht zwischengespeichert, sondern sind reine Signalleitungen. In dieser Modifikation des Micro16 müssen sie allerdings zwischengespeichert werden, wenn auch nur für Phase 4 (WB). Sie existieren also nur als N4 und Z4. Wie auch das Arbeitsregister S werden sie am Ende der Phase 3 (EXE) beschrieben. In der Phase 4 werden sie verwendet, um in Kombination mit eventuellen Sprungbefehlen im MIR4 zu bestimmen, auf welchen Wert das Register MIC für den nächsten Befehl gesetzt werden soll.

Abbildung 4.2 zeigt die veränderte Architektur des Micro16 mit Pipelining ohne Hazard-Erkennung.

4.1.3 Pipelining mit Hazard-Erkennung

Mit diesen Änderungen allein wäre das Pipelining auf dem Micro16 schon lauffähig. Da allerdings Hazards hier noch komplett ignoriert werden und bei der Programmierung manuell NOPs eingefügt werden müssten, wäre die Programmierung mit dieser Modifikation deutlich aufwändiger. Außerdem würden die NOPs wertvollen Speicherplatz im ohnehin schon knapp bemessenen Programmspeicher verschwenden. Deswegen wäre es sinnvoll, eine Hazard-Erkennung zu implementieren. Es gibt hierbei grundsätzlich drei verschiedene Arten von Hazards, die es zu erkennen gilt [PH07, A-11]:

- Strukturelle Hazards: Ein struktureller Hazard tritt auf, wenn mehrere Befehle die gleiche Struktur verwenden, wie z.B. ein paralleler Zugriff auf den Hauptspeicher mit nur einer Datenleitung.

- Control Hazards oder Branching Hazards: Diese Art von Hazard trifft auf, wenn ein bedingter Sprungbefehl verwendet wird. Hierbei weiß der Prozessor vor der Ausführung der Zeile nicht, ob der Sprung ausgeführt wird, oder nicht. Er kann somit nicht sicher sein, welche Code-Zeile er als nächstes laden muss.

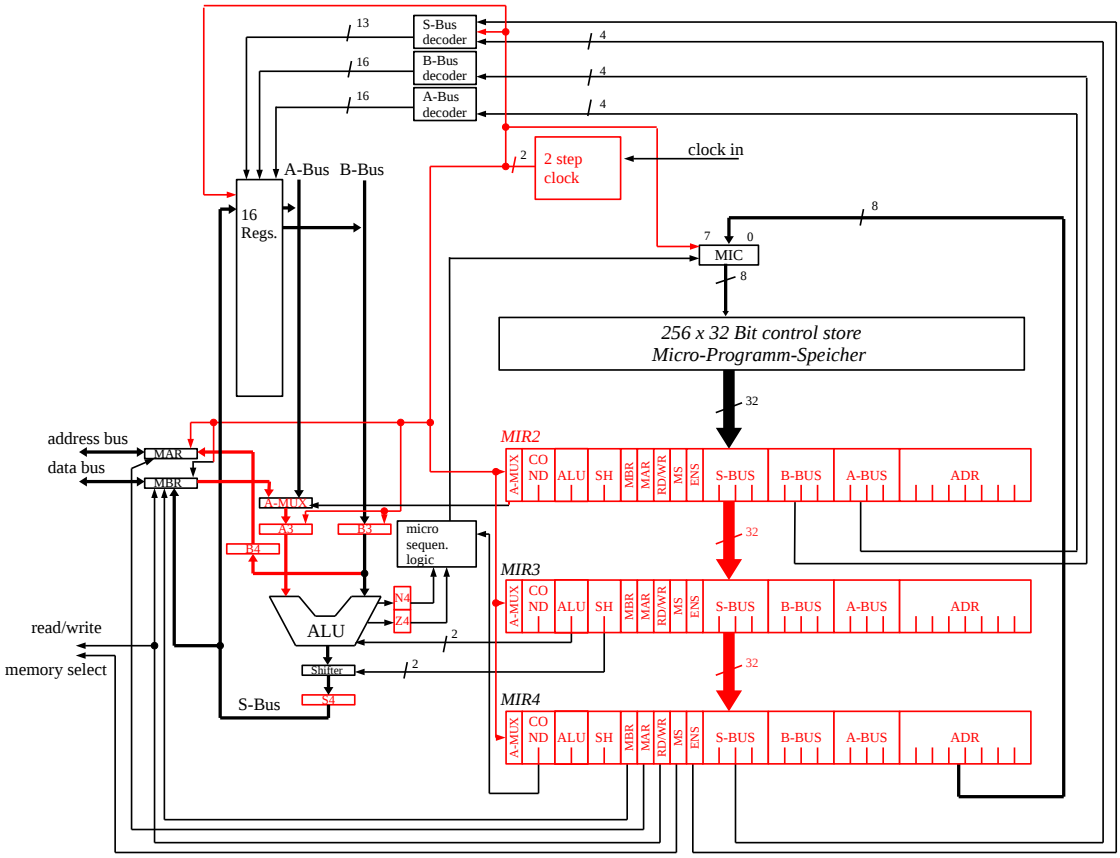

Abbildung 4.2: Erweiterte Micro16-Architektur mit Pipelining ohne Hazard-Erkennung (Erweiterungen in Rot)

- Data Hazards: Hierbei wird das Ergebnis eines Vorgängerbefehls benötigt, um den aktuellen Befehl auszuführen.

Um einen Hazard zu vermeiden, gibt es verschiedene Möglichkeiten: Die einfachste Möglichkeit, die gleichzeitig der letzte Ausweg ist, falls andere Methoden fehlschlagen, ist die Pipeline zu „stallen“. Hierbei wird die Ausführung weiterer Befehle durch eingeschobene NOPs verzögert, bis sich der Hazard aufgelöst hat, weil der vorhergehende Befehl vollständig abgearbeitet wurde. Stalls funktionieren immer, kosten aber viel Rechenzeit, da während des Stalls keine weiteren Befehle ausgeführt werden können. [PH07, A-11]

Die einzigen strukturellen Hazards, die es beim Micro16 gibt, treten auf, wenn eine Instruktion auf den Speicher mit einem der Speicherbefehle wr oder rd zugreift, während eine zweite Instruktion versucht, den jeweils anderen Befehl auszuführen. Um einen derartigen Stall zu vermeiden, würde es tiefgehende Veränderungen an der Architektur benötigen, wie z.B. eine Caching-Einheit, die es ermöglicht, ein rd oder wr in einem Takt auszuführen. [PH07, A-13]

Bei Control Hazards kann ein Stall vermieden werden, indem vorher korrekt vorhergesagt wurde, ob der Sprungbefehl ausgeführt wird oder nicht. Wurde der Sprung korrekt vorhergesagt, verliert der Prozessor nicht an Ausführungsgeschwindigkeit und muss die Pipeline nicht stallen. Wurde der Sprung falsch vorhergesagt, müssen die fälschlicherweise ausgeführten Befehle aus der Pipeline entfernt werden. Da diese Befehle ohnehin erst in der letzten Stufe Auswirkungen auf den Zustand des Prozessors haben und diese Befehle die letzte Stufe damit nicht mehr erreichen, verursachen diese fälschlicherweise ausgeführten Befehle beim Micro16 keine Probleme. Hierbei gehen genauso viele Taktzyklen verloren, als hätte der Prozessor die Pipeline direkt von Anfang an verzögert. Das heißt, es gibt hier keinen Verlust. Selbst im schlechtesten Fall ist ein Prozessor mit Sprungvorhersage nicht langsamer als ein Prozessor ohne Sprungvorhersage. Somit ist selbst eine schlechte Sprungvorhersage besser als gar keine. Ein simplistischer Ansatz für eine Sprungvorhersage wäre z.B. immer davon auszugehen, dass alle Sprünge ausgeführt bzw. nicht ausgeführt werden. Komplexere Sprungvorhersage-Algorithmen analysieren den Code vor Ausführung oder verwenden Statistiken, um die Sprungvorhersage während des Betriebes zu optimieren. [PH07, A-22]

Data Hazards treten auf, wenn für die Ausführung eines Befehls das Ergebnis eines vorhergehenden Befehls, der sich noch in der Pipeline befindet, benötigt wird. Ein Beispiel wäre die folgende Situation:

```
R1 ← R5 + R6
R2 ← R1
```

Hier kann der zweite Befehl nicht direkt ausgeführt werden, da erst das Ergebnis des ersten Befehls verarbeitet und gespeichert werden muss, bevor der zweite Befehl ausgeführt werden kann. Man müsste also die Pipeline verzögern und warten, bis der erste Befehl vollständig ausgeführt wurde.
Dieses Problem kann man mit „Forwarding“ umgehen. Forwarding wird dadurch ermöglicht, dass das Ergebnis der ersten Zeile in der zweiten Zeile erst in Phase 3 (Execution) benötigt wird, aber schon nach Phase 3 der ersten Zeile zur Verfügung steht. Beim Forwarding wird somit das Ergebnis aus der ALU nicht nur nach S4, sondern auch gleichzeitig nach A3 bzw. B3 geschrieben. Auf diese Weise kann der nachfolgende Befehl das Ergebnis schon verwenden, während es noch in Phase 4 in das Zielregister geschrieben wird [PH07, A-17].
Allerdings können nicht alle Data Hazards durch Forwarding umgangen werden. Vor allem Data Hazards im Zusammenhang mit Speicherzugriffen können nicht geforwarded werden, da beim Lesen aus dem Speicher die Daten noch nicht vorhanden sind. In diesem Fall muss verzögert werden oder der Hazard durch andere Methoden wie Out-of-Order-Execution vermieden werden.

Out-of-Order-Execution kann viele Hazards vermeiden, die sonst unvermeidbar wären. Bei dieser Optimierung wird versucht, durch Umsortieren der Befehle Hazards zu vermeiden. Der Prozessor verarbeitet hier nicht nur einen Befehl, sondern bezieht mehrere nachfolgende Befehle in die Verarbeitung mit ein. Wenn er einen zukünftigen Stall erkennt,

kann er dann die Befehle auf eine Weise umsortieren, die keine Stalls verursacht, aber das Ergebnis nicht verändert. Das folgende Beispiel soll diese Prozedur verdeutlichen:

```
R0 ← 1 + 1
R1 ← R0 + 1
R2 ← (-1) + (-1)
```

Hier verursacht der zweite Befehl einen Read-After-Write-Hazard, indem er versucht, R0 auszulesen, obwohl der erste Befehl noch nicht mit dem Schreiben nach R0 fertig ist. Der dritte Befehl hingegen ist vollständig unabhängig von den ersten beiden Befehlen. Er kann also vorgezogen ausgeführt werden.

Allerdings kann auch Out-of-Order-Execution nicht alle Hazards vermeiden, wie folgendes Beispiel zeigt:

```
R0 ← 1 + 1
R1 ← R0 + 1
R2 ← R1 + 1
```

In diesem Fall ist es nicht möglich, die Befehle umzusortieren, da jeder Befehl das Ergebnis des vorigen Befehls benötigt. Hier muss die Pipeline verzögert werden [PH07, A-66 bis A-70].

Für den Micro16 sollte die Hazard-Vermeidung möglichst einfach gehalten werden, weshalb grundsätzlich ein Verzögern der Pipeline vorzuziehen ist. Ein Stall ist sehr einfach zu implementieren, alle anderen Vermeidungsstrategien wären deutlich komplexer bei der Implementierung. Alles, was getan werden muss, ist, dass, wenn ein Hazard erkannt wurde, statt der eigentlichen Instruktion eine No-Operation in das MIR2 geschrieben wird. Dazu wird vor das MIR2 eine weitere Einheit geschaltet, die Hazard Detection. Diese Einheit bekommt am Anfang der ersten Phase den nächsten Befehl vom Programmspeicher. Löst dieser Befehl keinen Hazard aus, so wird er an MIR2 weiter gereicht. Die Hazard Detection signalisiert nun der Micro Sequencing Logic, dass der MIC um 1 erhöht werden darf. Wenn dieser Befehl jedoch einen Hazard auslösen würde, dann wird NOP an MIR2 weitergeleitet und der Micro Sequencing Logic signalisiert, dass MIC nicht erhöht werden soll.

So eine einfache Hazard-Vermeidung würde insofern gut in das Konzept der Vorlesung passen, da sie Schritt für Schritt eingeführt und erklärt werden kann. Komplexere Hazard-Vermeidungen müssten wieder als Black-Box implementiert werden, ohne dass den Studierenden detailliert vermittelt werden kann, wie sie funktioniert.

Die Hazard-Erkennung beim Micro16 wäre im Allgemeinen verhältnismäßig simpel. Wie zuvor erwähnt, können beim Micro16 nur Control Hazards, eine einzige Art von Data Hazard – nämlich der Read-After-Write-Hazard – sowie ein struktureller Hazard im Zusammenhang mit wr und rd auftreten. Für die Erkennung von Control Hazards sind einzig die beiden COND-Bits in MIR1 wichtig. Diese können insgesamt vier Kombinationen annehmen, von denen zwei immer einen Control Hazard bedeuten. Die anderen beiden sind sicher:

- 00 (kein Sprung): Bei diesem Wert kann kein Control Hazard auftreten.
- 01 (Sprung wenn N=1): Bei diesem Wert kann ein Control Hazard auftreten, die Pipeline muss verzögert werden.
- 10 (Sprung wenn Z=1): Auch bei diesem Wert kann ein Control Hazard auftreten und die Pipeline muss verzögert werden.
- 11 (unbedingter Sprung): Bei diesem Wert tritt kein Control Hazard auf, aber der MIC muss für die nächste Phase schon auf die Adresse des unbedingten Sprunges gesetzt werden.

Somit genügt für die Erkennung von Control Hazards der Vergleich von zwei Bits.

Die Erkennung eines Write-Before-Read-Hazards ist etwas komplexer. Diese muss nämlich sowohl die eigentlichen Register (mit Ausnahme der Konstantenregister) als auch das MBR schützen. Um einen Hazard auf den eigentlichen Registern zu erkennen, sind folgende Regeln notwendig:

- Wenn im MIR1 einer der Werte im A-Bus oder B-Bus größer ist als 0x2 (also, wenn wenn einer der Werte nicht auf ein Konstantenregister zeigt), dann werden diese Werte mit dem Wert am S-Bus aus MIR2, MIR3 und MIR4 verglichen. Stimmt A-Bus und/oder B-Bus mit S-Bus überein, dann tritt ein Data Hazard auf und der Wert von MIR1 wird mit NOP ersetzt und die eigentlich auszuführende Instruktion verzögert.
- Ähnliches gilt für das MBR: Ist das Bit A-MUX (dies bedeutet, dass das Arbeitsregister A mit dem Wert von MBR befüllt wird) aus MIR1 gesetzt, so wird überprüft, ob in MIR2, MIR3 oder MIR4 das Bit MBR (dies bedeutet, dass der Inhalt des Arbeitsregisters S nach MBR geschrieben wird) gesetzt ist. Ist das der Fall, dann tritt ein Data Hazard auf, und auch hier wird dann der Wert von MIR1 mit NOP ersetzt und die eigentlich auszuführende Instruktion verzögert.

Um den strukturellen Hazard zu erkennen, muss man überprüfen, ob in MIR1 sowie zumindest einem der anderen MIR-Registern MS gesetzt ist und sich RD/WR in diesem Register von RD/WR in MIR1 unterscheidet.

Die Abbildungen 4.3 und 4.4 veranschaulichen die Ausführung der Hazard-Dectection.

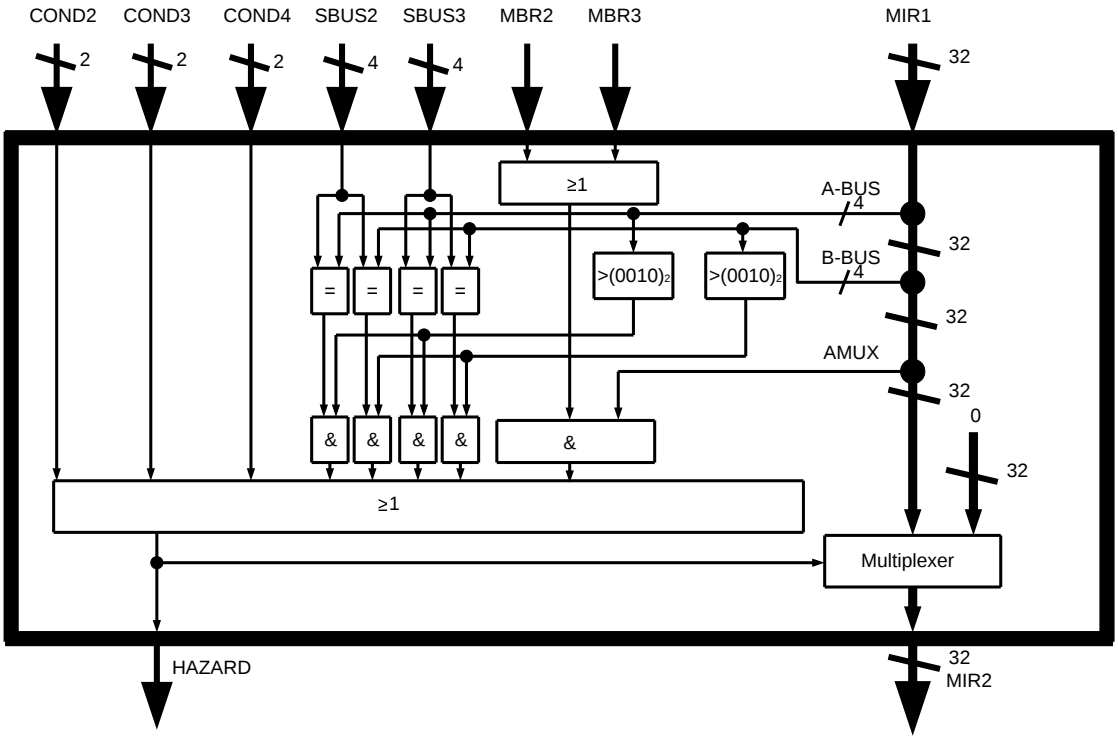

Abbildung 4.3: Hazard-Detection des Micro16

4.2 Caching

Auch Caching lässt sich in den Micro16 integrieren, da der Großteil des Cachings von einem dedizierten Bauteil (dem Cache) übernommen wird. In diesem Kapitel geht es nur um die Veränderungen, die für die Anbindung des Caches an den Micro16 umgesetzt werden müssen. Auf die Spezifika des Caches selbst wird nur soweit eingegangen, wie sie für die Änderungen am Micro16 relevant wären.

Es wird hierbei von einem autonomen und transparenten Cache ausgegangen, was bedeutet, dass es für den Prozessor keinen Unterschied macht, ob die Daten, die er lädt, sich im Cache befinden oder nicht. Die Ansteuerung ist hierbei gleich.

Da der Micro16 weder mehrere Prozessorkerne noch Interrupts oder andere Formen der Nebenläufigkeit unterstützt, muss der Cache nicht gegen Inkonsistenzen, die sich daraus ergeben könnten, abgesichert werden. Das Schreiben in den Hauptspeicher ändert sich somit nicht. Der Prozessor schickt den zu speichernden Wert an den Cache, so wie er ihn sonst an den Hauptspeicher geschickt hätte. Sobald der Cache diesen Wert empfangen hat, speichert er ihn ab und schreibt ihn weiter an den Hauptspeicher. Das kann er im Hintergrund tun, weswegen der Prozessor direkt weiter arbeiten kann. Sinnvollerweise würde man diesen Cache im (theoretischen) Optimalfall so designen, dass er nur einen Prozessortakt benötigt, um den Wert vom Prozessor in Empfang zu nehmen. Damit

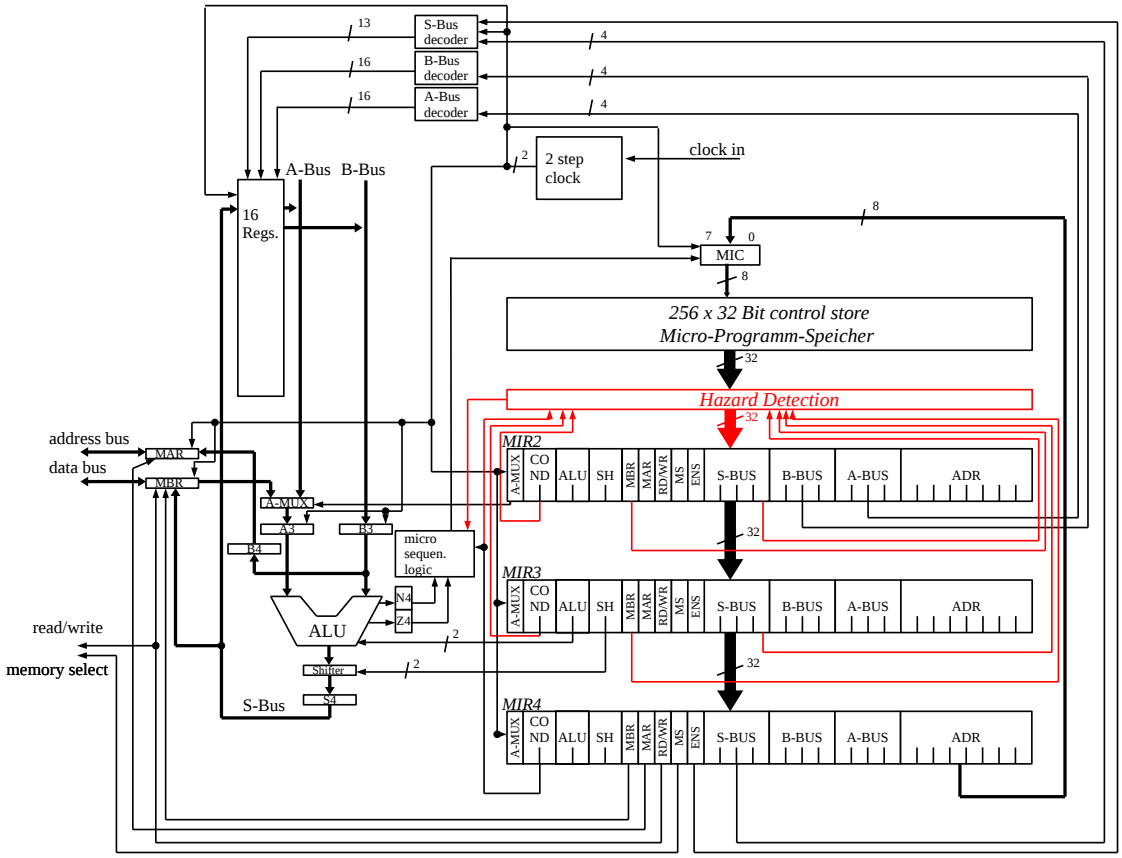

Abbildung 4.4: Einbettung der Hazard-Detection in den Micro16 (Erweiterungen zu Abbildung 4.2 sind Rot dargestellt)

benötigt der Prozessor in diesem Fall nur einen einzigen Takt, um den Befehl „wr“ auszuführen. Der Prozessor bleibt dabei immer noch Code-kompatibel zum originalen Micro16, da ein zweimaliges Schreiben desselben Wertes keinen Unterschied macht.

Für das Lesen aus dem Cache wird eine Änderung am Prozessor notwendig. Sollte der zu lesende Wert sich im Cache befinden, so kann der Wert schon innerhalb eines Prozessortaktes dem Prozessor zur Verfügung gestellt werden. Sollte er jedoch nicht verfügbar sein, so muss der Prozessor gegebenenfalls mehrere Taktzyklen warten, bis der Cache ihm signalisiert, dass der Wert verfügbar ist. Zu diesem Zweck wird eine neue Leitung mit dem Namen „Cache Read Ready (CRR)“ eingeführt. Der Wert dieser Leitung bedeutet, dass der angeforderte Wert sich im Cache befindet und vom Cache in das Register MBR geschrieben wurde. Dazu wird diese Leitung vom Cache auf 0 gesetzt, sobald der Micro16 den Befehl rd ausführt, sofern sich der angeforderte Wert nicht im Cache befindet. Der Cache lädt dann den Wert aus dem Hauptspeicher nach, schreibt ihn ins MBR und setzt den Wert von CRR auf 1. Befindet sich der Wert allerdings schon im Cache, so lädt der Cache den Wert direkt ins MBR und setzt CRR auf 1.

Der Prozessor muss – sofern CRR auf 0 steht und MBR als Quelle für den Befehl, der sich aktuell in der zweiten Phase befindet, verwendet wird – blockieren, solange bis CRR wieder auf 1 steht und somit signalisiert, dass das MBR einen gültigen Wert enthält. Am einfachsten ließe sich das umsetzen, indem diese Bedingung in die Hazard-Detection inkludiert wird. Dabei müsste die Hazard-Detection die Pipeline solange verzögern, wie der Wert des Feldes A-MUX (der anzeigt, dass das MBR als Quellregister für das Arbeitsregister A verwendet wird) auf 1 ist und gleichzeitig der Wert von CRR 0 ist. Auf diese Weise muss der Prozessor auch nicht blockieren, sofern MBR nicht gelesen wird.

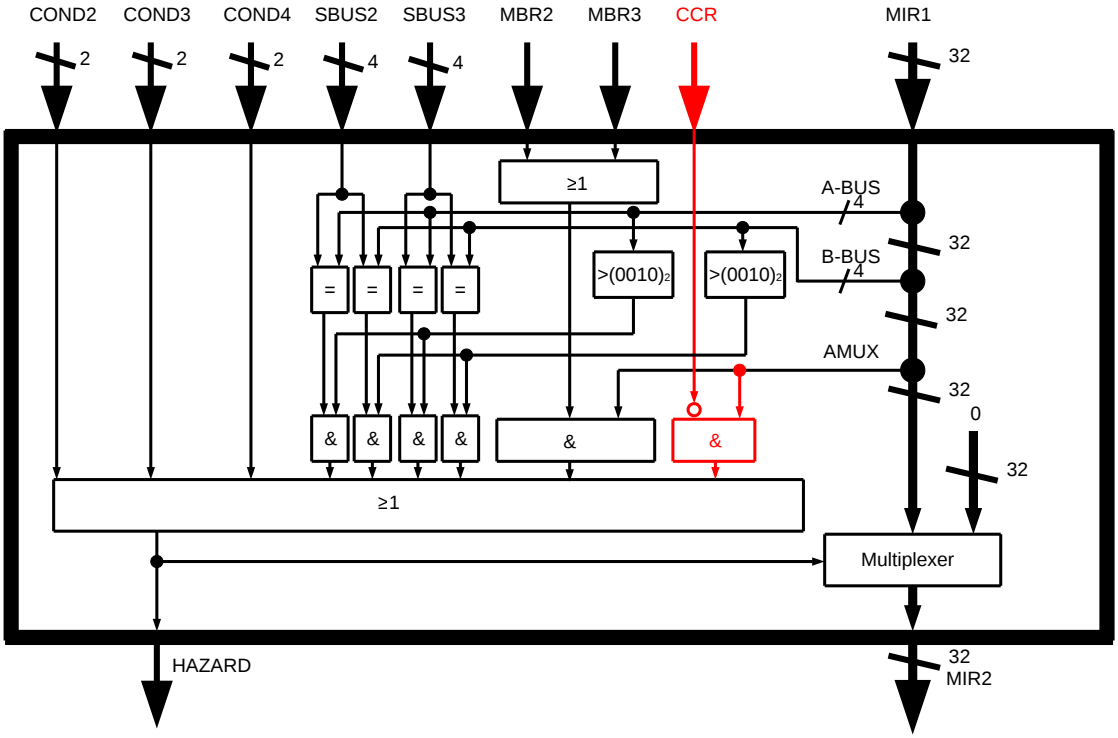

Abbildung 4.5: Erweiterung der Hazard-Detection, um mit Caching umgehen zu können (Erweiterungen in Rot dargestellt)

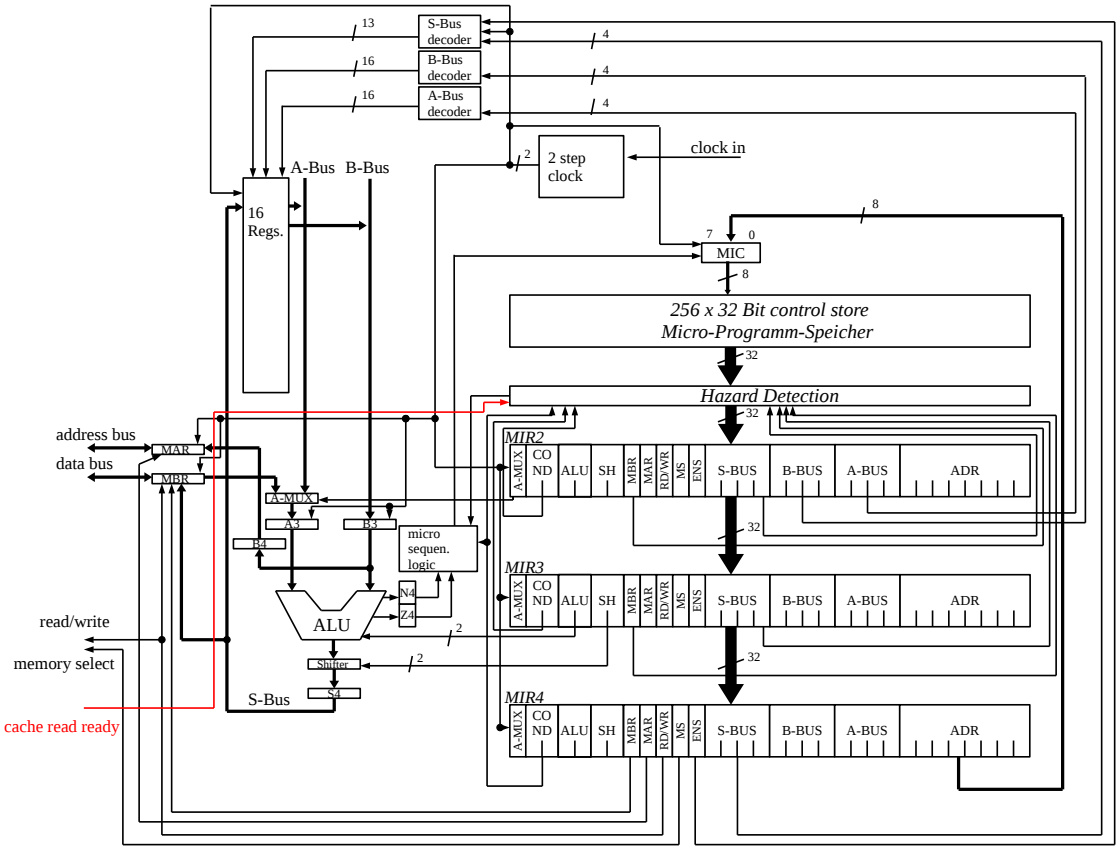

Abbildung 4.6: Micro16 mit Pipelining, Hazard-Detection und Caching (Rot dargestellt)

4.3 Mikrocode zur Implementierung einer Stackmaschine

In der Vorlesung wird der Mikrocode aktuell nur verwendet, um kleine Programme zu schreiben. Bei real verwendeten Prozessoren dient der Mikrocode hingegen dazu, erweiterte Befehle oder gar vollständig andere Prozessorarchitekturen zu implementieren. Ein Beispiel dafür sind Intels x86-Prozessoren: Schon seit dem Pentium Pro (P6-Mikroarchitektur) verwendet Intel keine echten x86-Kerne mehr, sondern stattdessen RISC-Prozessoren, deren Mikrocode x86-Code interpretiert. Das erlaubt einfachere und schnellere Prozessoren, die gleichzeitig die Kompatibilität mit weitverbreiteten x86-Architektur wahren. Auf diese Weise ist es auch möglich, durch ein einfaches Firmware-Update, bestehende Prozessoren zu verbessern, indem der Mikrocode angepasst wird. [Sch05b, 143f]

So wie A. Tanenbaum in seinem Buch „Structured Computer Organization“ auf seinem Beispielprozessor, den MIC-1, eine Stack-basierte Architektur – die IJVM – implementiert, wird in diesem Kapitel eine Architektur für eine Stackmaschine vorgestellt, die auf dem Micro16 implementiert werden kann. [Tan06, 246ff]

Im Rahmen der Vorlesung und der Übung wird eine Stackmaschine vorgestellt, die zwar

eine ähnliche Notation verwendet, wie der Micro16, aber gleichzeitig einen völlig anderen Befehlssatz besitzt. Auf diese Weise kann man die Beispiele auf dem Micro16 ausführen. Da im Laufe der Vorlesung und der Übung mehrere verschiedene Stackmaschinen gezeigt werden, die sich alle leicht von einander unterscheiden, wird hier eine Stackmaschine definiert, die die Fähigkeiten all dieser Stackmaschinen umfasst.

Da der Fokus dieser Maschine eindeutig am Stack liegt, besitzt sie nur ein einziges Register, das Register AC. Dieses Register wird einzig als Zwischenspeicher verwendet. Der Rest der Datenspeicherung geschieht auf dem Stack. Dieser beginnt am unteren Ende des Hauptspeichers, bei der Adresse 0xFFFF, und wächst nach unten. Das bedeutet, dass der erste Wert, der auf den Stack gelegt wird, an die Adresse 0xFFFF geschrieben wird. Der zweite Wert kommt dann an die Adresse 0xFFFE, der dritte auf 0xFFFD und so weiter. Der Stack-Pointer (R10) zeigt dabei immer auf den obersten Wert, der sich auf dem Stack befindet. Ist der Stack leer, so zeigt der Stack-Pointer auf die Adresse 0.

Der Mikrocode des Micro16 läuft in einer Schleife, in der nacheinander die einzelnen Befehle in R0 eingelesen und ausgeführt werden. Jeder der Befehle liegt als Opcode im Hauptspeicher, beginnend an der Adresse 0x0000. Die Befehle werden von oben nach unten ausgeführt. Die Stackmaschine unterstützt die in Tabelle 4.1 angeführten Befehle.

Um die Opcodes zu interpretieren, gibt es am Anfang des Mikrocodes einen Auswahl-Teil: Dabei wird der ausgelesene Opcode mit den möglichen Opcode-Werten verglichen. Stimmt der Opcode mit einem bekannten Opcode überein, so springt die Ausführung an die passende Stelle im Programm, die für die Ausführung dieses Opcodes zuständig ist. Ist die Ausführung des Opcodes beendet, dann wird an den Anfang der Hauptschleife gesprungen und der nächste Opcode geladen.

Um die Stackmaschine einfach zu halten, werden ungültige Opcodes als NOP interpretiert. Ungültige Werte (wie z.B. ein Multiplikation mit mehr als 255 oder eine Division durch 0) werden nicht abgefangen sondern sorgen üblicherweise für falsche Resultate.

Da die Befehle sehr simpel aufgebaut sind, lässt sich der Code für die Stackmaschine sehr einfach per Hand schreiben. Ein Beispielprogramm dazu findet sich in Tabelle 4.2.

Der kommentierte Micro16-Code, der diese Stackmaschine implementiert, befindet sich im Anhang und kann direkt in Vorlesung und Übung eingesetzt werden.

Tabelle 4.1: Befehle der Micro16-Stackmaschine

Befehl	**Opcode**	**Beschreibung**
NOP	0x0000	Keine Aktion
POP	0x0001	Entfernt den obersten Wert vom Stack und schreibt ihn in das Register AC.
PEEK	0x0002	Schreibt den obersten Wert des Stacks in das Register AC ohne ihn vom Stack zu entfernen.
PUSH	0x0003	Legt den Wert des Registers AC auf den Stack.
SWAP	0x0004	Vertauscht die obersten beiden Werte auf dem Stack.
ADD	0x0005	Entfernt die obersten beiden Werte vom Stack, addiert sie mit einander und legt das Ergebnis auf den Stack.
SUB	0x0006	Entfernt die obersten beiden Werte vom Stack, subtrahiert vom ersten den zweiten Wert und legt das Ergebnis auf den Stack.
MULT	0x0007	Entfernt die obersten beiden Werte vom Stack, multipliziert sie miteinander und legt das Ergebnis auf den Stack. Die Eingabewerte dürfen sich dabei nur im Wertebereich 0-255 befinden.
DIV	0x0008	Entfernt die obersten beiden Werte vom Stack, dividiert den ersten durch den zweiten Wert und legt das Ergebnis auf den Stack. Der Divisor darf sich dabei nur im Wertebereich 1-255 befinden.
SET	0x0009	Liest die nächste Zeile nach der aktuellen Codezeile aus dem Hauptspeicher und schreibt ihren Wert in das Register AC. Danach wird der Program Counter (PC) um eins erhöht, um diese Zelle zu überspringen. So wird sie nicht als Opcode ausgewertet.
END	0x000A	Wird am Ende des Programms verwendet, um selbiges zu beenden. Das Programm bleibt dann in einer Endlosschleife stecken, um zu verhindern, dass der Prozessor weiterläuft und eventuell den Stack erreicht. Sollte am Ende jedes Programms verwendet werden.

Tabelle 4.2: Beispielprogramm: Division von 200 durch 12

Befehl	**Opcode**	**Kommentar**
SET	0x0009	Setzt AC auf den nachfolgenden Wert.
12	0x000C	Dieser Wert wird nur als Parameter für SET benötigt und wird nicht als Programmcode interpretiert.
PUSH	0x0003	Legt den Wert 12 aus dem Register AC auf den Stack.
SET	0x0009	Setzt AC auf den nachfolgenden Wert.
200	0x00C8	Dieser Wert wird nur als Parameter für SET benötigt und wird nicht als Programmcode interpretiert.
PUSH	0x0003	Legt den Wert 200 aus dem Register AC auf den Stack.
DIV	0x0008	Nimmt die obersten beiden Werte (200 und 12) vom Stack und dividiert den obersten Wert (200) durch den Zweitobersten (12). Das Ergebnis ohne Rest (16) wird oben auf den Stack gelegt.
END	0x000A	Beendet die Ausführung des Programmes an dieser Stelle.

KAPITEL 5

Implementierungen des Micro16

Im Laufe der letzten Jahre wurden im Rahmen dieser Arbeit für den Micro16 verschiedene Implementierungen erstellt. Die Motivation dazu war, den Micro16 für die Studierenden praktisch und angreifbar zu machen. Vor diesen Implementierungen mussten Micro16-Programme im Rahmen der Übung per Hand ausgeführt werden. Man musste die Programme auf Papier schreiben und Schritt für Schritt manuell nachrechnen, was für Werte in den Registern des Micro16 an welchem Zeitpunkt stehen würden. Dies war äußerst zeitaufwändig, fehleranfällig und wurde von den Studierenden vielfach als rein theoretisch kritisiert. Deswegen wurden mehrere – teilweise aufeinander aufbauende – Implementierungen des Micro16 erstellt, von denen die erste seit 2012 aktiv in Vorlesung und Übung eingesetzt wird. [Mat13, 72]

5.1 Micro16-Emulator

Die erste Implementierung waren ein Bytecode-Emulator sowie ein Assembler, die auf auf Kommandozeilen-Basis arbeiteten und die beide im Java implementiert wurden. Der Emulator war in der Lage, Micro16-Bytecode korrekt auszuführen. Er emulierte das gesamte Innenleben des Micro16 sowie den Hauptspeicher. Der Assembler war in der Lage, Programme in der symbolischen Notation des Micro16 in Bytecode umzuwandeln. Die Dekompilation wurde auf Wunsch der Vorlesungsleitung nicht veröffentlicht, da einige Übungsaufgaben bewusst die händische Dekompilation von Micro16-Bytecode behandelten. Diese Programme wurden nicht in der Übung oder Vorlesung verwendet, waren aber ein Proof-of-Concept für die nächste Entwicklungsstufe.

5.2 Micro16-SDK

Das Micro16-SDK – SDK steht für Software Development Kit – ist eine vollständige Entwicklungsumgebung für den Micro16. Zusätzlich zum Emulator und Assembler erhielt

das Micro16-SDK eine grafische Oberfläche. Über diese Oberfläche lässt sich Micro16-Code erstellen und assemblieren, sowie in den Emulator laden, der den Code ausführen kann. Es lässt sich auch der Zustand der Datenregister sowie des Hauptspeichers beliebig editieren. Abbildung 5.1 zeigt das Hauptfenster des Micro16-SDK.

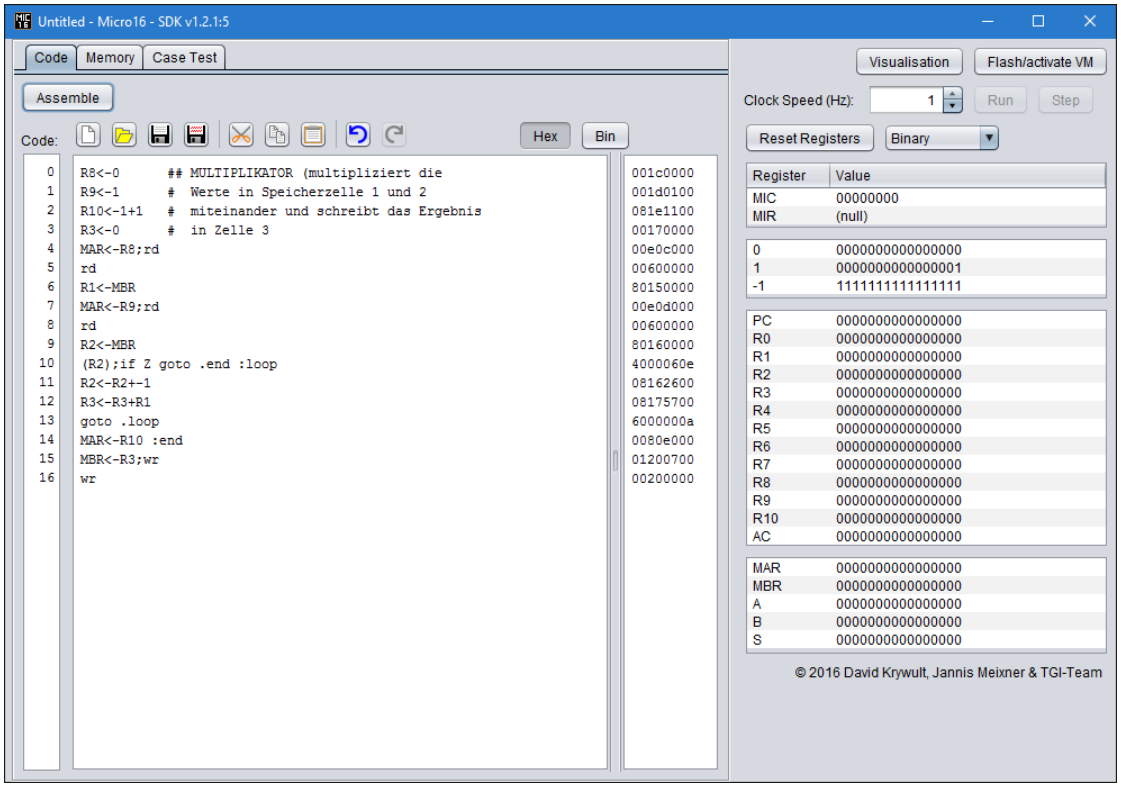

Abbildung 5.1: Hauptfenster des Micro16-SDK mit Code-Editor, assembliertem Hex-Code und Registeransicht

Da der Hauptspeicher des Micro16 mit 65536 Zeilen viel zu groß ist, um in der Hauptansicht unterzukommen, wurde die Hauptspeicheransicht auf einen eigenen Tab ausgelagert. In diesem Tab lassen sich alle Hauptspeicherzeilen binär, dezimal oder hexadezimal darstellen sowie modifizieren. Abbildung 5.2 zeigt die Ansicht des Hauptspeichers.

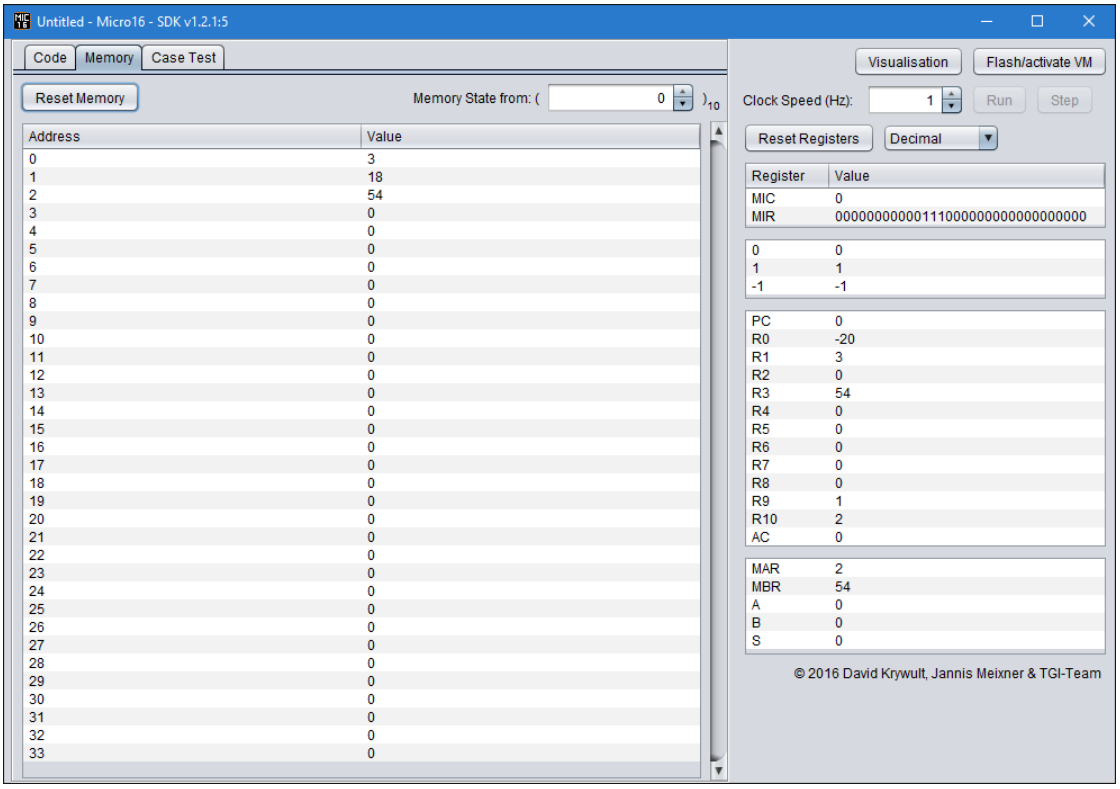

Abbildung 5.2: Hauptspeicheransicht des Micro16 in dezimaler Darstellung

Speziell für die Übung wurde auf Wunsch der Lehrveranstaltungsleitung ein Case-Test-System entwickelt, mit dem die Richtigkeit von Programmen fallgestützt überprüft werden kann. Dafür werden Case-Test-Dateien erstellt, die beliebig viele einzelne Case-Tests enthalten. Jeder dieser Tests hat einen Namen und kann Eingabe- und erwartete Ausgabewerte definieren. Wird ein Test gestartet, werden die Register und der Hauptspeicher des emulierten Micro16 mit den Eingabewerten befüllt. Dann wird das Programm ausgeführt, bis es das Ende des Programms erreicht. Danach werden die Register und der Hauptspeicher mit den erwarteten Ausgabewerten verglichen. Stimmen sie alle überein, dann gilt der Test als geglückt. Schlägt der Test fehl, dann wird ausgegeben, welche Register oder Hauptspeicherwerte nicht mit den erwarteten Ergebnissen übereinstimmen.

Durch Doppelklick auf einen einzelnen Test werden die Werte sofort in den Micro16 geladen. Der Test wird nicht direkt automatisch ausgeführt. Stattdessen kann man den Test wie normalen Code ausführen, was sinnvoll sein kann, um zu verstehen, wodurch der Fehler verursacht wird – hierfür bietet sich vor allem die schrittweise Ausführung an.

Die Studierenden können eigene Tests erstellen, die seitens der Lehrveranstaltungsleitung bewusst auch mit anderen Studierenden geteilt werden dürfen. Auf diese Weise können Studierende selbst überprüfen, ob ihre Lösungen korrekt sind.

In der Übung selbst wird durch die Übungsleitung überprüft, ob die Programme der Studierenden die zuvor festgelegten Testfälle korrekt abarbeiten. Das erleichtert die Arbeit im Übungsbetrieb enorm. Vor den Case Tests mussten die ÜbungsleiterInnen die Lösungen im Kopf bzw. per Hand nachrechnen, um herauszufinden, ob die Lösung Fehler aufwies. Dabei war es leicht, bei komplexeren Aufgaben Randfälle zu übersehen, die von anderen ÜbungsleiterInnern eventuell erkannt wurden. Mit den Case Tests wird der Code der Studierenden in das Micro16-SDK geladen, anschließend werden die Case Tests automatisiert ausgeführt. Das dauert, je nach Komplexität des Codes und Anzahl der Tests wenige Sekunden. Nach dieser kurzen Zeit ist sofort ersichtlich, welche Testfälle korrekt abgearbeiteten wurden und welche nicht. Fehlgeschlagene Tests können dann einzeln geladen werden, um Fehler genauer zu identifizieren.

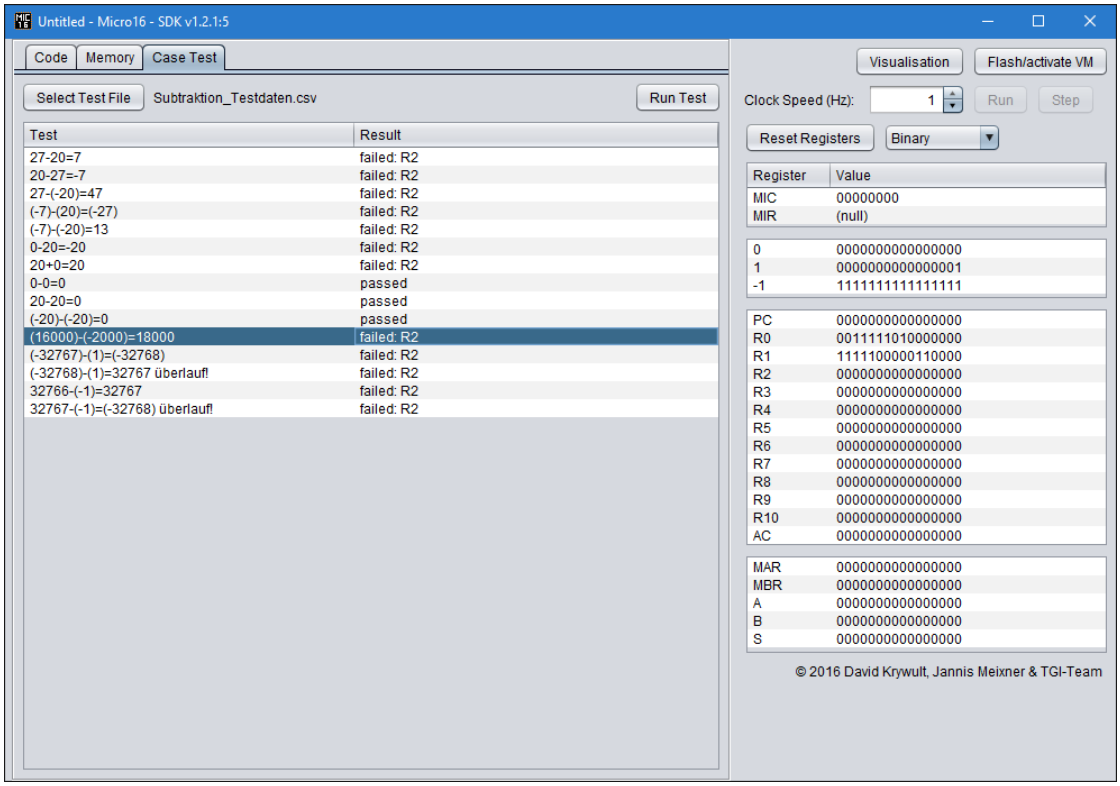

Abbildung 5.3: Case-Tests

Darüber hinaus wurde von Jannis Meixner eine Visualisierung erstellt, die für jedes Register die aktuellen Werte sowie alle aktiven Leitungen anzeigt. Diese Visualisierung verwendet das Blockschaltbild des Micro16, wie es in der Vorlesung präsentiert wird und erlaubt somit einen guten und übersichtlichen Einblick in die Funktionsweise des Micro16.

Dabei werden alle Teile des Micro16, die bei der Ausführung der aktuellen Codezeile aktiv sind, in einer gewünschten Farbe – in Abbildung 5.4 in der Farbe Rot – hervorgehoben.

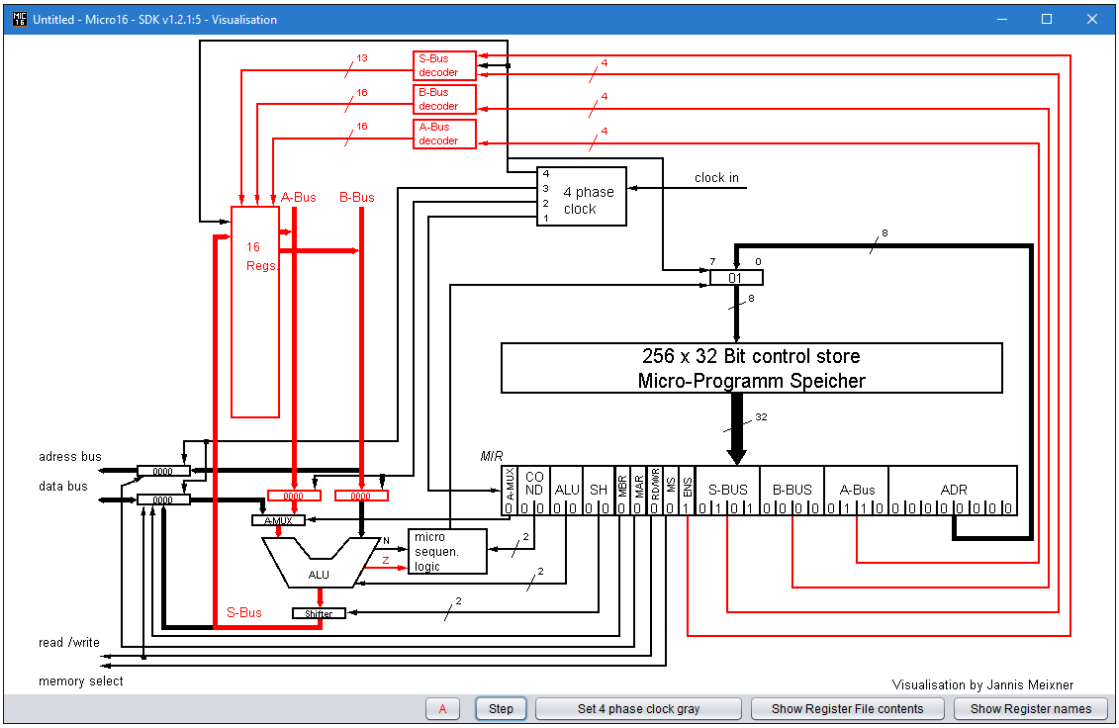

Abbildung 5.4: Visualisierung des Micro16

Um die Verwendung des Micro16-SDKs den Studierenden zu erleichtern, wurde von der Lehrveranstaltungsleitung unter Anderem eine Kurzbeschreibung erstellt. Diese befindet sich im Anhang unter dem Abschnitt 7.2.

In Vorbereitung auf die Implementierung des Micro16 auf einem FPGA wurden mehrere Prototypen von Eingabe- und Ausgabemöglichkeiten im Micro16-SDK erstellt. Anhand der Prototypen wurde ermittelt, welche Eingabe- und Ausgabemöglichkeiten am besten geeignet sind. Zwei dieser Prototypen sind in Abbildung 5.5 dargestellt.

Nach Erprobung von verschiedenen Eingabe- und Ausgabeschemata wurde eine finale Version erstellt, die sich auch an den Gegebenheiten auf dem DE1-Evaluierungsboard, das als Basis für die FPGA-Implementierung verwendet wurde, orientiert (vgl. Abbildung 5.6). Diese finale Version wurde dann auch auf dem FPGA implementiert. Sie wird den Studierenden im Rahmen der Vorlesung präsentiert und im Rahmen einer optionalen Praxis-Session interessierten Studierenden zur direkten Programmierung zur Verfügung gestellt. Eine genaue Erklärung der finalen Eingabe- und Ausgabemethoden findet sich im Kapitel 5.3.2.

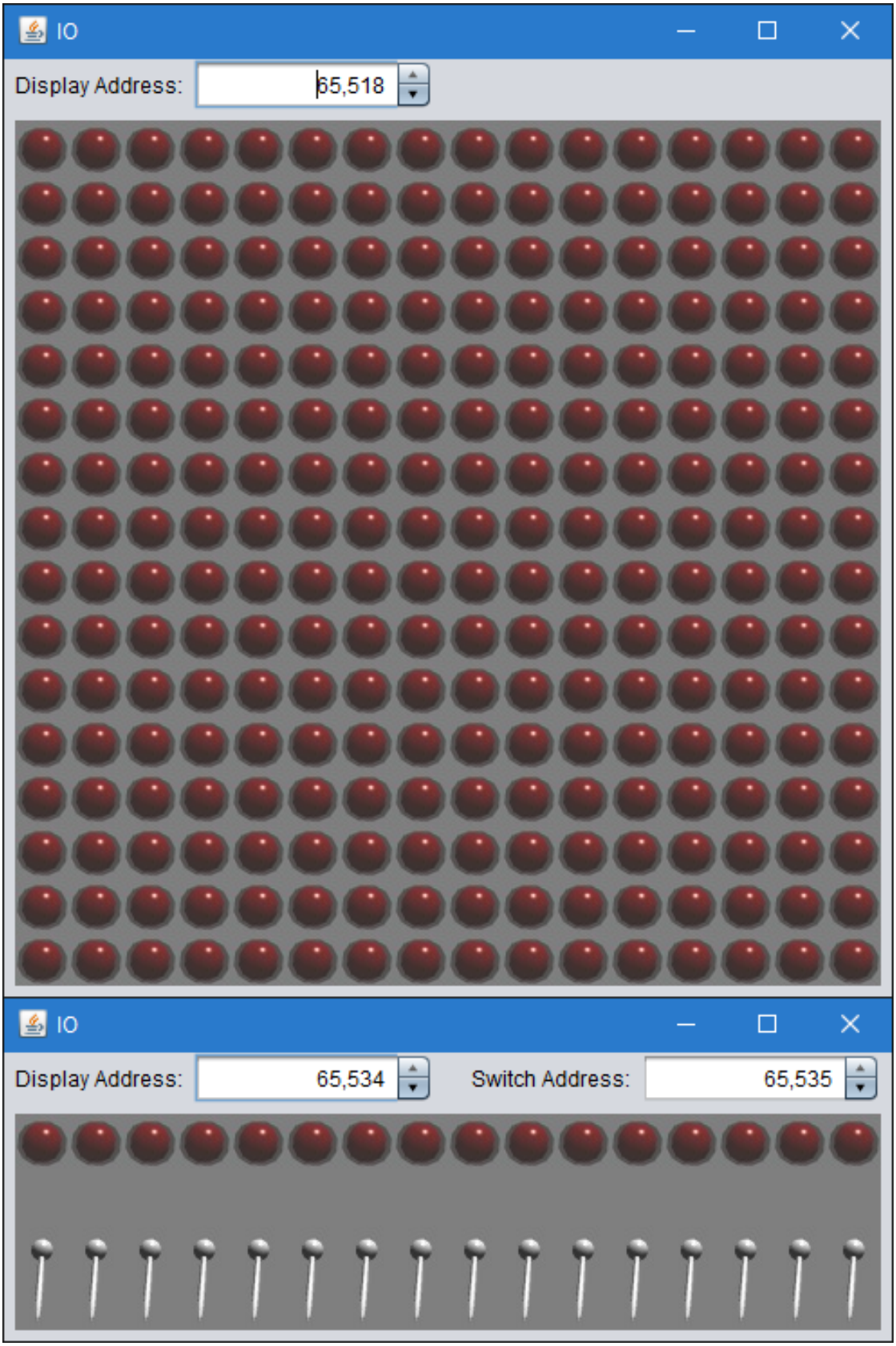

Abbildung 5.5: Zwei Prototypen von Eingabe- und Ausgabemöglichkeiten

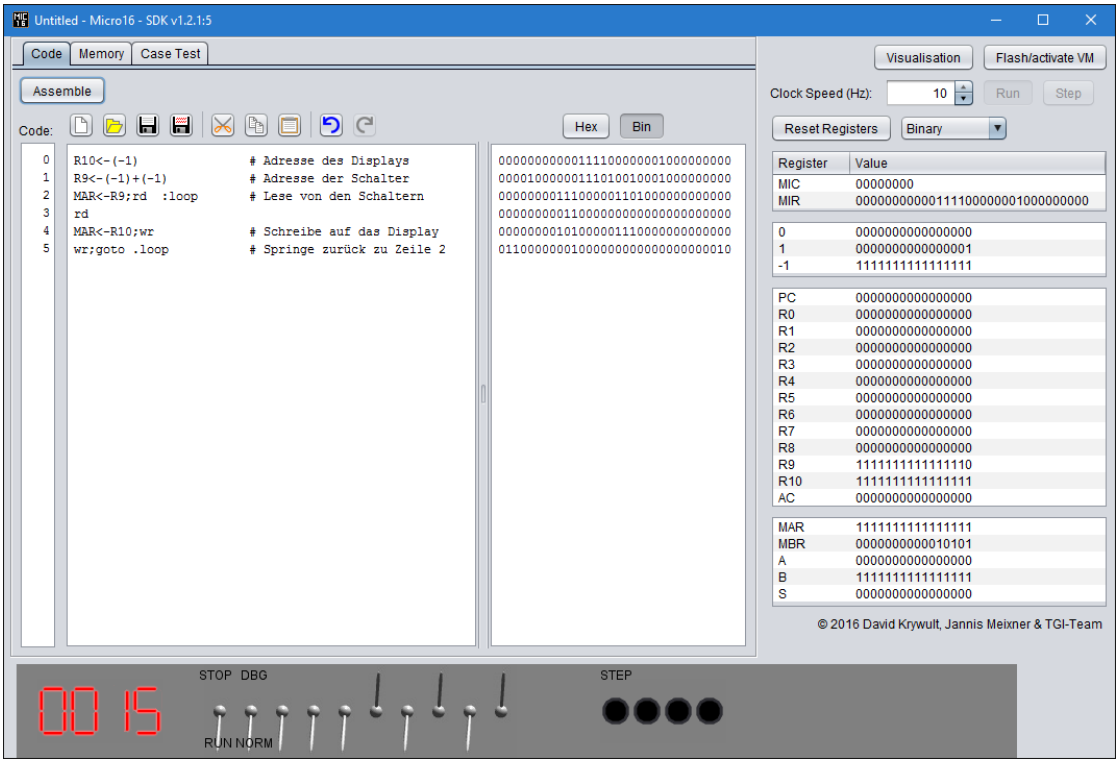

Abbildung 5.6: Finale Version der Eingabe- und Ausgabemöglichkeiten im Micro16-Simulator

5.3 Implementierung auf einem FPGA

Die Implementierung erfolgte mit den folgenden Zielen:

- Exakte Simulation des Innenlebens des Micro16 statt nur die Funktion zu emulieren
- Überprüfung, ob der Micro16 tatsächlich so funktioniert wie spezifiziert
- Erstellung eines Hardware-Objekts, das in der Vorlesung vorgeführt werden kann

Dazu wurde die gesamte Funktionsweise des Micro16 exakt in Verilog implementiert und auf einem DE1-Development Board von Altera umgesetzt.

5.3.1 Schaltkreis-Implementierung

Die Implementierung des Micro16 in Verilog war relativ einfach, da Verilog auf die Beschreibung von Hardware-Bauteilen spezialisiert ist. Verilog erlaubt es, Schaltnetze zu definieren, die unabhängig von einem Takt asynchron Berechnungen durchführen. Damit kann man z.B. die gesamte ALU als ein Schaltnetz definieren. Diesem Schaltnetz müssen

Eingangsvariablen übergeben werden, und nach kurzer Wartezeit kann man sich das Ergebnis abholen. Diese Schaltnetze führen ihre Berechnungen parallel und unabhängig von den anderen Schaltnetzen aus. Dazu verwendet man den Datentyp wire. Beispiele dafür sind die Implementierung der ALU sowie jene des Shifters der Micro16-Architektur.

Zuerst werden die wires definiert. Wires können eine beliebige Bit-Breite besitzen. Die wires amux, regSPreShift und regS sind 16 Bit breit, während die wires z und n jeweils nur ein Bit breit sind.

```
wire [15:0] amux;
wire [15:0] regSPreShift;
wire [15:0] regS;
wire z;
wire n;

assign amux = (MIRamux) ? mbr : registers[MIRregA];
```

Das wire amux beinhaltet entweder den Inhalt von MBR oder von Register A, je nach dem, was im MIR eingestellt ist.

```
assign regSPreShift = (MIRalu[1]==0) ?
    (
        (MIRalu[0]==0) ? amux : amux+regB
    ) : (
        (MIRalu[0]==0) ? amux&regB : ~amux
    );
```

Das wire regSPreShift beinhaltet den Output der ALU.

```
assign z = (regSPreShift == 0);
assign n = (regSPreShift[15] == 1);
```

Hier werden die Flags Z und N gesetzt, je nach dem, ob das Ergebnis null oder negativ ist.

```
assign regS = (MIRsh==2'b01) ? regSPreShift<<1 :
    (
        (MIRsh==2'b10) ? regSPreShift>>1 : regSPreShift
    )
```

In diesem letzten Schritt wird der Shifter ausgeführt, der das Ergebnis fertig stellt.

Diese wires arbeiten komplett asynchron, sind also vom Prozessortakt unabhängig. Dazu gibt es einen synchronen Code-Teil, der am Prozessortakt hängt. Alles was dieser Teil tun muss, ist, die aktuelle Codezeile aus dem Programmspeicher holen, sie in das Register MIR schreiben und am Ende die Ergebnisse wieder in die richtigen Register zurück

schreiben. Der Takt des synchronen Teils und somit des ganzen Prozessors ist im normalen Betriebsmodus auf 1 Hz gesetzt, damit man sehen kann, wie der Prozessor schrittweise arbeitet. Wird jedoch eine Beschleunigungstaste gedrückt, so wird der Prozessor auf 3,15 MHz hochgetaktet.

Auf diese Weise wurde der gesamte Micro16 in Verilog umgesetzt.

5.3.2 Eingabe und Ausgabe

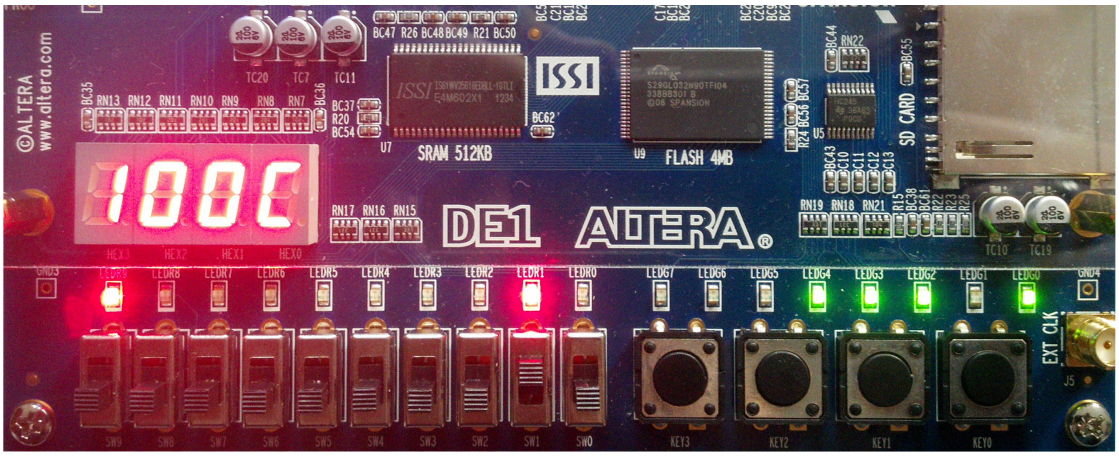

Abbildung 5.7: Eingabe/Ausgabe-Panel des Micro16 am DE1

Damit man mit dem Micro16 interagieren kann, wurden zusätzlich noch Ein- und Ausgabemöglichkeiten hinzugefügt, die bislang noch nicht vorhanden waren.

Um Ein- und Ausgabe realisieren zu können, ohne die Architektur des Micro16 verändern zu müssen, wurde Direct Memory Access verwendet. Das bedeutet, dass Peripherie-Geräte direkt auf den Hauptspeicher zugreifen können. Dabei bekommen die einzelnen Geräte verschiedene Hauptspeicherbereiche zugewiesen, die dann nicht mehr als normaler Hauptspeicher verwendet werden können. Stattdessen kann der Prozessor diese Bereiche verwenden, um direkt mit den Peripherie-Geräten zu kommunizieren.

Im konkreten Fall gibt es drei verschiedene Eingabe- und Ausgabegeräte: Das erste Gerät ist die vierstellige Siebensegmentanzeige am linken Rand von Abbildung 5.7. Diese Siebensegmentanzeige stellt den Wert an der Hauptspeicheradresse 0xFFFF als Hexadezimalwert dar. Was auch immer der Prozessor also an diese Stelle schreibt wird direkt auf der Anzeige ausgegeben.

Das zweite Gerät ist die Reihe an Schaltern am linken unteren Rand von Abbildung 5.7. Die linken beiden Schalter werden nicht an den Prozessor geleitet, sondern regeln den Debug-Modus, mit dem man z.B. den Prozessor pausieren kann, oder direkt Registerinhalte anzeigen kann. Die rechten acht Schalter hingegen werden bitweise an die Hauptspeicheradresse 0xFFFE geleitet. Das heißt, wenn der Prozessor diese Adresse ausliest, dann bekommt er immer den aktuellen Status der Schalter. Ist der Schalter

dabei auf der unteren Position, dann ist das Bit, das dem Schalter zugewiesen ist, 0. Ist er hingegen auf der oberen Position, so ist es 1. Der rechteste Schalter ist hierbei den niedrigstwertigsten Bit zugeordnet. Da hierfür nur acht Schalter zur Verfügung stehen, aber die Breite einer Hauptspeicherzelle 16 Bit beträgt, sind die höchstwertigsten acht Bit immer auf 0 gesetzt. Diese Adresse kann nur gelesen werden. Versucht der Prozessor diese Adresse zu überschreiben, so wird diese Aktion ignoriert.

Das dritte Ausgabegerät sind die Drucktaster am rechten unteren Rand von Abbildung 5.7: Hierbei sind die rechten drei Taster so wie die Schalter des zweiten Gerätes an die Speicheradresse 0xFFFD angebunden. Die Taste am weitesten links ist nicht an den Prozessor angebunden. Sie ist die Beschleunigungs-/Step-Taste und wird zur Steuerung der Ausführungsgeschwindigkeit des Prozessors verwendet. Ist der Prozessor pausiert (durch den Schalter am weitesten links vom zweiten Gerät), so führt ein Druck auf diesen Taster dazu, dass das Programm die nächste Instruktion abarbeitet (Step). Auf diese Weise können die Programmzeilen schrittweise abgearbeitet werden. Ist der Prozessor jedoch nicht pausiert, so wird er von 1 Hz auf 3,15 MHz hochgetaktet (Turbo). Auf diese Weise lassen sich Programme stark beschleunigen, was bei langen Programmen sinnvoll sein kann.

KAPITEL 6

Zusammenfassung und Ausblick

Ziel dieser Arbeit war es zunächst, zu überprüfen, ob der Micro16 nach wie vor die beste Wahl für einen Beispielprozessor für die Lehrveranstaltung „Technische Grundlagen der Informatik“ ist. Es hat sich nach eingehender Analyse herausgestellt, dass die größten Nachteile des Micro16 die fehlende Unterstützung von Pipelining und Caching sind. Die anderen Prozessoren, die dem Micro16 gegenübergestellt wurden – namentlich Atmel AVR, MIPS und Intel Skylake – waren vorwiegend aufgrund ihrer Komplexität weniger gut für die Lehrveranstaltung geeignet. Darum wurde entschieden, den Micro16 beizubehalten und dahin gehend zu erweitern, sodass er den Anforderungen entspricht.

Der Micro16 wurde zunächst um Pipelining, dann um eine Hazard-Detection erweitert. Darauf aufbauend folgte die Erweiterung um Caching. Der Aufbau dieser Erweiterungen folgt dem Stil der Vorlesung: So werden sie in dieser Arbeit detailliert und aufeinander aufbauend konstruiert. Dadurch sind die Erweiterungen leicht in die Lehrveranstaltung zu übernehmen. Aktuell wird in der Vorlesung für Pipelining die MIPS-Architektur verwendet und Caching wird ohne Bezug auf einen spezifischen Prozessor unterrichtet. Durch die Erweiterungen muss nicht mehr auf andere Prozessoren zurückgegriffen werden. Stattdessen kann der Micro16 für alle Themen der Lehrveranstaltung, die einen Beispielprozessor benötigen, herangezogen werden.

Eine weitere Verbesserung betrifft die Stackmaschinen, die in der Lehrveranstaltung verwendet werden. Aktuell werden mehrere verschiedene Stackmaschinen verwendet, die sich jeweils geringfügig voneinander unterscheiden. Von diesen Stackmaschinen existieren derzeit keine Implementierungen oder Simulatoren. Programmcode, den Studierende für diese Maschinen schreiben, muss immer noch per Hand gerechnet werden. In dieser Arbeit wurde, der Tradition von Tanenbaums Buch „Structured Computer Organization“ folgend, eine Stackmaschine in Micro16-Mikrocode implementiert. Diese Stackmaschine vereinigt die Fähigkeiten aller in der Lehrveranstaltung vorkommenden Stackmaschinen in sich und kann somit als direkter Ersatz für all diese Stackmaschinen verwendet werden. Somit werden Programme für diese Stackmaschine auf dem Micro16-SDK ausführbar.

Um den Micro16 auch für den Übungsteil besser einsetzbar zu machen, wurden eine Entwicklungsumgebung (Micro16-SDK) inklusive Simulator für den Micro16 entwickelt, die inzwischen seit mehreren Jahren in der Übung erfolgreich verwendet werden. Die Studierenden können darauf die Programme, die sie für die Übung schreiben, schrittweise ausführen und testen. Dadurch werden komplexere Programme ermöglicht, da die Studierenden die Programme nicht mehr von Hand ausführen müssen. Auch die Fehlersuche wird dadurch stark erleichtert, da die Studierenden sehen können, was genau ihr Code bewirkt. Eingebaute Debugging-Werkzeuge, wie ein Schritt-für-Schritt-Modus, mit dem das Programm Zeile für Zeile ausgeführt werden kann und eine Anzeige aller Register- und Speicherzellen-Zustände erlauben einen detaillierten Einblick auf die Auswirkungen von Codeänderungen. Die von Jannis Meixner eingebaute Visualisierung erlaubt zudem eine sehr übersichtliche Darstellung der Funktionsweise des Micro16.

Durch das Case-Testing-System des Simulators können Abgaben der Studierenden automatisch getestet werden. Dadurch müssen die Übungsleiterinnen und Übungsleiter die Abgaben nicht mehr manuell auf ihre Richtigkeit überprüfen, sondern können sich auf das Erklären der Lösung sowie der Fehler konzentrieren. Zudem erlernen die Studierenden im Rahmen der Übung fallbasiertes Testen von Software-Programmen.

Auch wurde eine Implementierung als IP-Core auf einem FPGA erstellt. Dadurch wurde die Korrektheit der Micro16-Architektur verifiziert. Dieser FPGA wird seit mehreren Jahren in der Vorlesungseinheit, in dem das Micro16-SDK vorgestellt wird, vorgezeigt. Der Hauptzweck dieser Präsentation ist, die Studierenden für den Micro16 zu begeistern. Das Interesse an der Implementierung und dem FPGA selbst war bislang jedes Mal sehr hoch.

Die praktischen Teile dieser Arbeit werden jetzt schon seit mehreren Jahren erfolgreich in der Lehrveranstaltung „Technische Grundlagen der Informatik“ eingesetzt. Das Feedback der Studierenden dazu ist sehr positiv. Im Wintersemester 2017 soll die Lehrveranstaltung überarbeitet und in ein neues Format gebracht werden. Die Ergebnisse dieser Arbeit könnten in diese Überarbeitung einfließen.

KAPITEL 7

Anhang

7.1 Stackmaschine in Micro16-Code

```
PC ← 0
AC ← 0
R10 ← 0
MAR ← PC;rd :decodeLine
R1 ← (-1);rd # R1: Opcode Counter
R0 ← MBR
(R0);if Z goto .NOP
(R0+R1);if Z goto .POP
R1 ← R1+(-1)
(R0+R1);if Z goto .PEEK
R1 ← R1+(-1)
(R0+R1);if Z goto .PUSH
R1 ← R1+(-1)
(R0+R1);if Z goto .SWAP
R1 ← R1+(-1)
(R0+R1);if Z goto .ADD
R1 ← R1+(-1)
(R0+R1);if Z goto .SUB
R1 ← R1+(-1)
(R0+R1);if Z goto .MULT
R1 ← R1+(-1)
(R0+R1);if Z goto .DIV
R1 ← R1+(-1)
(R0+R1);if Z goto .SET
R1 ← R1+(-1)
```

```
(R0+R1);if Z goto .END
goto .decodeLine :NOP # unknown Opcode or NOP

MAR ← R10;rd :POP
R10 ← R10+1;rd
AC ← MBR;goto .endLine

MAR ← R10;rd :PEEK
rd
AC ← MBR;goto .endLine

R10 ← R10+(-1) :PUSH
MAR ← R10;MBR ← AC;wr
goto .endLine;wr

MAR ← R10;rd :SWAP
R10 ← R10+1;rd
R8 ← MBR;MAR ← R10;rd
rd
R9 ← MBR
MBR ← R8;MAR ← R10;wr
R10 ← R10+(-1);wr
MBR ← R9;MAR ← R10;wr
goto .endLine;wr

MAR ← R10;rd :ADD
R10 ← R10+1;rd
R8 ← MBR;MAR ← R10;rd
rd
R8 ← R8+MBR
MBR ← R8;MAR ← R10;wr
goto .endLine;wr

MAR ← R10;rd :SUB
R10 ← R10+1;rd
R8 ← MBR;MAR ← R10;rd
rd
R9 ← ~MBR # invert R9
R9 ← R9+1
R8 ← R8+R9 # subtraction
MBR ← R8;MAR ← R10;wr
goto .endLine;wr
```

```
MAR ← R10;rd :MULT
R10 ← R10+1;rd
R8 ← MBR;MAR ← R10;rd #Multiplikand: R8
rd
R9 ← MBR       #Multiplikator: R9
R7 ← 0         #Result: R7
R6 ← lsh(1+1) #Helper: R6
R6 ← R6+R6
R9 ← lsh(R9+R9)
R9 ← lsh(R9+R9)
R9 ← lsh(R9+R9)
R9 ← lsh(R9+R9)
(R6);if Z goto .multEnd :multLoop
R7 ← lsh(R7)
(~R9);if N goto .mult0
R7 ← R7+R8
R9 ← lsh(R9) :mult0
R6 ← R6+(-1)
goto .multLoop
MAR ← R10;MBR ← R7;wr :multEnd
goto .endLine;wr

MAR ← R10;rd :DIV
R10 ← R10+1;rd
R8 ← MBR;MAR ← R10;rd #Divident: R8
rd
R9 ← ~MBR  #Divisor: R9
R7 ← 0     #Result: R7
R9 ← R9+(1)
R9 ← lsh(R9+R9)
R9 ← lsh(R9+R9)
R9 ← lsh(R9+R9)
R9 ← lsh(R9+R9)
R5 ← lsh(1+1) #Helper: R5
R5 ← lsh(R5+R5)
R5 ← lsh(R5+R5)
R5 ← lsh(R5+R5)
R6 ← rsh(-1) #Helper: R6
R6 ← ~R6
(R5);if Z goto .divEnd :divLoop
(R8+R9);if N goto .div0
R8 ← R8+R9
R7 ← R7+R5
```

```
R5 ← rsh(R5) :div0
R9 ← rsh(R9)
R9 ← R9+R6;goto .divLoop
MAR ← R10;MBR ← R7;wr :divEnd
goto .endLine;wr

PC ← PC+1 :SET
MAR ← PC;rd
rd
AC ← MBR;goto .endLine

goto .END :END

PC ← PC+1;goto .decodeLine :endLine
```

7.2 Befehlsreferenz Micro16

Micro16 - Befehlsreferenz

Version vom 28.04.2016
(Änderungen bzw. Erweiterungen vorbehalten!)

Nr.	Codierung	Symbolisch	Beschreibung	Beispiel
1	ALU = 00	R ← R	R unverändert durchschalten (A-Bus)	R5 ← R6; R5 ← 0; R5 ← 1; R5 ← -1
2	ALU = 01	R ← Rj + Rk	Wert in Rj und Rk addieren (linker Operand auf A-Bus)	R5 ← R6+R7
3	ALU = 10	R ← Rj ∧ Rk	Wert in Rj und Rk bitweise verknüpfen (linker Operand auf A-Bus)	R5 ← R5 ∧ R6; R5 ← R5 & R6;
4	ALU = 11	R ← ¬R	Wert in R bitweise negieren (A-Bus)	R5 ← ¬R5; R5 ← ~R5
5	SH = 01	R ← lsh(X*)	shift left	R7 ← lsh(R5)
6	SH = 10	R ← rsh(c)	shift right	R7 ← rsh(¬R5)
7	COND = 01	(X*); if N goto ADDR	Sprung, wenn N=1 ("negative"; Vorzeichenanzeige, überprüft *msb*)	(R5+R6); if N goto END
8	COND = 10	(X*); if Z goto ADDR	Sprung, wenn Z=1 ("zero"; Nullanzeige)	(R5); if Z goto START
9	COND = 11	goto ADDR	unbedingter Sprung	goto END
10	MAR = 1	MAR ← R	Memory Adress Register setzen (B-Bus)	MAR ← R7
11	A-MUX = 1	R ← MBR	Memory Buffer Register lesen (A-Bus)	R9 ← MBR
12	MBR = 1	MBR ← X*	Memory Buffer Register schreiben (S-Bus)	MBR ← R10
13	RD/RW = 1 MS = 1	rd	lesen (läuft über 2 Takte)	rd
14	RD/RW = 0 MS = 1	wr	schreiben (läuft über 2 Takte)	wr
		#	Kommentar	R5 ← lsh(1+1) # speichere 4 in R5

X*: Instruktionen 1-4 möglich

Anmerkung 1: Es sind Kombinationen von Befehlen möglich, sofern diese unterschiedliche Funktionseinheiten nutzen.
Beispiele: R5 ← lsh(R5 + R7); # Addiert die Werte in R5 und R7, führt anschließend einen Left-Shift durch
MAR ← R6; rd # Adressregister wird mit Wert in R6 beschrieben, Leseoperation vom Speicher wird gestartet

Anmerkung 2: Die Register R0 bis R10 bieten Lesezugriff (ENS = 0) und Schreibzugriff (ENS = 1).
Die Konstantenregister 0, +1 und -1 bieten nur Lesezugriff (ENS = 0). Wird trotzdem ein Schreibzugriff versucht, geht der geschriebene Wert verloren (die Konstante bleibt unverändert).

Abbildung 7.1: Befehlsreferenz Micro16

7.3 Micro16-SDK Kurzbeschreibung

Kurzbeschreibung Micro16-Simulator (Stand 28.04.2016)

Bedienung
Das Programm wird in symbolischer Notation in das Feld mit der Überschrift „Code“ eingegeben. Das Programm wird anschließend via „Assemble“ assembliert, bevor es via „Flash/activate VM“ auf den virtuellen Micro16 geflasht wird. Im Anschluss kann das Programm mit einer bestimmten Verarbeitungsgeschwindigkeit („Run“ mit „Clock Speed“) oder Schritt für Schritt („Step“) ausgeführt werden.

Kommentare
Alles nach einer Raute (#) wird als Kommentar gewertet und vom Assembler ignoriert.
Zeilen, die nur Kommentare enthalten, werden (vom MIC) ebenfalls mitgezählt!

Syntax

	Lehrbuch/Folien	Simulator
Zuweisung	←	<-
Logisches UND/AND	∧	&
Logische Negation/NOT	¬	~
Dekrementierung (-1)	R1 ← R1 – 1	R1<-R1+(-1)
Symbolische Sprungziele mit Bezeichnung/Label	Goto end end:	Goto .end :end

Zahlendarstellung
Zahlen werden im Simulator in Zweierkomplementdarstellung interpretiert.

Sprünge
Bei Sprüngen ist die als Sprungziel angegebene Instruktion noch inkludiert. Beispielsweise wird beim Sprung an .loop die Zuweisung an das MAR noch durchgeführt:
MAR←R3 :loop

Register
Der Zugriff auf die Register erfolgt grundsätzlich analog zum Lehrbuch/den Folien.
Die Register PC und AC können wie die Allzweckregister R0 bis R10 verwendet werden.
Die Adressierung der Register ist wie folgt festgelegt:

Adresse	Register
0000	Konstante 0
0001	Konstante +1
0010	Konstante -1
0011	PC
0100	R0
…	…
1110	R10
1111	AC

Initialisierung
Speicher und Allzweckregister können vor „Flash/activate VM“ initialisiert werden.

Abbildung 7.2: Micro16-SDK Kurzbeschreibung, Seite 1

Testfälle
Für automationsgestütztes Testen steht der Reiter „Case Test“ zur Verfügung. Über „Select Test File“ kann ein Testdatensatz geladen werden, die einzelnen Testfälle müssen wie folgt aufgebaut sein:

# Name/Beschreibung Testfall	z.B:	# 27-20=7 in R2 & Speicheradr. 10
Input,PC,R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,AC		Input,0,11011,10100
InputMemory, Adresse, Wert; Adresse, Wert		InputMemory, 0x000A, 0x0000
Output,PC,R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,AC		Output,*,*,*,111
OutputMemory, Adresse, Wert; Adresse, Wert		OutputMemory, 0x000A, 0x0007

Bei jeder Werteangabe werden automatisch führende Nullen (auf 16 Bit) ergänzt.
Nicht angegebene Register werden bei der Eingabe (Input) automatisch auf 0 gesetzt, bei der Ausgabe (Output) mit * (= beliebiger Wert wird akzeptiert).
Werte können in binärer oder hexadezimaler Darstellung (Präfix: 0x) angegeben werden.

Durch Doppelklick auf einen Testfall kann dieser auf die VM geladen (Register/Speicheradressen gemäß Input/InputMemory gesetzt) und im Anschluss direkt ausgeführt werden.

Für das Erstellen bzw. Editieren von Testdatensätzen (Dateityp .csv) empfiehlt sich die Verwendung eines simplen Texteditors.

Visualisierung
Über den Button „Visualisation“ kann man in einem neuen Fenster eine grafische Darstellung des Micro16 öffnen. In der Visualisierung kann bei Registern zwischen der Anzeige der Registernamen („showRegisterNames“) bzw. der Anzeige der Registerinhalte in hexadezimaler Darstellung („showRegisterValues“) umgeschaltet werden. Über den Button „Show Register File contents“ können die Register des Register Files und deren Inhalte detaillierter dargestellt werden.
Wird ein Programm ausgeführt, werden die durch die jeweilige Instruktion aktiven Leitungen bzw. Funktionseinheiten rot dargestellt, inaktive (bzw. auch daueraktive, wie z.B. im Zusammenhang mit dem Nachladen des MIR) schwarz. Die einzelnen Phasen der 4-Phase Clock werden nicht unterschieden.
Über den Button „Step“ im Visualisierungsfenster kann die Programmausführung – analog zum Button „Step“ im Hauptfenster – schrittweise ausgeführt werden.

Hinweis: Die Unterscheidung von aktiven und inaktiven Leitungen ist nicht immer ganz eindeutig zu treffen und Bedarf eines tieferen Blickes in die Funktionsweise des Micro16: So sind eigentlich auch bei einer reinen Kommentarzeile A- und B-Bus Decoder aktiv. In weiterer Folge liegt die Konstante 0 an A- und B-Bus an, ALU und Shifter schalten anschließend die Konstante 0 vom A-Bus unverändert durch.
Ziel der Visualisierung ist es, die Funktionsweise des Micro16 möglichst verständlich aufzuzeigen. Sofern Sie Verbesserungsvorschläge hinsichtlich der aktiv/inaktiv-Darstellung haben, teilen Sie uns diese bitte im Entwicklerforum mit. Für zweckdienliche Vorschläge werden Mitarbeitsplus vergeben!

Abbildung 7.3: Micro16-SDK Kurzbeschreibung, Seite 2

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[BGvN46] A. Burks, H. Goldstine, and J. von Neumann. *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument.* Ablex Publishing Corp., Norwood, 1946.

[Cle06] A. Clements. *Principles of Computer Hardware.* Oxford University Press Inc., New York, 2006.

[Cor11] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual volume 1: Basic architecture. Website, 2011. Online erhältlich unter `http://download.intel.com/design/processor/manuals/253665.pdf`; abgerufen am 12. November 2016.

[Cor15] Atmel Corporation. Atmega48a/pa/88a/pa/168a/pa/328/p datasheet. Website, 2015. Online erhältlich unter `http://www.atmel.com/images/atmel-8271-8-bit-avr-microcontroller-atmega48a-48pa-88a-88pa-168a-168pa-328-328p_datasheet_complete.pdf`; abgerufen am 8. November 2016.

[Cor16] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1. Website, 2016. Online erhältlich unter `https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html`; abgerufen am 15. November 2016.

[HH10] D. M. Harris and S. L. Harris. *Digital Design and Computer Architecture.* Morgan Kaufmann, Burlington, 2010.

[Hof13] D. Hoffmann. *Grundlagen der Technischen Informatik.* Hanser Verlag München, München, 2013.

[Mat13] J. Matiasch. Übungskonzept für Technische Grundlagen der Informatik. Master's thesis, Technische Universität Wien, 2013.

[MT08] Inc. MIPS Technologies. Mips32® m4k® processor core datasheet. Website, 2008. Online erhältlich unter `https://`

imagination-technologies-cloudfront-assets.s3.amazonaws.com/documentation/MD00247-2B-M4K-DTS-02.01.pdf; abgerufen am 18. November 2016.

[MT09] Inc. MIPS Technologies. Mips32® architecture for programmers volume ii: The mips32® instruction set. Website, 2009. Online erhältlich unter https://ti.tuwien.ac.at/cps/teaching/courses/cavo/files/MIPS32-IS.pdf; abgerufen am 24. Oktober 2016.

[PBLS16] W. J. Paul, C. Baumann, P. Lutsyk, and S. Schmaltz. *System Architecture an Ordinary Engineering Discipline*. Springer, Cham, 2016.

[PH05] D. Patterson and J. Hennessy. *Computer Organization And Design*. Morgan Kaufmann Publishers, Burlington, 2005.

[PH07] D. Patterson and J. Hennessy. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Burlington, 2007.

[RR98] S. Robins and K. A. Robins. A microprogramming animation. *IEEE Transactions on Education*, Vol. 41 No. 4:293–300, 1998.

[Sch05a] W. Schiffmann. *Technische Informatik 2: Grundlagen der Computertechnik*. Springer-Verlag, Hagen, 2005.

[Sch05b] G. Schildt. *Einführung in die Technische Informatik*. Springer-Verlag, Wien, 2005.

[Shi08] S. G. Shiva. *Computer Organization, Design, and Architecture*. CRC Press, New York, 2008.

[Tan06] A. S. Tanenbaum. *Structured Computer Organization*. Pearson Prentice Hall, Upper Saddle River, 2006.