FAKULTÄT FÜR !INFORMATIK

Selbsterklärender, plattformunabhängiger ALU-Simulator für die universitäre Lehre

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Markus Rossler
Matrikelnummer 9525750

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Em. Univ.Prof. Dipl.-Ing. Dr.techn. Richard Eier
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Friedrich Bauer

Wien, 09.09.2008 _____ _____
(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Kurzfassung

Die vorliegende Arbeit behandelt die Erstellung eines **MC8 ALU**-Simulators. Der *MC8* ist ein Modellcomputer, der am *Institut für Computertechnik (ICT)* der *Technischen Universität Wien* für die Lehre entwickelt wurde. Die *Arithmetisch Logische Einheit (engl. arithmetic logic unit, ALU)* ist ein Teil des Prozessors und ist unter anderem für arithmetische und logische Operationen zuständig.

Das Ziel dieser im World Wide Web verfügbaren *Adobe-Flash*-Applikation ist, die technischen Zusammenhänge sowie die zeitlichen Abläufe der *ALU* innerhalb der Hardware leicht verständlich zu visualisieren.

Zunächst wird auf Methoden moderner Wissensvermittlung und die Bedeutung von E-Learning eingegangen. Die folgenden Abschnitte befassen sich mit den fachlichen Grundlagen der Inhalte, soweit es für das Verständnis der vorliegenden Arbeit erforderlich ist.

Grundlegende elektronische Schaltungen werden beschrieben, die dazu genutzt werden können, arithmetische oder aussagenlogische Funktionen auszuführen. Essentiell bei der Erörterung der Grundlagen sind die Zustandsspeicher, die als Basis für Schaltwerke dienen. Diese können ausgehend von einem aktuellen Zustand und Eingabewerten einen Nachfolgezustand ermitteln und Ausgabewerte bzw. Steuersignale produzieren. Diese Eigenschaften werden benötigt, um Steuerwerke zu erzeugen, die das Verhalten einer *ALU* steuern. In dieser Einheit werden die zuvor gezeigten grundlegenden Schaltungen sowie einige zustandsabhängige Operationen durchgeführt. Das ist Gegenstand der Softwareimplementierung dieser Diplomarbeit.

Im Weiteren wird auf die Vereinfachungen des Modells und auf Kompromisse zugunsten der Verständlichkeit der Darstellung eingegangen. In Folge geht die Arbeit auf die technische Plattform ein. Sowohl ein kurzer geschichtlicher Abriss der Entwicklung von Programmiersprachen als auch die Kriterien für die Auswahl der *Adobe Flash*-Plattform werden erläutert. Ergebnisse der Recherche, welche Projekte ähnliche Ziele verfolgen, schließen den ersten allgemeinen Teil der Arbeit ab.

Darauf folgt eine Analyse, welche Ziele und Nichtziele die Applikation für die ermittelten Zielgruppen erfüllen soll, sowie ein Vergleich mit bisher vom *ICT* im Rahmen von Diplomarbeiten erarbeiteten E-Learning Simulatoren. Im weiteren Verlauf der Arbeit werden die Details und Abläufe der umgesetzten Berechnungs- und Logikoperationen dargestellt und es wird auf programmtechnische Besonderheiten bei der Umsetzung behandelt. Um eine konsistente Ansicht für alle Darstellungen von Befehlen zu ermöglichen wurden Standards erarbeitet, welche auch erläutert werden.

Danach erfolgt eine detaillierte Beschreibung der realisierten Operationen samt genauer Beschreibung der Abläufe. Diese Auflistung ist sehr umfangreich, da insgesamt zwölf Operationen in jeweils zwei unterschiedlich komplexen Darstellungen realisiert wurden.

Ein Ausblick über weitere mögliche Entwicklungen dieses Projekts wie die Zusammenarbeit mit anderen E-Learning-Projekten und -Plattformen befindet sich im letzten Kapitel.

Im Anhang befinden sich neben den obligatorischen Verzeichnissen und dem Glossar, Erläuterungen zur Architektur und relevante Details zur Implementierung in *Adobe Flash*.

Abstract

This present work covers the creation of a simulator of the **ALU** of the **MC8**, which was developed at the *Institute of Computer Technology* at the *Vienna University of Technology*. The *MC8* is a micro-computer designed for teaching purposes. An ALU is among other things a part of a processor, responsible for arithmetic and propositional logic operations.

Goal of this web published *adobe flash* application is to visualize interrelationships between time and hardware processes.

This work starts dealing with the meaning of e-learning and covers up to date knowledge transfer methods. The following chapters explain technical basics as far as they are required for understanding of terms and definitions.

Basic electronic circuits are described, which can be used to execute arithmetical and propositional logic functions. State memory is essential when talking about fundamental building sequential circuits. These circuits compute the following state, starting from a current state and using input values. Beyond that output values are being generated. This property is used to build a control unit, which is used to manage an arithmetic logical unit. In this unit the mentioned operations can be executed and are the subject of the software part in this work.

After describing the software, the work discusses, model simplifications. Possible compromises to technical correctness lead to an understandable presentation. After a short review of the technical platform there is a short explanation of the historical development of programming languages. After that follows a statement about the criteria for choosing *adobe flash* as a platform. This first part ends with an explanation of projects which goals are similar to those of this work.

In the following chapters, the goals and non-goals of the application of the target group are analysed. A comparison of realized simulators, done on the institute of computer technology follows. Furthermore, details of calculation- and propositional logic operations are discussed with a focus on realizing specific properties. In addition, established standards are explained to get a consistent user interface visualization.

After that, each single operation is described in much detail. This description is quite comprehensive, because there are twelve operations, each having two different views with variable difficulties.

An outlook about further development of this project and collaboration with other e-learning projects is discussed in the last chapter.

Beneath the mandatory directories and glossary, explanations about software architecture and relevant details about the realization in *adobe flash* conclude the thesis in the appendix.

Danksagung

Ich möchte all jenen Personen danken, die mich im Laufe dieser Arbeit unterstützt und durch Anregungen, Diskussionen und Motivation wesentlich zu dieser Arbeit beigetragen haben.

Allen voran danke ich herzlich Herrn *Em. Univ.Prof. Dipl.-Ing. Dr.techn. Richard Eier* für die freundliche Unterstützung.

Herrn *Univ.Ass. Dipl.-Ing. Dr.techn. Friedrich Bauer* gebührt ganz besonderer Dank für seine Geduld und für sein freundliches sowie konstruktives Feedback, welches mir stets weitergeholfen hat. Durch seine Unterstützung war es mir schlussendlich möglich, die Arbeit zügig fertigzustellen und einen früheren als den geplanten Einreichtermin zu nutzen.

Spezieller Dank gilt meiner Familie. Ohne ihre Geduld, Liebe, Verständnis und Motivation wäre diese Arbeit nicht gelungen! Ganz besonders möchte ich mich bei meiner Frau Sonja Brothanek für die liebevolle wie tatkräftige Unterstützung bedanken.

Zu guter Letzt danke ich auch allen Freunden für ihre Nachsicht, da sie vor allem in den letzten Wochen der Fertigstellung der Diplomarbeit zu kurz kamen: allen voran Stefan Fiedler, Sandra und Rainer Franzel, Ina und Florian Warnecke und natürlich Johnny Weingast und Julia Jäger, die netterweise das Abstract redigiert hat.

Herzlichen Dank!

Inhaltsverzeichnis

1 Abkürzungen

Actionscript	Programmiersprache von Adobe Flash
Adobe Flash	Design- und Programmierumgebung von Adobe Flash
AIR	Adobe Integrated Runtime
ALU	Arithmetic and Logic Unit, Mikroprozessor Recheneinheit
CMOS	Complementary MOS-Logic
CPU	Central Processing Unit, Mikroprozessor
ECL	Emitter Coupled Logic
FDT	Programmentwicklungsumgebung für Actionscript, die Programmiersprache von Adobe Flash
GND	Ground
HTML	Hypertext Markup Language
ICT	Institut für Computertechnik
MC8	Modell Computer 8 (8-Bit Lehrcomputer)
TTL	Transistor Transistor Logic
XOR	Exklusives Oder
VCC	Voltage Common Collector

2 Einleitung Ausgangssituation / Einführung

Das Internet verändert die Welt der Wissensvermittlung. Es ist nun möglich, allen Studierenden Zugang zu Ressourcen zu geben, wie es bisher nicht möglich war. Dabei verändern sich die Inhalte bei den Grundlagen naturgemäß nicht, die Methodik der Vermittlung ändert sich jedoch sehr wohl.

Bei der Übung der Vorlesung *Digitale Systeme* des *Instituts für Computertechnik* wird unter anderem der *MC8*[1] als Übungsgrundlage verwendet, der auch in Form eines Übungsboards[2] als Hardware zur Verfügung steht. Leider kann ein Fehlverhalten von Studierenden nicht ausgeschlossen werden, daher kommt es immer wieder zu Ausfällen der Hardware. Aber auch wenn niemals Fehler passieren würden, ist die Ressource knapp und muss für alle Studierenden in gleichem Maße zugänglich sein. Das bedingt wiederum eine zeitlich eingeschränkte Nutzung pro Person.

Daher ist es nahe liegend, diese Hardware softwaremäßig zu simulieren, da dann auch bei Fehlbedienung keine nachhaltigen Schäden erzeugt werden. Dies ist der Grundgedanke der vorliegenden Arbeit, denn über das Internet ist es leicht möglich, jeder Person Zugriff auf die Simulation des MC8 zu gewähren. Die Studierenden können nun ohne Schaden zu befürchten experimentieren – und das zeitlich und örtlich höchst flexibel.

Die Studierenden können sich intensiver damit auseinandersetzen, können lernen wann sie wollen, Inhalte beliebig oft erklären lassen, sich optimal für Prüfungen vorbereiten und am simulierten „lebenden“ Objekt testen und damit mehr Sicherheit erlangen.

Der Nachteil einer solchen individuell nutzbaren Simulation ist die fehlende Möglichkeit, Fragen stellen zu können. Bei einer realen Situation in einem Labor sitzen in der Regel mehrere Studierende in einem Raum, die sich gegenseitig weiterhelfen können. Da sich aber durch die Umstellung von Hardware auf Softwaresimulation ja kein Zwang ergibt, alleine zu lernen, bleibt es den Nutzenden überlassen, ihre bevorzugte Lernmethode anzuwenden.

E-Learning generell ersetzt *nicht* den bisherigen Lehrbetrieb, kann diesen aber hervorragend *ergänzen*. In diesem Fall kann die Simulation sowohl bei der Vorlesung mittels Einsatz eines Projektors, wie auch für individuellen Lern- und Prüfungseinsatz eingesetzt werden. Ein Vorteil hierbei ist die vereinheitlichte Darstellung für Studierende im Vortrag und der Übung.

Diese Arbeit trägt also mit dem Simulator dazu bei, die Lehre *noch anschaulicher und angreifbarer* zu gestalten.

1 Modellcomputer-8 (8-Bit Lehrcomputer), entwickelt für die Grundlagenvermittlung am Institut für Computertechnik
2 [BAUE1997]

3 GRUNDLAGEN

In diesem Kapitel werden die Grundlagen der vermittelten Inhalte diskutiert, mit dem Ziel zu beweisen, dass die Kenntnis der Grundlagen für Studierende der Elektrotechnik unabdingbar ist. Darüber hinaus wird die Landschaft der Softwareumgebungen untersucht, um zu begründen, warum dieses Projekt in Adobe Flash umgesetzt wurde. E-Learning Grundlagen und der Blick auf ähnliche Projekte runden das Bild zu einem vollständigen Ganzen ab.

3.1 E-Learning

Zu Beginn ist es zweckmäßig auf den Begriff *Lernen* einzugehen.

> *Unter **Lernen** versteht man den absichtlichen […] und den beiläufigen […], individuellen oder kollektiven Erwerb von geistigen, körperlichen, sozialen Kenntnissen und Fertigkeiten oder Fähigkeiten.*[3]

Wie gelernt wird und worum gelernt wird, ist von den handelnden Personen, ihrem Umfeld und vielen anderen Faktoren abhängig. Lernen bedeutet allerdings immer, dass etwas Neues dem bestehenden Wissen hinzugefügt wird. Lernen bedeutet auch Weiterentwicklung. Diese Weiterentwicklung war schon seit jeher für die Menschheit enorm wichtig.

Doch wie wurde Lernstoff weitergegeben? In der narrativen/oralen Lernkultur wurde Wissen über Erzählungen, Lieder, Gedichte usw. vermittelt. Das gekonnte Vortragen war ein wichtiges Element. Unterstützt wurde dies durch Mimik, Gestik, Musik uam. Dies erfuhr spätestens ab Erfindung des Buchdrucks eine starke Veränderung. Literale/wissenschaftliche Lernkulturen entwickelten sich und sind bis heute die am häufigsten angewendete Lernkultur. Etwas *schwarz auf weiß* zu lesen hat meist eine höhere Bedeutung als das gesprochene Wort.[4]

Das Vortragen von Lerninhalten ist zwar auch in unserer Zeit üblich, die Vorträge orientieren sich jedoch häufig am geschriebenen Wort oder sind sogar vollkommen ident mit diesem. Dies erschwert allerdings die Aufnahme der angebotenen Inhalte. Vorträge, die vom geschriebenen Wort abweichen und z.B. durch Präsentationstechniken unterstützt werden, sind demgegenüber im Vorteil.

---

[3] [WIKI2008A] Hervorhebungen im Text im Original.

[4] Vgl. [BROT2006, S. 69f]

Die visuelle/ästhetische Lernkultur ist eine Entwicklung der heutigen Zeit. Bilder – ob statisch oder bewegt – werden meist zur Verstärkung des geschriebenen Wortes herangezogen. Der Wahrheitsgehalt eines Fotos wird kaum bezweifelt, was allerdings in Zeiten von Bildbearbeitungsprogrammen gefährlich sein kann. Aber Abbildungen ermöglichen auch eine neue Lernqualität, da sich vor allem abstrakte Inhalte mittels Visualisierung leichter vermitteln lassen.[5]

Es lässt sich zusammenfassend sagen, dass unterschiedliche Inhalte eine ebenso vielfältige Aufbereitung erfordern. Abhängig vom Inhalt kann diese Aufbereitung gesprochen, geschrieben oder bildlich sein.

E-Learning ist eine eigene Lernkultur. Die Entwicklung von E-Learning ergibt sich einerseits aus einem geschichtlichen und andererseits aus einem technischen Kontext.

3.1.1 Entwicklung von E-Learning und Blended Learning

Der Wunsch, Lerninhalte möglichst automatisiert und unabhängig von Lehrenden anbieten zu können, bildete die Grundlage für die Erfindung so genannter *Lehrmaschinen*. Erste Lehrmaschinen wurden bereits im 16. Jahrhundert konstruiert[6]. 1928 entwickelte *Sidney Pressey* die erste Multiple-Choice-Lehrmaschine. Allerdings konnten sich zur damaligen Zeit Lehrmaschinen nicht durchsetzen. Dieter und Wiesner erklären dies mit folgenden Faktoren: schwere Rezessionen der Weltwirtschaft, hohe Arbeitslosigkeit und Rebellion des Lehrpersonals.[7]

Erst ab 1950, so Dieter und Wiesner, bekam die Idee der Lehrmaschine wieder neuen Aufschwung. In dieser Zeit begründet der Psychologe Burrhus Skinner seine Theorie vom programmierten Unterricht. Er geht davon aus, dass Handlungen, auf die unmittelbare und verstärkende Konsequenzen folgen, tendenziell wiederholt werden. Wird die Handlung oder das Verhalten nicht verstärkt, so wird es aus dem Verhaltensrepertoire gelöscht.[8]

Dieter und Wiesner stellen fest, dass Anfang der 60er Jahre die Popularität der Lehrmaschinen ihren Höhepunkt erreichte. Allerdings nimmt der Boom ab Mitte der 60er Jahre wieder ab[9]. Erst in den 80er Jahren ist für sie ein erneuter Aufschwung zu erkennen. Dies geschah durch die *„rasche Verbreitung der Computer-Hardware“*[10].

Die Technik machte es nun möglich, dass diese neuen Lehrmaschinen - die Computer - für unterschiedliche Lerninhalte verwendet werden konnten. Und nicht nur das, schließlich konnte und kann man einen Computer auch für andere Dinge verwenden, nicht nur um zu lernen.

Dieter und Wieser verweisen auf die Entstehung der Intelligenten Tutoriellen Systeme (ITS), welche sich von der Idee des programmierten Lernens wegbewegten. Im Mittelpunkt stand die *„Orientierung der Lernwege am Wissensstand des Lerners“*[11]. Die Mitte der 90er Jahre stellt für Dieter und Wieser

---

[5] Vgl. [BROT2006, S. 69f]

[6] In einem 1588 von Agostino Ramellis publizierten Buch „*Le deiverse et artificiose machine*“ findet man zahlreiche Erfindungen, unter anderem die eines Bücherrades, welches als erste Lehrmaschine gezählt wird, da dies eine Maschine zur Verwaltung von Wissen war. [BAYE2008]

[7] Vgl. [DIET2003] S. 4.

[8] Vgl. [DIET2003] S. 7f.

[9] Vgl. [DIET2003] S. 11.

[10] [DIET2003] S. 12.

[11] [DIET2003] S. 13.

durch die rasche Verbreitung des Internets einen weiteren Meilenstein bei der Entwicklung von E-Learning dar. [12]

3.1.2 Begriffsdefinition

Der Begriff E-Learning wird heutzutage wie folgt definiert:

> *Unter E-Learning (auch eLearning, englisch electronic learning – elektronisch unterstütztes Lernen), auch E-Lernen genannt, werden – nach einer Definition von Michael Kerres – alle Formen von Lernen verstanden, bei denen digitale Medien für die Präsentation und Distribution von Lernmaterialien und/oder zur Unterstützung zwischenmenschlicher Kommunikation zum Einsatz kommen. Für E-Learning finden sich als Synonyme auch Begriffe wie Online-Lernen, Telelernen, Computer Based Training, multimediales Lernen, Open and Distance Learning, computergestütztes Lernen u. a.*[13]

oder auch:

> ***E-Learning** das, Kurzwort für **Electronic Learning**, **elektronisches Lernen,** computergestütztes Lernen unter Nutzung von Multimedia- und Netzwerktechnologien, das mit strukturellen Veränderungen der Bildungsorganisation und der Lernkultur verbunden ist.*[14]

Zusammengefasst bedeutet das, dass es beim E-Learning um die Vermittlung von Lerninhalten unter Zuhilfenahme moderner Technologien geht.

Im Meyers Online-Lexikon wird zum Begriff E-Learning weiter ausgeführt,

> *… [es wurde] davon ausgegangen, dass E-Learning zu einer Verbesserung traditioneller Lehr-Lern-Zusammenhänge beiträgt, weil diese durch den Einbezug elektronischer Hilfsmittel abwechslungsreicher, motivierender und anregender gestaltet sowie damit verbundene Lernzeiten beziehungsweise -orte flexibilisiert werden können.*[15]

Diese Darstellung beschreibt zwei Rahmenbedingungen von E-Learning:

Erste Rahmenbedingung ist die **Aufbereitung der Lerninhalte**. Wobei es allerdings sehr kurzsichtig ist, anzunehmen, dass alle Lerninhalte, die unter dem Titel E-Learning angeboten werden, automatisch die Kriterien *abwechslungsreich*, *motivierend* und *anregend* erfüllen. Grundlegend geht es bei E-Learning darum, das Lernen durch Computer bzw. digitale Medien zu unterstützen. Darunter kann man nun von der Folienpräsentation im Internet bis zur hoch entwickelten Lernumgebung alles verstehen. Die Qualität der angebotenen Inhalte ist sehr unterschiedlich und hängt stark vom jeweiligen Anbieter ab.

Zweitens wird die **zeitliche und örtliche Unabhängigkeit** genannt. E-Learning Lerninhalte können meist *beliebig oft* angesehen werden, *zeitliche Gestaltung* und *Örtlichkeit* können die Lernenden meist selbst bestimmen. Dies kann tatsächlich als Vorteil zu traditionellen Lehr- und Lernsituationen gesehen werden.

Wie sinnvoll ist allerdings das ausschließliche Darstellen von Texten und Abbildungen, sei es auch in ausgeklügelten Lernumgebungen? Kann Lernen auf diese Weise sinnvoll funktionieren?

---

[12] Vgl. [DIET2003] S. 15.
[13] [WIKI2008B]
[14] [MEYE2008A]
[15] [MEYE2008B]

Wie Dr. Kienle in ihrer Arbeit ausführt, ist die lernende Person gemäß der Lerntheorie Konstruktivismus *„ein aktiv konstruierendes Wesen, das im sozialen Kontext in reger Auseinandersetzung mit der Umwelt Wissen erwirbt."*[16] Das Lernen erfolgt ihrer Theorie zufolge *„durch Erleben, Interpretieren und Konstruieren"*[17]. Kienle erweitert ihre Ausführungen noch dahingehend, dass für sie Lernen in einen sozialen Kontext eingebettet sein soll und dass auch Experten am Lernprozess beteiligt sein sollen.[18]

Daraus lässt sich ableiten, dass E-Learning in seiner Reinform für eine qualitative Wissensvermittlung in vielen Fällen nicht ausreicht, denn Fragen zu stellen und Inhalte infrage zu stellen sind wesentliche Elemente des wissenschaftlichen Lernens. Ohne Diskussion und Diskurs gibt es keine Weiterentwicklung, keine neuen Ideen und keinen Fortschritt.

Texte und Abbildungen müssen durch weitere Elemente ergänzt werden. Dies können beispielsweise Foren sein, in denen sich Teilnehmer austauschen, oder Filme, die die angebotenen Lerninhalte unterstützen. Aber auch Tutorials oder die gemeinsame Bearbeitung von Dokumenten zählen dazu, ebenso wie die Kombination aus Seminaren mit Anwesenheit und E-Learning Anwendungen.

Dieser Methodenmix trägt den Namen Blended Learning. Definiert wird der Begriff wie folgt:

> *Das Schlagwort „Blended Learning" bezeichnet Kombinationsformen des Lernens, die den Einsatz moderner Informations- und Kommunikationstechnologien mit dem Lernen in Präsenzseminaren verknüpfen.*[19]
>
> *Blended Learning ist eine mögliche Form von eLearning. Im universitären Raum setzt sich diese Form zunehmend durch. Blended Learning ist die angemessene Kombination von Präsenz-Lehr- und -Lernphasen und Online-Lehr- und -Lernphasen (Lernumgebung mit Informations- und Kommunikationsmedien und Lehr- und Lernaktivitäten) über einen bestimmten Zeitraum (z. B. ein Semester).*[20]

Das bedeutet, E-Learning ist eine Lern- und Lehrmethode, die im Blended Learning zum Einsatz kommen kann, aber nicht zwangsläufig muss.

[16] [KIEN2005] S. 10
[17] [KIEN2005] S. 10
[18] Vgl. [KIEN2005] S. 10
[19] [REGL2005] S. 3
[20] [ELEA2008A]

3.2 Fachliche Grundlagen

Das Ziel dieser Diplomarbeit ist es, einen Simulator einer **Arithmetisch-Logischen Einheit (ALU)** umzusetzen. Dieser Abschnitt motiviert die Grundlagen, um eine **ALU** eines Prozessors zu entwerfen. Dazu sind logische Funktionen und Zustandsspeicherungen erforderlich. Diese stehen zu Beginn dieses Abschnitts.

Die Elektrotechnik kann in zwei Bereiche geteilt werden: in die *Analog-* und die *Digitaltechnik*[21].

Die *Analogtechnik* wird heutzutage tendenziell für die Erzeugung und den Transport von Energie bzw. für die Umwandlung in andere Energieformen benutzt. Während es bei der *Digitaltechnik* ausschließlich um den Transport und die Verarbeitung von Daten und Informationen geht.

Eine exakte Trennung von *Analog-* und *Digitaltechnik* ist nicht möglich, da sich die beiden Bereiche überschneiden[22]. Abgesehen davon muss erwähnt werden, dass *Digitaltechnik* immer mit Analogtechnik realisiert wird.

Ein **digitales Signal** ist ein diskretes Signal, bei dem den Werten der Informationsparameter Worte entsprechen (i>=1, k>=2). Die Wortzuordnung wird als **Codierung** bezeichnet.

Digitale Signale sind *codierte diskrete Signale*.

Die Erweiterung dazu ist das **binäre Signal**[23]. Der Informationsparameter kann genau zwei Werte annehmen (i=1; k=2). Entsprechend dem binären Zahlensystem (Dualsystem) werden die beiden Werte als Signalwert 0 und Signalwert 1 bezeichnet[24].

[21] [LIND2004]

[22] Beispiele hiefür sind die analoge Frequenzmodulationstechnik, die im Hörfunk eingesetzt wird, und das D-Netz der mobilkom (welches 2002 eingestellt wurde) über welche Informationen in einer für den Menschen verständlichen Form übermittelt werden.

[23] [LIND2004] S. 425ff.

[24] Korrekterweise müsste man, um zu kennzeichnen, dass es sich um ein eigens definiertes Alphabet handelt, „0“ bzw. „1“ (also mit Anführungszeichen) schreiben. Zur besseren Lesbarkeit wird in dieser Arbeit diese Kennzeichnung weggelassen und nur 0 bzw. 1 (ohne Anführungszeichen) geschrieben.

Da auch in der Digitaltechnik nur mit analogen Signalen gearbeitet werden kann, müssen Wertebereiche für analoge Signale definiert werden, innerhalb derer ein anliegendes analoges Signal als 0 (beziehungsweise 1) eindeutig interpretiert werden kann. Befindet sich ein Signal außerhalb dieser zwei Wertebereiche, ist eine Zuordnung nicht eindeutig möglich. Eine Zuordnung könnte beispielsweise wie folgt lauten:

Digitales Signal	Schaltkreisfamilien	
	TTL oder CMOS (positive Logik)	ECL (negative Logik)
0	GND (Ground, Masse) 0,0 .. 0,8V	-1,7V
1	VCC (+5V bei TTL) 2,4 .. 5,0V	-0,9 V

Tab. 1: Schaltkreisfamilien und ihre Spannungsintervalle

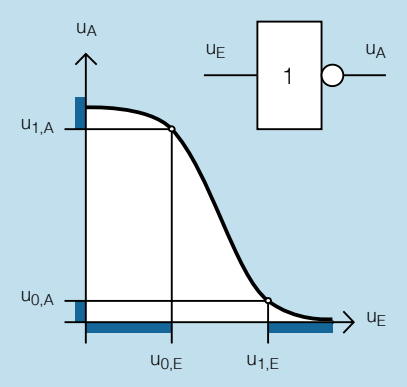


Abb. 1: Kennlinie mit Spannungsintervallen

Mit diesen Festlegungen kann man also stets eine eindeutige Zuweisung zu Symbolen treffen. In diesem Kontext spricht man von einem **Wort**[25], wenn man eine geordnete Menge von Zeichen eines Alphabets hat, die erst in ihrer Gesamtheit eine Bedeutung hat bzw. eine Information enthält.

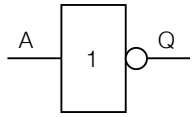
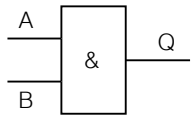
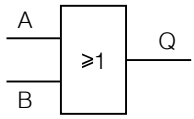
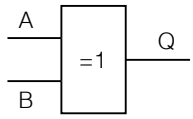
Ein **Alphabet**[26] ist eine Liste von vereinbarten Zeichen oder Symbolen. Bei der Digitaltechnik nutzt man das binäre Zahlensystem, woraus sich das Alphabet festlegen lässt:
Es besteht nur aus den Werten 0 und 1.

Digitale Schaltungen[27] bestehen aus einem oder mehreren miteinander verbundenen *Gattern*. Ein Gatter stellt die logische und technische Realisierung einer boolschen Funktion dar. Digitale Schaltungen werden in Elementarschaltungen und komplexe Schaltungen gegliedert. Elementarschaltungen sind kombinatorische Schaltungen (Schaltnetze) oder sequenzielle Schaltungen (Schaltwerke).

[25] [LIND2004] S. 425ff.
[26] [LIND2004] S. 425ff.
[27] [LIND2004] S. 426ff.

3.2.1 Logische Grundfunktionen

Man unterscheidet bei den Grundfunktionen zwischen jenen mit einem oder zwei Operanden. Diese Unterscheidung ist in weiterer Folge für die Abarbeitung von Befehlen einer **ALU** relevant.[28]

Operation	Beschreibung	Wahrheitstabelle	Schaltbild
Negation (NICHT, NOT)	Diese Operation invertiert den anliegenden Wert. Aus 0 wird 1 und umgekehrt.	**Eingabe (A)** / **Ausgabe (Q)** 0 / 1 1 / 0	
Konjunktion (UND, AND)	Diese Verknüpfung ergibt nur dann im Wert 1, wenn beide anliegende Werte 1 sind.	**Eingabe (A)** / **Eingabe (B)** / **Ausgabe (Q)** 0 / 0 / 0 0 / 1 / 0 1 / 0 / 0 1 / 1 / 1	
Disjunktion (ODER, engl. OR)	Durch diese Funktion wird die Ausgabe 1, wenn mindestens einer der beiden Eingänge 1 ist, sonst 0.	**Eingabe (A)** / **Eingabe (B)** / **Ausgabe (Q)** 0 / 0 / 0 0 / 1 / 1 1 / 0 / 1 1 / 1 / 1	
Antivalenz (Exklusives Oder, XOR, engl. XOR)	Wenn genau ein Eingang 1 ist, dann ergibt diese Operation 1, sonst 0.	**Eingabe (A)** / **Eingabe (B)** / **Ausgabe (Q)** 0 / 0 / 0 0 / 1 / 1 1 / 0 / 1 1 / 1 / 0	

Tab. 2: Boolsche Grundfunktionen

3.2.2 Schaltnetz

Schaltnetze sind *speicherfreie*, *digitale Schaltungen*, die Realisierungen boolscher Funktionen darstellen. Das heißt, dass sich zu jedem Zeitpunkt die binären Ausgänge *A* eindeutig aus der Verknüpfung der binären Eingangswerte *E* ergeben.

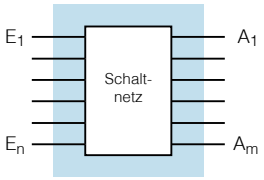


Abb. 2: schematische Darstellung eines Schaltnetzes

Ein Beispiel dafür ist ein *Volladdierer (VA)*, der auch in der ALU zum Einsatz kommt. Der Volladdierer wird im nachfolgenden Abschnitt behandelt.

[28] Je nachdem, ob die ALU einen oder zwei Parameter für eine Funktion benötigt, werden die Eingangssignale entsprechend angelegt.

3.2.2.1 Volladdierer

Ein Volladdierer addiert zwei Eingangsbits sowie den Übertrag aus einem Vorgänger. Dabei ergibt sich folgende Wahrheitstabelle:

Eingang *a*	0	1	0	1	0	1	0	1
Eingang *b*	0	0	1	1	0	0	1	1
Übertrag $\ddot{u}_{ein}$	0	0	0	0	1	1	1	1
Ergebnis *s*	0	1	1	0	1	0	0	1
Übertrag $\ddot{u}_{aus}$	0	0	0	1	0	1	1	1

Tab. 3: Wahrheitstabelle des Volladdierers

Mit verschiedenen Methoden (zum Beispiel mit dem KV-Diagramm[29]) kann man diese Funktion optimal darstellen.[30]

Es ist offensichtlich, dass die Ergebnisse *s* und $\ddot{u}_{aus}$ ausschließlich von den Eingängen *a*, *b* und $\ddot{u}_{ein}$ abhängen. Ein Schaltnetz bildet somit stets eine idempotente Funktion ab, ohne Notwendigkeit der Speicherung von Informationen.

Man kann nun ausgehend von dieser logischen Funktion (und deren optimaler Darstellung) eine Schaltung entwerfen:

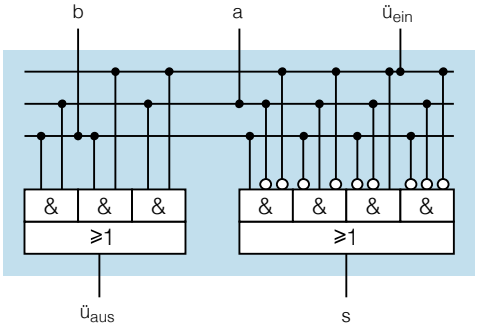


Abb. 3: Schaltung eines Volladdierers

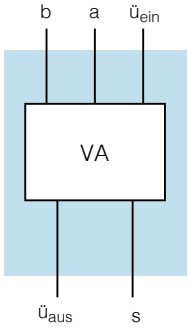


Abb. 4: schematische Darstellung eines Volladdierers

Verkettet man mehrere dieser Volladdierern miteinander, so spricht man von einer **Kaskade**. Da man diese Kaskade theoretisch beliebig lang machen kann, kann man auch beliebig viele Binärstellen addieren.

[29] Karnaugh-Veight-Diagramm: Systematisches Vereinfachungsverfahren für logische Funktionen. Es würde den Rahmen dieser Arbeit sprengen, die Optimierung von logischen Funktionen zu erläutern (weiterführende Informationen siehe [BAUE2008]).

[30] Diese optimierte Funktion lässt sich wie folgt darstellen:
s = (a XOR b) XOR $\ddot{u}_{ein}$
$\ddot{u}_{aus}$ = (a AND b) OR (a AND $\ddot{u}_{ein}$) OR (b AND $\ddot{u}_{ein}$)

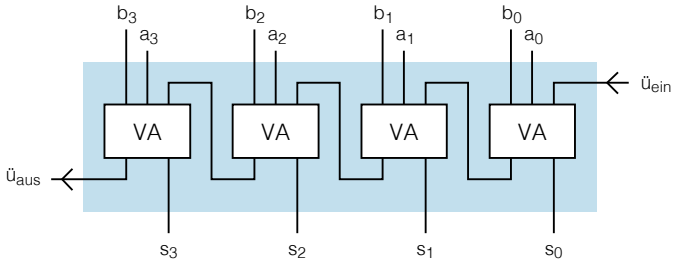


Abb. 5: Volladdierer in Kaskade

3.2.2.2 Subtraktion und Ersatzaddition

Analog zur Addition kann man eine Subtraktion im Binärsystem durchführen. Man benötigt zunächst die zwei Eingangswerte von Minuend und Subtrahend. Sind beide Werte ident oder der Subtrahend kleiner als der Minuend ist das Ergebnis trivial und ein Übertrag zur nächsten Stelle ist 0. Ist der Subtrahend größer, muss man den Minuend um eine Zweierpotenz erhöhen (man muss sich von der nächsten Stelle eine 1 „borgen“). Somit ist das Ergebnis und der Übertrag zur nächsten Stelle 1.

Die einfache Binärarithmetik macht es möglich, die Subtraktion ohne eigenes Subtrahierwerk durchzuführen. Man nutzt dafür die **Ersatzaddition**: Dabei geht man von der Überlegung aus, dass man eine Zahl von einer anderen subtrahieren kann, wenn das Vorzeichen der abzuziehenden Zahl gewechselt und dann eine Addition durchgeführt wird. Man ersetzt also den Ausdruck

d = a – b

durch

d = a + (-b).

Dafür benötigt man eine binäre Darstellung negativer Zahlen, welche im binären Zahlensystem durch das **Zweierkomplement** ermöglicht wird.

Diese negative, binäre Zahlendarstellung erreicht man ausgehend von der positiven Darstellung. Man komplementiert jede einzelne Stelle (aus 0 wird 1 und umgekehrt) und zählt 1 zum Gesamtergebnis hinzu.

Beispiel: 4-Bit Zahl Zweierkomplement

Ausgehend von der dezimalen Zahl 7 mit folgender binären Darstellung:	`0111`
Alle Stellen einzeln invertieren	`1000`
Zum Gesamtwert 1 dazuzählen (binär 0001)	`+0001`
Zweierkomplement	**`1001`**

Tab. 4: Umwandlung einer vierstelligen Binärzahl in ihre Zweierkomplementdarstellung

Das Zweierkomplement leistet gute Dienste, denn es lässt sich schaltungstechnisch sehr einfach realisieren. Ausgehend von dem zuvor gezeigten 4-Bit-Addierwerk invertiert man die Eingänge des Subtrahenden mit NOT-Gatter. Dem ersten Volladdierer führt man als ersten Übertrag den Wert 1 zu und erhält durch diese Maßnahme eine Zweierkomplementdarstellung. Diese Schaltung unterscheidet

sich nur unwesentlich von der Additionsschaltung:

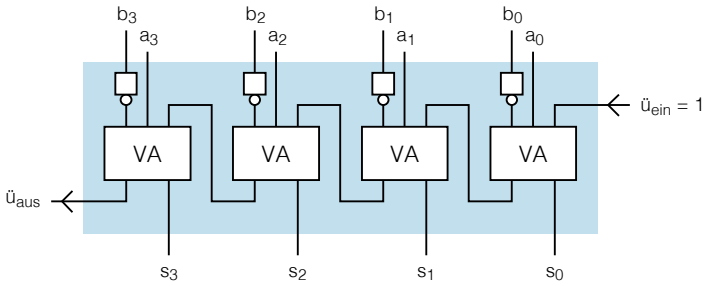


Abb. 6: Volladdierer-Kaskade für die Ersatzaddition (automatische Zweierkomplementbildung des Wertes b)

Es liegt daher nahe, das Verhalten der Schaltung so weit steuerbar zu machen, dass sie entweder eine Addition oder eine Subtraktion (mittels Ersatzaddition) durchführt. Man ersetzt hiefür die NOT-Gatter durch XOR-Gatter und legt am ersten Eingang des XOR-Gatters den Wert von b_n an. Am zweiten Eingang wird ein Steuersignal zugeführt – dies ist auch jener Wert, der beim Übertragseingang des ersten VA anliegt. Dadurch verhalten sich die XOR-Gatter bei dem Wert 0 des Steuersignals wie eine direkte Verbindung (wird für die *Addition* benötigt). Wird der Wert 1 angelegt, verhalten sich die XOR-Gatter wie ein NOT-Gatter (benötigt man für die *Ersatzaddition*):

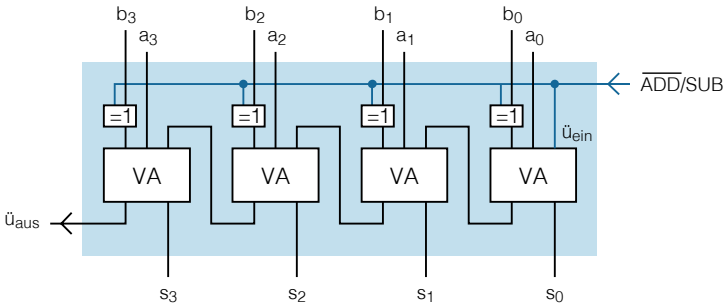


Abb. 7: Volladdierer-Kaskade für wahlweise Addition bzw. Subtraktion

Das Steuersignal $\overline{\text{ADD}}/\text{SUB}$ kann zum Beispiel durch ein Steuerwerk gesetzt werden (siehe Abschnitt 3.2.5).

Mit Steuerwerken kann man auch aufwändigere Operationen steuern, die aus mehreren Einzelschritten bestehen können. So kann es notwendig sein, in einem ersten Schritt anliegende Werte zu laden und erst in einem weiteren Schritt zu verarbeiten. Dieses Vorgehen verlangt aber Wissen über den aktuellen Zustand, was eine Speicherung von Information voraussetzt (in diesem Fall den aktuellen Zustand).

3.2.3 Zustandsspeicherung

Die Zustandsspeicherung findet sich in der **ALU** an vielen Stellen, nicht zuletzt bei den Flags, die gewisse Zustände abbilden und sich nur innerhalb gewisser zeitlicher Grenzen ändern dürfen[31].

[31] Genau genommen sind die Flags Bestandteil der CPU, nicht der ALU. Da die Flags aber für die Betrachtung wichtig sind und bei Rotate With Carry (siehe Abschnitt 4.2.3.3) operationsrelevante Werte für die Berechnung liefern, werden die Flags im Rahmen dieser Arbeit immer gemeinsam mit der ALU betrachtet.

Die Frage, wann man Zustandspeicherung benötigt, kann beantwortet werden, wenn man sich die Aufgaben der ALU genauer ansieht: Einige der Schaltungen benötigen Informationen aus vorherigen Zuständen oder müssen auf verzögerte Eingänge warten.

Um Zustände zu speichern setzt man **Flip-Flops** ein. Ein Flip-Flop kann ein Bit an Information speichern.

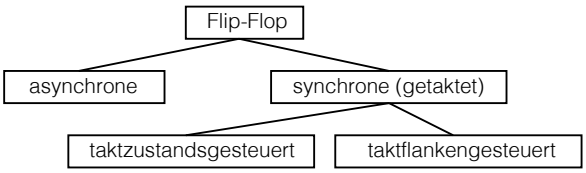


Abb. 8: Kategorisierung von Flip-Flops[32]

Asynchrone Flip-Flops sind Schaltungen, die einen Wert unmittelbar übernehmen, ohne auf ein Taktsignal zu warten. Ein typischer Vertreter ist das **RS-Flip-Flop** (*Reset-Set-Flip-Flop*), welches einen *Set*- und einen *Reset*-Eingang hat.

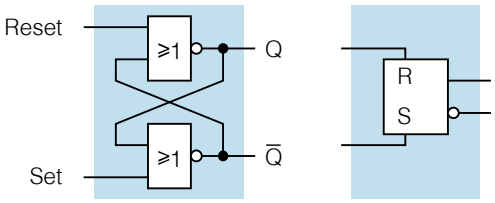


Abb. 9: RS-Flip-Flop (Ausführung und Symbol)

Wenn am Set-Eingang 1 anliegt, speichert das RS-Flip-Flop diesen Wert. Liegt am Reset-Eingang 1 an, wird das RS-Flip-Flop den Wert 0 speichern. Liegt auf beiden Eingängen jeweils 1 an, ist das Flip-Flop in einem ungewünschten Zustand und legt sowohl auf dem Ausgang Q als auch am negierten zweiten Ausgang ($\neg$Q) 0 an. Aufgrund dieses und anderer Nachteile betrachten wir in weiterer Folge nur getaktete Flip-Flops[33].

[32] Vgl. [LIND2004] S. 462ff

[33] Der Vollständigkeit halber seien noch weitere asynchrone Flip-Flops erwähnt: SL-Flip-Flop (RS-FF mit Setzvorrang) und EL-Flip-Flop (RS-FF mit Löschvorrang) vgl. dazu Linder, H./Brauer, H./Lehmann, C., JAHRESZAHL, S. 464.

3.2.3.1 Taktzustandsgesteuertes D-Flip-Flop (D-Latch)

Dieses Element übernimmt einen am D-Eingang anliegenden Wert sofort, solange der Takt (via Eingang C) auf 1 ist. Dieser übernommene Wert wird jedoch gehalten, wenn der Takt auf 0 ist. Folgende Darstellung zeigt das Schaltbild und einen möglichen Signal/Wert-Verlauf auf den Eingängen D (Daten) und C (Takt) sowie am Ausgang (Q).

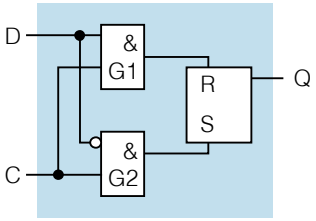


Abb. 10: Ausführung eines D-Latch

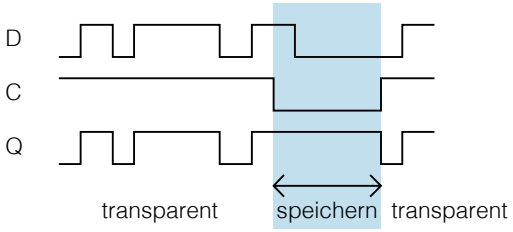


Abb. 11: Werteübernahme-Bereich anhand eines beispielhaften Signalverlaufs

3.2.3.2 Taktflankengesteuertes D-Flip-Flop

Im Gegensatz zum D-Latch wird beim **taktflankengesteuerten D-Flip-Flop** der anliegende Wert beim Eingang D *nur bei Flankenwechsel* des am Eingang C (Takt) anliegenden Signals übernommen. Üblicherweise nutzt man dabei eine steigende Taktflanke. Dies kann man auf zwei Arten erreichen:

Einerseits durch ein *Master-Slave-Flip-Flop*, bei dem zwei Latches in Serie geschaltet sind, andererseits durch ein *Latch mit kurzen Taktimpuls*.

Beim **Master-Slave-Flip-Flop** übernimmt das *Master-Latch* den Wert des Eingangs D, wenn Eingang C gleich 0 ist. Ist C gleich 1, übernimmt das *Slave-Latch* den Wert vom Master. Wird dann der Eingang C wieder 0, übernimmt wieder nur das Master-Latch die Werte von Eingang D und der Slave ist gesperrt.

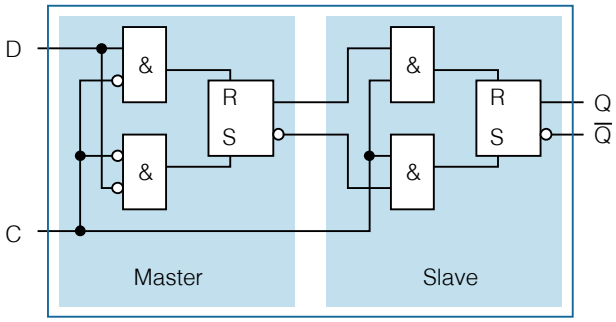


Abb. 12: Aufbau eines taktflankengesteuerten D-Flip-Flops

Beim **Latch mit kurzem Taktimpuls** wird durch das Ausnutzen der Verzögerungszeit einer Negation ein kurzer Impuls erzeugt, indem man den Eingang C und das durch ein Negations-Gatter laufende Signal in einem weiteren Gatter (z.B. AND oder NOR) zusammenführt.

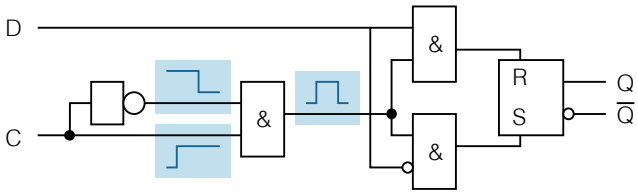


Abb. 13: Taktflankengesteuertes D-Flip-Flop mit kurzem Taktimpuls

Eine Erweiterung des D-Flip-Flops stellt das **JK-Flip-Flop** dar. Es erfüllt vier Funktionen

1) SET (Setzen des Ausgangs auf 1)

2) RESET (Rücksetzen des Ausgangs auf 0)

3) STORE (alten Zustand beibehalten)

4) TOGGLE (Zustand ändern)

J	K	Ausgang Q
0	0	STORE (unverändert)
0	1	RESET (0)
1	0	SET (1)
1	1	TOGGLE (0→1, 1→0)

Tab. 5: Zuordnung der Funktionen eines JK-Flip-Flops zu Eingangswerten

3.2.4 Schaltwerk

Schaltwerke sind wie Schaltnetze Schaltungen, die logische Funktionen abbilden. Diese können jedoch nicht nur von den Eingängen, sondern zusätzlich auch noch von im Schaltwerk gespeicherten *Zuständen* abhängen. Diese Zustände werden in Registern, die zum Beispiel aus Flip-Flops gebildet werden können, gespeichert.

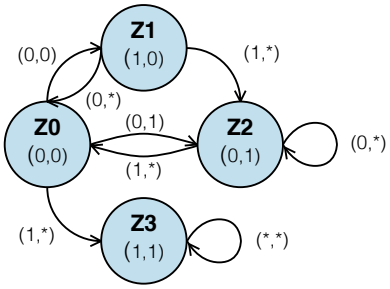


Abb. 14: möglicher Zustandsgraph eines Schaltwerks

Dieser Graph zeigt eine Übergangsfunktion anhand eines Beispiels:

Es entsprechen Z0 bis Z3 je einem Zustand, der aus dem Zustandsspeicher **Z** stammt. Die Werte der Pfeile zwischen den Zuständen stellen die anliegenden Werte der Eingänge dar. Interpretiert man den Graph als Darstellung für die Übergangsfunktion **G**, entspricht der neue Zustand dem Ausgang der Übergangsfunktion *G*.

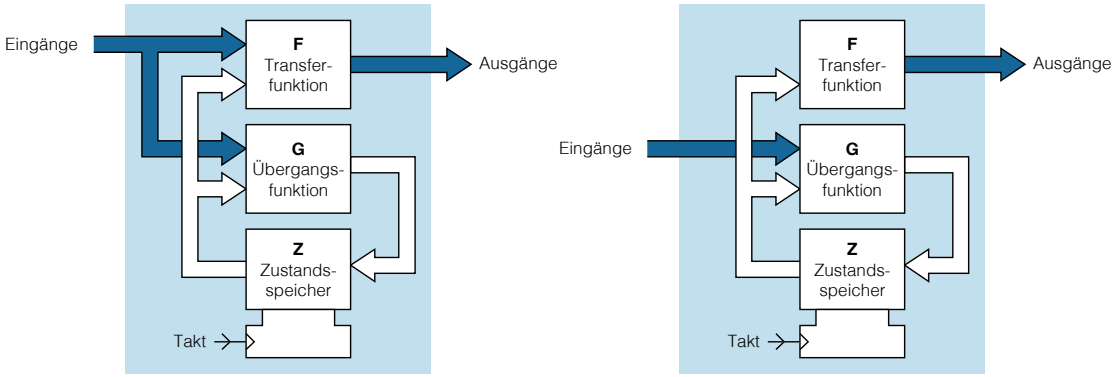


Abb. 15: links: *Mealy-Automat*, rechts: *Moore-Automat*, beides sind Beispiele für synchrone Schaltwerke[34]

Die Werte der Eingänge liegen sowohl der Transferfunktion **F**, als auch der Übergangsfunktion *G* vor. Schaltwerke haben einen inneren Zustand, der im Zustandsspeicher *Z* gespeichert wird.

[34] Beide Schaltwerke nutzen die Eingänge und einen Zustand, um den nächsten Ausgangswert zu ermitteln. Beim Moore-Schaltwerk (rechts) fließt der Wert des Eingangs ausschließlich in die Übergangsfunktion ein, beim Mealy-Schaltwerk (links) auch in die Transferfunktion.

Die Transferfunktion *F* bildet aus diesen Eingangswerten und dem aktuellen Zustand die Werte für die Ausgänge. Die Übergangsfunktion *G* erhält ebenso die Werte der Eingänge sowie den aktuellen Zustand.

Beim nächsten Taktimpuls ermittelt die Übergangsfunktion *G* aus den anliegenden Eingangswerten und den Werten aus dem Zustandsspeicher *Z* den nächsten Zustand und überträgt diesen in den Zustandsspeicher *Z*. Von den Werten des Zustandsspeichers *Z* und den Eingängen bildet die Transferfunktion *F* nun die neuen Ausgangswerte.

Optional kann man auch die Transferfunktion *F* mit dem Takt versehen, so dass diese ihren Wert für die Ausgänge wie die Übergangsfunktion *G* nur zu gewissen Zeiten ändert.

Oft wird die Übergangsfunktion nicht durch eine fest verdrahtete Lösung, sondern durch ein ROM (EPROM, EEPROM) ersetzt. Dabei werden alle möglichen Eingangssituationen (Eingänge und mögliche Zustände) und die zu erzeugenden Ausgangswerte in das ROM geschrieben. Dadurch ist es sehr einfach, den nächsten Zustand zu bestimmen. Abgesehen davon kann man diese Zuordnung bei (E)EPROMs verhältnismäßig schnell ändern.

Alter Zustand	Z0				Z1				Z2				Z3			
Eingang 0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Eingang 1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Zustandsvariable Z_0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Zustandsvariable Z_1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Gewünschter neuer Zustand	**1**	**3**	**2**	**3**	**0**	**2**	**0**	**2**	**2**	**0**	**2**	**0**	**3**	**3**	**3**	**3**
Vorbereitung für Z_0	1	1	0	1	0	0	0	0	0	0	0	0	1	1	1	1
Vorbereitung für Z_1	0	1	1	1	0	1	0	1	1	0	1	0	1	1	1	1

Tab. 6: Darstellung der Übergangsfunktion des Zustandsgraphen

3.2.5 Steuerwerke

Ein **Steuerwerk** ist eine spezielle Form des *Schaltwerks*. Es dient der Kontrolle eines Vorgangs, welcher eine Speicherung des aktuellen Zustands erforderlich macht. Der nächste Arbeitsschritt hängt also vom *Ergebnis einer vorhergehenden Operation* sowie vom *inneren Zustand* des Steuerwerks ab. Eine wesentliche Aufgabe eines Steuerwerks ist es, *Datenwegsschaltungen* vorzunehmen. In der Regel handelt es sich um eine große Anzahl von Steuerleitungen, die benötigt werden, um eine Schaltung zu konfigurieren. Datenwegsschaltungen sind einerseits eine Vorbereitungsaktivität, damit benötigte Werte an den Eingängen einer Schaltung anliegen, um die Operation abarbeiten zu können. Andererseits muss das Ergebnis der Operation zu Registern, Flags und andere Zielen[35] gelangen.

Die *CPU* (siehe Abschnitt 3.2.7) besteht aus einer *Registerbank* (mit *Flags*), einer *ALU* und einem *Steuerwerk*. Dieses Steuerwerk ist dafür verantwortlich die Eingangs- und Ausgangsdatenwege zu schalten, Maschinenbefehle aus dem Speicher in ein internes Register (*IR*, *Instruction Register*) zu laden, zu dekodieren und der ALU weiterzuleiten. Das Steuerwerk ist auch für die Steuerung der internen Abläufe der ALU verantwortlich.

Das Steuerwerk besitzt folgende Zustände, die auch in dieser Reihenfolge pro Befehl durchlaufen werden.

[35] Hier wäre z.B. das Ergebnis einer Programm-Sprung-Adresse zu nennen, die am Adressbus anliegen muss. Eine weitere Möglichkeit ist das Anlegen eines Ergebnisses einer Rechenoperation am Datenbus, um das Ergebnis in den Hauptspeicher zu schreiben.

1) **FETCH** (Daten aus dem Arbeitsspeicher holen, kann auch mehrmals durchlaufen werden, wenn mehrere Parameter geladen werden müssen)
2) **DECODE** (Befehl dekodieren)
3) **EXECUTE** (geladenen Befehl ausführen)

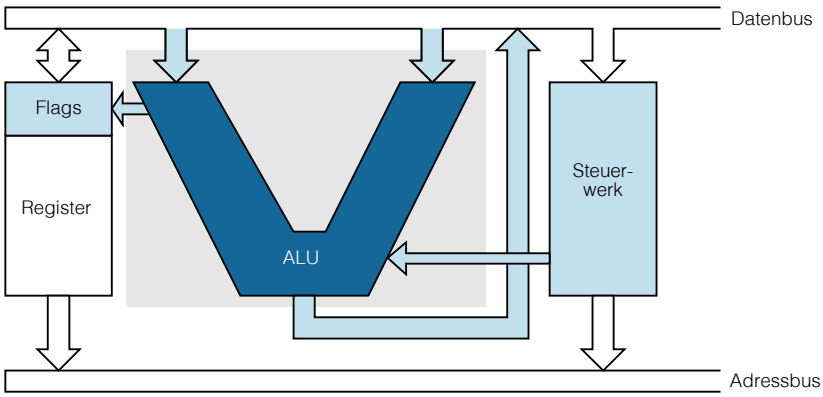


Abb. 16: Zusammenhänge zwischen der ALU, den Flags, den Registern und dem Steuerwerk der CPU

3.2.6 Rechenoperationen des MC8

Neben den bereits beschriebenen binären Funktionen Negation, Konjunktion, Disjunktion und Antivalenz kann die ALU des MC8 auch Rechenoperationen durchführen:

ADD: Addition zweier achtstelliger Bitmuster

SUB: Subtraktion[36] zweier achtstelliger Bitmuster

CP: Vergleich (Compare) zweier achtstelliger Bitmuster

Diese achtstelligen Bitmuster interpretiert man üblicherweise als vorzeichenbehaftete oder -lose Zahlen.

Diese Operationen haben gemein, dass sie aus jeweils acht binären Operationen bestehen, die jeweils eine Stelle des Ergebnisses berechnen und zusätzlich vom Ergebnis der Operation der nächst niederwertigen Stelle abhängen.

Darüber hinaus gibt es auch hier wieder die Verantwortlichkeit des Steuerwerks die Datenwege, dem Zustand entsprechend, richtig zu schalten. Zur besseren Übersicht kann man die Zustände wie folgt beschreiben:

1) **Laden** der acht Einzelwerte in die Operationen
2) **Berechnen** der Einzelwerte inklusive Übertragsberechnung und -übernahme
3) **Setzen der Flags** entsprechend dem Ergebnis der Operation
4) **Speichern** des Ergebnisses

[36] Die Subtraktion wird in Form der *Ersatzaddition* durchgeführt.

3.2.7 Central Processing Unit

Die **CPU** (*Central Processing Unit*, *Zentralprozessoreinheit*) ist für die Abarbeitung von Programmen sowie die Konfiguration der Datenwege und der beteiligten Komponenten (z.B. ALU) zuständig. Die CPU unterhält auch Register, die für die Zwischenspeicherung von Operationsergebnissen verwendet werden können. Diese so genannten *Register* haben gegenüber dem Hauptspeicher einen erheblichen Geschwindigkeitsvorteil.

Da es für das weitere Verständnis notwendig ist, wird kurz auf die Register eingegangen: Register sind Speicher für ein acht- oder sechzehnstelliges Datenwort. Die Register befinden sich in der CPU und werden unter anderem für die Befehlsabarbeitung benutzt. So gibt es zum Beispiel ein *Program Counter-Register (PC)*, welches angibt, an welcher Stelle im Programm die CPU sich gerade mit der Ausführung befindet. Wichtig für die vorliegende Arbeit sind die Register A, B und C. Diese Register sind acht Bit breit und können als Parameter für Befehle sowie als Eingangswerte für Operationen genutzt werden. Dem so genannten *Akkumulator* A kommt dabei eine besondere Bedeutung zu, da er bei den betrachteten Operationen zwingend als Zielregister eingesetzt wird.

Ebenfalls eine wesentliche Bedeutung kommt dem Flag Register zu: Die acht einzeln zu betrachtenden Bits zeigen bestimmte Eigenschaften des Ergebnisses einer Operation an. Die einzelnen Bits werden **Flags** genannt, wobei für diese Arbeit vier erwähnenswert sind:

Das **Carry Flag** zeigt an, ob bei einer Rechen- oder Vergleichsoperation ein Übertrag vorhanden ist. Bei Logikoperationen wird das Carry Flag nicht verändert und bei Schiebe- und Rotationsoperationen entsprechend gesetzt. Das **Sign Flag** signalisiert ein negatives Vorzeichen bei einer vorzeichenbehafteten Zahl. Das **Overflow Flag** zeigt an, ob bei einer Rechen- oder Vergleichsoperation das Ergebnis zu groß für den darstellbaren Zahlenbereich ist (bei vorzeichenlosen 8-Bit-Zahlen ist dies 0-255, bei vorzeichenbehafteten Zahlen ist das der Wertebereich -128 bis +127). Dieses Flag teilt sich mit dem **Parity Flag** die Speicherzelle, da dieses nur bei Logik-, Schiebe- und Rotationsoperationen gesetzt wird. Das Parity Flag zeigt an, ob die Anzahl der, im Ergebnis enthaltenen unterschiedlichen, Werte gerade ist. Durch das **Zero Flag** wird angezeigt, ob alle Stellen des Ergebnisses gleich 0 sind.

Die Befehle des MC 8 setzen sich immer aus einem Befehl und einem Parameter zusammen. Der Parameter kann entweder auf ein Register referenzieren oder einen konkreten Zahlenwert repräsentieren. Da die Befehle in Maschinensprache und damit ausschließlich als binäre Wortfolgen vorliegen und für Menschen nicht leicht interpretierbar sind, wird in weiterer Folge eine einfache Darstellung mittels mnemonischen Abkürzungen der Befehlen (ADD, SUB, CP, etc.) verwendet.

In der Praxis haben moderne CPUs erheblich mehr Aufgaben als in diesem für die Lehre ausgelegten Übungsboard (bzw. Simulator) MC8. Zu diesen Aufgaben zählen unter anderem Multiplikation und Division. Moderne CPUs können oft um mehr als eine Stelle *Schieben* (Schiebeoperation siehe Abschnitt 3.2.8.2). Auch bezüglich der Wortbreite sind Unterschiede zu erkennen: Viele Prozessoren haben ein 16-, 32- oder 64-Bit-Wortbreite. Ein weiteres Thema ist Unterstützung der Parallelisierung von Prozessen. Darüber hinaus sind werden CPUs auf gewisse Aufgabengebiete hin entwickelt, zum Beispiel Prozessoren, die digitale Audiosignale von einem Format in ein anderes umwandeln können.

3.2.8 Arithmetic Logic Unit

Die **ALU** (*Arithmetic Logic Unit*, *Arithmetisch logische Einheit*) ist für Berechnungen, Boolsche sowie Schiebeoperationen zuständig.

3.2.8.1 Aufbau

Die ALU kann zwei Eingangswerte aufnehmen und legt das Ergebnis einer Operation an den Ausgangsleitungen an. Zusätzlich gibt es Verbindungen zu Flags der CPU, da diese bestimmte Eigenschaften von Operationsergebnissen aufnehmen können. Vom CPU-Steuerwerk erhält die ALU mittels zahlreicher Steuerleitungen Informationen zu den durchzuführenden Operationen. Schließlich enthält die ALU natürlich die Schaltungen, um mathematische Berechnungen, boolsche und Schiebe-Operationen vorzunehmen.

3.2.8.2 Operationen

Die ALU des MC8 kann folgende Operationen durchführen:

Rechenoperationen

Dazu zählen die **Addition** (**ADD**) und die **Subtraktion** (**SUB**). Diese Befehle sind samt allen möglichen Parametern in dieser Tabelle zusammengefasst:

Befehl mit Parameter	Durchgeführte Operation
ADD A	A := A + A
ADD B	A := A + B
ADD C	A := A + C
ADD dat8	A := A + dat8
SUB A[37]	A := A – A
SUB B	A := A – B
SUB C	A := A – C
SUB dat8	A := A – dat8

Tab. 7: Übersicht der Rechenoperationen der ALU des MC8

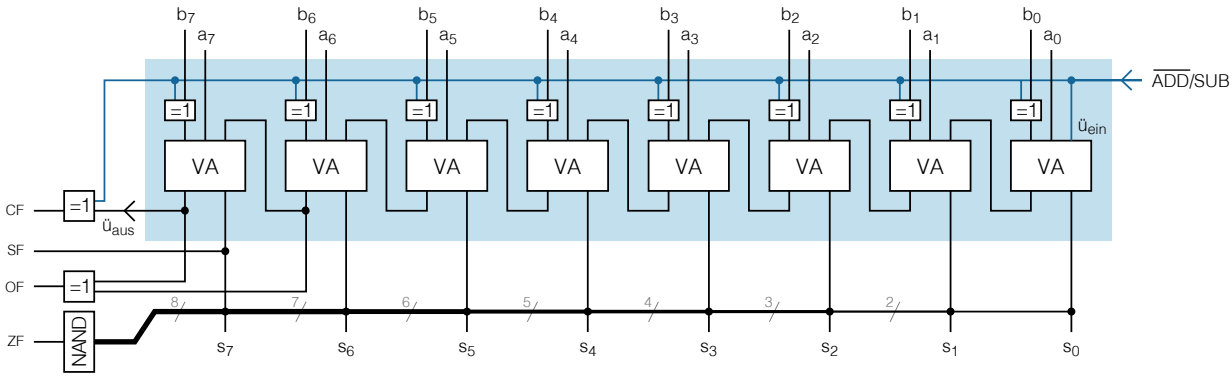


Abb. 17: Aufbau eines Rechenwerks für ADD, SUB und CP[38]

[37] SUB A hat stets das Ergebnis 0, da Minuend und Subtrahend ident sind (in beiden Fällen der Akkumulator A). So findet dieser Befehl in der Praxis keine Verwendung.

[38] CP ist eine Vergleichsoperation zwischen zwei achtstelligen Bitmustern, sie im nächsten Abschnitt erläutert

Das Rechenwerk verfügt neben den Eingängen für die zu verarbeitenden Daten auch über einen Steuerleitungseingang, der darüber entscheidet, ob eine Addition oder Subtraktion durchgeführt wird.

Das Steuerleitungssignal wird dem ersten Volladdierer als Übertrag $ü_{ein}$, allen XOR-Gattern der Eingangsleitungen des Subtrahenden (siehe *Ersatzaddition*, Abschnitt 3.2.2.2) und in weiterer Folge dem *Carry Flag* vorgelagerten XOR-Gatter zugeführt.

Ist diese Steuerleitung nicht gesetzt, wird eine Addition durchgeführt, da der erste Übertrag 0 ist und alle mit der Steuerleitung beschickten XOR-Gatter das Eingangssignal unverändert übernehmen. Wenn die Steuerleitung aktiv ist, ist auch der erste Übertrag 1 und die zuvor genannten XOR-Gatter verhalten sich nun wie ein NOT-Gatter. Das entspricht der *Ersatzaddition*, da auf diese Weise ein *Zweierkomplement* des Subtrahenden gebildet wird.

Vergleichsoperation

Zwei Zahlenwerte können mit dem Befehl **Compare** (**CP**) verglichen werden. Dieser Vorgang unterscheidet sich von der Subtraktion nur dadurch, dass an den Ausgängen kein Ergebnis anliegt. Die Flags werden allerdings wie gewohnt gesetzt. Die Aufgabe, die Ergebniswerte in das Register A zu speichern (oder eben nicht beim Befehl CP), übernimmt das Steuerwerk und wird daher an dieser Stelle nicht weiter behandelt.

Befehl mit Parameter	Durchgeführte Operation
CP A	A – A
CP B	A – B
CP C	A – C
CP dat8	A – dat8

Tab. 8: Übersicht der Vergleichsoperationen der ALU des MC8

Es werden sowohl bei den Rechen- als auch bei den Vergleichsoperationen stets alle Flags gesetzt. Dies geschieht immer nach dem gleichen Prinzip:

CF := $ü_{aus,7}$
SF := s_7
OF := $ü_{aus,7}$ XOR $ü_{aus,6}$
ZF := NOT(s_0) AND NOT(s_1) AND NOT(s_2) AND NOT(s_3) AND
NOT(s_4) AND NOT(s_5) AND NOT(s_6) AND NOT(s_7)

Eine Übersicht aller Flags und ihrer Werte findet sich am Ende des Abschnitts.

Anmerkungen zur Compare-Operation:

- Die Berechnungsergebnisse der Compare-Operation (welche eigentlich eine Subtraktion – genau genommen eine Ersatzaddition – ist), werden wie bereits erwähnt nicht gespeichert, daher steht in der Tabelle vor der durchgeführten Operation kein Zuweisungsoperator.
- Will man Aussagen über die Beziehung zweier Zahlen zueinander machen, muss man zunächst unterscheiden, ob es sich um eine vorzeichenlose oder eine Zahl in Zweierkomplement-Darstellung handelt:

Auswertung	vorzeichenlose Zahl	Zahl in Zweierkomplement-darstellung
Register A < Vergleichswert	CF == 0 ZF == 0	ZF == 0 (CF XOR OF) == 0
Register A = Vergleichswert	ZF == 1	ZF == 1
Register A > Vergleichswert	CF == 1	(OF XOR SF) == 1

Tab. 9: Auswertung der Flags nach einer Compare-Operation

Logische Operationen

Das sind die binären logischen Operationen **Und** (**AND**), **Oder** (**OR**) und **Exklusives Oder** (**XOR**). Diese Operationen werden durch einfache Logikgatter, die der zugehörigen Funktion entsprechen, realisiert. Die technische Realisierung sieht vor, dass alle drei Operationen parallel durchgeführt werden und nur das Ergebnis der gewählten Funktion an den Ausgang gelangt. Technisch wird das mittels Steuerleitungen, die vom Steuerwerk gesetzt werden, gelöst. Diese Steuerleitungen führen zu je einem *Mulitplexer* (*MUX*)[39], der das richtige Signal zum Ausgang durchschaltet.

Bei diesen und den folgenden Operationen wird das, bei den Rechen- und Vergleichsoperationen verwendete, Overflow Flag durch das Parity Flag ersetzt. Das Carry Flag wird dabei nicht verändert, da es keine Bedeutung hat.

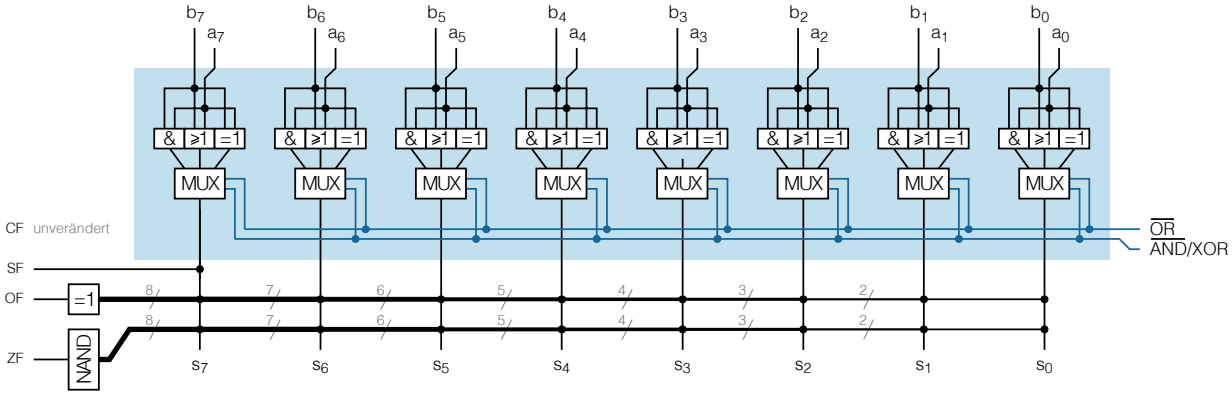


Abb. 18: Aufbau des Schaltnetzes der boolschen Operationen

[39] Ein Multiplexer (MUX) ist ein Selektionsschaltnetz, mit dem aus einer Anzahl von Eingangssignalen eines ausgewählt werden kann, etwa bei einem Speicherzugriff oder der Anwahl oder Durchschaltung analoger und digitaler Signalkanäle.
Vgl. auch [WIKI2008C]

Befehl mit Paramter	Durchgeführte Operation
AND A	A := (A AND A)
AND B	A := (A AND B)
AND C	A := (A AND C)
AND dat8	A := (A AND dat8)
OR A	A := (A OR A)
OR B	A := (A OR B)
OR C	A := (A OR C)
OR dat8	A := (A OR dat8)
XOR A	A := (A XOR A)
XOR B	A := (A XOR B)
XOR C	A := (A XOR C)
XOR dat8	A := (A XOR dat8)

Tab. 10: Übersicht der aussagenlogischen Operationen der ALU des MC8

Schiebe- und Rotationsoperationen

Beim MC8 ist es möglich, den Inhalt des Registers A bitweise zu **schieben** (engl. **shift**, **SH**). Wird dabei der herausfallende Wert am Anfang wieder hereingeschoben, spricht man von **Rotation** (engl. **rotation**, **RO**). Wird dieser Wert jedoch in das Carry Flag gespeichert und der alte Wert des Carry Flags am Anfang hereingeschoben, handelt es sich um eine **Rotation mit Carry** (**rotation with carry**, **RC**). Alle diese Operationen können nach *links* und nach *rechts* durchgeführt werden: **SHL** (shift left), **SHR** (shift right), **ROL** (rotate left), **ROR** (rotate right), **RCL** (rotate with carry left) und **RCR** (rotate with carry right).

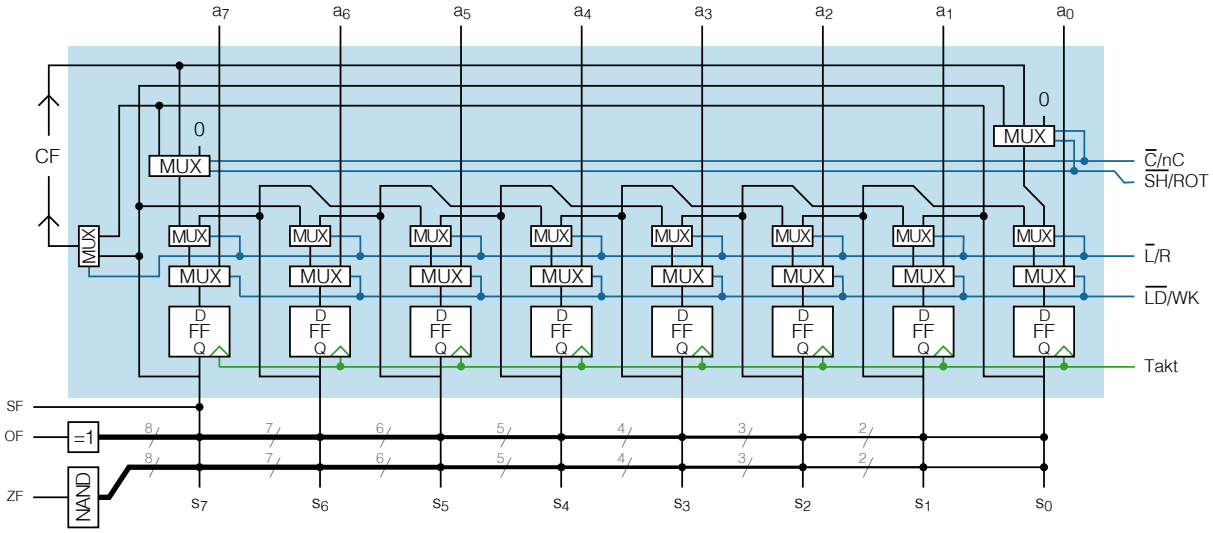


Abb. 19: Aufbau der Schiebe- und Rotationsoperationen

Bei der folgenden Tabelle werden die Bit der Werte dargestellt, da diese für das Verständnis notwendig sind. A steht für das Akkumulator-Register A, CF für das Carry Flag.

Befehl und Parameter[40]	Durchgeführte Operation
SHL	$A(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) := (a_6, a_5, a_4, a_3, a_2, a_1, a_0, 0)$
SHR	$A(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) := (0, a_7, a_6, a_5, a_4, a_3, a_2, a_1)$
ROL	$A(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) := (a_6, a_5, a_4, a_3, a_2, a_1, a_0, a_7)$
ROR	$A(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) := (a_0, a_7, a_6, a_5, a_4, a_3, a_2, a_1)$
RCL	$A(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) := (a_6, a_5, a_4, a_3, a_2, a_1, a_0, CF)$ $CF := a_7$
RCR	$A(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) := (CF, a_7, a_6, a_5, a_4, a_3, a_2, a_1)$ $CF := a_0$

Tab. 11: Übersicht der Schiebe- und Rotationsoperationen der ALU des MC8

Operation	Carry Flag (CF)	Sign Flag (SF)	Overflow Flag (OF)	Parity Flag (PF)	Zero Flag (ZF)
ADD	$ü_{aus,7}$	s_7	$ü_{aus,7}$ XOR $ü_{aus,6}$		NOT(s_0) AND NOT(s_1) AND NOT(s_2) AND NOT(s_3) AND NOT(s_4) AND NOT(s_5) AND NOT(s_6) AND NOT(s_7)
SUB	NOT($ü_{aus,7}$)				
CP					
AND	unverändert			XOR ($s_{0..7}$)	
OR					
XOR					
SHL	a_7				
SHR	a_0				
ROL	a_7				
ROR	a_0				
RCL	a_7				
RCR	a_0				

Tab. 12: Flags und deren Werte

[40] Diese Befehle beziehen sich immer auf den Akkumulator A, daher benötigen diese Befehle keinen Parameter. Aus Konsistenzgründen wurde diese Tabelle jedoch gleich strukturiert wie jene davor.

3.2.9 Besonderheiten bei der Simulation

Um die Visualisierung der Abläufe in der ALU für die Lernenden möglichst verständlich zu gestalten, müssen im Hinblick auf die technische Korrektheit Kompromisse eingegangen werden. Folgende Änderungen wurden vorgenommen:

3.2.9.1 Warten auf alle notwendigen Eingabewerte

Bei der Animation der Werte auf den Leitungen wurde darauf geachtet, dass die Leitungslängen für die Dauer der Animation nicht relevant sind. Es wird gewartet, bis alle notwendigen Eingangswerte vorliegen und erst dann wird die Operation durchgeführt.
Beispiel: Die Eingangswerte liegen in der Simulation visuell auf unterschiedlichen Höhen. Durch eine einheitliche Animationsgeschwindigkeit würde der Wert auf der längeren Leitung mehr Zeit bis zum Ziel (z.B. ein AND-Gatter) benötigen, als jener Wert auf der kürzeren. Durch diese Verzögerung würde sich genau genommen das Ergebnis schon vor Anliegen des zweiten, länger laufenden Wertes bilden. In diesen Fällen wartet die Animation bis alle Werte am Gatter anliegen und führt erst dann die Operation durch.
Dies wird am Beispiel Subtraktion (Ersatzaddition) veranschaulicht:

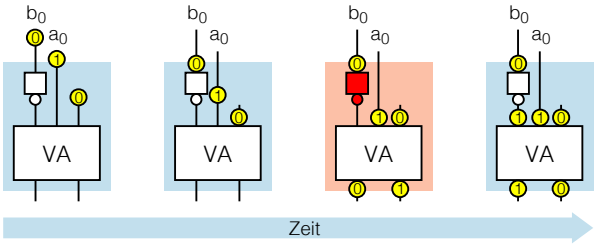


Abb. 20: zeitlicher Ablauf der Animation mit verzögerter Wertbildung

Diese vier Abbildungen zeigen den zeitlichen Verlauf einer Animation. Im ersten Bild sieht man die Ausgangssituation – die Animation wurde gerade gestartet. Die Werte liegen am Beginn der Leitungen. Im nächsten Bild ist die Animation so weit fortgeschritten, dass der Wert von b_0 am NOT anliegt. In der dritten Abbildung kann man erkennen, wie durch die Gatterverzögerung das Zwischenergebnis nach dem NOT noch nicht gebildet wurde, die anderen Werte jedoch schon am VA anliegen. Theoretisch würde der nicht synchrone Volladdierer sofort einen Ausgangswert bilden, der dem tatsächlichen Ergebnis diametral entgegensteht. Im letzten Bild sieht man die korrigierte Version: Die Ausgangswerte bilden sich erst beim Anliegen aller Eingangswerte beim VA.

3.2.9.2 Takt und Steuersignale statisch

Der Takt und die Steuersignale des Steuerwerks werden nicht mit Hilfe animierter Werte dargestellt (wie Eingangswerte), um Laufzeiteffekte zu vermeiden. Das Ziel ist, dass der Takt und die Steuersignale bei allen Flip-Flops, Gattern und Flags gleichzeitig anliegen. Diese Approximation ist für die Verständlichkeit förderlich und stellt ein hinreichend genaues Modell für die Lernenden dar.

Eine Ausnahme bilden hierbei die Schiebe- und Rotationsoperationen, da diese zwei Takte benötigen. Die Darstellung des Taktimpulses erfolgt durch 250 ms gelb eingefärbte Steuerleitungen, die vor und nach dieser Zeit inaktiv (grau) gezeigt werden.

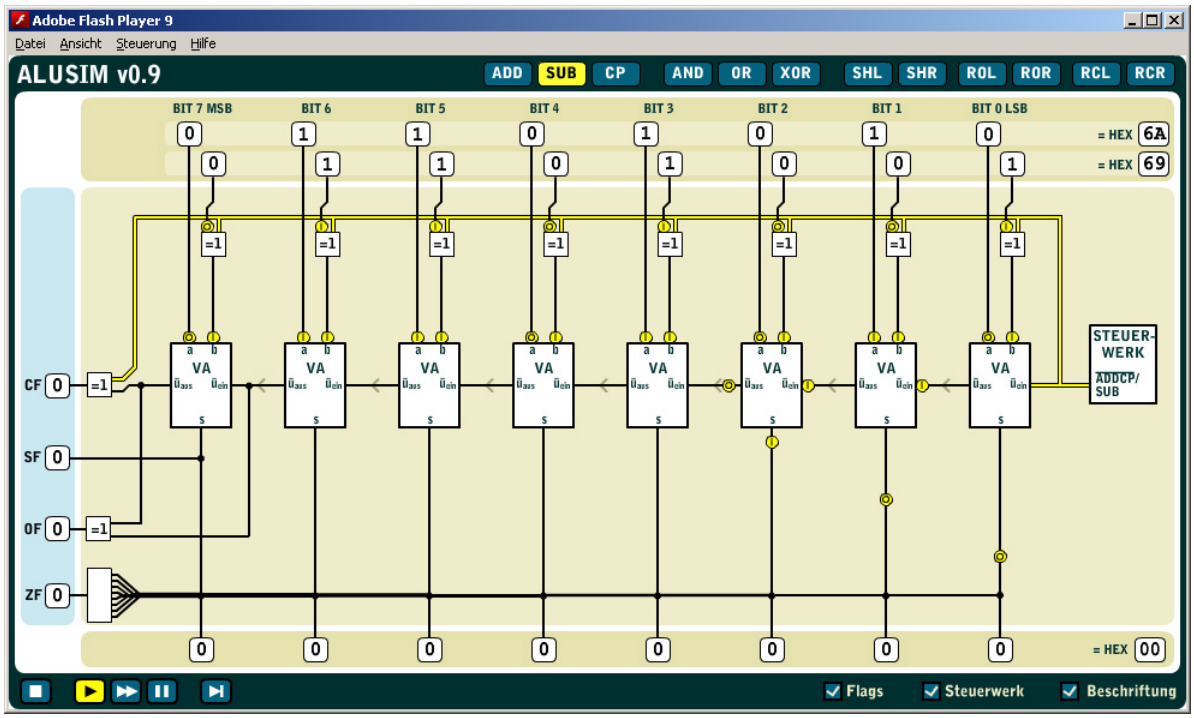


Abb. 21: Animation einer Subtraktion mit statischer Steuerleitung

3.2.9.3 Vereinfachung des Modells

Es ist zweckmäßig, für Lehrzwecke ein Modell zu entwerfen, das darauf ausgelegt ist, dass die Studierenden die Abläufe gut nachvollziehen können und ihrem Wissenstand entsprechend differenzierte Vorgehensmodelle zu präsentieren. In diesem Abschnitt wird auf den Vereinfachungsprozess des Modells eingegangen.

Flags und Steuerleitungen können ausgeblendet werden. Es ist möglich, eine vereinfachte Darstellung der Abläufe einzustellen. Das Abstraktionsniveau beim Modell ist dabei so hoch, dass selbst Flags (die normalerweise stets vorhanden sind und in der Regel auch einen neuen Wert nach der Operation erhalten) in dieser Darstellung ausgeblendet werden. Bei den *Rotation mit Carry*-Funktionen RCL und RCR ist es allerdings auch bei ausgeblendeten Flags für das Verständnis notwendig, das Carry Flag zu zeigen. Ohne dieses ist es nicht möglich, eine schematische Darstellung von RCL und RCR zu visualisieren. Am Beispiel der Operation SHL wird der Vereinfachungsprozess dargestellt:

Ausgangssituation

Ausgehend von der Schaltung mit Steuerwerk, Flags und Beschriftung werden Schritt für Schritt Leitungen und Bauelemente entfernt oder vereinfacht.

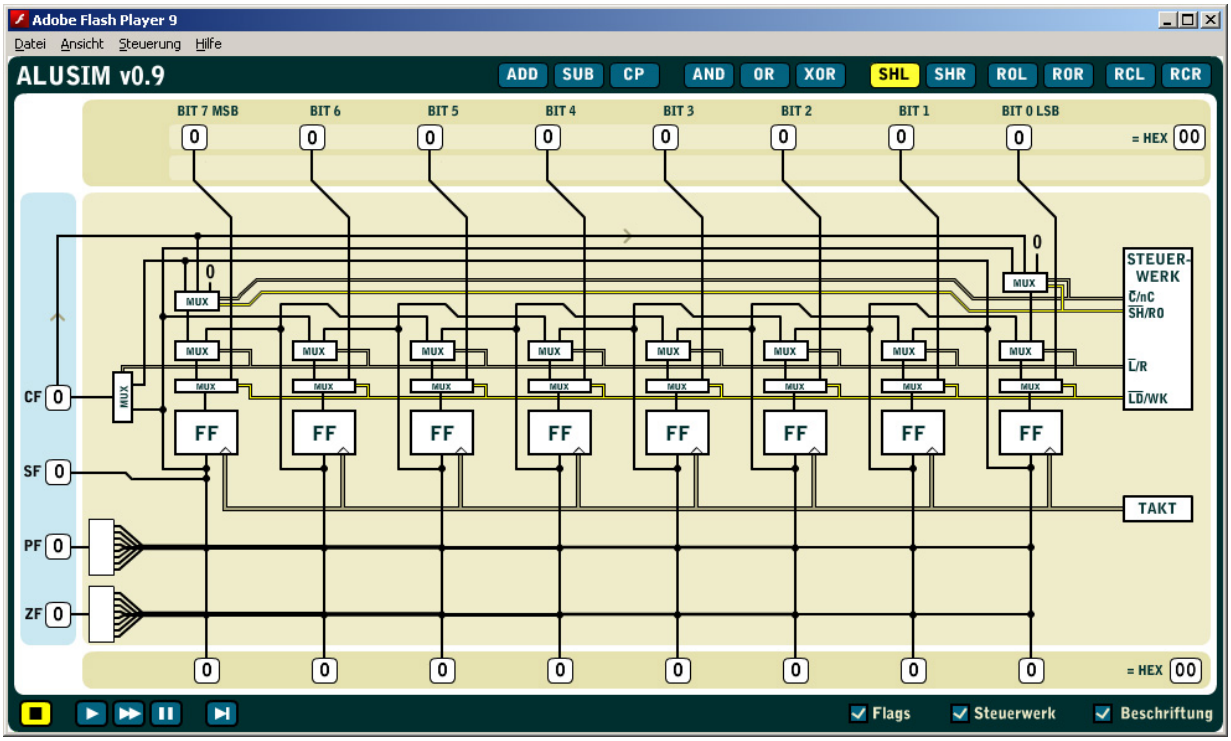


Abb. 22: Komplexe Darstellung der Schiebe- und Rotationsoperationen, die vereinfacht werden kann

In dieser Darstellung befinden sich alle notwendigen Details. Da das Parity Flag und das Zero Flag bei allen Operationen auf die gleiche Weise gesetzt werden und für das Verständnis des Befehls SHL nicht notwendig sind, kann man die Zuleitungen in einem ersten Durchgang entfernen.

Parity Flag und Zero Flag entfernt

Die Schaltung bietet immer noch genügend Potential für Vereinfachung. Im vorherigen Schritt wurden Schaltungsteile entfernt, die für das Verständnis nicht unmittelbar notwendig sind und bei vielen Operationen auf gleiche Weise erfolgen.

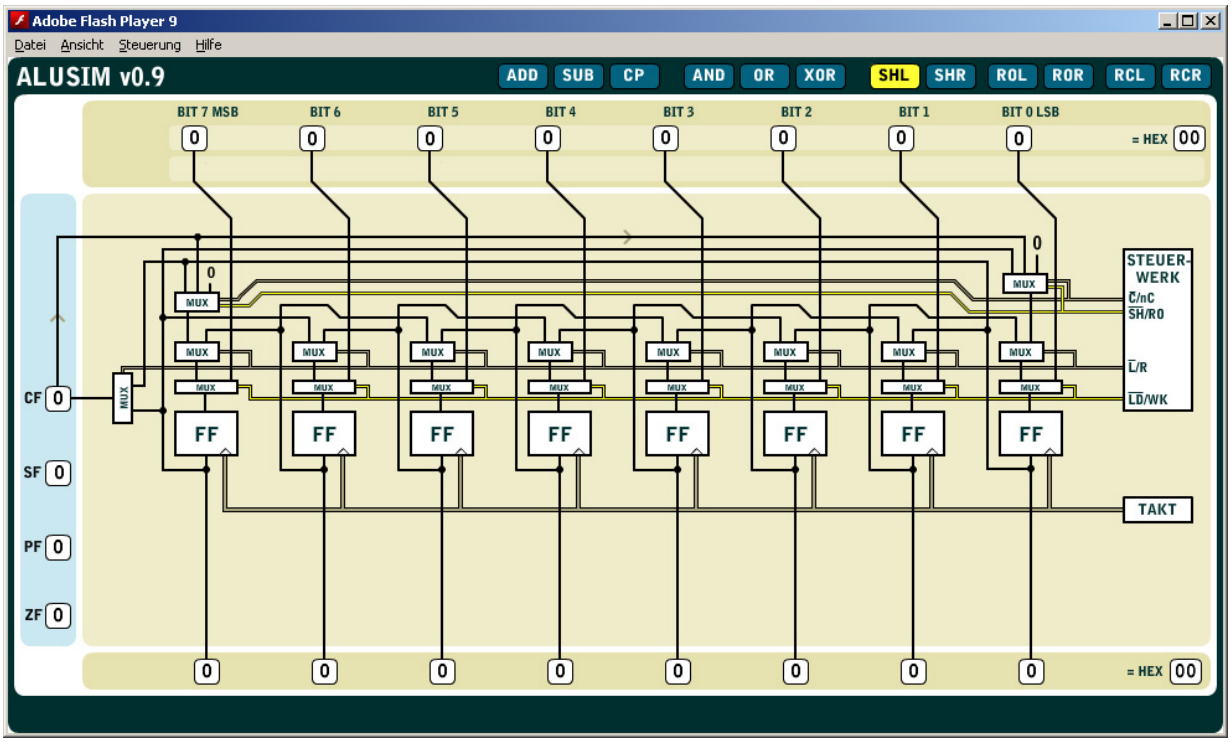


Abb. 23: Vereinfachung der Darstellung von SHL: Entfernte Parity Flag- und Zero Flag-Zuleitungen.

Nun kann man die Anzahl der Leitungen erheblich reduzieren (und damit die Übersichtlichkeit erhöhen), indem man die Steuerleitungen ausblendet. Die Idee, auch den Takt und dessen Leitung zu entfernen, ist nicht umsetzbar: Da die Schaltung zwei Taktzyklen benötigt, um die Operation SHL durchführen, muss der Taktgeber als zentrales Element erhalten bleiben.

Steuerwerk ausblenden

Der nächste Ansatz begründet sich in der Überlegung, dass bei SHL das Carry Flag zwar gesetzt wird, aber nicht für die Verarbeitung des Befehls notwendig ist. Dies ist bei RCL und RCR nicht so: Bei diesen Operationen wird das Carry Flag nicht nur beschrieben, sondern auch der Wert des Carry Flags der Schaltung zur Verarbeitung zugeführt.

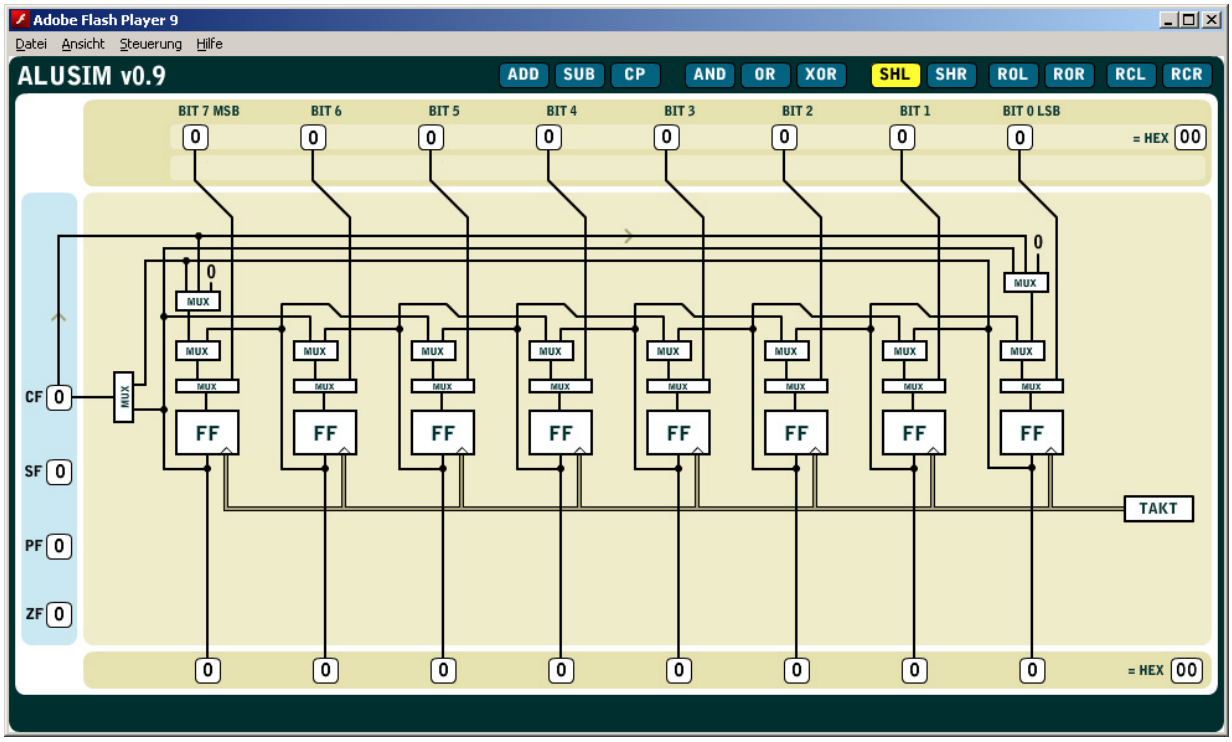


Abb. 24: Vereinfachung der Darstellung von SHL: Entfernung des Steuerwerks und der Steuerleitungen

Im nächsten Schritt werden alle Leitungen vom und zum Carry Flag entfernt. Da der MUX vor dem Carry Flag ohne Zu- und Ableitungen sinnlos ist, wird dieser ebenfalls aus der Visualisierung entfernt.

Entfernung aller Leitungen zum und vom Carry Flag

Diese Darstellung ist schon stark vereinfacht. Jeder Ausgang eines Flip-Flops ist immer noch mit seinem Vorgänger und seinem Nachfolger verbunden (mit Ausnahme des LSB- und MSB-Flip-Flops).

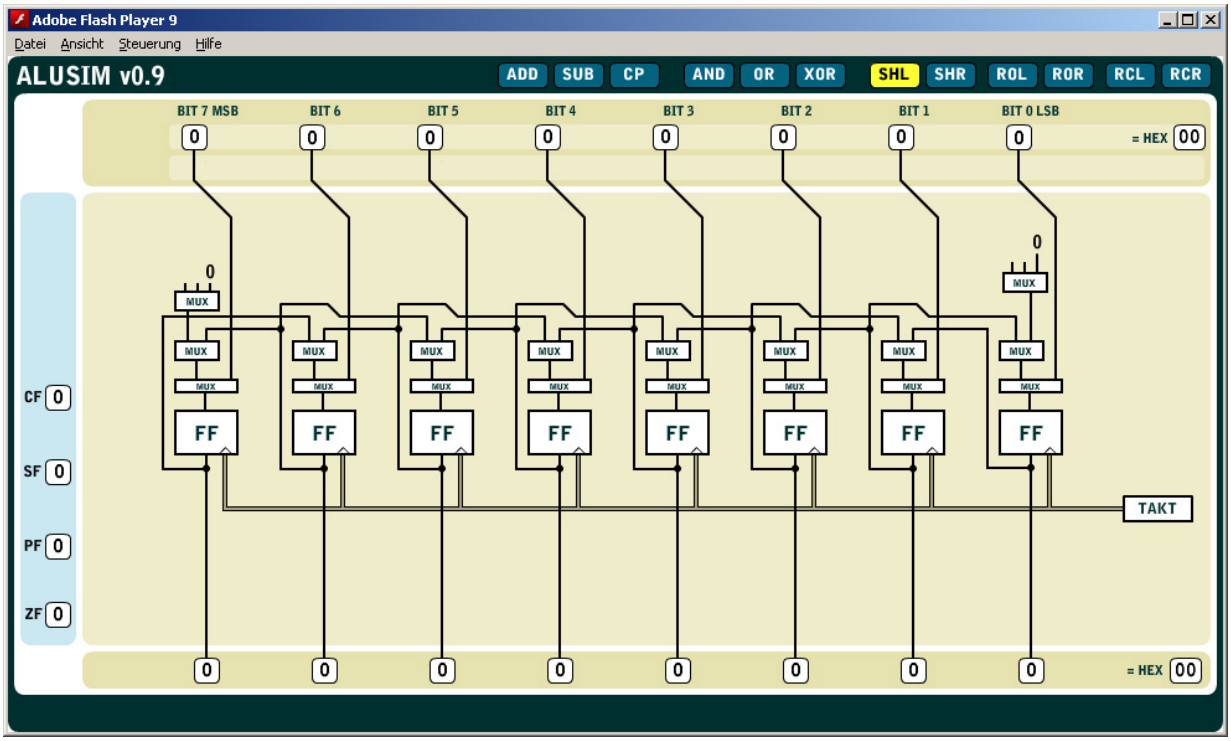


Abb. 25: Vereinfachung der Darstellung von SHL: Leitungen vom und zum Carry Flag wurden entfernt

Der nächste Schritt ergibt sich aus der Tatsache, dass die Umsetzung des Befehls SHL visualisiert werden soll. Für SHL sind nämlich die Verbindungen eines Flip-Flops nur zu seinem linken Nachbarn notwendig

Entfernung der Leitungen zum niederwertigeren Flip-Flop

Nach dem Entfernen der Verbindungen zum rechten Nachbarn ist man schon fast am Ziel. Es fällt jedoch auf, dass noch einiger Optimierungsbedarf für die vorhandenen Multiplexer besteht.

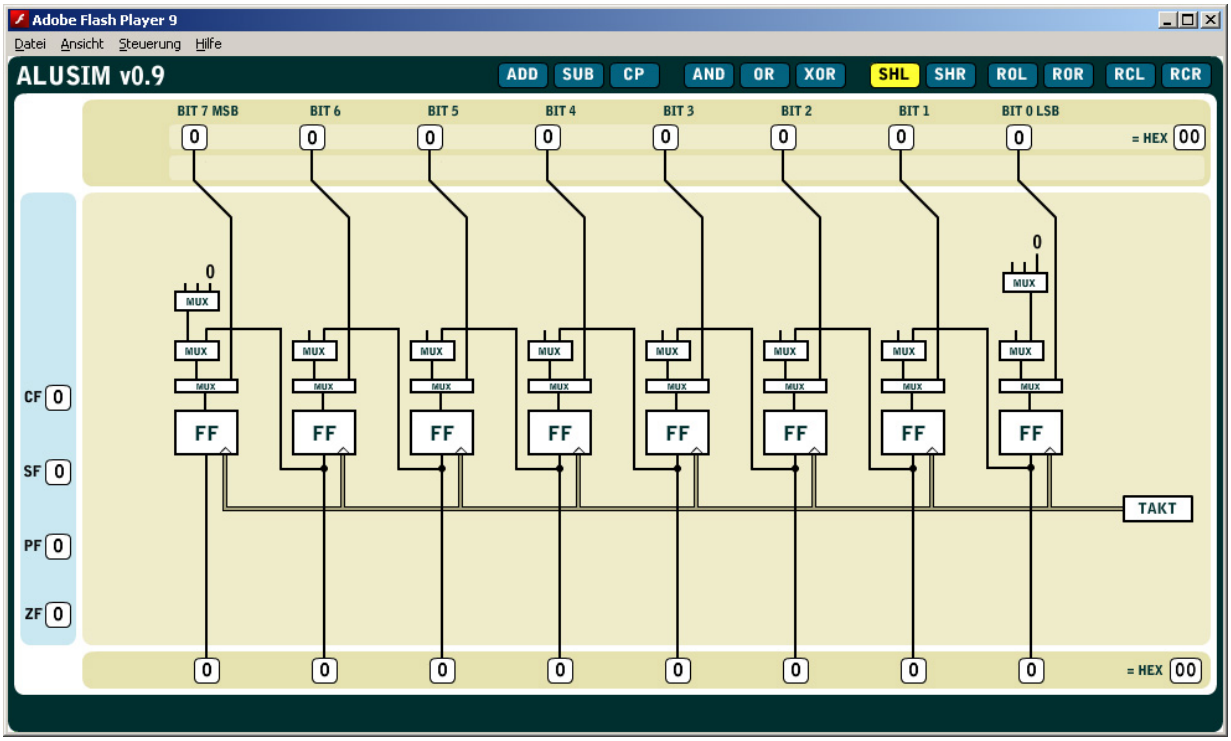


Abb. 26: Vereinfachung der Darstellung von SHL: Leitungen zum rechten Nachbarn

Die Multiplexer jeder Stelle werden nun zu einem konsolidierten, schematisierten Multiplexer pro Flip-Flop zusammengefasst.

Konsolidierung der Multiplexer

Die eingeführten Multiplexer schaffen schon eine übersichtliche Darstellung. In einem letzten Schritt wird nun noch einmal Hand angelegt, um die Schaltungsvereinfachung zu finalisieren.

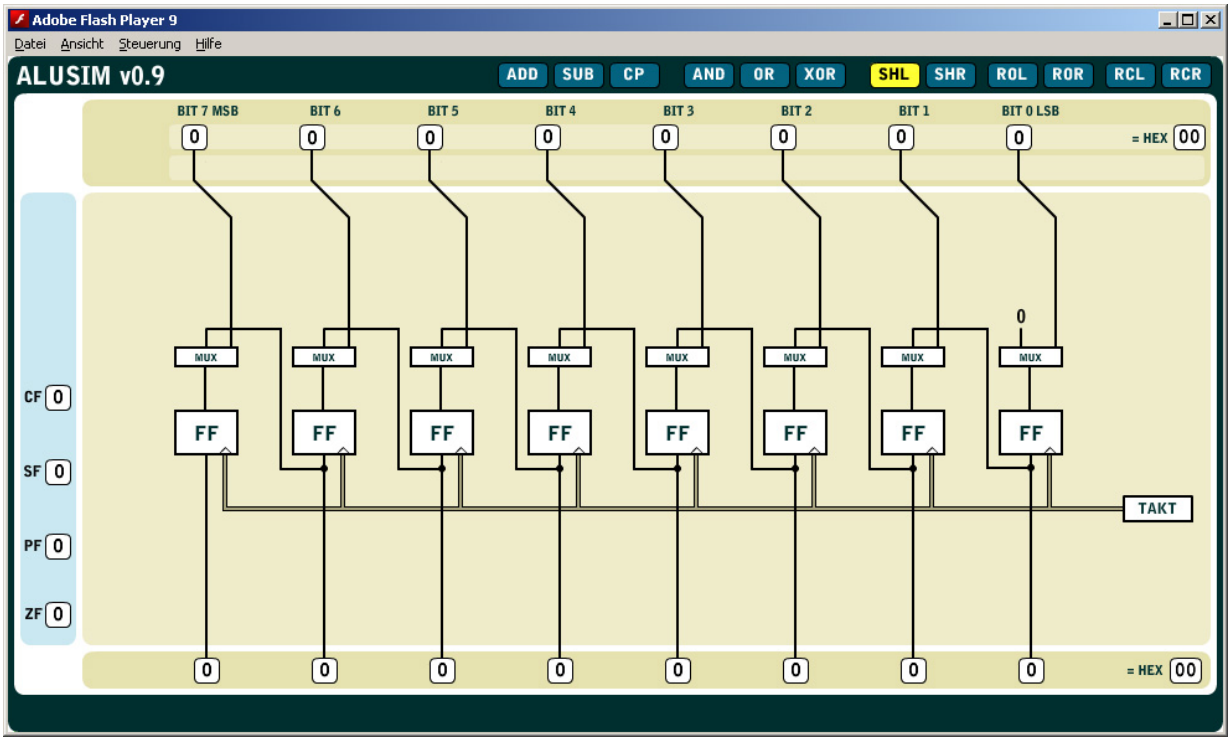


Abb. 27: Vereinfachung der Darstellung von SHL: Konsolidierte Multiplexer

Im letzten Schritt verändert man die Leitungsführung der Eingänge der Multiplexer, so dass die Eingangsleitungen auf geradem Weg in diese geführt werden können.

Finalisierung

Zusätzlich zur Leitungsführung wurde der Multiplexer nicht mehr als MUX bezeichnet, sondern zeigt die durchgeschaltete Leitung wie ein Schalter. Das dient dem Verständnis, da ohne Steuerleitungen nicht klar wäre, welche Leitung der Multiplexer auswählen würde.

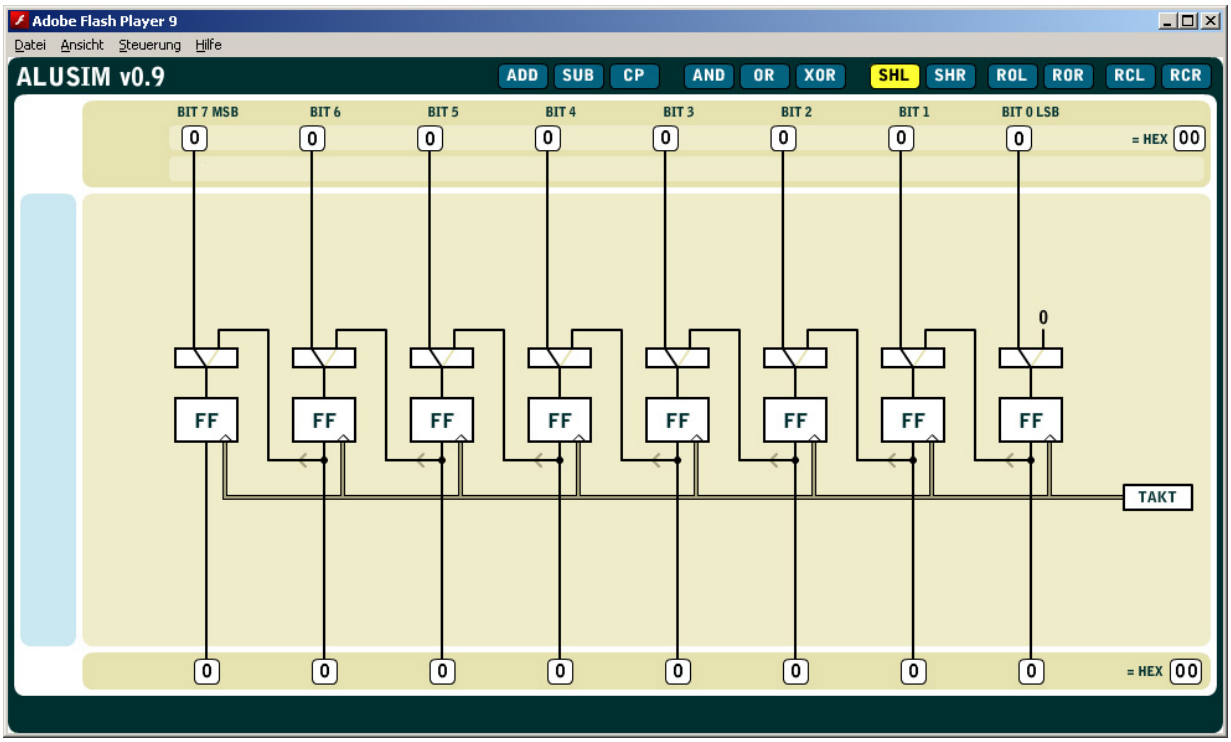


Abb. 28: Vereinfachte Darstellung von SHL

Die Schaltungsvereinfachung ist hiermit abgeschlossen.

Zusammenfassend lässt sich sagen, dass zur Vereinfachung der Schaltung folgende Regeln beachtet wurden:

- **Entfernen von Schaltungsteilen**, die für die Durchführung dieses konkreten Befehls **nicht unbedingt notwendig** sind.
- **Ausblenden des Schaltwerks und der Steuerleitungen** zur Steigerung der Übersichtlichkeit.
- Flags sind für das Verständnis nicht relevant und deren Werte werden unabhängig von der Operation immer auf die gleiche Weise ermittelt. Deshalb können **Flags ausgeblendet** werden. Eine Ausnahme stellt das Carry Flag dar, dem bei RCL und RCR eine besondere Bedeutung zukommt und das in diesem Fall dennoch gezeigt wird.
- **Zusammenfassen von Schaltungsteilen**, sofern die Veränderungen Vorteile bei der Darstellung oder Wissensvermittlung bringen.

3.3 Adobe Flash

Dieser Abschnitt zeigt kurz die historische Entwicklung von Programmiersprachen auf, um schlussendlich auf die aktuelle Situation einzugehen. Darüber hinaus wird auch dargestellt, warum der Simulator in **Adobe Flash** realisiert wurde.

3.3.1 Arten und Entwicklung von Programmiersprachen

1. Generation	Maschinensprache	Die Programme und notwendigen Daten werden in einem maschinenlesbaren Format zur Verfügung gestellt. Beispiel: Maschinenbefehl `00110001 11111111 11111111`
2. Generation	Assembler	Programme werden in - für den Menschen - leichter interpretierbarer Form geschrieben. Immer noch sehr hardwarenah, jeder Befehl wird in einen oder wenige Maschinensprachenbefehle umgesetzt. Maschinenbefehl des letzten Beispiels in Assembler: `MOV SP, FFFFh`
3. Generation	Höhere Programmiersprachen	Erlauben prozedurales Programmieren: Funktionen, die man öfter (und an unterschiedlichen Position) benötigt, kann man in Prozeduren kapseln und parametrisierbar machen.
Heute relevant	Objektorientierte Programmiersprachen	Diese Sprachen ziehen eine höhere Abstraktionsebene ein: Eigenschaften (Properties) und Funktionen (Methods) ergeben zusammen ein so genanntes *Objekt*.
	Skriptsprachen	Durch das Internet und dessen Erfordernisse wuchsen die Anzahl und die Qualität der Skriptsprachen. Diese werden in der Regel von einem Interpreter-Programm Schritt für Schritt abgearbeitet. Bis zur Version 8 (mit der Programmiersprache Actionscript 2) war Adobe Flash eine Interpretersprache.

Tab. 13: Übersicht über die Entwicklung der Programmiersprachen

3.3.2 Geschichtliche Entwicklung von Adobe Flash

Adobe Flash gibt es seit 1997 mit der Programmiersprache *Actionscript 1*. *Adobe Flash* ist ein Programm zur Erstellung von Grafiken, die mittels *Actionscript* Aktionen durchführen können. Dadurch wurden Animationen und Benutzereingaben möglich.

Relevant ist *Adobe Flash* erst seit 2003, denn zu diesem Zeitpunkt hat der Hersteller (damals Macromedia) die Skriptsprache (nun Actionscript 2) erheblich verbessert und die Animationsmöglichkeiten verfeinert. Allerdings war eine, aus informationstechnischer Sicht, brauchbare Software-Architektur immer noch nur mit umständlichen Hilfskonstruktionen möglich.

Die wesentlichen Stärken spielte damals *Adobe Flash* im Umgang mit Multimedia-Elementen aus, denn es war nun leicht, aufwändige Grafiken, Animationen und Sound in eine *Adobe Flash*-Datei zu verpacken. Auch kümmerte sich *Adobe Flash* um kleine Dateigrößen (damals wichtiger als heute), welche es möglich machten, dass sich eine immer breiter werdende Menge an Personen die Inhalte im Internet mit dem - zu diesem Zeitpunkt schon weit verbreiteten - *Adobe Flash*-Player ansehen konnte.

Nachdem Adobe den ursprünglichen Hersteller *Macromedia* übernommen hatte, wurde die Programmiersprache *Actionscript* komplett neu geschrieben (jetzt Version 3). Aus technischer Sicht ein Meilenstein, da *Adobe Flash* nun mit einer, dem modernsten Stand der Softwaretechnik entsprechenden, Programmiersprache ausgestattet ist, die den großen, verbreiteten und gewachsenen Vorbildern (*Java*, *C#*) um nichts nachsteht.

3.3.3 Vergleich von Adobe Flash mit anderen technischen Plattformen

Bis vor zwei Jahren waren *Adobe Flash* und die Programmiersprache Java die einzigen wesentlichen Möglichkeiten im Web dynamischen Inhalt zu präsentieren. Das hat sich geändert, denn *Adobe Flash* steht nun in direkter Konkurrenz zu *Microsofts Silverlight* und *JavaFX*.

Microsoft Silverlight ist *Adobe Flash* sehr ähnlich und ist auch fast annähernd gleich mächtig. Die Programmierumgebung ist weniger multimedialastig, aber sehr gut geeignet, Projekte umzusetzen. *Microsoft Silverlight* ist noch zu kurz am Markt, um wesentliche Marktanteile zu haben (der *Microsoft Silverlight*-Player ist noch nicht auf ausreichend vielen Computern installiert). Daher fällt diese Lösung aus, da in der Regel zunächst *Microsoft Silverlight* installiert werden müsste und erst dann das E-Learning Modul verwendet werden kann.

JavaFX ist ein Ansatz[41] von *Sun* (dem Hersteller von *Java*[42]), multimediale Anwendungen zu entwickeln. Da *JavaFX* aber noch in der Entwicklung steckt und der vorhandene Zwischenstand nicht den Komfort von *Adobe Flash* oder *Microsoft Silverlight* bietet – inklusive der kaum vorangeschrittenen Verbreitung – ist *JavaFX* obgleich seiner wachsenden Angebote nicht als Plattform geeignet, um E-Learning-Inhalte zu produzieren.

Adobe Flash hat eine sehr gute Entwicklungsumgebung für Multimediainhalte und für die Erstellung der Actionscript-Programme. Abgesehen davon gibt es Open-Source- und Closed-Source-Lösungen zur Optimierung der Arbeitsvorgänge beim Programmieren. Darüber hinaus ist die Verbreitungsrate des aktuellen *Adobe Flash*-Players in der Region Europa 96,5%[43], was einer nahezu vollständigen Abdeckung gleichkommt. Die Benutzer müssen also de facto nichts installieren, wollen sie dieses E-Learning Modul verwenden.

Aufgrund der hervorragenden Entwicklungstools und der weiten Verbreitung wurde Adobe Flash als Umsetzungsplattform gewählt. Als positiv erweist sich dabei, dass die Zusammenführung (siehe Abschnitt 5.4 Vernetzung der Module) mit dem MC8-Simulator und dem Adobe Flash-Assembler in ein großes E-Learning Modul durch die gemeinsame technische Basis bei der Umsetzung erleichtert wird.

[41] Die Alternative von Sun zu JavaFX sind Java Applets, die jedoch eine sehr uneinheitliche Verbreitung haben (jeder Browser unterstützt andere Versionen) und kommen daher auch nicht in Frage.
[42] Java ist eine Programmiersprache, die sowohl im wissenschaftlichen als auch im wirtschaftlichen Bereich verwendet wird. Sie ist zum Zeitpunkt der Erstellung dieser Diplomarbeit eine der dominierenden Programmiersprachen am Markt.
[43] [ADOB2008A]

Bei der Umsetzung werden Animationen und Grafiken in *Adobe Flash CS3* umgesetzt und über ein *Adobe-Flash*-Bibliotheksdateiformat exportiert. Diese Bibliothek wird eingebunden und so der Workflow der grafischen Arbeiten und Programmiertätigkeit separiert. Das hat den Vorteil, dass es möglich ist, Grafiken zu ändern, ohne den Programmcode zu modifizieren und umgekehrt.

Die Erstellung des Programmcodes wurde mit einer Entwicklungsumgebung eines Drittanbieters (*Powerflasher - Flash Developer Toolkit, FDT*) optimal unterstützt. Diese bietet erheblich mehr Komfort bei der Erstellung des Programmcodes als die von Adobe angebotene Entwicklungsumgebung.

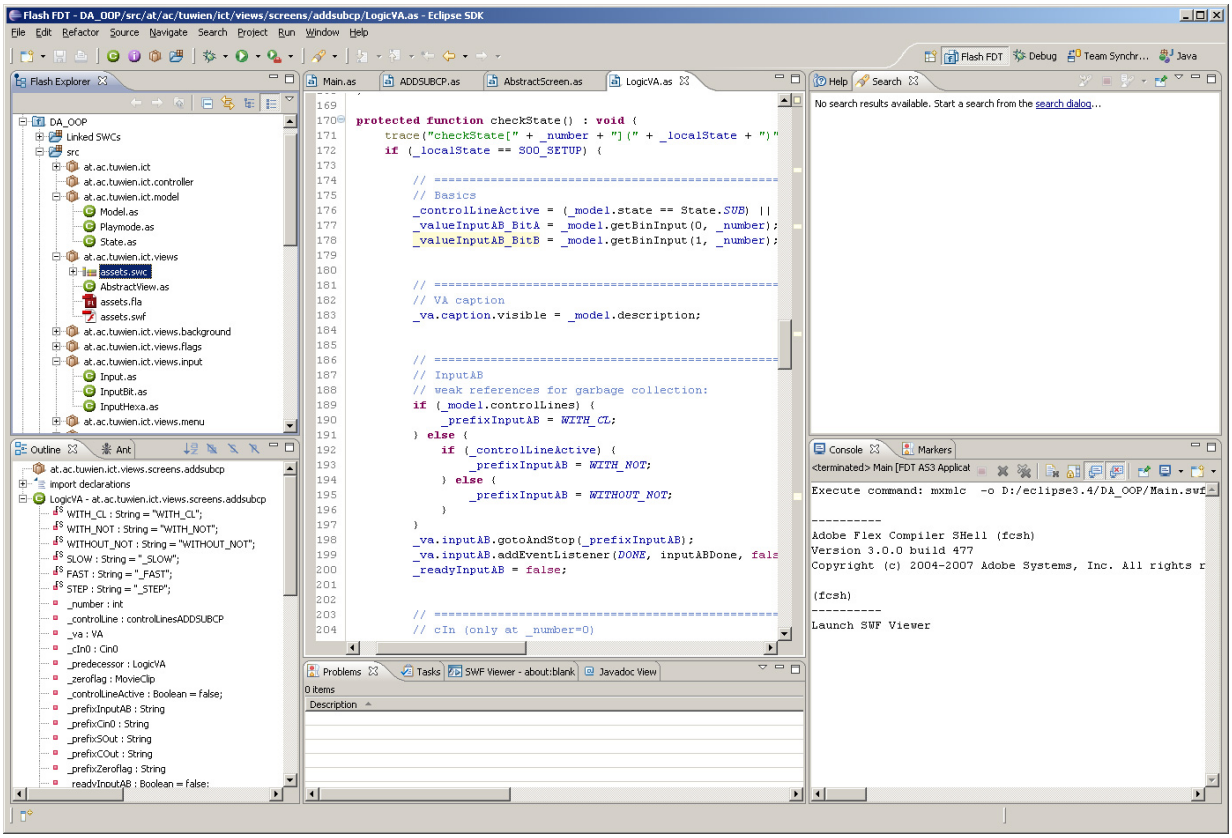


Abb. 29: Entwicklungsumgebung *Flash Developer Toolkit* von *Powerflasher*

3.4 Ähnliche Projekte

Herr Dr. Bauer (der Betreuer dieser Diplomarbeit) hatte die Idee, den bereits existierenden MC8 Assembler Compiler sowie den MC8-Simulator um einen ALU-Simulator zu ergänzen. Dieser Abschnitt motiviert die Erstellung des ALU-Simulators aufgrund einer Diskussion der Zusammenarbeit mit anderen Simulatoren und der Ergebnisse der Recherche bezüglich Projekte anderer Organisationen mit ähnlichen Aufgabenstellungen. Zunächst werden die Ergebnisse der Recherche zusammengefasst präsentiert und in weiterer Folge drei repräsentative Projekte kurz vorgestellt.

3.4.1 Umfeld

Bei der Beantwortung der Frage, wie man die Abläufe in einer ALU gut aufbereiten kann, stößt man auf folgende Problematik:

Erfolgt der **Einstieg über eine Abstraktion hohen Niveaus**, werden Gesamtzusammenhänge gut erkennbar, Details können jedoch aufgrund der Abstraktion nicht unmittelbar gezeigt werden. Man kann wesentliche Komponenten gut darstellen und in dem Modell Vorgänge visualisieren.

Eine gute Umsetzung eines solchen Simulators stellt der am Institut für Computertechnik entwickelte MC8 Simulator dar:

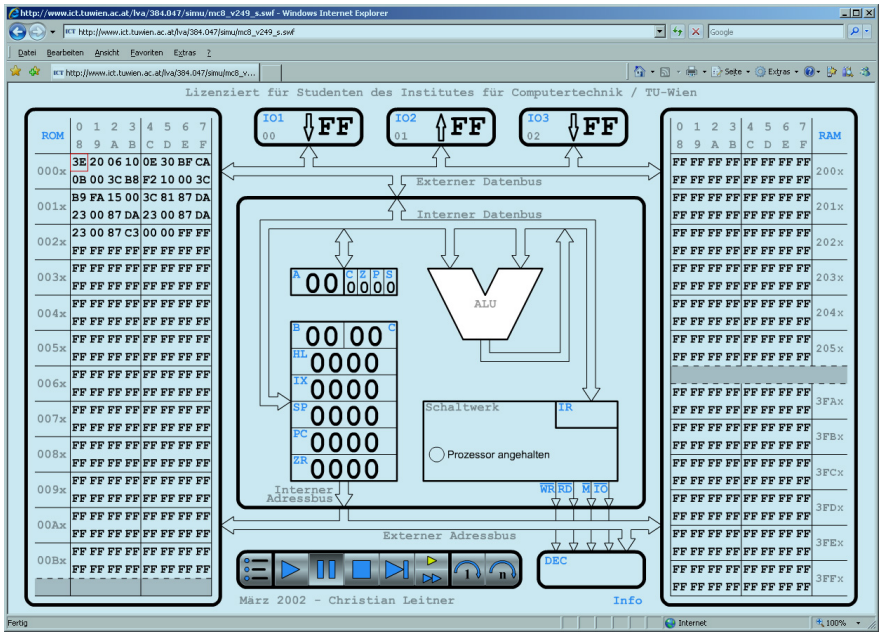


Abb. 30: MC8-Simulator Bildschirmabbild mit hervorgehobener ALU

In dieser Darstellung wurde die ALU hervorgehoben. Man kann dadurch gut erkennen, dass bei einer Darstellung des gesamten Systems prinzipbedingt wenig Platz für die Darstellung der ALU und Abläufe in der ALU bleibt.

Ein anderer Ansatz ist, einen **Simulator im gewünschten Detaillierungsgrad** zu entwickeln. Dieser hat naturgemäß den Vorteil, das Modell beliebig genau zu darstellen zu können – jedoch mit dem Nachteil, dass die Gesamtzusammenhänge nur teilweise dargestellt werden können.

Um die Nachteile beider Varianten zu kompensieren, gibt es die Möglichkeit, **beide Arten zu kombinieren**. Die Simulation läuft zunächst auf einer hohen Abstraktion und man gelangt automatisch (oder auf Wunsch) in die nächste, detailreichere Darstellung. Nachdem die Schritte in der Detaildarstellung abgearbeitet sind, wechselt man wieder auf die Darstellung mit höherem Abstraktionsniveau. Das funktioniert natürlich auch mit Detaildarstellungen verschiedener Teile.

Im folgenden Beispiel sieht man eine mögliche Zusammenarbeit von Simulatoren auf unterschiedlichen Abstraktionsniveaus. In der dritten Ebene befinden sich dem MC8 ALU-Simulator auch noch ein weiterer, derzeit fiktiver Steuerwerks-Simulator und ein Platzhalter für weitere Simulatoren, die vom MC8 SIM aus gestartet werden.

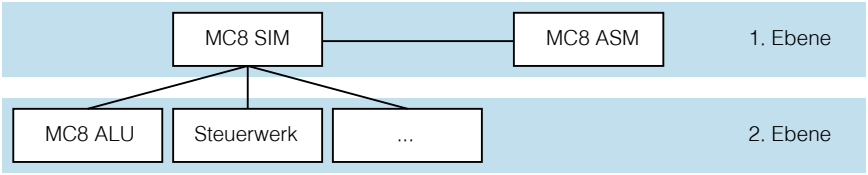


Abb. 31: Beispiel mit zwei Detailebenen

In den folgenden Abschnitten wird erörtert, welche Konzepte von anderen Institutionen realisiert wurden und vergleicht diese.

3.4.2 Erkenntnisse der Recherche

Wenn man im Umfeld der technischen (Hoch-)Schulen recherchiert, ergibt sich eine erstaunliche Anzahl an Ansätzen, eine standardisierte oder eine für Schulungszwecke eigens konstruierte CPU oder ALU (oder Teile davon) zu simulieren. Eine zusammengefasste Darstellung der Ergebnisse der Recherche-Tätigkeiten ist hier angeführt:

- Die größte Anzahl der Umsetzungen decken den Bereich der kompletten CPU ab. Oftmals liegt hierbei der *Schwerpunkt auf Abarbeitung von Maschinen- oder Assemblercode* und nicht auf der grafischen Visualisierung der Abläufe innerhalb der CPU.
- Zahlreiche *Schulen* setzen sich mit diesem Thema auseinander (also keine technischen Hoch- oder Fachhochschulen). Der Schluss liegt nahe, dass dieses komplexe Thema in Form von Schulprojekten umgesetzt wird, um das Thema besser zu visualisieren, da dieses abstrahierte Verhalten der Schaltungen im Frontalunterricht schwer verstanden wird und als E-Learning-Modell besser begriffen werden kann.
- Einige Arbeiten wurden *vor vielen Jahren fertig gestellt und nicht weiter aktualisiert*. Die älteste Lösung[44] stammt aus den Jahren 1993/1994 und ist eine Prozessorsimulation des Intel 8085-Prozessors.

[44] [SAUS1995]

- Mit Ausnahme trivialer Simulatoren sind die Umsetzungen *nicht intuitiv* benutzbar. In vielen Fällen muss man eine vorhandene Dokumentation lesen, um mit der Benutzerschnittstelle umgehen zu können. Existiert keine Dokumentation, geht der Autor von einer Unterweisung im Vortrag aus, sofern es sich um einen Simulator zu Lehrwecken handelt.
- Viele Arbeiten sind triviale Simulatoren. Oftmals sind keine oder nur sehr eingeschränkte Konfigurationsmöglichkeiten gegeben.

Der Autor stellt im Folgenden zwei Ansätze exemplarisch vor, die den Charakter hochwertiger Lösungen wiedergeben.

3.4.3 Implementierungen

3.4.3.1 VIP-Simulator

Der **VIP**[45] ist ein einfacher *Ein-Adress-Rechner*, der für die Visualisierung der Inhalte von Vorträgen konstruiert wurde. Zu diesem virtuellen Computer existiert ein Simulator, mit dem große Teile der Funktionalität mittels Java-Software simuliert werden können. Es kommt in diesem Fall eine Mehrfenster-Technik zum Einsatz, wobei im Hauptfenster in der ersten Karteikarte die Assembleranweisungen eingegeben und auch abgearbeitet werden. Nicht nur das *Instruction Register*, der *Akkumulator*, sondern auch der *Programcounter* und die Flags werden angezeigt. Auf der zweiten Registerkarte befindet sich ein Speicherabbild. Im zweiten Fenster wird die Struktur visualisiert. Änderungen von Werten werden durch rote statt blaue Schrift gekennzeichnet. Abläufe im Sinne einer Animation gibt es jedoch nicht.

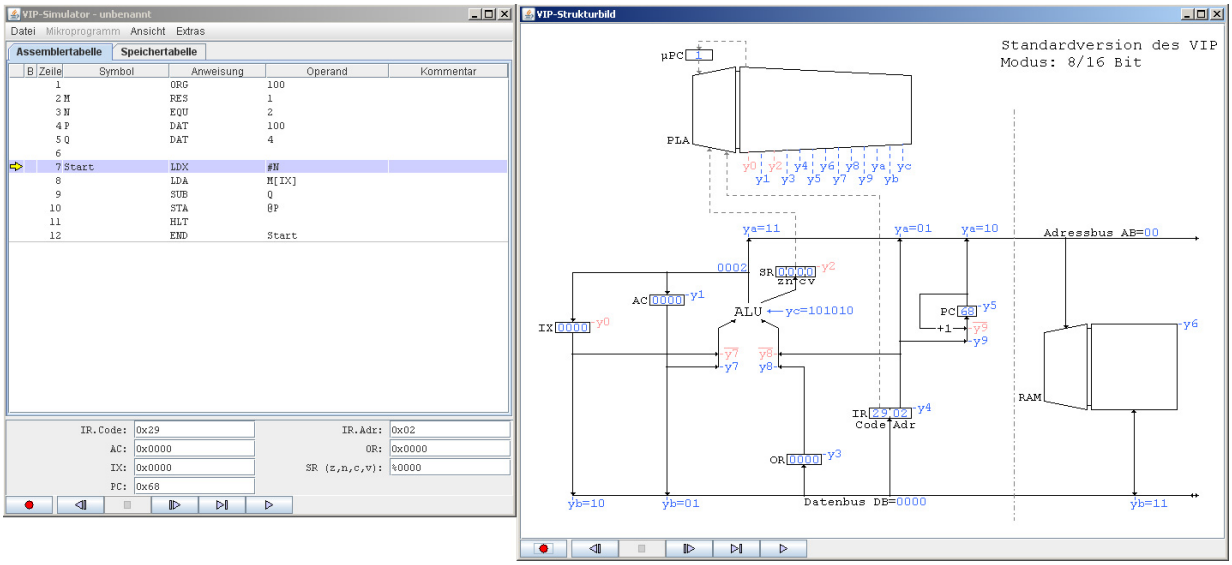


Abb. 32: VIP-CPU-Simulator der TU Berlin

[45] VIP steht für Virtueller Info 2-Prozessor in Anlehnung an die Kurzbezeichnung der Vorlesung „Informatik 2“ an der TU Berlin, Institut für technische Informatik und Mikroelektronik.

3.4.3.2 Relatively Simple CPU Simulator

Der CPU-Simulator mit dem Namen *Relativ einfacher CPU-Simulator* ist die Beilage eines Buches[46]. Er ist der einzige dem Autor der vorliegenden Arbeit bekannte kommerziell publizierte Simulator. Er ist ebenso wie der vorher beschriebene vorwiegend für den Einsatz bei Vorträgen an Hochschulen konzipiert. Er kommt dem Verhalten des *ALUSIM* am nächsten, da er Abläufe mittels einfacher Simulationen darstellt. Das Mehrfenster-Konzept zeigt die CPU, den Speicher und das Steuerwerk in jeweils einem Fenster.

Dieser Simulator hat eine umfangreiche HTML-Hilfe, mit der man sich gut zurechtfindet und rasch mit der Bedienung zurechtkommt. Negativ fällt vor allem die holprige Animation auf, deren Nachvollziehbarkeit durch das Unterlassen von flüssigen Bewegungen etwas leidet. Der Vorteil: Der Java-Sourcecode ist im Internet frei verfügbar[47].

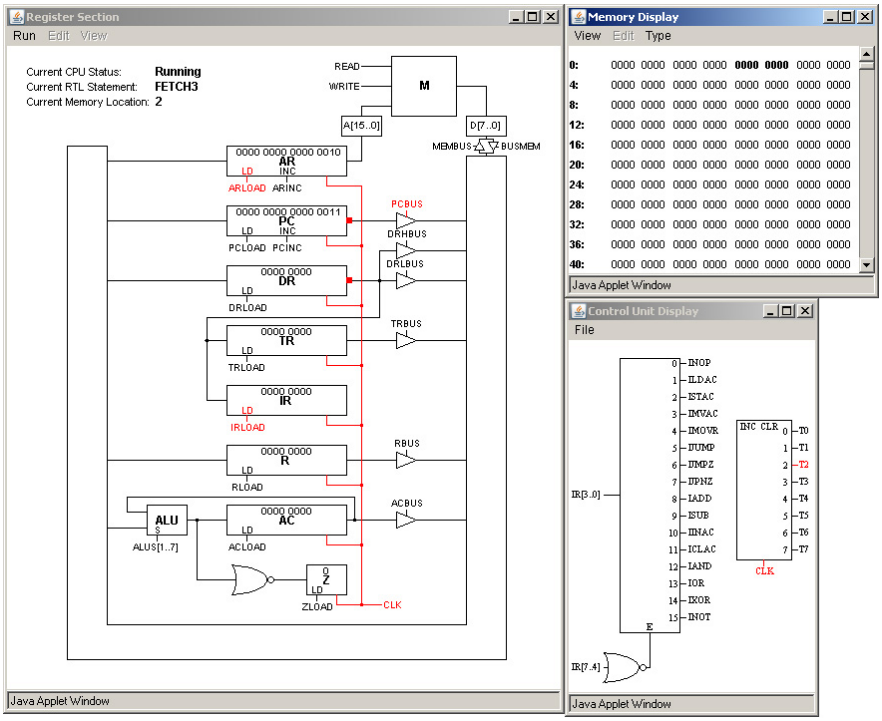


Abb. 33: CPU-Simulator als Beilage von [CARP2000]

3.4.3.3 Weitere Simulatoren

Die große Masse an CPU- oder ALU-Simulatoren sind de facto gut aufbereitete Vortragsfolien, die in der Regel nicht oder nur gering konfigurierbar sind. Der Vollständigkeit halber sei daher ein Beispiel herausgegriffen:

Der als Simulator angekündigte CPU- und ALU-Simulator der Kelso High School ist eine *Adobe Flash* Applikation. Sie simuliert vier fixe Befehle, die der Prozessor Schritt für Schritt abarbeitet und dabei den *Program Counter*, den *Akkumulator* der ALU sowie das *Instruction-Register* abbildet. Diese Applikation

[46] [CARP2000]
[47] [PEAR2000]

hat ohne Zweifel einen Nutzen für die Studierenden, da sie die Vorgänge visualisiert. Durch die fehlende Möglichkeit der Beeinflussung ist jedoch der Einsatzbereich stark eingeschränkt und ist als Simulator für die weitere Betrachtung irrelevant.

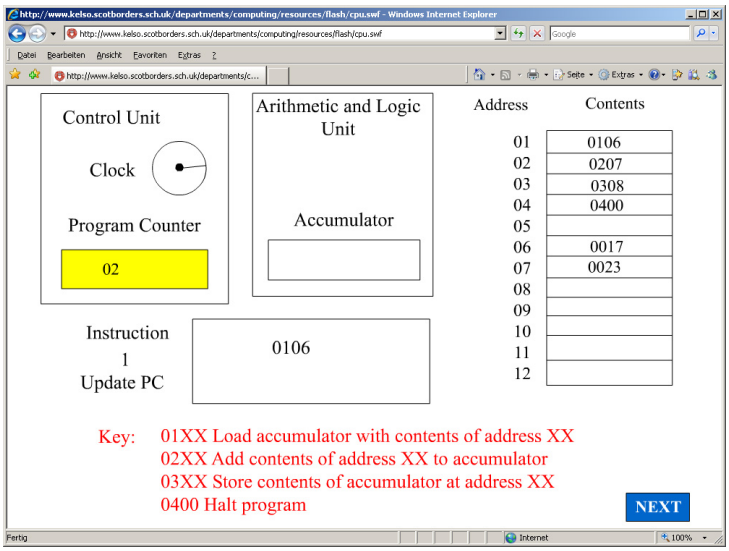


Abb. 34: CPU- und ALU-Simulator, entwickelt an der Kelso High School

4 MC8 ALUSIM Ausführung, Herangehensweise

Der **MC8 ALUSIM** ist ein Simulator der ALU des *MC8* und wurde im Rahmen dieser Arbeit als E-Learning-Modul erstellt. In den folgenden Abschnitten werden die bestehenden Simulatoren MC8ASM und MC8 Simulator vorgestellt, die Entstehung und die Umsetzung dargestellt, sowie die realisierten Lösungskonzepte diskutiert.

4.1 Analyse

Grundlage dieses Abschnitts ist die folgende Ausgangsituation: Es sollte eine Möglichkeit geschaffen werden, die Vorgänge innerhalb der ALU des MC8 zu visualisieren. Einerseits für den Einsatz im Vortrag, andererseits für das Selbststudium und zur Prüfungsvorbereitung.

4.1.1 Zielgruppe

Die Zielgruppe für den ALUSIM sind die Studierenden der Elektrotechnik, insbesondere die Zuhörenden der Vorlesung und Übung Digitale Systeme[48].

In der Vergangenheit konnten die Studierenden die Hardware nutzen, um Vorgänge nachzuvollziehen. Aufgrund der Anzahl der Teilnehmenden an der Übung war das aber nur in einem engen zeitlichen Rahmen möglich, da sich alle die Hardware teilen mussten. Abgesehen davon kann die Hardware durch unsachgemäßen Gebrauch Defekte aufweisen und muss dann aufwändig wieder Instand gesetzt werden.

Die Anforderungen der Zielgruppe sind neben der didaktischen Unterstützung auch die organisatorischen Rahmenbedingungen: Der Simulator muss auf den technisch verbreiteten PC-Plattformen lauffähig sein und jederzeit zur Verfügung stehen. Dadurch kann der Simulator zur Übungs- und Prüfungsvorbereitung genutzt werden.

Diese Voraussetzungen führen direkt zur Formulierung der Ziele und Nichtziele des Systems, welche im Folgenden behandelt werden.

[48] Institut für Computertechnik der TU Wien, LVA-Nr. 384.046 und 384.047

4.1.2 Ziele und Nichtziele

Folgende Ziele verfolgt das Projekt:

1) **Unterstützung beim Vortrag**
   Der ALU-Simulator soll in der Vorlesung als Live-Demonstrationsobjekt verwendet werden, um Vorgänge innerhalb der ALU des MC8 zu visualisieren. Diese Vorgänge konnten bisher nur verbal beschrieben und mit Hilfe von Skizzen erklärt werden.

2) **Unterstützung beim Selbststudium** der Studierenden
   Die Studierenden können den beim Vortrag verwendeten Simulator am eigenen PC starten, konfigurieren, Versuche durchführen und dabei das Verhalten der ALU genau verfolgen. Darüber hinaus kann der Simulator zur Prüfungsvorbereitung genutzt werden.

3) **Intuitive, einfache Verwendung**
   Der Simulator soll ohne Einschulung, Dokumentation oder zusätzliche Hilfe intuitiv verwendbar sein. Von der Umsetzung einer intuitiven Benutzeroberfläche hängt der Erfolg bei den Studierenden ab, da diese den Simulator nur nutzen werden, wenn es einfach ist, schnell zu Erfolgen zu kommen. Zu diesem Zweck wurde bei der Erstellung auf ein modellbasiertes Vorgehen geachtet. Zunächst wurden Zeichnungen, Grafiken und Prototypen entwickelt und erst in weiterer Folge die erforderlichen Programme geschrieben. Durch diese qualitätssichernde Maßnahme wurde die Einfachheit der Bedienung gewährleistet.

4) **Leichte Verbreitung**
   Durch den Einsatz der marktführenden *Adobe Flash*-Technologie ist eine technisch und organisatorisch einfache Verbreitung gewährleistet. Der Simulator muss lediglich in die Institutswebsite eingebunden werden. Da die Verbreitung des *Adobe Flash*-Players sehr hoch ist und der Player für alle relevanten PC-Betriebssysteme zur Verfügung steht, ist dieses Ziel erfüllt.

Die Nichtziele sind auch klar definiert:

1) **Schnittstellen** zu anderen Systemen
   Derzeit soll es keine Schnittstellen zu anderen Simulatoren oder E-Learning-Plattformen geben. Diese ist eine der möglichen Erweiterungen, die auch im Abschnitt 5.4 (Vernetzung der Module) beschrieben sind.

2) **Detailreiche technisch korrekte Darstellung**
   Durch die Abstrahierung des Modells und Vereinfachung der Abläufe mit dem Ziel, die Verständlichkeit der Darstellung zu maximieren, sind semantische Lücken gegeben. Diese sind jedoch akzeptabel, da ein theoretisches Modell stets nur einen Teil der Wirklichkeit abbildet. In diesem Fall ist die Grenze die verständliche und intuitive Aufbereitung der Abläufe in der ALU.

4.1.3 Modellcomputer-8 (MC8)

Im Rahmen einer Diplomarbeit[49] wurde eigens für den Übungsbetrieb der Modellcomputer MC8 entwickelt.

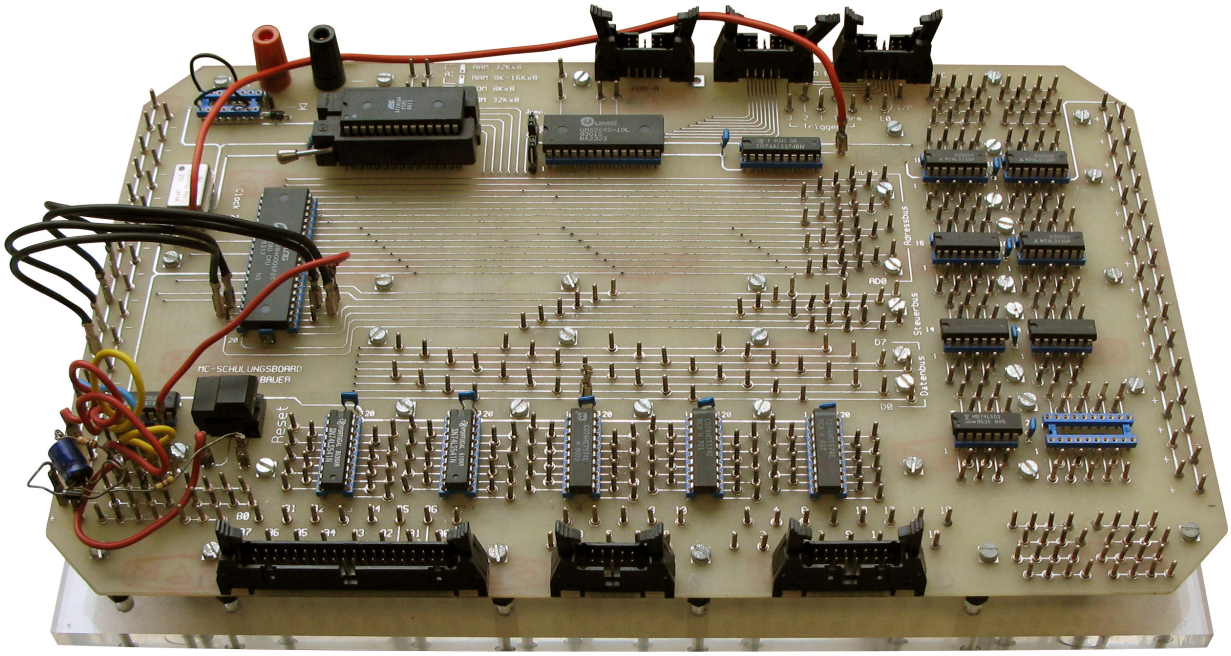


Abb. 35: Übungsaufbau des MC8

Der Übungsaufbau dieses Modellcomputers besteht im Wesentlichen aus einer Hauptplatine, auf der mit Hilfe von einer Logic Probe[50] einzelne logische Statusse (Leitungsspannungen als binäre Werte interpretiert) gemessen werden können.

Das Übungsboard besteht aus:

- Einem *Zilog Z80* Mikroprozessor
- Logikbauelementen der 74er Reihe
- Gut zugänglichen Leiterbahnen der Bussysteme
- Anschlussmöglichkeiten für einen Logikanalysator
- RAM als Arbeitsspeicher
- Sockel für (E)EPROM als Programmspeicher

[49] [BAUE2008]

[50] Ein Logic Probe ist eine Schaltung, die die Spannung an einer Leitung von einer Schaltung messen kann und entsprechend der binären Werte interpretiert. Oft wird dabei eine grüne LED zur Low-Level- und eine rote LED zur High-Level-Visualisierung verwendet.

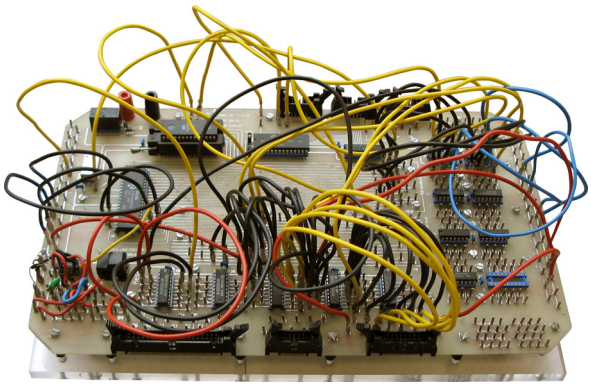


Abb. 36: Übungsaufbau des MC8 mit Verdrahtung

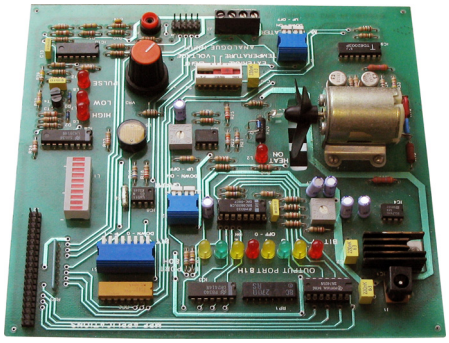


Abb. 37: Übungsboard mit LEDs, Heizwiderstand, Temperatursensor und Motor

Damit Studierende anschauliche Beispiele realisieren können, existieren am ICT auch Übungsboards mit angeschlossenen LEDs, einem Heizwiderstand, einem Temperatursensor mit Analog Digital Converter (ADC) und einem Motor (mit Ventilator).

Da jedoch jeglicher Umgang mit der Übungshardware auch Probleme mit sich bringt (Hardwaredefekte, nur begrenzte Anzahl an Übungsgeräten verfügbar, nur in den Laborstunden zugänglich), ist es sinnvoller, diesen Modellcomputer in einer E-Learning-Umgebung zu simulieren.

Dieser Modellcomputer bildet die Grundlage dieser Arbeit und somit auch des ALU-Simulators.

4.1.4 MC8 ASM

Der MC8 ASM wurde im Rahmen einer Diplomarbeit von Joseph Gernot Otto Wenninger[51] erstellt. Er bietet die Möglichkeit Assemblercodeprogramme auf dem MC8 zu simulieren. Der Ablauf stellt sich wie folgt dar: Man schreibt oder kopiert ein Assemblerprogramm in das Eingabefeld für den Sourcecode und geht daraufhin in den Wiedergabemodus. Hier wird nun der Assemblercode übersetzt. Bei fehlerhaften Anweisungen, die nicht übersetzt werden können, hält der Simulator die Übersetzung an und zeigt den Fehler in der Anweisung samt detaillierter Fehlermeldung an. Die Fehlermeldungen sind dabei so aussagekräftig, dass die Studierenden anhand der Beschreibung den Programm-Fehler selbst beheben können.

[51] [WENN2008]

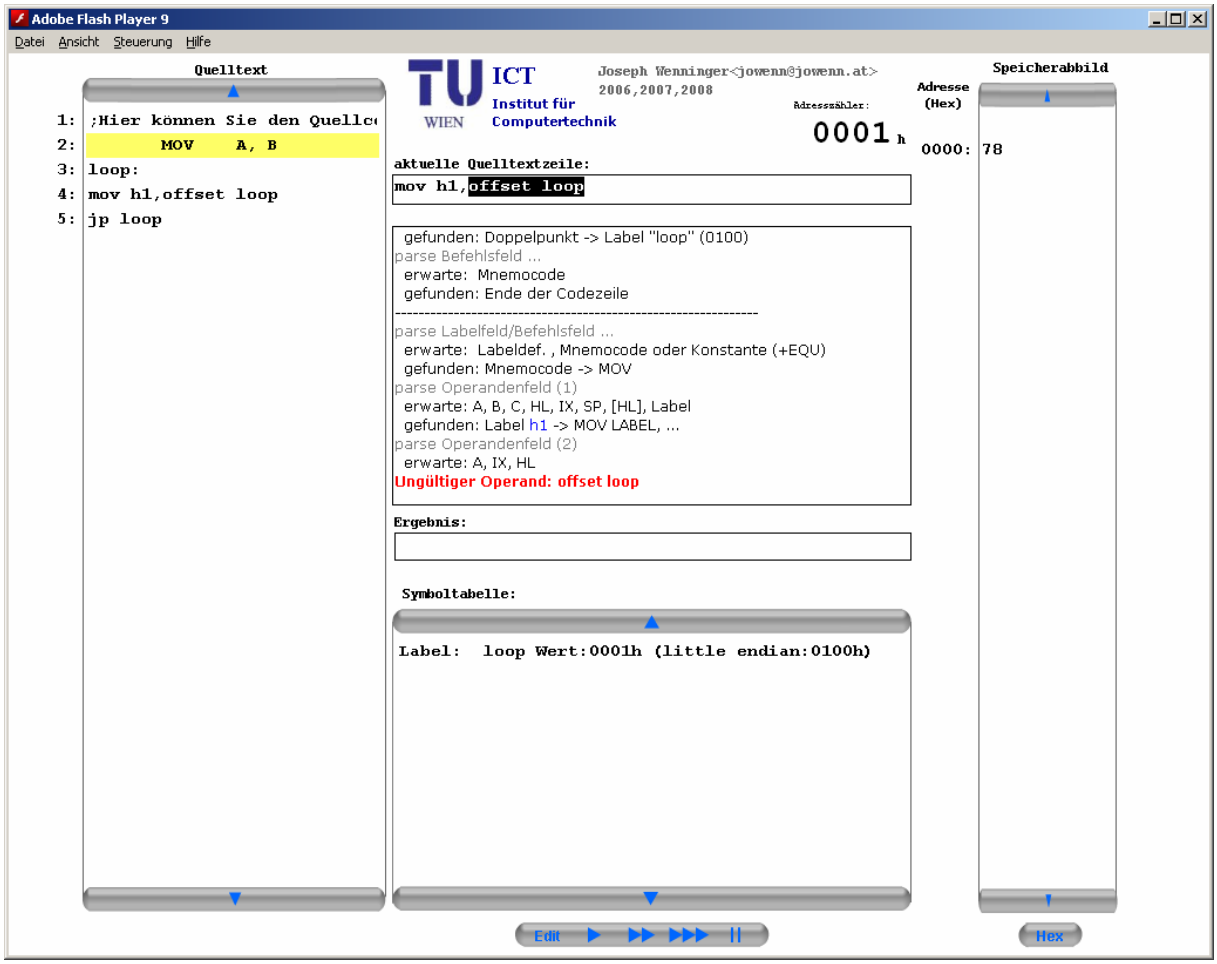


Abb. 38: MC8 ASM

Auch der MC8 ASM setzt auf die bewährten Konzepte des MC8 Simulators:

- **Intuitives Bedienkonzept**
- **Animation in verschiedenen Geschwindigkeiten** sowie Einzelschritt-Technik
- **eindeutige Trennung** der Komponenten und **übersichtliche Gliederung**

4.1.5 MC8 Simulator

Es gibt vom MC8 einen Simulator, der den grundsätzlichen Aufbau und die Funktionsweise beschreibt. Davon existieren zwei Versionen. Die erste wurde im Rahmen einer Diplomarbeit[52] erstellt. Es handelt sich dabei um eine Windowsanwendung mit fixer Größe der Benutzeroberfläche. Sie wurde im Vorlesungs- und Übungsbetrieb jedoch nie eingesetzt.

Eine zweite Implementierung wurde während eines Programmierpraktikums erstellt. Diese Version wurde in *Adobe Flash* realisiert und ist über die Instituts-Website erreichbar und wird auch in der Lehre erfolgreich eingesetzt. Sowohl Vortragende als auch Studierende nutzen den Simulator in der Praxis.

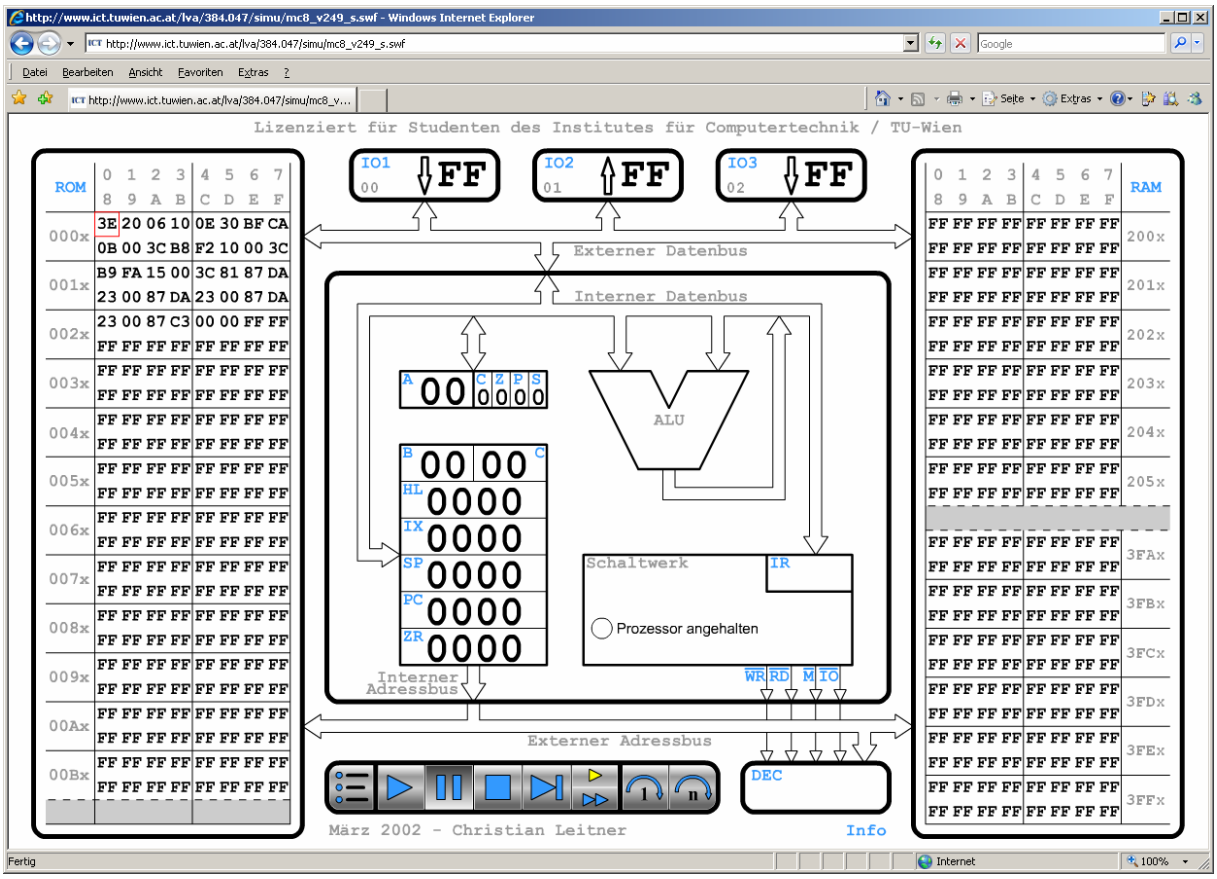


Abb. 39: MC8 Simulator

[52] [LANG1997]

Diese Version verwendet folgende Mittel, um den Lerneffekt zu maximieren:

- Sehr **intuitive Bedienung** (Anleitung oder umfangreiche Hilfe nicht notwendig)
- **Animation in verschiedenen Geschwindigkeiten** sowie Einzelschritt-Technik, um den Studierenden die Zusammenhänge ihren Lernbedürfnissen entsprechend anzuzeigen
- **Tooltipps** (kurze Erläuterungen beim Überfahren einer Stelle mit dem Mauszeiger)
- Klare Formensprache und **übersichtliche Gliederung** der Inhalte

Diese Funktionalitäten wurden auch bei der vorliegenden Arbeit berücksichtigt.

4.2 Entwurf

In diesem Abschnitt wird auf die Eigenschaften des Systems aus Sicht der Benutzerschnittstelle eingegangen. Das nächste Thema sind die Steuerungsmöglichkeiten des Simulators und die Ausformung der Benutzerschnittstelle. Im Anschluss folgen die Beschreibungen des Animationsmodells. Diese beinhalten die Darstellung des Modells und dessen Umsetzung in Form der Animation.

4.2.1 Gestaltung der Benutzerschnittstelle

Um eine Kontinuität in der Qualität der Benutzerschnittstelle zu erhalten, werden die Grundsätze der bisher vorhandenen Simulatoren übernommen:

- sehr **intuitive Bedienung** (Anleitung oder umfangreiche Hilfe nicht notwendig)
- **Animation in zwei verschiedenen Geschwindigkeiten** sowie Einzelschritt-Technik, damit sich die Studierenden die Zusammenhänge in ihrer bevorzugten Lerngeschwindigkeit zeigen lassen können.
- **Tooltipps** (kurze Erläuterungen beim Überfahren einer Stelle mit dem Mauszeiger)
- **Klare Formensprache** und übersichtliche Gliederung der Inhalte
- der **Abstraktionsgrad kann** von den Studierenden selbst **gewählt werden**
- Übernahme von technischen Symbolen und Normen so weitreichend wie möglich, um auch eine **enge Bindung zum existierenden Skriptum** der Vorlesung und Übung Digitale Systeme zu erreichen.

Die einheitliche Gestaltung ist wesentlich für eine intuitive Bedienung und eine klare Formensprache. Daher wurden folgende Farben für die Gestaltung der Oberfläche festgelegt:

Abb. 40: Farbvarianten für die Gestaltung der Benutzerschnittstelle

Gelb hat dabei einen Sonderstatus, es kennzeichnet alles Aktive (Schaltflächen, aktive Leitungen, Animierte Werte auf einer Leitung). Die restlichen Farben wurden so gewählt, dass sie einen harmonischen Eindruck erwecken. Auch Personen mit Einschränkung im Farbsehen haben mit dieser Farbwahl keine Schwierigkeiten.

Die Benutzerschnittstelle wurde hinsichtlich der Gliederung von eigenständigen Bereichen ähnlich dem *MC8 ASM* und dem MC8 Simulator umgesetzt. Im Gegensatz zu diesen beiden Applikationen werden jedoch Rahmen durch vollflächige Felder ersetzt, da sich sonst sehr viele Linien überschneiden würden und dadurch die Übersichtlichkeit negativ beeinflusst würde.

Damit auch bei Projektion im Hörsaal die Studierenden die Leitungen gut erkennen können, wurde die Leitungsdicke auf zwei Pixel erhöht. Steuerleitungen haben ebenso eine Dicke von zwei Pixel, jedoch eine Umrandung von einem Pixel, lediglich bei der komplexen Darstellung der Shift- und Rotate-Befehle werden die Steuerleitungen aufgrund der Übersichtlichkeit nur ein Pixel dick dargestellt.

4.2.1.1 Technische Umsetzung der Grafiken der Benutzerschnittstelle

Sämtliche Grafiken wurden in der grafischen Benutzeroberfläche von *Adobe Flash* erstellt. Diese bietet neben zahlreichen reinen Zeichenfunktionen die Möglichkeit, zusammengehörige grafische Elemente zu gruppieren und in so genannten *Bibliotheken* abzulegen. Eine Gruppierung kann nicht nur aus Grafiken, sondern auch aus Programm-Code bestehen. So war es in *Adobe Flash* früher notwendig, Funktionalität und grafische Umsetzung eng miteinander zu verknüpfen[53]. Dies ist seit der Einführung der Programmiersprache *Actionscript 2* nicht mehr zwingend notwendig und wurde mit *Actionscript 3* gänzlich überflüssig (blieb aber möglich). So kann man nun Funktionalität und Darstellung trennen. Diese Teilung ist erstrebenswert, damit sowohl die grafische Darstellung als auch die Programmlogik unabhängig voneinander umgesetzt werden kann.

Die Grafiken wurden, wie im vorigen Absatz bereits erwähnt, gruppiert und als so genannte *Symbole* in *Bibliotheken* angelegt. Diese Symbole werden ähnlich einem Dateisystem in Ordnern abgelegt, die zusammengehörende Symbole zusammenfassen. Beim Anlegen der Symbole kann man *Adobe Flash* anweisen, automatisch Klassen für ausgewählte Symbole anzulegen. Diese automatisch generierten Klassen haben keinerlei Programmlogik, ermöglichen aber eine einfache Verwendbarkeit in *Actionscript 3.*

[53] Flash ist ursprünglich von Macromedia entwickelt worden, welche später von Adobe übernommen wurde. So gesehen müsste man in diesem konkreten Fall korrekterweise Macromedia Flash schreiben.

Die Ordnerstruktur dient nicht nur einer optischen Strukturierung in der Übersicht, sondern steht in dieser Struktur auch in Actionscript zur Verfügung:

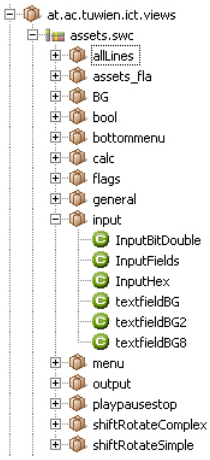


Abb. 41: Ansicht in der Bibliothek in der FDT-Programmierumgebung

Man kann diese Bibliothek in der Entwicklungsumgebung importieren. Nun ist ein direktes Zugreifen auf Grafiksymbole möglich, wobei die Ordnerstruktur der Bibliothek in der Entwicklungsumgebung erhalten bleibt. Beispielsweise kann man nun ein Symbol in Actionscript 3 einfach anzeigen und um 90° im Uhrzeigersinn drehen:

```
var _inputHex : InputHex = new InputHex(); // Das Symbol als Variable anlegen
addChild(_inputHex); // Die Variable mit dem Symbol am Bildschirm sichtbar machen
_inputHex.rotation = -90; // 90° gegen den Uhrzeigersinn drehen.
```

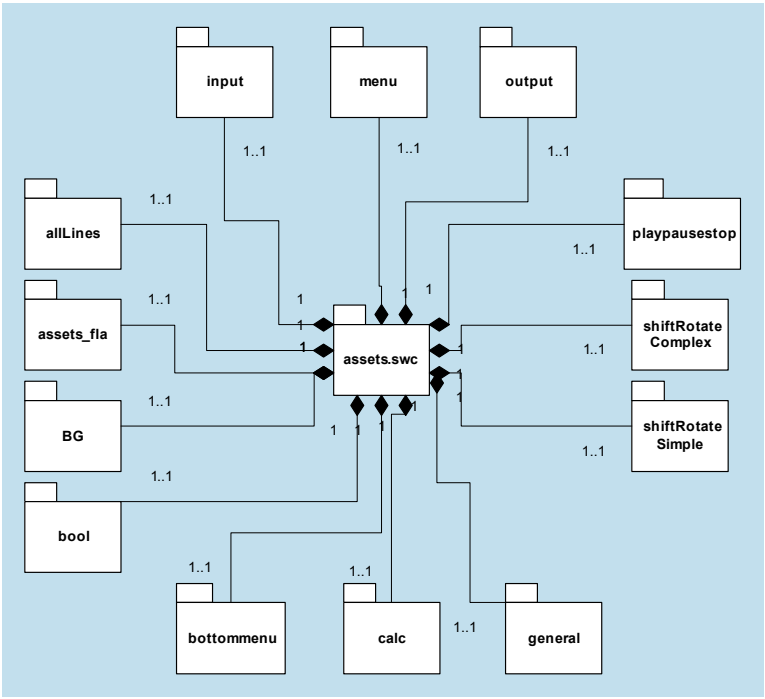


Abb. 42: Übersicht Grafiksymbole

Man kann in *Adobe Flash* mit Hilfe von Grafiksymbolen Animationen erstellen, wie zum Beispiel die animierten Bits auf einer Leitung. Um jedoch das Verhalten steuern zu können – wie zum Beispiel:

- das *Anhalten* oder *erneute Starten* einer Animation,
- die *Detailkonfiguration* einer Grafik (Details müssen ausgeblendet werden) oder
- das *Senden eines Events*[54] an andere Symbole, damit diese entsprechend reagieren können

– muss man die Symbole mit Programmcode verknüpfen.

Jedem der erstellten Symbole kann man nun ein Verhalten zuordnen. Dies erreicht man, indem man dem Symbol eine Klasse zuweist.

4.2.2 Grundlegende Funktionalitäten zur Steuerung des Simulators

Neben den in Abschnitt 3.2.8.2 bereits beschriebenen Rechen-, Vergleichs-, Logik- und Schiebefunktionen hat der Simulator zusätzliche Funktionalitäten, um das Verhalten des Programms zu steuern. Um den unterschiedlichen Fortschritten beim Selbststudium der Studierenden gerecht zu werden, gibt es drei Geschwindigkeiten in denen die Simulation ablaufen kann: *langsam*, *schnell* und *in einem Schritt*. Im Folgenden wird auf diese Punkte eingegangen.

4.2.2.1 Simulationsgeschwindigkeiten

Bei der **langsamen Animationsgeschwindigkeit** werden die Werte, die an Leitungen anliegen (welche die Eingangswerte, Gatter, Flags und Ausgangswerte verbinden) mit Hilfe von kleinen Kreisen, die den konkreten Wert beinhalten, symbolisiert. Der Wert läuft entlang der Leitung, der er anliegt, von der Quelle zum Ziel. Die Animation der Werte wird mit einer Geschwindigkeit von 3,6 Pixel pro 40 ms[55] ausgeführt. Dadurch erhalten die Studierenden die Möglichkeit, die Abläufe sehr detailliert zu studieren. Die Animation kann zur besseren Veranschaulichung an beliebigen Stellen angehalten und wieder fortgesetzt werden.

[54] Ein Event ist eine Methode des Objektorientierten Programmierens, zwischen Programmteilen mittels Nachrichten zu kommunizieren, ohne dass die einzelnen Programmteile viel über das kommunikationstechnische Gegenüber wissen zu müssen. Eine tiefer gehende Ausführung würde diesen Rahmen sprengen.

[55] Dieser Wert ist empirisch ermittelt und beträgt 90 Pixel richtungsgebundene Bewegung pro Sekunde. Da der Mensch 25 Bilder pro Sekunde benötigt, um den Eindruck einer flüssigen Bewegung zu erhalten, wird auch die Animation 25 Mal pro Sekunde aktualisiert. Daraus ergibt sich der Wert 90 / 25 = 3,6 Pixel / 40 ms.

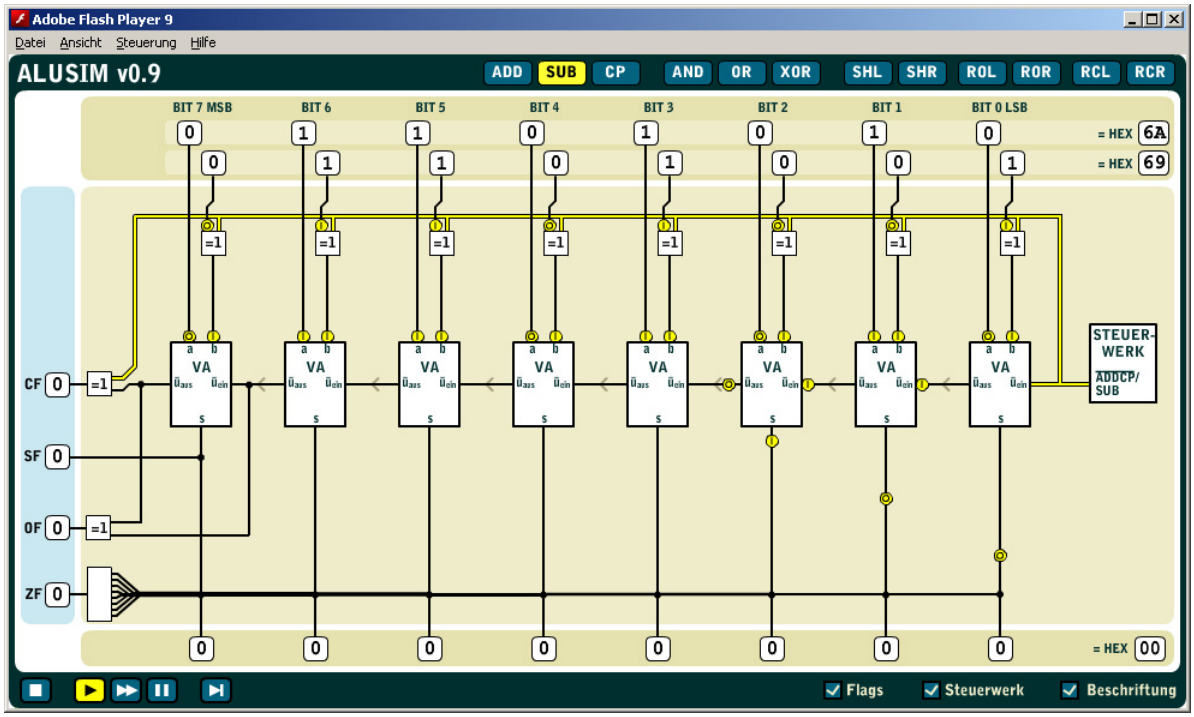


Abb. 43: Bildschirmabbild bei der Simulation einer Subtraktion bei langsamer Animation

Genau genommen müssten, diesem Modell folgend, die Werte des Steuerwerks ebenso auf den Leitungen vom Steuerwerk zu den Gattern animiert werden. Aus Gründen der Vereinfachung des Modells (hinsichtlich des zu vermittelnden Verständnisses für die Abläufe) wäre dies aber zu weitführend. Daher wurde darauf verzichtet und diese Animation durch zwei Zustände ersetzt: Wenn die Steuerleitung aktiv ist, dann wird sie gelb dargestellt. Ist sie nicht aktiv, wird sie grau dargestellt. Es laufen also keine Werte entlang der Steuerleitung, sondern die ganze Leitung ist eindeutig als aktiv oder nicht aktiv gekennzeichnet.

Dieses Prinzip kommt auch bei der **schnellen Animation** zum Tragen: Die Leitungen, an deren Quelle ein neuer Wert entstanden ist (etwa angeregt durch einen Taktimpuls oder durch den Start der Animation), werden durch – im Verhältnis zur Leitungsdicke – massive gelbe Pfeile visualisiert. Diese Pfeile werden nach dem Entstehen eingeblendet und bleiben bis zum Ende der Animation oder bis zum nächsten Bilden eines Wertes auf der gleichen Leitung sichtbar, wobei keine weitere Bewegung oder andere Animation der Pfeile erfolgt. Die zeitlichen Abläufe werden jedoch eingehalten: Leitungslaufzeiten werden entsprechend der Leitungslänge und heuristischen Maßstäben realisiert[56].

[56] Als Ergebnis zahlreicher Versuche wurden folgende Werte als Richtwerte festgelegt: Bei langen Leitungslängen (> rund 100 Pixel) wird der Pfeil eine Sekunde angezeigt, bevor die Folgeoperation simuliert wird. Bei sehr kurzen und kurzen Leitungen ist die Anzeigedauer ¼ bzw. ½ Sekunde.

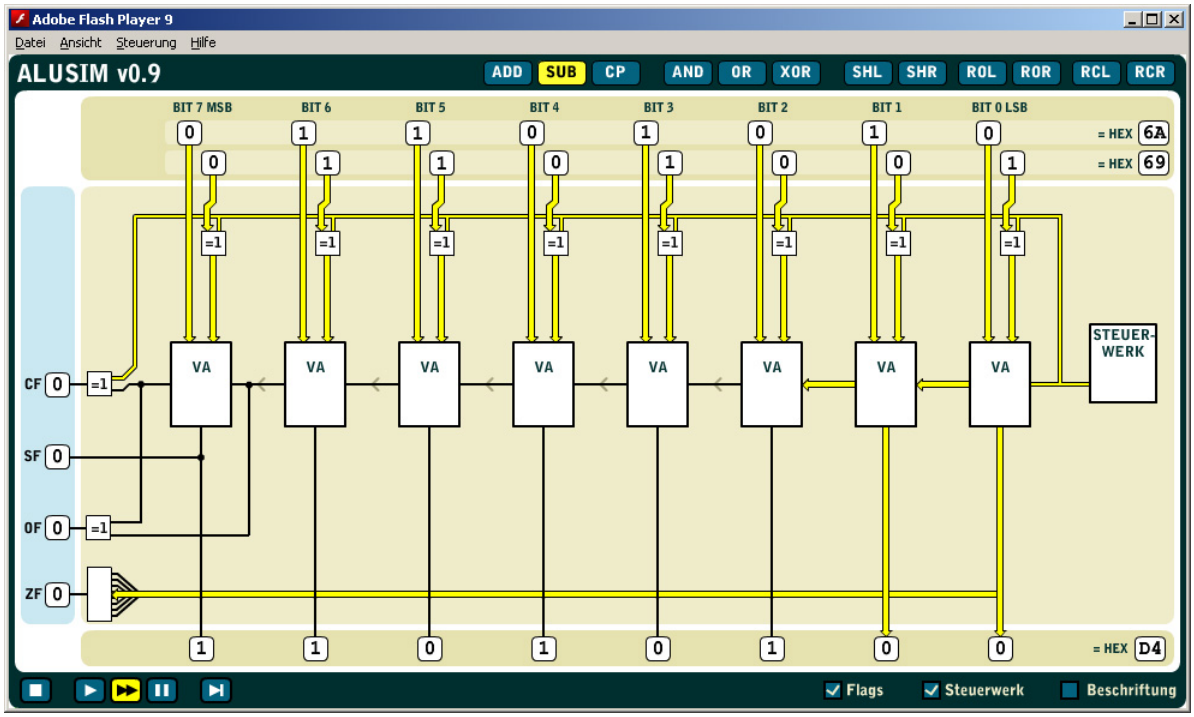


Abb. 44: Bildschirmabbild bei der Simulation einer Subtraktion bei schneller Animation

Das Starten, Stoppen und Pausieren kann gemeinsam mit der Animationsgeschwindigkeit mittels fünf Schaltflächen, am Bildschirm links unten, ausgewählt werden.

Abb. 45: Detail der Benutzerschnittstelle: Steuerung der Animation

Steuerungsmöglichkeit für: **Stopp**, **Abspielen** der Animation in **langsamer** und in **schneller** Geschwindigkeit, **Pause** sowie **Einzelschritt** (die Animation zeigt sofort den endgültigen Zustand).

4.2.2.2 Darstellungsvereinfachungen

Um den Lernenden die Möglichkeit zu geben, den Simulator ihren Lernbedürfnissen anzupassen, ist eine Konfiguration der angezeigten Detaillierungstiefe möglich. Dafür befinden sich in der rechten unteren Ecke der Benutzerschnittstelle drei Kontrollkästchen, die jeweils eine Eigenschaft in der Anzeige ein- bzw. ausschalten:

Abb. 46: Detail der Benutzerschnittstelle: Steuerung der Komplexität der Darstellung

Flags

Im linken Teil des Simulators befindet sich ein hellblaues Feld, welches die Flags zeigt.

Diese sind in vielen Fällen für das Verständnis nicht unmittelbar relevant und können daher ausgeblendet werden. Mit dem Ein- und Ausschalten werden auch alle Zuleitungen ein- bzw. ausgeblendet.

Eine Sonderfunktion erfüllt das *Overflow*- und *Parity Flag*, da es je nach Operation für den jeweiligen Zweck genutzt wird. Bei den arithmetischen Operationen *ADD* und *SUB*, sowie bei Vergleichsoperation *CP* wird das *Overflow Flag* gesetzt und als solches interretiert. In allen anderen Fällen wird das gleiche Flag als *Parity Flag* genutzt und durch ein achtstelliges XOR-Gatter über alle Ausgangsleitungen gebildet.

In diesem Zusammenhang wird noch einmal erwähnt, dass die Flags (wie auch die Eingabe- und Ausgabewerte) nicht Teil der simulierten ALU sind.

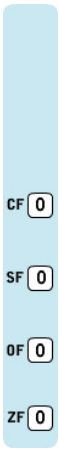


Abb. 47: Darstellung der Flags

Steuerwerk

Es gibt je Operationsgruppe ein individuelles Steuerwerk, welches samt den, zur Schaltung führenden, Leitungen ein- oder ausgeblendet werden kann.

Schaltwerk für ADD, SUB und CP

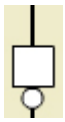
Schaltwerk für AND, OR und XOR

Schaltwerk für SHL, SHR, ROL, ROR, RCL und RCR

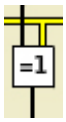
Abb. 48: Vergleich der Steuerwerke

In allen Fällen bedingt das Einblenden von Steuerwerk und Steuerleitungen eine *Umstellung* des Detaillierungsgrades der Visualisierung. Folgende Abbildungen verdeutlichen dies anhand eines

Beispiels der Negation am Eingang bei der Subtraktion. Die Negation ist bei der Addition nicht sichtbar.



Negation bei Subtraktion



„schaltbare“ Negation durch XOR-Gatter realisiert

Abb. 49: Negation bei der Subtraktion im Vergleich zur allgemeinen Darstellung bei eingeschaltetem Steuerwerk

Die Subtraktion und der Compare-Befehl wird mit Hilfe der Ersatzaddition durchgeführt. Diese benötigt unter anderem für jede Stelle des zweiten Eingabewertes eine Negation. Diese wird von der Addition jedoch nicht benötigt. Daher benötigt man eine Negation, die man deaktivieren kann. Das löst man, indem man die Negation mittels XOR-Gatter realisiert. Der erste Eingang des Gatters wird auf den Wert 1 gesetzt, wodurch der andere Eingang auch beim Ausgang anliegt – jedoch negiert. Wird an den ersten Eingang der Wert 0 gelegt, wird der Wert des zweiten Eingangs unverändert am XOR-Gatter ausgegeben. Beim XOR-Gatter steuert also der erste Eingang – abhängig von dessen Wert 0 oder 1 – ob das XOR-Gatter den zweiten Eingangswert negiert oder nicht.

Bei allen Operationen muss die Darstellung der Schaltung umgestellt werden, damit die Steuerleitungen die Schaltung konfigurieren können. In jedem Fall wird die Schaltung detaillierter.

Beschriftung

Die Beschriftung der Gatter und des Steuerwerks – wenn dieses angezeigt wird – kann ein- und ausgeblendet werden.

Ein- und Ausgänge

Die Ein- und Ausgänge an der oberen bzw. unteren Seite des Simulators haben eine grundsätzliche Funktionalität: Sie zeigen die binär vorliegenden Werte auch in hexadezimaler Darstellung an.

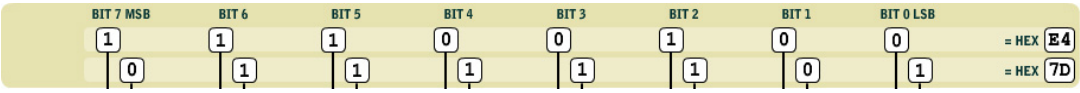


Abb. 50: Detail der Benutzerschnittstelle: Eingabebereich der Eingangswerte

Da für die Simulation die Eingabewerte veränderbar sein müssen, sind diese durch die Benutzerin oder den Benutzer setzbar. Dabei kann man Darstellung der Bits durch Anklicken auf den jeweils anderen Wert setzen (aus *0* wird *1* und umgekehrt).

Der angezeigte hexadezimale Wert ändert sich dabei sofort mit. Auf der anderen Seite kann man auch die hexadezimalen Werte angeben und das Bitmuster für den jeweiligen Ausgang wird automatisch berechnet. Eine Änderung eines Wertes hat zur Folge, dass eine gerade ablaufende Animation unterbrochen wird.

In der Beschriftung der Eingangsbits sieht man beim ersten zusätzlich die Bezeichnung **MSB**, was ein Akronym der englischen Bezeichnung des *höchstwertigen Bits* (*most significant bit*) ist. Analog gilt dies für die Bezeichnung **LSB** beim *niederwertigsten Bit* (*least significant bit*).

4.2.3 Allgemeine und detaillierte Beschreibung der Abläufe

Nach den im vorigen Abschnitt beschriebenen allgemeinen Steuerungs- und Beeinflussungsmöglichkeiten des Simulators geht es in diesem Abschnitt um die konkreten Abläufe und deren Darstellung. Für jede einzelne Operation werden Darstellung und Vorgang erörtert.

4.2.3.1 ADD, SUB und CP

Obwohl die **Addition** (**ADD**) und die **Subtraktion** (**SUB**) arithmetische Operationen sind, befinden sie sich in der gleichen Gruppe wie **Compare** (**CP**). Das rührt daher, dass sich die *Subtraktion* und *Compare* nicht unterscheiden, bis auf die Tatsache, dass bei *Compare* im Gegensatz zur *Subtraktion* die Ausgänge nicht aus der ALU herausgeführt werden.

Addition (ADD)

Bei der Addition werden acht Volladdierer in Serie geschaltet. Je Volladdierer werden die zugehörigen beiden Eingangsbits (sie werden in weiterer Folge an die Eingangsleitungen als Wert angelegt) über Leitungen zugeführt. Die Ergebnisse der acht Einzelberechnungen liegen an den Ausgängen an. Die Grundkonfiguration der Addition sieht in ihrer einfachsten Form wie folgt aus:

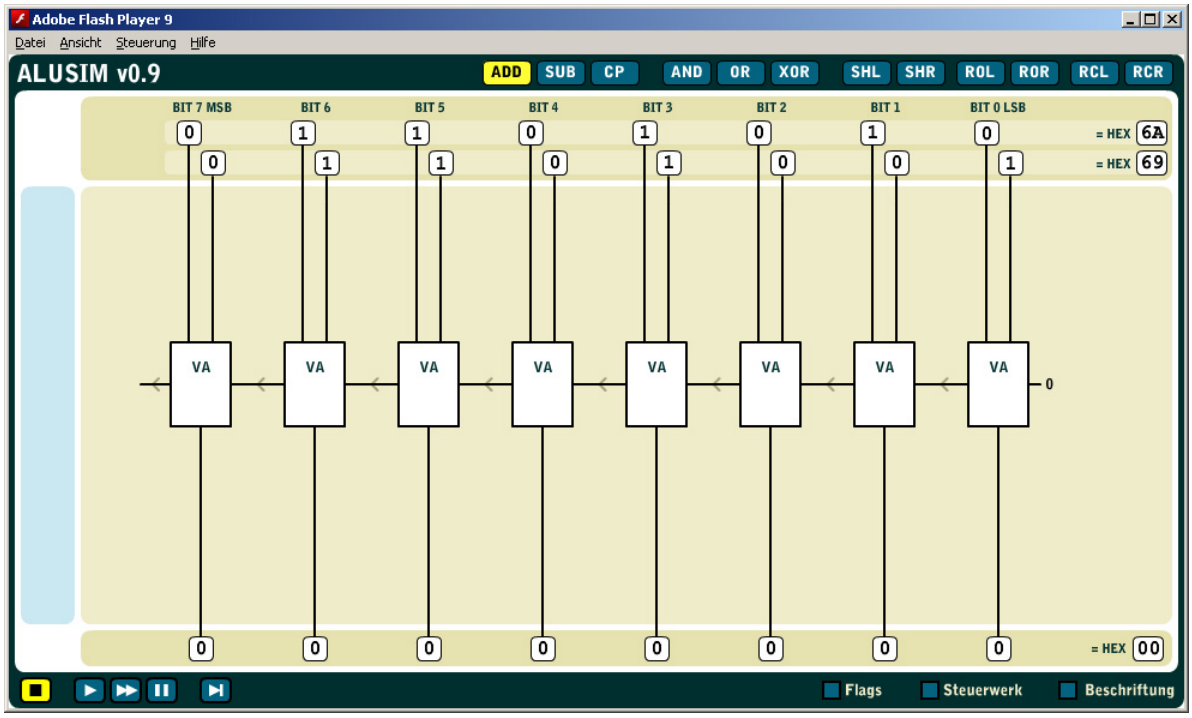


Abb. 51: Bildschirmabbild der einfachen Addition

Beim Starten der Animation werden zunächst die Eingangswerte an die Leitungen zu den Volladdierern gelegt. Darüber hinaus wird am ersten Übertragseingang der Wert 0 erzeugt. Der logisch erste Volladdierer kann, sobald beide Eingangswerte anliegen und der angelegte Wert 0 angekommen ist, die Berechnung durchführen und den Übertrag dem nächsten Volladdierer zuführen und das erste berechnete Ergebnis an die Ausgangsleitung führen.

Jeder folgende Volladdierer erhält nun den Übertragsausgang des Vorgängers als Übertragseingang und kann seine Berechnung durchführen. Der letzte Volladdierer führt den Übertrag zwar heraus, das Ergebnis bleibt aber als eines der Endergebnisse stehen.

Subtraktion (SUB)

Im Falle der Subtraktion, die der Addition sehr ähnlich ist, nutzt man das arithmetische Prinzip, dass die Ausdrücke a-b und a+(-b) das gleiche Ergebnis haben. Um an Stelle der Subtraktion eine Addition durchführen zu können, muss der Subtrahend in Zweierkomplementdarstellung umgewandelt werden. Man erreicht dies, indem man die zugehörigen Eingangsleitungen negiert und den ersten Übertrag auf den Wert 1 setzt. Mit dieser kleinen Modifikation wird aus der Addition die Subtraktion. Man nennt dies die *Ersatzaddition*, welche bereits in Abschnitt 3.2.2.2 erörtert wurde.

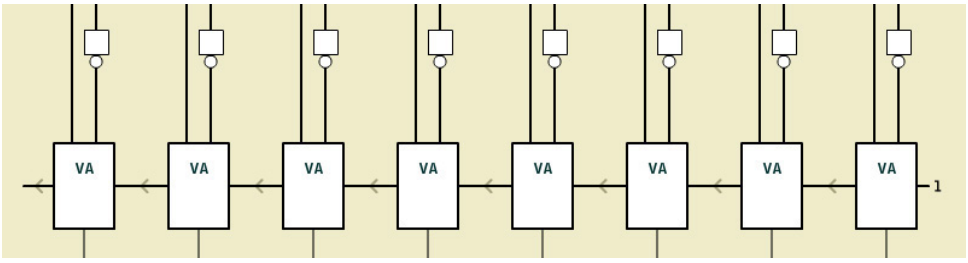


Abb. 52: Bildschirmausschnitt der Subtraktion

Der Ablauf der Animation ist analog der Addition mit den Unterschieden, dass der zweite Eingangswert jedes Volladdierers durch ein NOT-Gatter negiert wird und der erste Überlauf-Eingang den Wert *1* statt *0* erhält.

Compare (CP)

Die Funktion des Vergleichs basiert auf den Überlegungen, dass man zwei zu vergleichende Zahlen voneinander abziehen muss, und dass das Ergebnis etwas über das Verhältnis der Zahlen zueinander aussagt. Es ist jedoch nicht das Ergebnis an sich notwendig, sondern nur die, anhand der Ausgänge, gesetzten Flags. Deshalb wird das Ergebnis auch nicht herausgeführt.

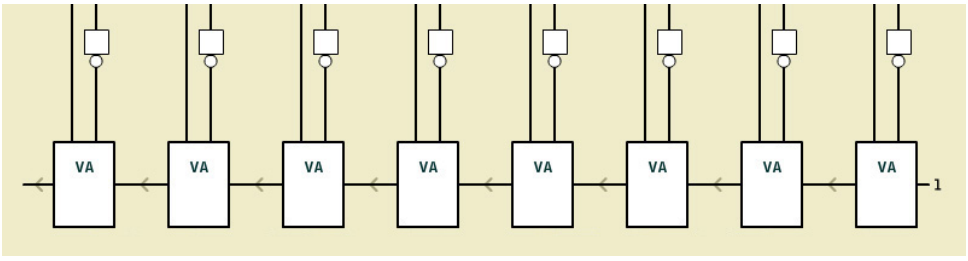


Abb. 53: Bildschirmausschnitt der Vergleichsfunktion

Die Animation ist gleich wie bei der Subtraktion mit dem Unterschied, dass das Ergebnis der VA nicht zu den Ergebnisbits geführt wird.

Variante mit Steuerwerk

Wird das Steuerwerk eingeblendet, ändert sich die Schaltung dahingehend, dass mit Hilfe des Steuerwerks und dessen Steuerleitungen mit ein- und derselben Schaltung die Operationen ADD, SUB und CP abgearbeitet werden können.

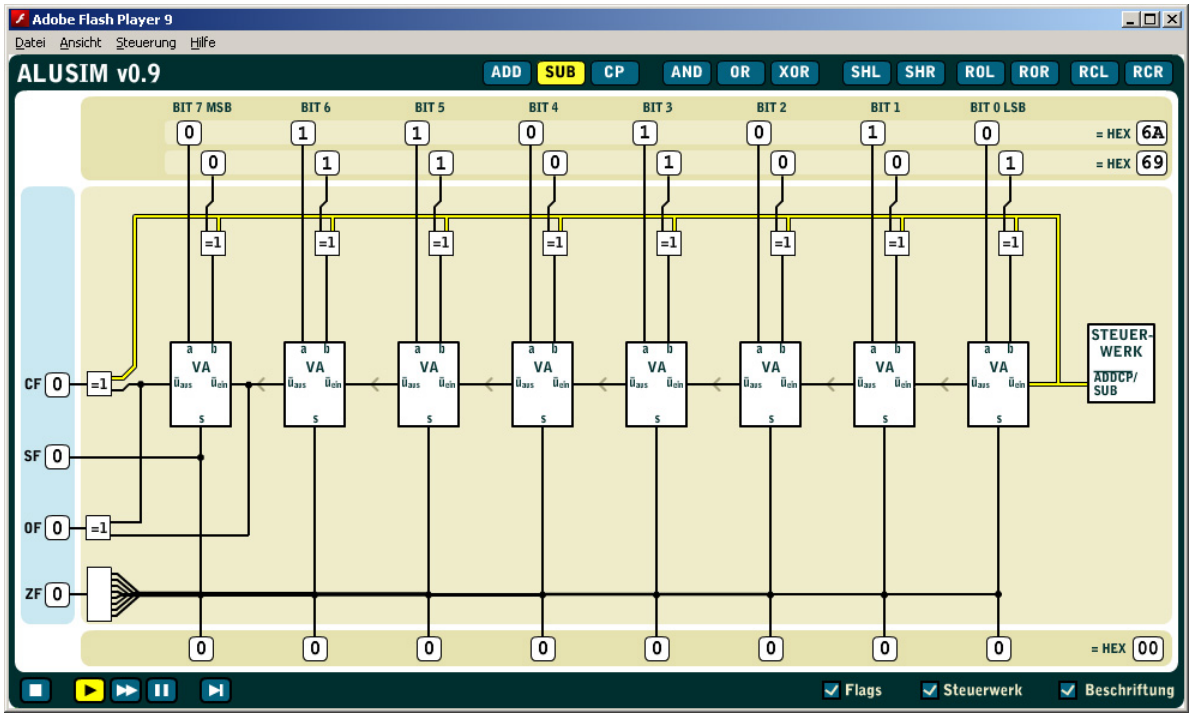


Abb. 54: Bildschirmabbild der Subtraktion mit Steuerwerk

4.2.3.2 AND, OR und XOR

In dieser Gruppe stehen die Operationen, die *ausschließlich* von den Eingangswerte abhängen und nicht von Ergebnissen anderer Gatter.

Konjunktion (Und, AND)

Diese und die nachfolgenden Operationen führen aus aussagenlogischer Sicht nur jeweils eine Grundoperation aus. In diesem Fall werden die Eingangswerte mit einem AND-Gatter verknüpft und an den Ausgang angelegt.

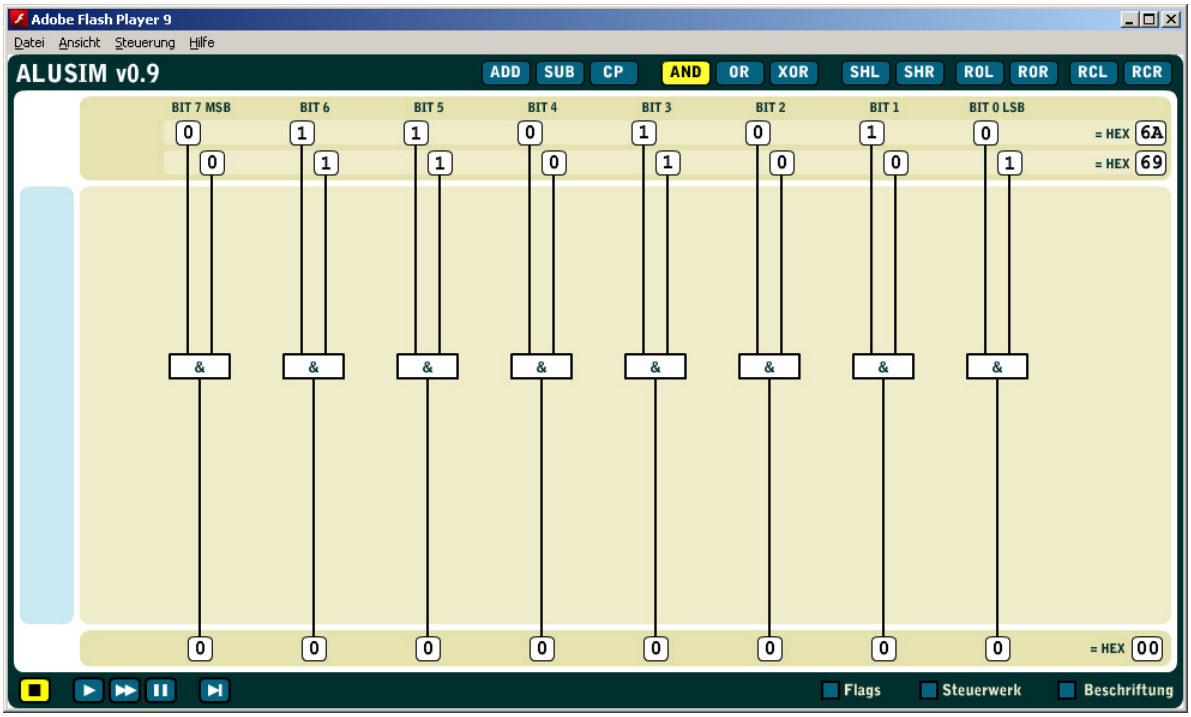


Abb. 55: Bildschirmabbild der Konjunktion

Die Animation erfolgt beginnend bei den beiden Eingangswerten und führt die beiden Werte diese zum Gatter. Nach der Ermittlung des Ergebniswertes wird dieser an die Leitung zu den Ausgängen gelegt.

Disjunktion (Oder, OR)

Die Oder-Verknüpfung ist nahezu ident mit der Konjunktion. Der einzige Unterschied besteht darin, dass in der Mitte nun ein OR-Gatter das Ergebnis ermittelt.

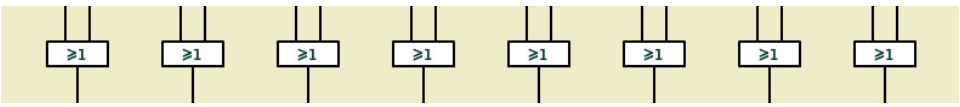


Abb. 56: Bildschirmausschnitt der Disjunktion

Die Animation ist ident mit der Konjunktion.

Antivalenz (Exklusives Oder, XOR)

Das Exklusive Oder hat fachlich insofern eine gewisse Sonderstellung, als diese Operation ebenso durch eine Kombination von AND- und OR-Gattern erreicht werden kann.

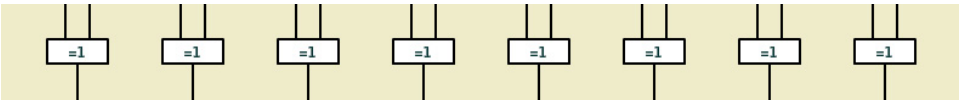


Abb. 57: Bildschirmausschnitt der Antivalenz

Auch in diesem Fall läuft die Animation gleich wie bei der Konjunktion ab.

Variante mit Steuerwerk

Bei der Variante mit Steuerwerk werden stets alle drei Operationen gleichzeitig durchgeführt. Mithilfe eines *Multiplexers (*kurz: *MUX)* wird das gewünschte Ergebnis selektiert und den Ausgängen zugeführt.

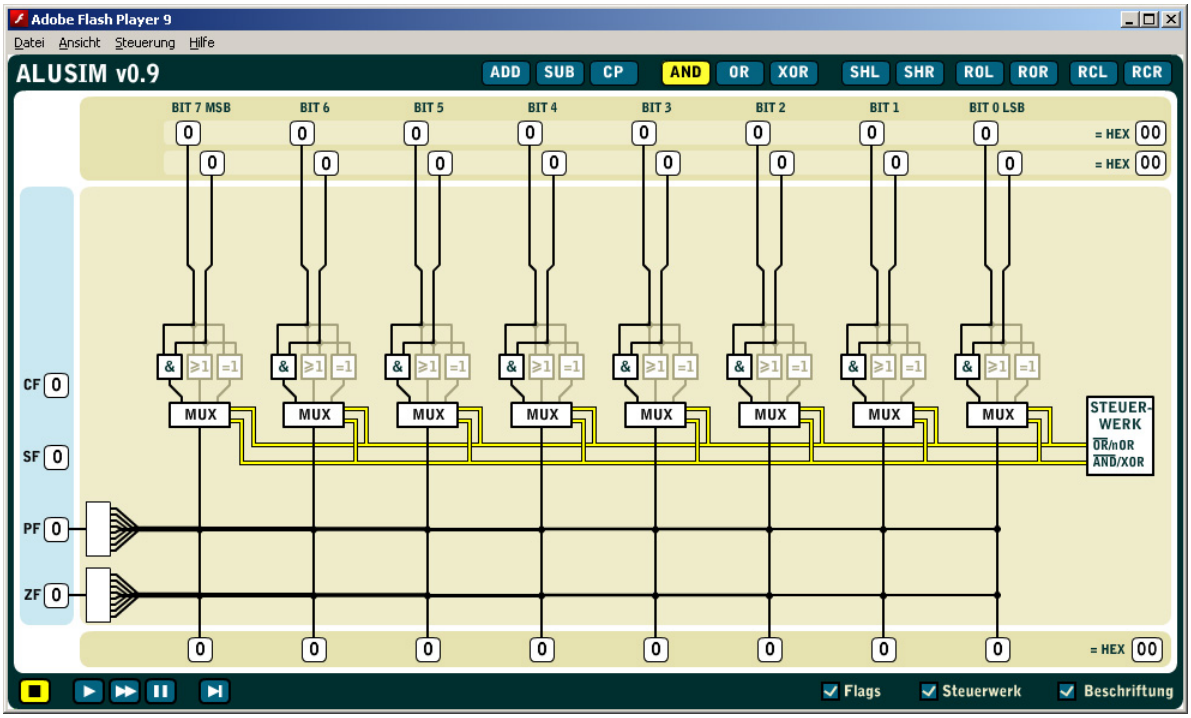


Abb. 58: Bildschirmabbild der Konjunktion mit Steuerwerk

Die Animation zeigt dies etwas simplifiziert, indem die Eingangswerte ausschließlich dem gewählten und nicht jeweils allen drei Gattern zugeführt werden. Diese Darstellung ist eine explizite Modellvereinfachung, da die Eingangswerte der Konjunktion-, der Disjunktion- als auch der Antivalenz-Gatter zugeführt werden müssten. Die entstehende semantische Lücke kann jedoch akzeptiert werden. Die gewählte Operation wird daraufhin durchgeführt und transportiert den Ergebniswert zum MUX, welcher

durch Wahl der Funktion im Menü bereits die richtigen Steuerleitungen gesetzt hat. Vom MUX wird nun der Ergebniswert auf der Leitung zu den Ausgängen transportiert.

4.2.3.3 SHL, SHR, ROL, ROR, RCL und RCR

Diese Operationen sind von der visuellen Aufbereitung her die komplexesten in dieser Arbeit. Das liegt daran, dass es sich hierbei nicht um *Schaltnetze* handelt (wie bei den bisher erläuterten Operationen), sondern um *getaktete Schaltwerke.* Das heißt, dass die Ermittlung der Ausgangswerte erst bei einer Taktflanke erfolgt. Darüber hinaus sind mehrere Taktzyklen zur Abarbeitung notwendig. Der *grundlegende Ablauf* lässt sich wie folgt beschreiben: **Laden eines Eingangswertes** beim ersten Takt und **Ausführen der Schiebe- oder Rotationsoperation** beim zweiten Takt.

In den folgenden *einfachen Modellen ohne Steuerwerk* wurde eine Veränderung vorgenommen, die von anderen Modell-Vereinfachungen abweicht: Das Problem der Eingangsauswahl beim ersten und zweiten Takt wurde mit Hilfe eines stilisierten Multiplexers gelöst, der als Schalter ohne Steuerleitungen ausgeführt wurde. Genau genommen müsste der Schalter die Information erhalten, wann er welchen Eingang weiterleiten muss. Aus Gründen der Vereinfachung wurde aber sowohl auf die technisch korrekte Bezeichnung MUX als auch auf die notwendige Zuführung der Steuerleitung verzichtet. Die Darstellung ist jetzt analog der eines Schalters. Im Folgenden wird dieser Schalter als *Eingangswahlschalter* bezeichnet.

Schiebeoperation nach links (shift left, SHL)

Hier werden die Eingangswerte (es handelt sich hier jeweils um die Bits des achtstelligen Eingangswerts) um jeweils eine Stelle nach links verschoben. Das heißt, dass an der *n*-ten Stelle der *(n-1)*-te Wert am Ausgang ausgegeben wird. Das LSB wird dabei auf den Wert *0* gesetzt.

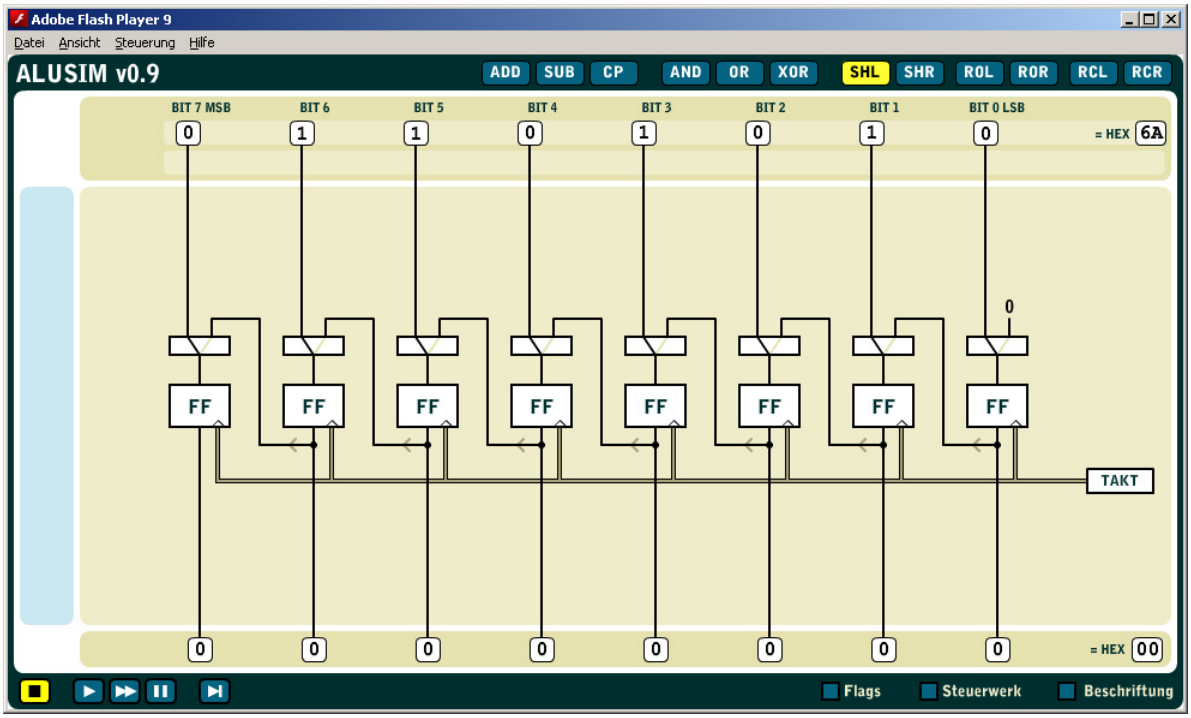


Abb. 59: Bildschirmabbild der Schiebeoperation nach links

Der Ablauf der Animation beginnt damit, dass die Eingangswerte über die Leitung vom Eingang zum Eingangswahlschalter kommen.

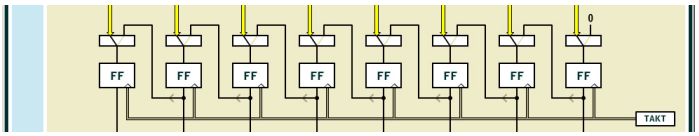


Abb. 60: SHL: Eingangswerte liegen am Eingangswahlschalter an

Dieser ist so geschaltet, dass der nun anliegende Wert über die Leitung zum Flip-Flop weitergeleitet wird.

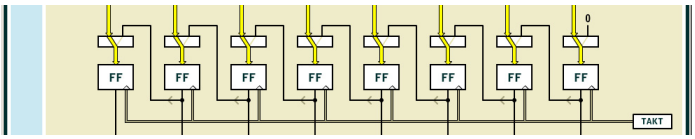


Abb. 61: SHL: Eingangswerte werden zum Flip-Flop weitergeleitet

Nachdem die Eingangswerte bei allen Flip-Flops anliegen, erfolgt eine Taktflanke, die durch eine 250ms lang gelb eingefärbte Taktleitung symbolisiert wird.

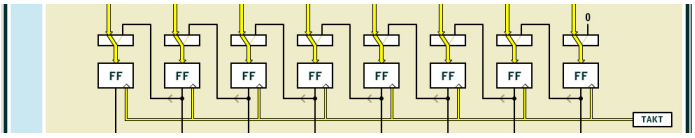


Abb. 62: SHL: erste Taktflanke

Nach einer weiteren kurzen Verzögerung – die das zeitliche Verhalten des Flip-Flops verdeutlichen soll – wird das angelegte Signal mit dem symbolisierten Wert auch am Ausgang des Gatters angelegt. Dieses Signal läuft sowohl zu den Ausgängen der ALU (die diese Werte aber nicht übernimmt) als auch zum nachfolgenden Flip-Flop mit dem davor befindlichen Eingangswahlschalter. Gleichzeitig wird die Animation des Wertes 0 für den nächsten Schritt am Eingang des LSB-Flip-Flop Eingangswahlschalter durchgeführt.

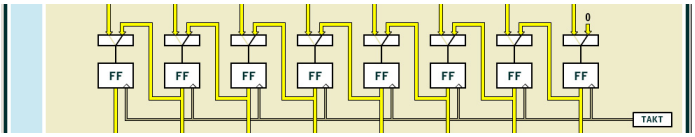


Abb. 63: SHL: Ausgänge der Flip-Flops liegen bei Nachfolgern an

Der Eingangswahlschalter wird nun umgeschaltet. Einkommende Werte können daher sofort zum Eingang des jeweiligen Flip-Flops weitergeleitet werden.

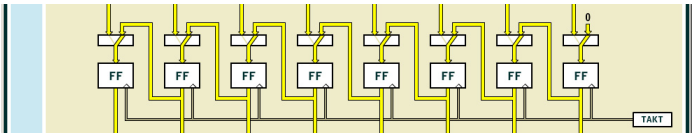


Abb. 64: SHL: weiterleiten des Werts mittels Eingangswahlschalter

Wenn alle Werte des Vorgängers als auch der explizit generierte Wert 0 an den Eingangswahlschaltern anliegen, erfolgt erneut die Darstellung einer Taktflanke durch die gelb gefärbte Taktleitung.

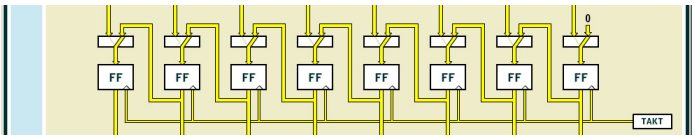


Abb. 65: SHL: zweite Taktflanke

Nach kurzer Verzögerung übernimmt nun der an jedem Flip-Flop anliegende Wert abermals und wird an den Ausgang gelegt.

Diesmal werden die Werte von den Ausgängen der ALU übernommen, da diese nun endgültig sind. Die Werte liegen in weiterer Folge wieder beim nachfolgenden Eingangswahlschalter an (und damit am zugehörigen Flip-Flop), was aber nicht weiter stört, da keine weitere Taktflanke das Übernehmen des Wertes erzwingt.

Schiebeoperation nach rechts (shift right, SHR)

Diese Operation ist analog der Schiebeoperation nach links, nur mit dem Unterschied, dass sich der *n*-te Wert aus dem *(n+1)*-ten Wert ergibt. Das *MSB* wird auf *0* gesetzt.

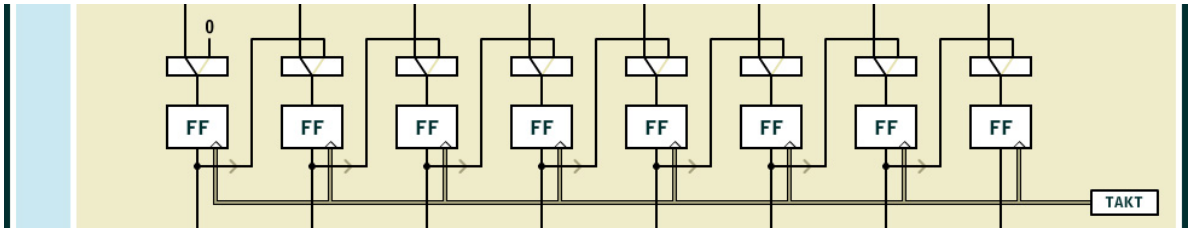


Abb. 66: Bildschirmausschnitt der Schiebeoperation nach rechts

Die Schiebeoperation nach rechts verläuft analog zur Schiebeoperation nach links mit den zwei Unterschieden, dass die Werte nach rechts weitergeleitet werden und das MSB anstatt des LSB auf 0 gesetzt wird.

Rotation nach links (rotate left, ROL)

Auch bei dieser Operation werden wie bei SHL die Werte an der *n*-ten Stelle von der *(n-1)*-ten Stelle übernommen. Der Wert des *LSB* wird jedoch vom *MSB* übernommen, so dass kein Wert dem LSB explizit zugeführt werden muss (bei SHL und SHR werden die Stellen, von denen die Werte in die nachfolgende Stelle übernommen werden, aber kein Eingabewert vorhanden ist, der Wert *0* erzeugt und zugeführt).

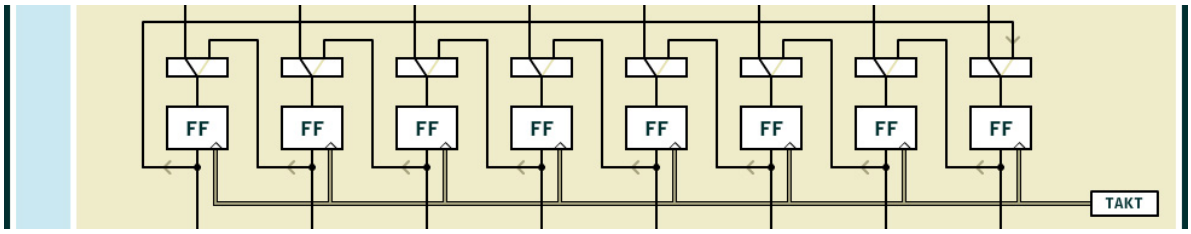


Abb. 67: Bildschirmausschnitt der Rotation nach links

Zunächst werden die Werte aus den Eingängen an die jeweiligen Eingangswahlschalter zugeführt.

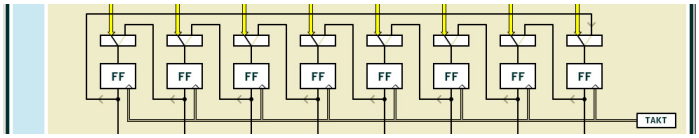


Abb. 68: ROL: Eingangswerte liegen am Eingangswahlschalter an

Die Eingangswahlschalter leiten die Werte zum Flip-Flop weiter.

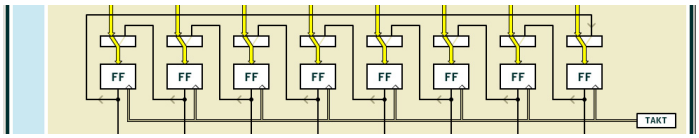


Abb. 69: ROL: Eingangswerte werden zum Flip-Flop weitergeleitet

Der nächste Schritt ist die erste Taktflanke.

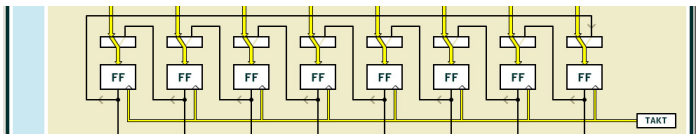


Abb. 70: ROL: erste Taktflanke

Die Ausgänge der Flip-Flops werden nach der Taktflanke generiert und diese Werte dem nachfolgenden Eingangswahlschalter zugeführt. Das Ausgangssignal des Flip-Flops vom MSB wird dem Eingangswahlschalter vom LSB angelegt.

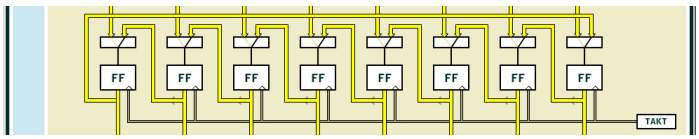


Abb. 71: ROL: Ausgänge der Flip-Flops liegen bei Nachfolgern an

Die Werte werden über die Eingangswahlschalter dem zugehörigen Flip-Flop zugeführt.

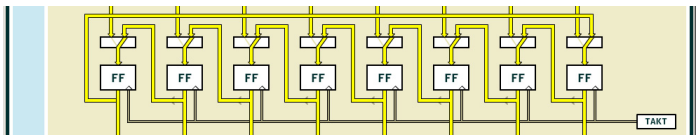


Abb. 72: ROL: weiterleiten des Werts mittels Eingangswahlschalter

Die anliegenden Werte werden wieder beim zweiten Takt übernommen und an die Ausgänge der ALU geführt.

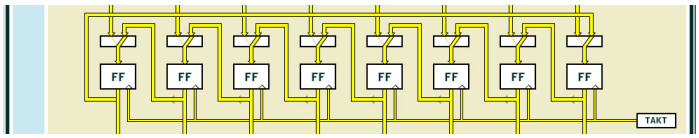


Abb. 73: ROL: zweite Taktflanke

Rotation nach rechts (rotate right, ROR)

Das Verhalten dieser Operation ist dem der Rotation nach links sehr ähnlich. Der *n*-te Wert ergibt sich aus dem *(n+1)*-ten Wert, mit Ausnahme des *MSB*. Dieser erhält den Wert vom *LSB*.

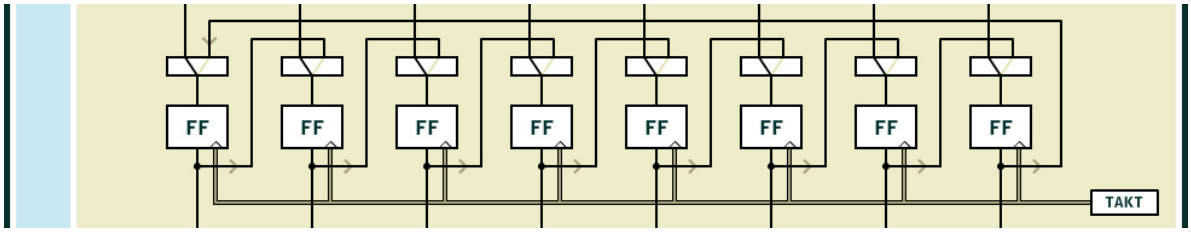


Abb. 74: Bildschirmausschnitt der Rotation nach rechts

Die Rotation nach rechts erfolgt wie die Rotation nach links, jedoch werden die Werte der Flip-Flops zum jeweils rechten Nachbar (anstatt zum linken) weitergeführt. Der Wert des *LSB*-Flip-Flops wird dem Eingangswahlschalter vom *MSB* zugeführt.

Rotation mit Carry Flag nach links (rotate with carry left, RCL)

Die Rotationen mit Carry Flag beziehen ihren Namen entsprechend das Carry Flag mit ein. Die Randwerte, die bei den bisherigen Operationen nicht benötigt wurden (Shift-Operationen) beziehungsweise die den jeweils komplementären Positionen zugeführt wurden (bei den Rotationsoperationen werden Werte der höchstwertigen den niederwertigsten Flip-Flops angelegt), werden nun in das Carry Flag übernommen, respektive von dort übernommen. Wichtig dabei ist die zeitliche Reihenfolge: Das *LSB* Flip-Flop übernimmt den Wert des Carry Flags und erst danach wird der neue Wert vom Flip-Flop-Ausgang des MSB übernommen.

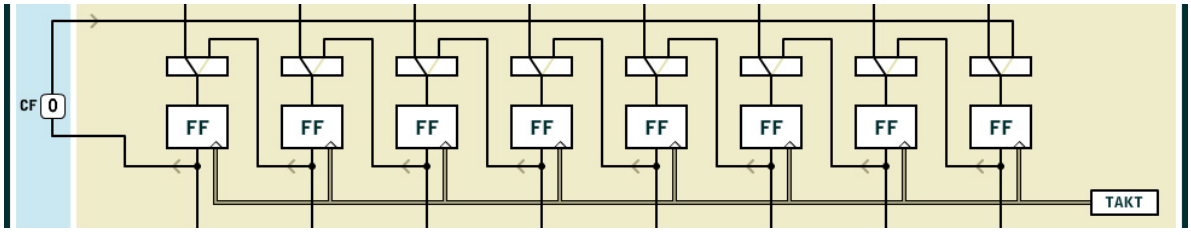


Abb. 75: Bildschirmausschnitt der Rotation mit Carry nach links

Die Animation beginnt wie bei den anderen Operationen mit der Übernahme des Eingangswertes in die Eingangswahlschalter.

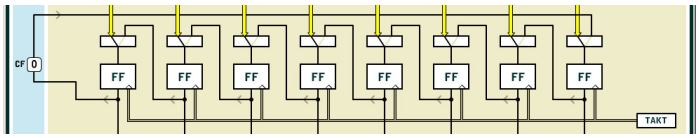


Abb. 76: RCL: Eingangswerte liegen am Eingangswahlschalter an

Die Eingangswerte liegen nun auch an den Flip-Flops an.

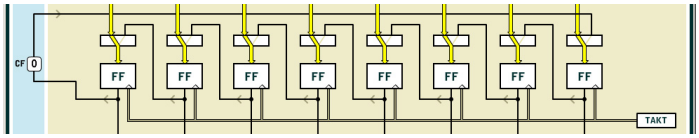


Abb. 77: RCL: Eingangswerte werden zum Flip-Flop weitergeleitet

Im Anschluss wird der erste Taktimpuls gesendet.

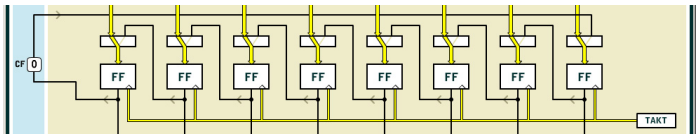


Abb. 78: RCL: Erste Taktflanke

Nach dem ersten Takt startet sofort der Wert aus dem Carry Flag auf der Leitung zum LSB- Eingangswahlschalter, um diesen Wert an das zugehörige *LSB*-Flip-Flop weiterzuleiten.

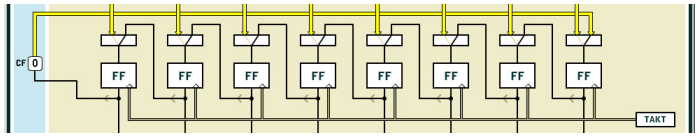


Abb. 79: RCL: Carry Flag wird sofort dem LSB zugeführt

Zuerst wird der Inhalt des Carry Flags in das *LSB* übernommen. Dann übernimmt das Carry Flag den Inhalt des *MSB*. Das Carry Flag wird hier also wie ein neuntes Bit verwendet („Rotate with carry“).

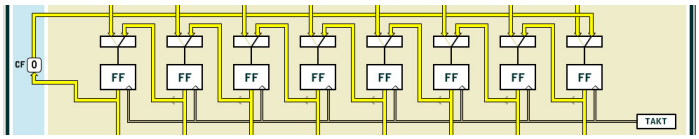


Abb. 80: RCL: Ausgänge der Flip-Flops liegen bei Nachfolgern und dem Carry Flag an

Die Werte werden von den Eingangswahlschaltern zu den Flip-Flops geführt.

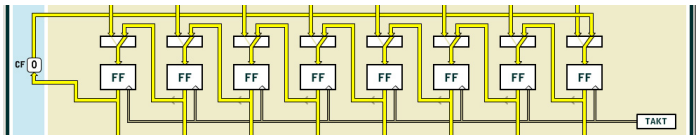


Abb. 81: RCL: weiterleiten des Werts mittels Eingangswahlschalter

Nachdem die Werte an den Leitungen wie erläutert anliegen, erfolgt die zweite Taktung und die rotierten Werte liegen an den Ausgängen der ALU an.

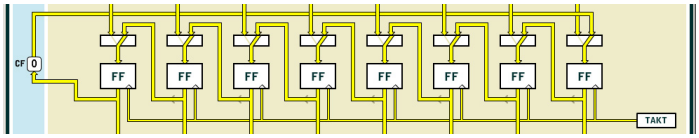


Abb. 82: RCL: zweite Taktflanke

Als Besonderheit bei RCL sei erwähnt, dass das Carry Flag hier immer angezeigt wird, auch wenn das Kontrollfeld für das Einblenden der Flags nicht aktiviert ist, da das Carry Flag für diese Operation zwingend erforderlich ist.

Rotation mit Carry Flag nach rechts (rotate with carry right, RCR)

Wie auch schon bei *Rotation mit Carry Flag nach links* erläutert, werden die Werte nicht nur um eine Stelle verschoben, sondern es wird zusätzlich das Carry Flag mit einbezogen. Das *n*-te Flip-Flop erhält den Wert des *(n+1)*-ten Flip-Flops. Das *MSB* wird durch den Wert des Carry Flags zum Zeitpunkt des Starts der Operation ersetzt.

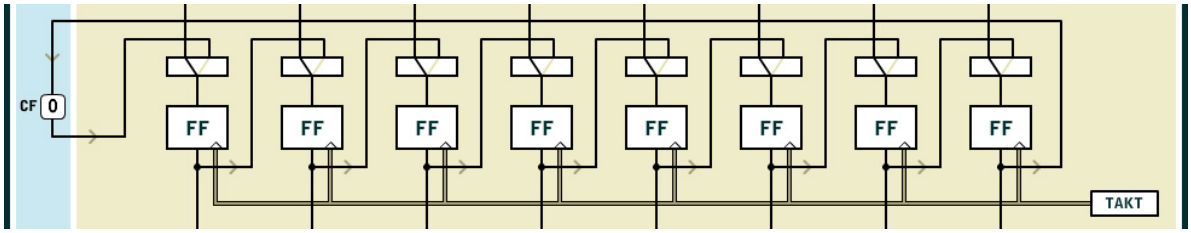


Abb. 83: Bildschirmausschnitt der Rotation mit Carry nach rechts

Analog zu den bisherigen Operationen erfolgt auch hier die Rotation mit Carry Flag nach *rechts* gleichartig wie die nach *links*. Nach der ersten Taktflanke läuft der Wert der Ausgangsleitung jedes Flip-Flops zum Eingangswahlschalter des nächsten niederwertigeren Flip-Flops. Der Ausgangswert vom *LSB*-Flip-Flop läuft zum Carry Flag, welches zeitlich vorher seinen aktuellen Wert als Eingabewert für den Eingangswahlschalter des *MSB*-Flip-Flops zur Verfügung stellt.

Wie bereits bei RCL erwähnt, gilt hier analog: Das Carry Flag wird in jedem Fall eingeblendet, da dies für die Abarbeitung der Operation unbedingt erforderlich ist.

Variante mit Steuerwerk

In dieser Version sind alle Operationen dieser Gruppe umsetzbar. Der Eingangswahlschalter wurde durch einen MUX ersetzt. Die Wahl der Zuführung des Wertes des linken oder rechten Flip-Flops wurde durch einen zusätzlichen Multiplexer realisiert, ebenso wie die Wahl der Eingänge der *LSB*- und *MSB*-Flip-Flops. Diese äußersten Flip-Flops müssen hinsichtlich der Wahl der Eingänge gesondert behandelt werden, da diese nicht nur das Signal des direkten Nachbarn übernehmen, sondern auch den Wert *0*

(bei Schiebeoperationen), den Wert des Flip-Flops am anderen „Ende“ der Schaltung (bei Rotationsoperationen) oder den Wert des Carry Flags (bei *Rotation mit Carry* Flag-Operationen).

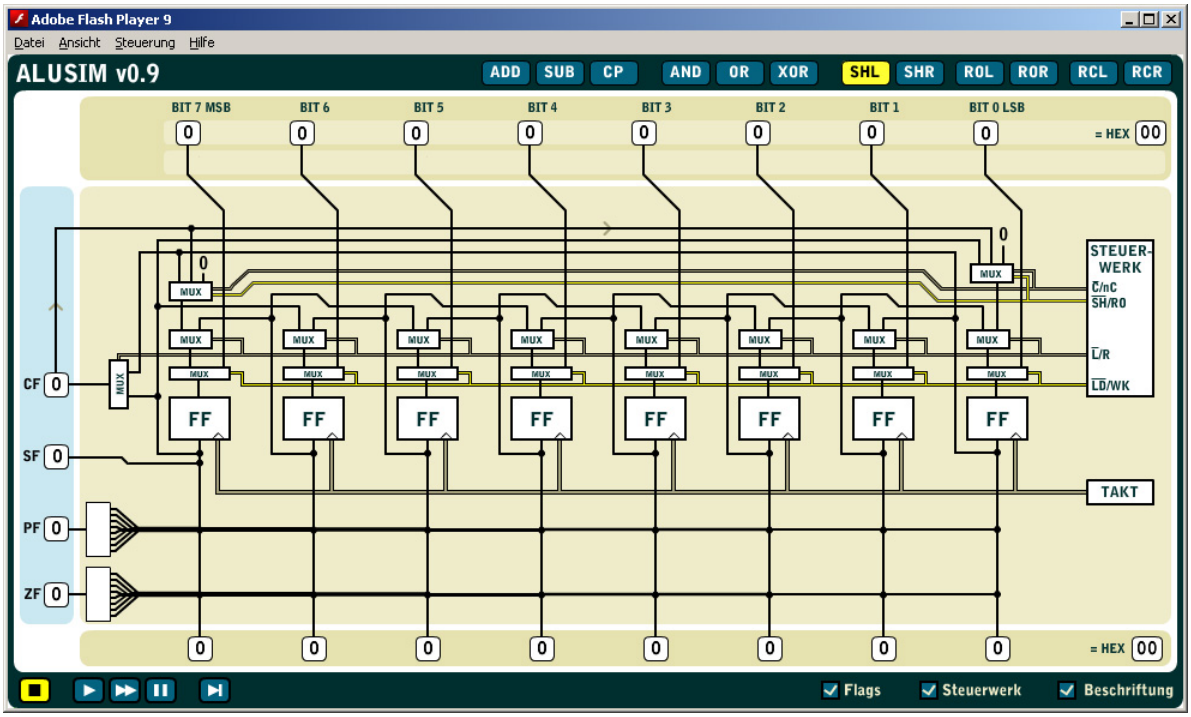


Abb. 84: Bildschirmabbild der Schiebe- und Rotationsoperationen mit Steuerwerk

Zweifellos ist durch die zahlreichen Verbindungsmöglichkeiten, dem relativ aufwändigen Steuerwerk und den Multiplexern für die Verbindungssteuerung das am meisten komplexe aller vorhandenen Modelle. Die Studierenden haben jedoch so die Möglichkeit, eine mögliche technische Umsetzung vollständig zu durchdenken.

Für jede der acht Stellen gibt es ein Flip-Flop als zentrales Speicherelement. Direkt davor sitzt ein MUX, der entweder beim ersten Schritt (dem *Laden*) den Eingangswert oder den von einem anderen Flip-Flop kommenden, weitergeschobenen Wert übernimmt (das *Ausführen der Schiebe- oder Rotationsoperation*). Dieser geschobene Wert wird durch einen zweiten MUX ausgewählt, der in der Regel den Wert vom linken oder rechten Nachbarn auswählen kann.

Beim LSB und MSB verhält es sich geringfügig anders: Der übernommene Wert wird von einem weiteren MUX vorausgewählt. Dies ist, je nach Operation, der Wert *0* beim *Schieben*, der des *LSB oder MSB* bei *Rotationen* oder der des *Carry Flags* bei *Rotationen mit Carry*.

Die Aufgabe des vor dem Carry Flag sitzenden MUX ist es, den Wert des Ausgangs entweder vom LSB oder vom MSB weiterzuleiten, je nach Operation.

Man könnte die zu je einem Flip-Flop gehörenden zwei Multiplexer (bzw. drei bei den äußeren beiden Flip-Flops) zu einem einzigen vereinfachen. Von dieser Optimierung wurde jedoch zugunsten der Nachvollziehbarkeit der Vorgänge Abstand genommen.

4.2.4 Vorgehensweise

Nach den ersten Besprechungen im Rahmen dieser Diplomarbeit und der Recherche anderer Projekte wurden die Schwerpunkte auf die Erstellung eines *didaktisch wirkungsvollen Modells* und einer *intuitiven Benutzeroberfläche* gelegt.

Aus diesem Grund wurden vor der tatsächlichen Umsetzung in *Adobe Flash* zahlreiche Entwürfe als Handskizzen oder digitale Grafiken erstellt und diskutiert. Dieser Abstimmungsprozess war von der zeitlichen Ausdehnung erheblich länger als die tatsächliche Programmierung. Da dieser Vorgang eine Vereinigung von kreativem Gestaltungsprozessen einerseits und technischen Modellierungsvorgehen andererseits darstellt, kann das Vorgehen in dieser Phase nur durch zahlreiche Iterationen die notwendige Qualität sicherstellen.

Der Grund dafür ist der kreative Teil der Gestaltungsleistung, der nicht (oder nur zum Teil) vorgefertigten Mustern oder Prozessen folgen kann. Das gewählte Vorgehen stellte sicher, dass bei der Umsetzung nur mehr bei kleinen Details Fragen entstanden sind, die allerdings rasch geklärt werden konnten.

Die Umsetzung erfolgte ebenso in zahlreichen Interationen: Zunächst wurden die Grundfunktionen grafisch und ohne Programmlogik in Flash realisiert und animiert. Sukzessive wurden reine Animationen durch programmgesteuerte Animationen ersetzt, bis die Programmlogik alle Animationen kontrollierte.

4.2.5 Besonderheiten und Schwierigkeiten

Aus technischer Sicht sind hinsichtlich der Umsetzung keine Schwierigkeiten zu erwarten gewesen und bis auf eine Ausnahme auch nicht aufgetreten: Leider stieß der Autor bei der Umsetzung auf einen Bug im *Actionscript 3*-Compiler, der ein *Refactoring*[57] des Programmcodes notwendig machte. Durch dieses Problem verzögerte sich die Arbeit um rund eine Woche, was aber auf die Gesamtlaufzeit der Umsetzung keine wesentliche Auswirkung hatte.

4.2.6 Eigene Standards

Neben den grafischen Vereinheitlichungen hinsichtlich der Benutzerschnittstelle (zum Beispiel das Farbkonzept) wurden bei der technischen Umsetzung die Grundsätze der objektorientierten Programmierung und darüber hinaus Entwurfsmuster eingesetzt. Diese garantieren eine Implementierung, die eine spätere Erweiterung oder ein Zusammenführen mit anderen Lösungen einfacher macht. Diese Konzepte werden in den Abschnitten 4.2.1 und I erläutert.

[57] Als Refactoring bezeichnet man das Umschreiben von Programmcode, mit dem Ziel, die gleiche Funktionalität mit einer anderen Programmstruktur zu erreichen. Das wirkt auf den ersten Blick wenig sinnvoll, hat aber im vorliegenden Fall des Bugs im Compiler seine Berechtigung. Refactoring wird in der Praxis auch oft bei gewachsenen Programmstrukturen eingesetzt, an deren Erstellung über einen längeren Zeitraum viele Personen gearbeitet haben und durch die Überarbeitung eventuell inkonsistente Umsetzungen neu strukturiert und gut weiterverwendbar gemacht werden können, da sich im Laufe der Zeit die Anforderungen an die Strukturen ändern.

4.3 Zusammenfassung

Der ALUSIM stellt für Studierende eine wertvolle Unterstützung dar, um die Vorgänge in der ALU des MC8 beobachten und beeinflussen zu können. Das Verhalten der zwölf simulierten Operationen der ALU kann dabei beliebig oft wiederholt und zu jedem Zeitpunkt und an jedem Ort durchgeführt werden. Der ALUSIM stellt zu den am Institut für Computertechnik bereits vorhandenen Lernhilfen für den MC8 (den MC8 Simulator und den MC8 ASM) eine gute Ergänzung dar und rundet damit das Leistungsspektrum der unterstützenden Lernhilfen ab.

Um dieses Ziel zu erreichen, wurde auf die Benutzerschnittstelle viel Wert gelegt. Aber auch die Erstellung des Modells wurde intensiv betrieben. Bei sämtlichen simulierbaren Operationen der ALU (das sind drei Gruppen: Arithmetische Funktionen ADD, SUB, CP. Logische Grundoperationen AND, OR, XOR sowie die Gruppe der der Schiebe- und Rotationsoperationen SHL, SHR, ROL, ROR, RCL und RCR) können die Eingangswerte gesetzt, aber auch der Detailreichtum des Modells variiert werden. So können beispielsweise die Flags und Beschriftungen ein- und ausgeschaltet werden. Außerdem ist es möglich, ein Steuerwerk einzublenden, was das Modell soweit generalisiert, dass via Steuerleitungen alle Befehle einer Gruppe abgearbeitet werden können.

Wichtig bei der Umsetzung waren das iterative Vorgehen bei der Erstellung des Modells und die Einhaltung von Standards für die Gestaltung der Benutzerschnittstelle und des Programmcodes.

5 Ausblick und Diskussion

Die folgenden Seiten widmen sich der Erweiterung des ALUSIM hinsichtlich einer konvergenten Lernumgebung. Einerseits wird kurz auf die Grundlagen von E-Learning Plattformen eingegangen, um im Weiteren die Möglichkeiten der Einbindung in bestehende Systeme der Technischen Universität Wien vorzustellen und die damit verbundenen technischen Anpassungen zu erklären. Andererseits kann eine engere Zusammenarbeit des **MC8 Simulators**, des **MC8 ASM** und des **ALUSIM** erfolgen. Beide Ansätze werden nun motiviert und diskutiert. Dabei wird auch auf technische Aspekte eingegangen.

5.1 Grundlagen von E-Learning Plattformen

Die Frage, woraus eine E-Learning Plattform (auch Lernplattform) besteht und welche Funktionalität sie bietet, kann man sehr gut mit folgender Definition beantworten:

> *Lernplattform: Ein Softwaretool, auf welches im Intranet/Internet zugegriffen werden kann, und das über eine entsprechende Oberfläche bestimmte Funktionalitäten, wie den Aufruf und die Administration von Lernern, Lerninhalten, Übungsaufgaben, Kommunikationstools usw. von einer zentralen Stelle aus ermöglicht. Sie ist die zentrale Schnittstelle einer Lernumgebung zwischen Trainingsanbietern und Trainingskunden (Dr. Günter Pees). Eine Plattform verfügt gewöhnlich nicht über Autoren-Tools zur Erstellung von Kursen. Neudeutsch wird auch das Wort LMS verwendet. [ELEA2001]*

Ein **LMS** ist ein Akronym von *Learning Management System*, also der englischen Bezeichnung einer *Lernplattform*.

In einem *LMS* befindet sich ein *Datenspeicher* (*Object Repository*), der alle Daten zentral verwaltet. Die Vortragenden stellen nun ihre Inhalte in das System und verknüpfen sie miteinander. Es gibt Systeme, die eine umfangreiche Benutzerverwaltung zulassen, so dass festgelegt werden kann, wer Inhalte einpflegen und warten darf und wer auf diese Inhalte Zugriff bekommt. In solchen Systemen ist es möglich, nicht nur Inhalte zur Durcharbeitung zur Verfügung zu stellen, sondern auch Prüfungen durchzuführen. So könnte beispielsweise festgelegt werden, dass Studierende gewisse Inhalte zunächst durcharbeiten müssen und am Ende jedes Teils Fragen zum Inhalt gestellt bekommen, die dann auf Korrektheit überprüft werden.

Eines dieser *LMS* ist *Moodle*[58]. Dies ist ein Open Source Produkt, das bereits eine weite Verbreitung und damit gute Unterstützung bietet. Die Technische Universität Wien nutzt dieses System, wie im folgenden Abschnitt erläutert wird.

5.2 TUWEL

Die TU Wien betreibt ein zentrales Kompetenzzentrum zur Bündelung der E-Learning Aktivitäten. Es bietet Lehrenden als auch Studierenden Serviceleistungen im Zusammenhang mit E-Learning. Beide Personengruppen haben so einen zeitlich und örtlich flexiblen Zugang zu den Lerninhalten und zum Support. Seit Beginn des Sommersemesters 2006 steht die auf *Moodle* basierende Lernplattform *TUWEL* zur Verfügung. Die Zahl der Lehrveranstaltungen, die *TUWEL* nutzen, um Inhalte zur Verfügung zu stellen, steigt sehr rasch[59]:

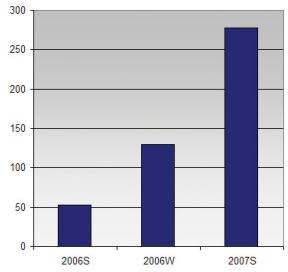


Abb. 85: Zahl der Lehrveranstaltungen, die TUWEL nutzen

Das Ziel des E-Learning Zentrums ist es, ein breites Spektrum an alternativen Lernformen und innovativen Lehrmethoden zu unterstützen. Darüber hinaus wird Unterstützung für die Lehre an der Technischen Universität Wien angeboten.

Aufgrund der zunehmenden Ergänzung der Lehre durch E-Learning ist dies eine gute Möglichkeit, den ALU-Simulator in einer solchen Umgebung zu implementieren.

5.3 Technische Anpassungen für TUWEL

In einem ersten Schritt ist es möglich, den ALUSIM wie eine Abbildung in *Moodle* einzufügen. Eine weitergehende Konfiguration ist nicht notwendig. Der Simulator erscheint dann an der gewünschten Stelle an Stelle eines Bildes und kann von den Studierenden einfach benutzt werden.

Kommt es jedoch zu Erweiterungen, die die Kommunikation des Simulators mit dem *LMS* notwendig machen (zum Beispiel Rückmeldungen, ob eine spezielle Person den Simulator bereits benutzt und danach inhaltliche Fragen beantwortet hat), gibt es zwei Möglichkeiten, diese in *TUWEL* einzubinden:

[58] http://www.moodle.org
[59] [TUWE2007]

Einerseits das *LAMS*[60] Kursformat, andererseits das *SCORM/AICC*[61]-Format. Beide stellen einen kompletten Lerninhalt als Paket dar, in das man den ALUSIM einbringen muss. Dieses Paket kann dann auch auf anderen Lernplattformen installiert und verwendet werden. Auf diese Anpassungen im Detail einzugehen, sprengt jedoch den Rahmen dieser Arbeit.

5.4 Vernetzung der Module

Denkbar ist eine Zusammenführung mit dem MC8 Simulator und dem MC8 ASM. Ein mögliches Szenario sieht vor, zunächst Assemblercode zu schreiben, diesen von MC8 ASM übersetzen zu lassen. Das Ergebnis, ein Maschinencode, kann im MC8 Simulator geladen und abgearbeitet werden. Sind darunter Befehlsteile, die die ALU abarbeiten muss, öffnet sich der ALU-Simulator und zeigt den Ablauf der jeweiligen Operation. Das Ergebnis wird in den MC8 Simulator rückgeführt und es geht mit dem nächsten Befehl weiter. Natürlich muss es dennoch möglich sein, alle drei Simulatoren getrennt voneinander zu verwenden.

[60] Learning Activity Management System, kurz LAMS ist ein Werkzeug, um die Zusammenarbeit bei E-Learning Aktivitäten zu designen, zu organisieren und zu verteilen.
[61] SCORM (Sharable Content Object Reference Model) und AICC (Aviation Industry CBT Committee, CBT steht dabei für Computer Based Training) sind zwei so genannte *Containerformate*. Das bedeutet, dass man sämtliche Unterlagen und Lernaktivitäten in einem SCORM/AICC-Container ablegen und in LMS, die mit diesen Formaten umgehen können, einsetzen kann.

ANHANG

I. Implementierungsübersicht

Adobe Flash bietet zahlreiche Ansätze, um ein Problem auf unterschiedliche Arten zu lösen. Um dem Charakter einer Diplomarbeit gerecht zu werden, wurden moderne Softwarearchitekturen und agile Projektmanagementmethoden eingesetzt.

Als grundlegende Softwarearchitektur wurde das **Model-View-Controller-Designpattern**[62] gewählt. Es entkoppelt die *Daten- bzw. Zustandsspeicherung* (*Model*) von der *Darstellung* (*View*). Als dritte Komponente gibt es *Controller*, der sich um das Zusammenspiel von *Model* und *View* kümmert. Im vorliegenden Fall ist die Anwendung sehr *View*-orientiert, da alle Animationen den meisten programmtechnischen Aufwand bedingen. *Controller*-Funktionalität wurde daher an einigen Stellen näher zur *View* gebracht, damit der Overhead bei der Umsetzung möglichst gering bleibt. Darüber hinaus wurde ein Abstraktionslevel eingezogen, um die Komplexität zu optimieren. Eine Konsequenz daraus ist, dass für je eine der drei Gruppen (ADD/SUB/CP, AND/OR/XOR und SHL/SHR/ROL/ROR/RCL/RCR) eine *View* existiert, die sich um alles innerhalb dieser Anzeige kümmert.

In der **Hauptklasse** (Main) werden alle Komponenten initialisiert und mit dem Model lose verbunden. Dadurch können sich die Komponenten für Änderungen bei den Daten an einem *Benachrichtigungsdienst* (*Eventdispatcher*) registrieren und bekommen so automatisch eine Nachricht, wenn sich Daten im *Model* geändert haben. Liegen modifizierte oder neue Daten im *Model* vor, werden durch die Nachricht alle registrierten Komponenten benachrichtigt, ohne dass das *Model* weiß, wer aller benachrichtigt wird. Das ist das Prinzip der losen Kopplung, das ein Grundsatz moderner Softwareentwicklung ist[63].

Ein anderer wichtiger Grundsatz in der modernen objektorientierten Programmierung ist es, *Komposition vor Vererbung* zu stellen. Um diesen Zusammenhang zu erläutern, ist es von Bedeutung, wie *Adobe Flash* Animationen und Grafiken handhabt. Diese werden immer in der Bibliothek abgelegt.

---

[62] Designpattern (Entwurfsmuster) werden in der Architektur eingesetzt, um ähnliche, immer wieder auftretende Probleme nach einem vorgegebenen Muster zu lösen. Diese Überlegung, Lösungen, die immer wieder auftreten, auf eine bestimme Art zu lösen, setzt sich auch in der Softwareentwicklung immer weiter durch. Designpatterns beschreiben dabei Gedanken zur optimalen Umsetzung konkreter Anwendungsfälle. Wichtig dabei ist jedoch, sich nicht stur auf die Umsetzung nach der Literatur zu halten, sondern die sinnvollen Teile der Lösung in die eigene Arbeit zu übernehmen.

[63] Eine *lose Verbindung* ist eine architektonisch gute Methode, um Klassen, die miteinander kommunizieren müssen, zu verbinden. Dies geschieht mittels Benachrichtigungen (Events), die die Klasse aussendet, sobald sich etwas für andere Klassen relevantes getan hat. Bei dieser losen Verbindung *registriert* sich die Klasse, die sich dafür interessiert bei der aussendenden, um solche Benachrichtigungen ebenfalls zu erhalten. Auf diese Weise vermeidet man eine zu enge Bindung durch zu viel Wissen über die Struktur der jeweils anderen Klasse; was ein Vorteil ist, wenn sich Änderungen im Programmcode ergeben und man alle abhängigen Klassen nachziehen müsste. Bei einer engen Bindung müsste zum Beispiel eine Klasse eine andere Klasse genau kennen und eine Funktion in dieser Aufrufen, um einen Status nachzufragen.

Möchte man nun mittels *Actionscript* darauf zugreifen, muss man die Grafik (bzw. Animation) in ein so genanntes *Symbol* umwandeln. Bei jedem Übersetzungsvorgang (Kompilieren) wird automatisch von jedem *Symbol* eine Klasse angelegt. Diese Klasse könnte man nun als Grundlage nehmen und Funktionalitäten ergänzen (Vererbung). Aus Sicht der Softwarearchitektur ist es jedoch zweckmäßiger, eigene Klassen zu schreiben, die diese *Symbol-Klassen* einbinden (instanzieren[64]). Dadurch wird man unabhängiger von Änderungen bei den Symbolen. Die weiter unten zuletzt aufgeführten Komponenten bedienen sich dieses Konzepts exzessiv. Jeder Komponente (Klasse) ist bekannt, welche Symbole benötigt werden und spricht diese explizit an, konfiguriert diese und kümmert sich um die Sichtbarkeit der angezeigten Symbole.

Die relevanten Komponenten sind:

- **Datenmodell** (Model)
  Das Datenmodell muss zunächst einmal erstellt werden. Damit es jedoch nicht mehrmals gleichzeitig erstellt werden kann, kommt das Singleton-Designpattern zum Einsatz. Dieses verhindert die mehrfache Erstellung vom Datenmodell, da dies zu einem Problem würde. Existieren mehrere Datenmodelle, können diese unterschiedliche Daten beinhalten und so zu Inkonsistenzen führen (Programmteile greifen auf unterschiedliche Datenmodelle zu und haben für ihre Durchführung unterschiedliche Inhalte).
- **Menü**
  Dieser Teil ist für die Auswahl der Operation (ADD, SUB, etc.), die Anzeige-Eigenschaften (Flags, Steuerleitungen und Beschriftung anzeigen) sowie für die Auswahl des Animationsmodus (Stopp, schnelle oder langsame Animation, schrittweise Abarbeitung) zuständig.
- **Simulationen**
  Die ADD/SUB/CP-, AND/OR/XOR- und die SHL/SHR/ROL/ROR/RCL/RCR-Simulation haben alle ähnliche Aufgaben: Sie konfigurieren die Anzeige (welche Schaltung wird in welchem Detaillierungsgrad angezeigt?) und führen Animationen und Berechnungen durch. Diese Komponenten vereinen *View* und *Contoller*, da eine Trennung eine unnötige Anhebung des Komplexitätsgrades zur Folge hätte.

Es gibt noch weitere Komponenten (zum Beispiel die Darstellung des Hintergrunds), die aber keine wesentliche Rolle spielen.

In weiterer Folge erläutert diese Arbeit das *Datenmodell* (*Model*), um das dahinter stehende Konzept zu beleuchten. Danach wird auf die Klassenstruktur eingegangen

[64] Eine Klasse ist ein Muster, in dem man Eigenschaften und Verhalten festgelegt. *Instanziert* man eine Klasse, entsteht ein so genanntes *Objekt*, das eine eigenständige Speicherung der Eigenschaften nach dem Muster der Klasse hat und das Verhalten der Klasse übernimmt.

II. Datenmodell

Die Simulation der MC8 ALU benötigt *kein* umfangreiches Datenmodell, da die gespeicherten Daten im Wesentlichen die Eingangs- und Ausgangsdaten sind. Dazu kommen noch Werte der Flags und der Status der Menüs: Ausgewählte *Operation*, Art des *Simulationsmodus* sowie *Geschwindigkeit* der Simulation.

Abgesehen davon wurde im Datenmodell eine Liste vorgesehen, die dazu dient, nicht mehr benötigte Benachrichtigungen (Events) abzustellen. Das ist erforderlich, da bei jedem Beginn einer Animation die Benachrichtigungen neu eingestellt und zuvor alle alten Benachrichtigungsdienste zurückgesetzt werden. Damit dies automatisch geschehen kann, wurde diese Liste erstellt:

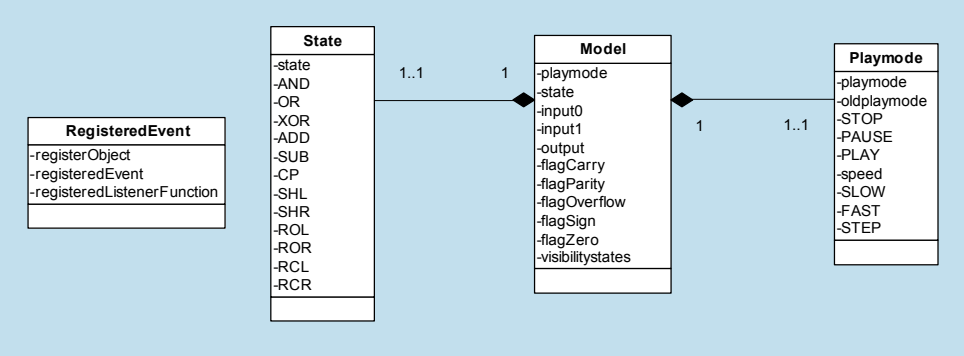


Abb. 86: Datenmodell des ALUSIM

III. Klassenmodell

In diesem Abschnitt wird auf die Programmstruktur eingegangen, wobei der Blick auf die wesentlichen Klassen gerichtet ist. Das untenstehende Klassendiagramm zeigt eine vereinfachte, aber für das Verständnis ausreichende Darstellung.

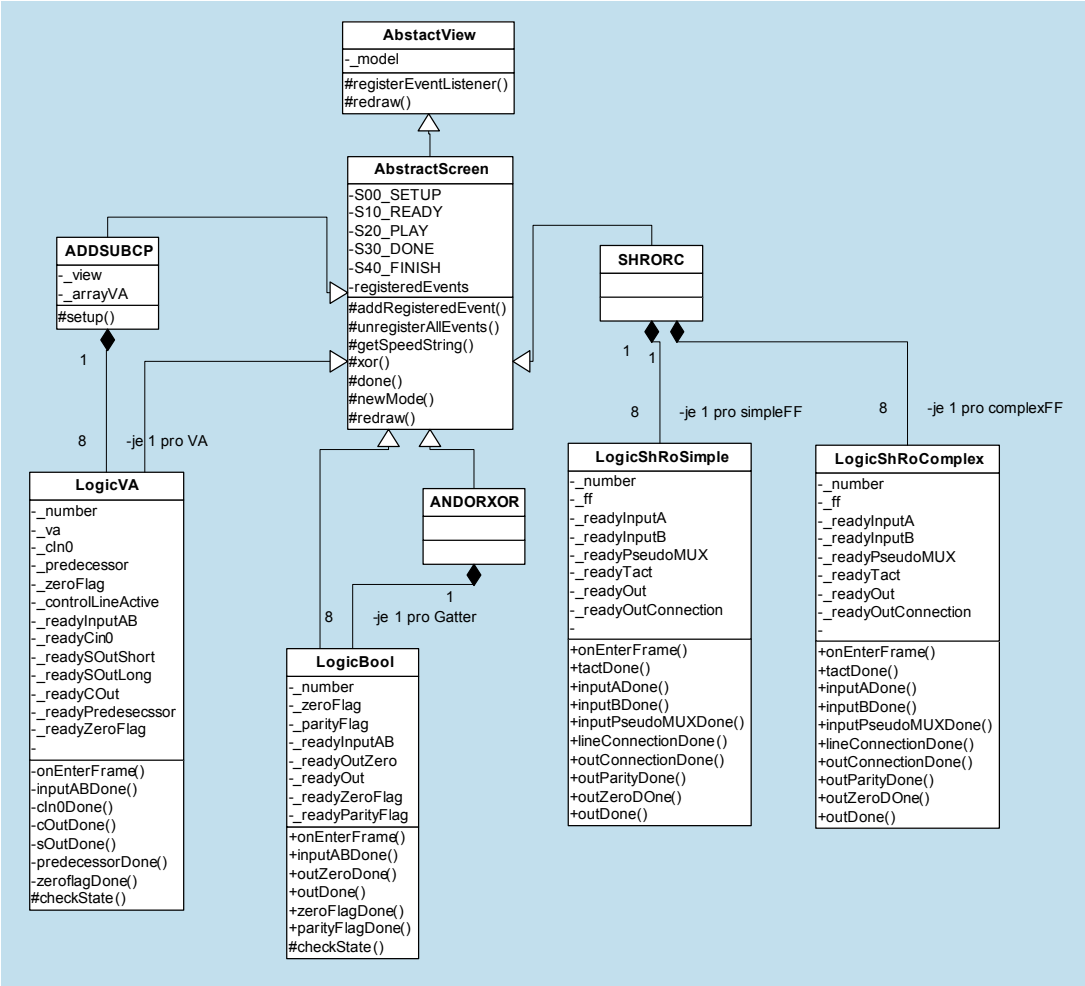


Abb. 87: Klassendiagramm des ALUSIM

Die abgebildeten Klassen werden im Folgenden beschrieben:

- **AbstractView**
  Diese Klasse stellt grundlegende Model-Verbindungsfunktionalitäten für jede View zur Verfügung. Das heißt, dass neben einer Verbindung zum Model auch eine Behandlung der relevanten Events vorhanden ist.

- **AbstractScreen**
  Für die grundlegende Statusverwaltung ist diese Klasse zuständig. Sie beinhaltet neben Statuscode-Definitionen auch allgemein gültige Funktionen, die alle Views benötigen. Dazu

zählen zum Beispiel die XOR-Funktion, die in der benötigten Form nicht in Actionscript 3 vorliegt und die Funktion getSpeedString, die die gewählte Geschwindigkeit als String retourniert.

- **ANDORXOR**
  Es werden acht *LogicBool*-Instanzen erstellt und konfiguriert. Darüber hinaus beschränken sich die Aufgaben dieser Klasse auf die Konfiguration der allgemeinen Teile für die Operationen AND, OR und XOR: Konfigurieren des Steuerwerks, der Flags und deren Zuleitungen. Das beinhaltet sowohl das Ein- und Ausblenden, als auch die richtige Beschaltung der Steuerleitungen.

- **LogicBool**
  Hier wird die Funktionalität eines Schaltnetzes, das die Operationen AND, OR und XOR ausführen kann, simuliert und visualisiert. Es gibt zwei unterschiedliche Visualisierungen: Eine mit dem gewählten Gatter (AND, OR, XOR) und eine Version mit allen Gattern gleichzeitig, inklusive Steuerwerk und MUX zur Auswahl der gewünschten Operation.

- **ADDSUBCP**
  In dieser Klasse werden, ähnlich wie bei ANDORXOR, das Steuerwerk, die Flags und deren Zuleitungen konfiguriert. Viel wichtiger ist in diesem Fall die Instanzierung der acht Volladdierer-Schaltnetze (LogicVA) und deren Konfiguration. ADDSUBCP stellt die richtige Verbindung zwischen den Instanzen her und ermöglicht so erst die Berechnung, denn die Volladdierer sind auf Ergebnisse aus ihren Vorgängern angewiesen.

- **LogicVA**
  Die Operationen eines Volladdier-Schaltnetzes mit Ersatzaddition wird in dieser Klasse abgebildet. Zusätzlich kann ein Compare-Befehl (CP) durchgeführt werden, was der Subtraktion entspricht, jedoch ohne ein Ergebnis an die Ausgänge zu legen (die Flags werden dabei sehr wohl gesetzt). Die Instanzen von LogicVA haben Kenntnis über ihre Position und können sich dementsprechend selbst konfigurieren. Sie schaffen dabei mittels Events lose Bindungen zu anderen Instanzen von LogicVA.

- **SHRORC**
  Diese Klasse beinhaltet mehr Logik als ADDSUBCP oder ANDORXOR. Der Grund dafür ist einerseits die doppelte Anzahl an Operationen gegenüber den zuvor genannten Klassen. Andererseits unterscheiden sich die Schaltungen komplett – in Abhängigkeit von der Visualisierung mit oder ohne Steuerwerk. Hinzu kommt, dass diese beiden Operationen taktabhängig sind und einer übergeordneten Steuerung des Takts bedürfen, welche durch SHRORC gewährleistet wird. Natürlich ist auch diese Klasse wieder für die Instanzierung der Schaltwerke für Shift-, Rotate- und RotateWithCarry-Operationen zuständig. Jede der genannten Operationen kann nach links oder rechts durchgeführt werden.

- **LogicShRoSimple** und **LogicShRoComplex**
  Diese beiden Klassen unterscheiden sich durch die Visualisierung mit (LogicShRoComplex) und ohne (LogicShRoSimple) Steuerwerk. In beiden Fällen kümmern sich die jeweils acht Instanzen um das Laden des Eingangswertes, das Weiterleiten des geladenen Wertes nach links oder rechts (bzw. zum oder vom ersten bzw. letzten Element) sowie zum oder vom Carry Flag.

IV. Funktionsimplementierungen

In diesem Abschnitt werden Kernfunktionen anhand eines Programmcode-Listings erklärt. Die abgedruckte Klasse ist für je ein Gatter samt Ein- und Ausgängen für die Operationen AND, OR und XOR zuständig. Das Vorgehen ist bei den anderen Operationen äquivalent. Der hier abgedruckte Code ist eine vollständige Klasse, die an wichtigen Stellen für Erklärungen, die direkt auf den Programmcode referenzieren, unterbrochen ist.

```
package at.ac.tuwien.ict.views.screens.bool {
    import flash.display.MovieClip;
    import flash.events.Event;

    import at.ac.tuwien.ict.model.Model;
    import at.ac.tuwien.ict.model.Playmode;
    import at.ac.tuwien.ict.model.State;
    import at.ac.tuwien.ict.views.screens.AbstractScreen;

    import bool.Bool;
    import bool.Out;
```

Die *import*-Statements stellen die Verbindung zu anderen Klassen und Symbolen in Bibliotheken her. In der letzten Zeile sieht man beispielsweise den Import des Symbols *Bool* aus dem Bibliotheksordner *bool*[65]. Damit kann man auf die Grafiken, die für die Darstellung der Boolschen Operationen notwendig sind, zugreifen.

```
    public class LogicBool extends AbstractScreen {

        private const ZERO_DONE  : String = "ZERO_DONE";

        private const DETAIL_AND : String = "DETAIL_AND";
        private const DETAIL_OR  : String = "DETAIL_OR";
        private const DETAIL_XOR : String = "DETAIL_XOR";

        private const SIMPLE_AND : String = "SIMPLE_AND";
        private const SIMPLE_OR  : String = "SIMPLE_OR";
        private const SIMPLE_XOR : String = "SIMPLE_XOR";

        private const ANDORXOR : String = "ANDORXOR";

        private var _number : int = 0;
        private var _bool : Bool;
        private var _zeroFlag : MovieClip;
        private var _valueInputAB_BitA : Boolean = false;
        private var _valueInputAB_BitB : Boolean = false;
        private var _valueOut : Boolean = false;

        private var _prefixInputAB : String;
        private var _prefixBoolBoxes : String;
        private var _prefixOut : String;
        private var _prefixZeroflag : String;

        private var _readyInputAB : Boolean = false;
        private var _readyOutZero : Boolean = false;
        private var _readyOut : Boolean = false;
        private var _readyZeroFlag : Boolean = false;

        private var _startedOut : Boolean = false;
        private var _startedZeroFlag : Boolean = false;
```

In diesem Teil sieht man die Definition der einzelnen Zustände, welche ihre Entsprechung auch in den Symbolen in der Bibliothek haben. Das zuvor importierte Symbol *Bool* besteht im Wesentlichen aus vier Teilen:

[65] Actionscript unterscheidet zwischen Groß- und Kleinschreibung

1. Zunächst sind hier die **Eingänge** zu erwähnen. Diese sind die Verbindungen von den Eingabebereichen an der oberen Seite des Simulators bis zur Mitte. Diese Verbindungen bestehen entweder aus einer Direktverbindung vom Eingabewert zum Gatter – wenn man keine Steuerleitungen eingeblendet hat. Dafür sind die Zustände SIMPLE_AND, SIMPLE_OR und SIMPLE_XOR da.
   Sind Steuerleitungen sichtbar, sind in der Mitte alle drei Gatter (AND, OR und XOR) und die Leitungen verzweigen zu jedem dieser einzelnen Gatter, wobei nur die relevanten Leitungen schwarz sind – während die restlichen Leitungen ausgegraut sind. Die zugehörigen Zustände dafür sind DETAIL_AND, DETAIL_OR und DETAIL_XOR.
2. Als nächstes kommt die Darstellung der **Gatter** dran, die entweder das gewünschte Gatter mit der Funktion anzeigt (keine Steuerleitungen) oder alle drei Gatter nebeneinander zeigt (mit Steuerleitungen).
3. Die **Ausgangsleitungen** führen die Ergebniswerte direkt zum Ausgang, wenn keine Steuerleitungen eingeblendet sind. Im Falle von Steuerleitungen muss das Ergebnis der richtigen Operation noch mit Hilfe eines Multiplexers ausgewählt werden und erst nach dem Multiplexer geht die Verbindung zu den Ausgängen.
4. Das **Steuerwerk** und die zugehörigen Steuerleitungen sind die vierte Komponente. Auch diese benötigen diese Zustände, da dementsprechend die zwei Steuerleitungen, die zu den Multiplexern führen gesetzt werden müssen.

```
public function LogicBool( model : Model,
                        number : int,
                        bool : Bool,
                        zeroFlag : MovieClip ) : void {
    super(model);
    _number = number;
    _bool = bool;
    _zeroFlag = zeroFlag;
    _localState = S00_SETUP;
    checkState();
}
```

Im *Konstruktor*[66] der Klasse wird zunächst der *Konstruktor* der *Superklasse*[67] *AbstractScreen* aufgerufen und die Referenz auf das Model übergeben. In der Superklasse werden die Events registriert, die für den Ablauf notwendig sind. Bei der Standardeinstellung der *Superklasse* ist es so, dass bei jeder Änderung der Eingangs- oder Ausgangsanzeige ein Neuzeichnen des Bildschirminhalts erfolgt und damit auch ein Rücksetzen der Animation. Das ist im Normalfall auch gewünscht, da bei Benutzereingaben während einer Animation das Verhalten der Schaltung nicht konsistent wäre. Die durchgeführten Operationen liefern ihr Ergebnis jedoch in das Model, welches die Ausgangsfelder mittels Event von der Änderung benachrichtigt, damit diese sich neu darstellen, da es ja eine Änderung gegeben hat. Das hätte auch ein Neuzeichnen und Rücksetzen bei der gerade durchgeführten Operation zur Folge. Sobald also der erste Ergebniswert in das Model geschrieben wird, würde die Operation durch die Neuzeichnung abgebrochen, was nicht sinnvoll ist. Daher wird im Folgenden die Registrierungsfunktion überschrieben und dieser Effekt umgangen:

```
override protected function registerEventListener() : void {
    _model.addEventListener(Model.MENU_CHANGED, redraw);
    _model.addEventListener(Model.INPUT_CHANGED, redraw);
    _model.addEventListener(Model.OUTPUT_CHANGED, outputChanged);
```

[66] Der Konstruktor wird in der Regel automatisch bei einer Instanzierung aufgerufen und hat die Aufgabe, das durch die Instanzierung erstellte Objekt in einen Anfangszustand zu bringen.
[67] Der Begriff Superklasse wird bei der Vererbung in der Objektorientierten Programmierung verwendet. Sie kennzeichnet diejenige Klasse, von der geerbt wird. Vererben heißt, dass man alle Eigenschaften und Verhaltensweisen übernimmt.

```
        _model.addEventListener(Model.FLAGS_CHANGED, redraw);
        // little change here:
        _model.playmode.addEventListener(Model.PLAYMODE_CHANGED, newMode);

        addEventListener(Event.ENTER_FRAME,  onEnterFrame, false, 0, true);
    }
    private function outputChanged(event : Event) : void {
        // do nothing when output changes.
    }
```

In der letzten Zeile der Funktion *registerEventListener()* wird ein *ENTER_FRAME*-Event registriert. Das wird benötigt, da die Bits auf den Leitungen, die für je einen Wert 0 oder 1 stehen, diesen Wert auch anzeigen müssen. Das hätte man einerseits mittels einer Funktion im jeweiligen Bibliothekssymbol lösen können. Das hätte leider mit sich gezogen, dass nur wegen dieser Wertanzeige für *jedes* Symbol eine Klasse erstellt werden müsste. Da dies viele fast idente Klassen mit sich bringt und der ALUSIM keine hochperformante Applikation ist, erlaubt sich der Autor einen Überhang an vermeidbaren Arbeitsschritten: Alle 40 ms wird der Wert gesetzt, wenn er sichtbar ist. Ob er sichtbar ist oder nicht, wird anderer Stelle entschieden. Der erwähnte Überhang liegt also darin, dass sichtbare, bereits gesetzte Werte erneut auf den gleichen Wert gesetzt werden und nicht vorhandene Werte auf Vorhandensein geprüft werden. Es handelt sich in dieser Klasse im schlimmsten Fall um elf Operationen, die alle 40 ms zu viel ausgeführt werden, also 275 Befehle pro Sekunde zu viel. In Zeiten von Gigahertz-Prozessoren in handelsüblichen Personalcomputern ist das ein verschwindend geringer Anteil und daher aus Sicht des Autors ein guter Kompromiss im Vergleich zum Aufwand, den die Erstellung der Klassen mit sich gebracht hätte.

```
    private function onEnterFrame(event : Event) : void {
        // inputA
        if (_bool.inputAB.bitA) { _bool.inputAB.bitA.bit_1.visible = _valueInputAB_BitA; }
        // inputB
        if (_bool.inputAB.bitR) { _bool.inputAB.bitR.bit_1.visible = _valueInputAB_BitB;  }
        // bit left
        if (_bool.out.bitL) {_bool.out.bitL.bit_1.visible = _valueInputAB_BitA && _valueInputAB_BitB; }
        // bit middle
        // also visible when only one line is visible, so lets see what operation is going on
        if (_bool.out.bitM) {
            if (_model.state == State.AND) {
                _bool.out.bitM.bit_1.visible = _valueInputAB_BitA && _valueInputAB_BitB;
            } else if (_model.state == State.OR) {
                _bool.out.bitM.bit_1.visible = _valueInputAB_BitA || _valueInputAB_BitB;
                } else if (_model.state == State.XOR) {
                    _bool.out.bitM.bit_1.visible = xor(_valueInputAB_BitA, _valueInputAB_BitB);
                }
        }
        // bit right
        if (_bool.out.bitR) {_bool.out.bitR.bit_1.visible = xor(_valueInputAB_BitA, _valueInputAB_BitB); }
    }
```

Drei der vier zuvor genannten Komponenten *feuern*[68] Events beim Erreichen eines relevanten (Zwischen-)Ziels. Diese Events werden benötigt, um der nachfolgenden Komponente Bescheid zu geben, dass diese den Wert übernimmt und ihrerseits mit der Animation fortfährt. In diesen Funktionen, die beim Eintreten des Events automatisch vom System aufgerufen werden, werden Statusvariablen gesetzt, die den geänderten Zustand festhalten. Danach rufen diese Funktionen die weiter unten beschriebene Funktion *checkState()* auf, da diese anhand der Statusänderung ähnlich einem *Schaltwerk* arbeitet. Die Funktion *inputABDone()* wird aufgerufen, sobald die Input-Werte auf den Leitungen bei den Gattern angekommen sind. *outZeroDone()* kommt zur Ausführung, wenn der animierte Wert auf der Ausgangsleitung des Gatters (oder des nachgeschaltenen Multiplexers) bei dem Kreuzungspunkt zur Zero Flag bildenden Leitung angelangt ist. *outDone()* ist die Funktion, die bei Erreichen des Ausgangswertes und *zeroFlagDone()* die Funktion, die bei Erreichen der Zero Flag Leitung aufgerufen wird.

[68] *Feuern* ist die Bezeichnung für das Benachrichtigen aller registrierten Objekte eines Events.

```
private function inputABDone(event : Event) : void {
    _readyInputAB = true;
    _bool.inputAB.stop();
    checkState();
}

private function outZeroDone(event : Event) : void {
    _readyOutZero = true;
    checkState();
}

private function outDone(event : Event) : void {
    _readyOut = true;
    checkState();
}

private function zeroFlagDone(event : Event) : void {
    _readyZeroFlag = true;
    checkState();
}
```

Die folgende Funktion bildet den Zustandsautomaten ab. checkState() hat vier Zustände:

1. Im Anfangszustand **S00_SETUP** werden alle Grafiken zurückgesetzt, Animationen beendet, die Eingangswerte aus dem Model abgegriffen und alle internen Zustandsvariablen auf einen Anfangswert gesetzt.

```
protected function checkState() : void {
    if (_localState == S00_SETUP) {
        // ================================================================================
        // Basics
        _valueInputAB_BitA = _model.getBinInput(0, _number);
        _valueInputAB_BitB = _model.getBinInput(1, _number);
        // ================================================================================
        // InputAB
        // weak references for garbage collection:
        if (_model.controlLines) {
            if (_model.state == State.AND) { _prefixInputAB = DETAIL_AND;
            } else if (_model.state == State.OR) { _prefixInputAB = DETAIL_OR;
            } else { _prefixInputAB = DETAIL_XOR; }
        } else {_prefixInputAB = ANDORXOR;}
        _bool.inputAB.gotoAndStop(_prefixInputAB);
        _bool.inputAB.addEventListener(DONE, inputABDone, false, 0, true);
        _readyInputAB = false;
        // ================================================================================
        // boolBoxes
        if (_model.controlLines) {
            // details
            if (_model.state == State.AND) { _prefixBoolBoxes = DETAIL_AND;
            } else if (_model.state == State.OR) { _prefixBoolBoxes = DETAIL_OR;
            } else { _prefixBoolBoxes = DETAIL_XOR; }
        } else {
            // simple
            if (_model.state == State.AND) { _prefixBoolBoxes = SIMPLE_AND;
            } else if (_model.state == State.OR) { _prefixBoolBoxes = SIMPLE_OR;
            } else { _prefixBoolBoxes = SIMPLE_XOR; }
        }
        _bool.boolBoxes.gotoAndStop(_prefixBoolBoxes);
        // ================================================================================
        // Out
        if (_model.controlLines) {
            // details
            if (_model.state == State.AND) {
                _valueOut = _valueInputAB_BitA && _valueInputAB_BitB;
                _prefixOut = DETAIL_AND;
            } else if (_model.state == State.OR) {
                _valueOut = _valueInputAB_BitA || _valueInputAB_BitB;
                _prefixOut = DETAIL_OR;
            } else {
                _valueOut = xor(_valueInputAB_BitA, _valueInputAB_BitB);
                _prefixOut = DETAIL_XOR;
            }
        } else {
            // simple
            if (_model.state == State.AND) {
                _valueOut = _valueInputAB_BitA && _valueInputAB_BitB;
            } else if (_model.state == State.OR) {
                _valueOut = _valueInputAB_BitA || _valueInputAB_BitB;
            } else {
                _valueOut = xor(_valueInputAB_BitA, _valueInputAB_BitB);
            }
            _prefixOut = ANDORXOR;
        }
```

```
        _startedOut = false;
        _readyOut = false;
        _readyOutZero = false;
        _bool.out.gotoAndStop(_prefixOut);
        _bool.out.addEventListener(DONE, outDone, false, 0, true);
        _bool.out.addEventListener(ZERO_DONE, outZeroDone, false, 0, true);
        // ================================================================================
        // zeroflag (gets value from sout)
        if (_model.flags) {
            _prefixZeroflag = "BIT";
            _zeroFlag.visible = true;
            _zeroFlag.gotoAndStop(_prefixZeroflag);
            _zeroFlag.addEventListener(DONE, zeroFlagDone, false, 0, true);
        } else {
            _zeroFlag.visible = false;
        }
        _startedZeroFlag = false;
        _readyZeroFlag = false;
        _localState = S10_READY;
    }
```

2. Im Zustand **S10_READY** befindet sich der Automat, wenn das Setup beendet ist oder nach Betätigung der Pausetaste. Hier wird keine Aktion durchgeführt

3. **S20_PLAY** ist der Zustand, wenn die Animation durchgeführt wird. Anhand der durch Events gesetzten Variablen kann das Verhalten der Animation gesteuert werden. Von allen Animationen, die aufeinander folgen, gibt es jeweils zwei Variablen: Eine beginnend mit *_started* (die Animation wurde gestartet) und eine beginned mit *_ready* (die Animation ist beendet). Die Variablen enden jeweils mit einer namentlichen Referenz auf die Animation, also zum Beispiel *_startedOut* und *_readyOut* für die Animation der Ausgangsleitung.

```
    if (_localState == S10_READY) {
    }
    if (_localState == S20_PLAY) {
        if (_readyInputAB) {
            if (!_startedOut) {
                if (_model.state == State.AND) {
                    _valueOut = _valueInputAB_BitA && _valueInputAB_BitB;
                } else if (_model.state == State.OR) {
                    _valueOut = _valueInputAB_BitA || _valueInputAB_BitB;
                } else {
                    _valueOut = xor(_valueInputAB_BitA, _valueInputAB_BitB);
                }
                _bool.out.gotoAndPlay(_prefixOut + getSpeedString());
                _startedOut = true;
            } else {
                if (!_startedZeroFlag && _readyOutZero) {
                    _zeroFlag.gotoAndPlay(_prefixZeroflag + getSpeedString());
                    _startedZeroFlag = true;
                } else {
                    if (_startedZeroFlag && (!_readyZeroFlag)) {
                        _zeroFlag.play();
                    }
                }
                if (_readyOut) {
                    _model.setBinOutput(_number, _valueOut);
                    _localState = S30_DONE;
                }
            }
        }
    }
```

4. Der Endzustand ist **S30_DONE** und hat keine Funktionalität. Dieser Zustand wird nach Abschluss der letzten Animation in *S20_PLAY* automatisch erreicht.

```
    if (_localState == S30_DONE) {
    }
}
```

Diese Funktion hat die Aufgabe, bei Eingaben die Animation zu stoppen und die Neukonfiguration einzuleiten. Sie fragt dabei ab, ob es sich um eine der von dieser Klasse behandelten Operationen handelt (AND, OR und XOR) und reagiert nur in diesem Fall auf die Benutzereingabe. In jedem anderen Fall wird der Zustand *S00_SETUP* erzwungen, sofern dieser nicht der aktuelle Zustand der Klasse ist.

```
        override protected function redraw(event : Event) : void {
            if ((_model.state == State.AND) || (_model.state == State.OR) ||(_model.state == State.XOR)) {
                _localState = S00_SETUP;
                checkState();
            } else {
                if (_localState != S00_SETUP) {
                    _localState = S00_SETUP;
                    checkState();
                }
            }
        }
```

Diese Funktion wird aufgerufen, sobald der Abspielmodus geändert wird (Stopp, langsam, schnell, Schritt oder Pause). Im Falle der Pause wird hier darauf geachtet, dass alle laufenden Animationen korrekt angehalten und beim Beenden wieder fortgesetzt werden. Man könnte diese Funktionalität auch in der Zustandsautomaten-Funktion checkState() unterbringen.

```
        override protected function newMode(event : Event) : void {
            if ((_model.state == State.AND) || (_model.state == State.OR) ||(_model.state == State.XOR)) {
                switch (_model.playmode.playmode) {
                    case Playmode.PLAY :
                        if (_model.playmode.oldPlaymode == Playmode.STOP) {
                            // start new animation
                            _bool.inputAB.gotoAndPlay(_prefixInputAB + getSpeedString());
                        } else {
                            // resume after break
                            if (!_readyInputAB) _bool.inputAB.play();
                            if (!_readyOut) _bool.out.play();
                            if ((!_readyZeroFlag) && _zeroFlag.visible) _zeroFlag.play();
                        }
                        _localState = S20_PLAY;
                        checkState();
                        break;

                    case Playmode.PAUSE :
                        if (_bool.inputAB) _bool.inputAB.stop();
                        if (_zeroFlag.visible) _zeroFlag.stop();
                        _localState = S10_READY;
                        checkState();
                        break;

                    case Playmode.STOP :
                        _localState = S00_SETUP;
                        checkState();
                        break;
                }
            }
        }
    }
}
```

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

WISSENSCHAFTLICHE LITERATUR

[BAUE1997] BAUER Friedrich: Schulungsboard auf Basis des Mikroprozessors Z80, Diplomarbeit, Technische Universität Wien, 1997

[BAUE2008] BAUER Friedrich, Skriptum zur Vorlesung und Übung Digitale Systeme (LVA-Nr. 384.046 und 384.047), Wien 2008

[BROT2006] BROTHANEK, Sonja / NOWACEK, Manuela: Lehrpfade und Erlebniswelten in Theorie und Praxis / Eine analytische Betrachtung in- und ausländischer Beispiele; Diplomarbeit, Universität f. Bodenkultur, Wien, 2006

[CARP2000] Computer Systems Organization and Architecture, John D. Carpinelli, Addison Wesley, ISBN-13: 978-0201612530, 30. Oktober 2000

[DIET2003] DIETER, Sventje / WIESNER, André: E-Learning / Einführung und Überblick; Vortragsunterlage im Rahmen des 11. AIK-Symposium „E-Learning – Strategien für die Unternehmenspraxis“, Universität Karlsruhe, Karlsruhe, 2003,
online: http://www.aifb.uni-karlsruhe.de/AIK/veranstaltungen/aik11/presentations/aifb.pdf

[ELEA2001] Handbuch E-Learning, Andreas Hohenstein, Karl Wilbers (Hrsg.), Stand Oktober 2005

[LANG1997] LANG Marco: Simulation für den Modellcomputer MC8, Institut für Computertechnik der TU Wien, Diplomarbeit, 1997

[LIND2004] LINDNER, Helmut / BRAUER, Harry / LEHMANN, Constans: Taschenbuch der Elektrotechnik und Elektronik; Fachbuchverlag Leipzig, Leipzig 2004[8]

[KIEN2005] KIENLE, Andrea: E-Learning: Konzepte, Beispiele und Nutzen für Auszubildende und Studierende; Fraunhofer Institut, Integrierte Publikations- und Informationssysteme; Vortragsunterlage im Rahmen der GI-Jahrestagung, Bonn, 2005, online: http://www.informatik2005.de/fileadmin/dateien_redaktion_2005/downloads/VF/SP_VF_Kienle.pdf

[PROJ2008] PROJEKTZENTRUM LEHRENTWICKLUNG, Porzellangasse 33a, A-1090 Wien online: http://elearningcenter.univie.ac.at

[REGL2005] REGLIN, Thomas: Blended-Learning-Angebote richtig vermarkten – Ergebnisse einer qualitativen Analyse von Leistungsversprechungen; In: HOHENSTEIN, Andreas; WILBERS, Karl (Hrsg.): Handbuch E-Learning, 13. Ergänzungslieferung August 2005, Deutscher Wirtschaftsdienst

[WENN2008] WENNINGER, Joseph Gernot Otto: Ein selbsterklärender, visueller Maschinencode-Assembler für die universitäre Lehre

INTERNETREFERENZEN

[ADOB2008A] Adobe Flash Player Version Penetration
http://www.adobe.com/products/player_census/flashplayer/version_penetration.html

[BAYE2008] Bayerische Staatsbibliothek, Fachkoordination Geschichte, 06.09.2008,
http://www.historicum.net/news/notizendetails/ca/d3177a5136/browse-news/1/news/714/unser-titelb/

[ELEA2008A] PROJEKTZENTRUM LEHRENTWICKLUNG,
http://elearningcenter.univie.ac.at/index.php?id=535 [2008-08-25]

[KELS2008] Kelso High School, CPU-Simulator,
http://www.kelso.scotborders.sch.uk/departments/computing/resources/flash/cpu.swf
[2008-08-27]

[MEYE2008A] MEYERS ONLINE LEXIKON, Bibliographisches Institut & F.A. Brockhaus AG,
Dudenstraße 6, D-68167 Mannheim, online:
http://www.lexikon.meyers.de/meyers/E-Learning [2008-08-27]

[MEYE2008B] Meyers Online Lexikon, http://lexikon.meyers.de/E-Learning [2008-08-27]

[PEAR2000] Website zu [CARP2000],
http://media.pearsoncmg.com/aw/aw_carpinel_compsys_1/rscpu/web.html [2008-08-27]

[SAUS1995] Technisches Gymnasium der BBS Winsen, Betreuer Stephan Sausel, Fachlehrer im Bereich Elektrotechnik, http://old.bbs-winsen.de/projekte/wilusim/index.html [2008-08-27]

[TUWE2007] ZIDline Juni 2007: E-Learning mit TUWEL
http://www.zid.tuwien.ac.at/zidline/zl16/tuwel/ [2008-08-27]

[VIPT2006] Virtueller Info 2 Prozessor Simulator der Technischen Universität Berlin, Aufbau und Funktionsweise programmierbarer digitaler Systeme, Dozent Dr.-Ing. Thomas Flik,
Institut für Technische Informatik und Mikroelektronik,
Fachgebiet Rechnerorgansisation und Schaltwerksentwurf (ROSW)
http://aes.cs.tu-berlin.de/info2/ss06/ [2008-08-27]

[WIKI2008A] WIKIPEDIA ein Angebot von Wikimedia Deutschland – Gesellschaft zur Förderung Freien Wissens E.V., Bolongarostraße 103, D-65929 Frankfurt,
http://de.wikipedia.org/wiki/Lernen [2008-08-27]

[WIKI2008B] WIKIPEDIA ein Angebot von Wikimedia Deutschland – Gesellschaft zur Förderung Freien Wissens E.V., Bolongarostraße 103, D-65929 Frankfurt,
http://de.wikipedia.org/wiki/E-Learning [2008-08-25].

[WIKI2008C] WIKIPEDIA ein Angebot von Wikimedia Deutschland – Gesellschaft zur Förderung Freien Wissens E.V., Bolongarostraße 103, D-65929 Frankfurt
http://de.wikipedia.org/wiki/Multiplexer [2008-08-28]