Hessische Hochschulen

Philipps-Universität Marburg

Der CPU-Simulator MikroSim

H. Peter Gumm, Martin Perner
FB Mathematik, FG Informatik, Philipps-Universität Marburg
Hans-Meerwein-Straße (Lahnberge)
D-35032 Marburg
Telefon: (0 64 21) 28-54 47, Fax -54 19

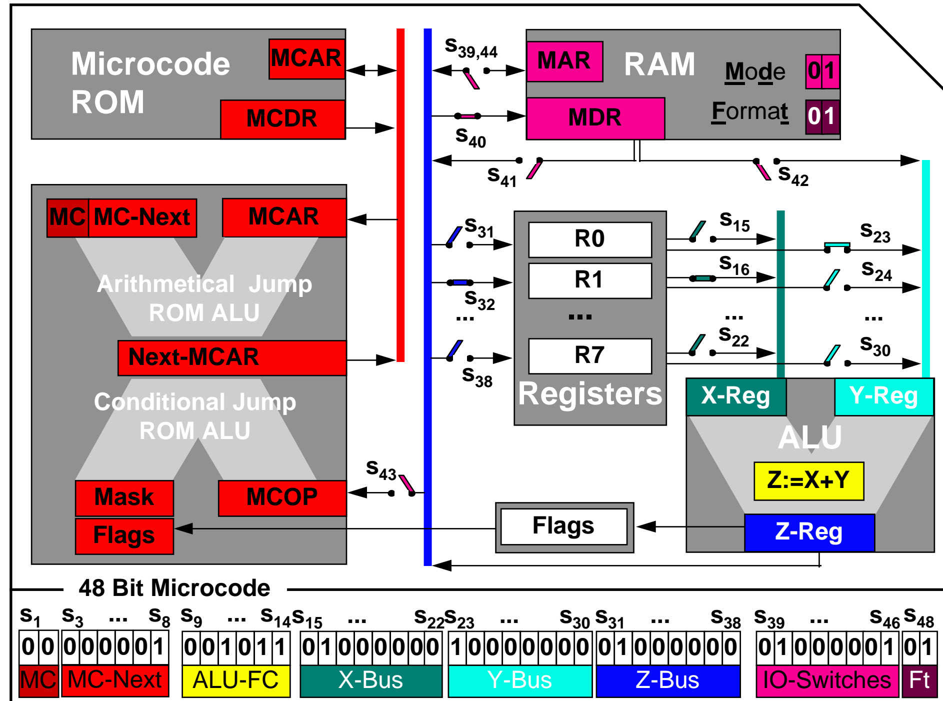
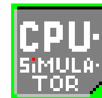


Abb. 1: Das Modell einer CPU, das von MikroSim simuliert wird.

Der CPU-Simulator MikroSim ist eine Windows-Anwendung, mit der sich die Funktionsweise eines Mikroprozessors (CPU) simulieren läßt. Der Benutzer kann auf verschiedenen Abstraktionsebenen verfolgen, wie Mikrobefehle als Codierung von Schalterstellungen zu verstehen sind, wie sie in Takte zerlegt ausgeführt werden, wie im ROM Folgen von Mikrobefehlen zu Mikroprogrammen gruppiert werden und wie schließlich ein universelles Mikroprogramm konstruiert werden kann, das als Interpreter für Maschinensprache im RAM abgelegte Maschinenprogramme ausführt. So wird die Spannweite von den Grundbausteinen ALU und Speicher bis zum programmierbaren Universal-Rechner versteh- und erfahrbar.



Während ALU und Speicher in ihrer Funktionalität vorgegeben sind, läßt sich das ROM beliebig gestalten. Dadurch kann der Benutzer seine eigene Maschinensprache entwerfen, als Mikroprogramm im ROM implementieren und in beliebig feinen Auflösungen (Phasen, Takte oder Maschinenzyklen) nachvollziehen und testen. Die Bedienung des Programms ist denkbar einfach und intuitiv. Auf dem Windows-Bildschirm entstehen die graphischen Abbilder der in der jeweiligen Abstraktionsebene benötigten Bauteile. Durch Anklicken entsprechender Buttons öffnen sie sich und gestatten die gezielte Modifikation von Speicher-Inhalten, Funktionscodes oder Mikroprogrammen.

Einleitung

In der Vorlesung Informatik III im Grundstudium Informatik sollen die Studenten die Spannweite von den physikalischen Bausteinen elektronischer Rechner bis hin zu der abstrakten Sicht eines intelligenten Desktops exemplarisch, aber lückenlos nachvollziehen können. Auf dem Weg vom Transistor zum Desktop gibt es eine Kette von Abstraktionsebenen, auf denen man verweilen kann. Am unteren Ende entstehen zunächst aus den Transistoren logische Schaltglieder; die Boolesche Algebra erklärt deren Zusammenwirken und die gezielte Konstruktion gewünschter Schaltungen (Gattern) aus den Grundbausteinen. Die komplexeste so gewonnene Schaltung ist eine ALU (arithmetische logische Einheit). Durch einfache Rückkopplung entstehen einfache Schaltungen mit "Gedächtnis", zunächst ein Flip-Flop, dann daraus ein 1-Bit-Speicher und durch geeignete Kombination in Verbindung mit einem Codierer/Dekodierer ein linearer Speicher. Damit sind die wichtigsten Bausteine einer CPU eingeführt. Die Abstände zwischen den einzelnen Abstraktionsebenen: Transistor - Logikglieder - Gatter - Flip-Flop - 1-Bit-Speicher - Register - RAM sind jeweils einfach überbrückbar. Zudem ist jede Abstraktionsebene in dem Sinne vollständig, daß die nächsthöhere Ebene gänzlich in den Konzepten der gegenwärtigen Ebene ausdrückbar und erklärbar ist und keinen Durchgriff zu niedrigeren Ebenen erfordert.

Als nächster Schritt ist der Aufbau eines Mikroprozessors (CPU) aus den bereitgestellten Bauteilen - ALU, Register, Busse, Speicher zu beschreiben. Dies ist der deutlich komplexeste Schritt und es gibt keine klaren Abstraktionsebenen, auf denen man verweilen kann. Wesentliche und problematische Designentscheidungen erschließen sich erst bei intensiver Beschäftigung mit Systemdetails. Dies erfordert mehr als ein theoretisches Umgehen mit den Konzepten, eine experimentelle Beschäftigung mit einem Mikroprozessor ist erforderlich. Diese experimentelle Beschäftigung kann heute am besten mit einem guten Simulationsprogramm stattfinden. Die Vorteile sind vielfältig - der Umgang erfordert keine besondere Geschicklichkeit mit elektronischen Bauteilen. Dadurch kann die Aufmerksamkeit voll auf die logische Funktion gerichtet werden - der Simulator ist wandlungsfähiger als ein Stück Hardware, und es können alternative Funktionskonzepte erprobt werden. Außerdem ist ein Simulationsprogramm auf einer weit verbreiteten Plattform (MS-Windows) jedem Studenten unmittelbar zugänglich. Mit diesen Zielsetzungen entstand der CPU-Simulator MikroSim in der Fachgruppe Informatik der Philipps-Universität Marburg.

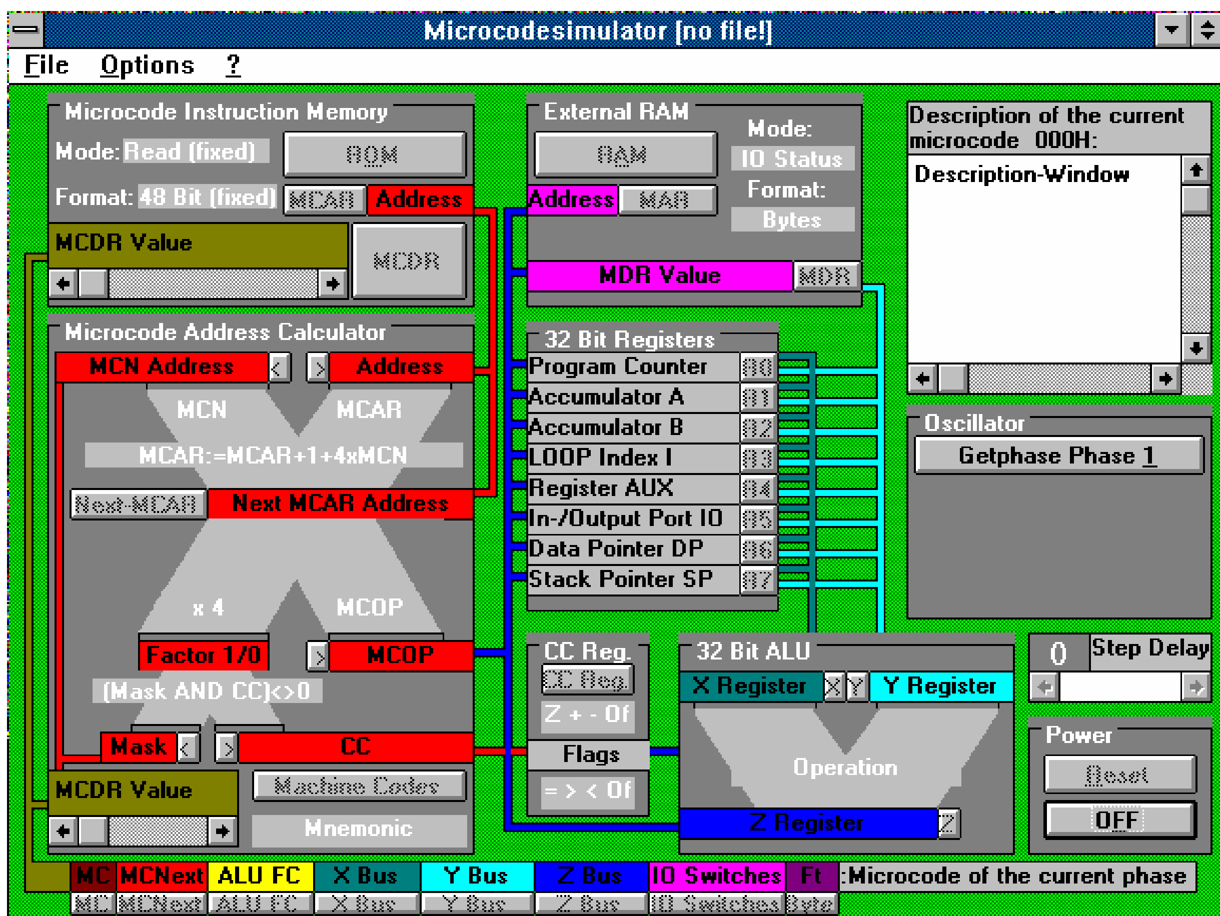


Abb. 2: MikroSim in dem Erkundungsmodus. Die Register tragen noch die erklärenden, vorgegebenen Bezeichnungen. Mit dem Menüpunkt Optionen läßt sich die CPU stufenweise zusammensetzen.

Das didaktische Programm-Konzept in vier Lernschritten

Grundlage bildete ein Modell (siehe Abb. 1) einer mikroprogrammierten CPU wie sie in der Vorlesung Informatik III behandelt wird und in dem Lehrbuch [GS] eingeführt wird. Ein erster Versuch, Teile dieses Modells in einem Turbo-Pascal Programm zu simulieren, stammte von H. Gasiorowski. Im Sommer 1992 wurde dann der Simulator von M. Perner völlig neu konzipiert und eine erste Version (siehe Abb. 2 und 11) in Visual Basic implementiert. Ausgerichtet an den Bedürfnissen der Vorlesung veränderte sich dann das Konzept bis zur heute vorliegenden Version, die ein fester Bestandteil der Ausbildung in Rechnerarchitektur geworden ist. Das gegenwärtige Konzept unterstützt ein Verständnis der CPU in vier Schritten, die wir zunächst überblicksweise erläutern:

1. Lernschritt

Im ersten Schritt wird das Zusammenspiel zwischen ALU, Registern und Bussen erklärt. Alle übrigen Bestandteile der CPU sind dabei im Simulator ausgeblendet. Die Funktionsweise dieses Teils der CPU ergibt sich aus der Stellung von Schaltern, die in den einzelnen Phasen eines Taktes die Verbindung zwischen Registern, Bussen und ALU herstellen oder unterbrechen. Eine Schalterstellung läßt sich als Teil eines Maschinenwortes auffassen. Dieses wird ergänzt um den Funktionscode, der in der ALU eingestellt ist (siehe Abb. 3). So lassen sich bereits einfache Operationen ausdrücken und simulieren, wie das Addieren von Registerinhalten (R1 := R1+R0) oder das Erhöhen eines Registerwertes (R0 := R0+1). Wir verwenden hier eine Pseudo-Notation um die Semantik einfacher Operationen zu erklären. Die Simulation erfolgt nach Belieben phasenweise oder im Drei-Phasen-Takt (Hol-, Rechen- und Bring-Phase).

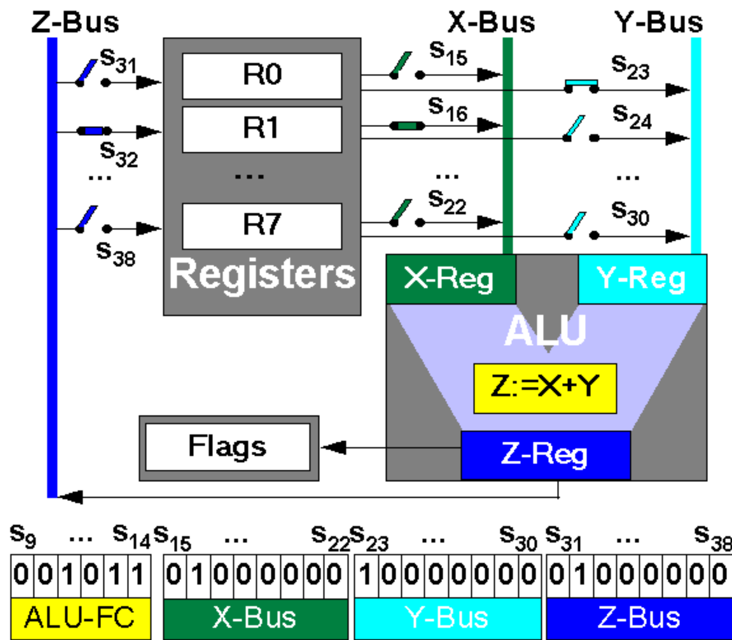


Abb. 3: Die Recheneinheit der CPU: ALU, Register und die Datenbusse. Der gezeigte Mikrobefehl codiert die Anweisung R1:=R1+R0.

2. Lernschritt

Im zweiten Schritt wird ein RAM-Speicher hinzugefügt (siehe Abb. 4). Das Maschinenwort verlängert sich um die Bits, die Schaltern zwischen Bussen und Adress- bzw. Datenregistern entsprechen. Zusätzliche Bits drücken den Speichermodus (Schreiben, Lesen, Warten) und das Format der bewegten Datenblöcke aus. In dieser Simulationsebene lassen sich nun auch Datenbewegungen zwischen Registern und Speicher

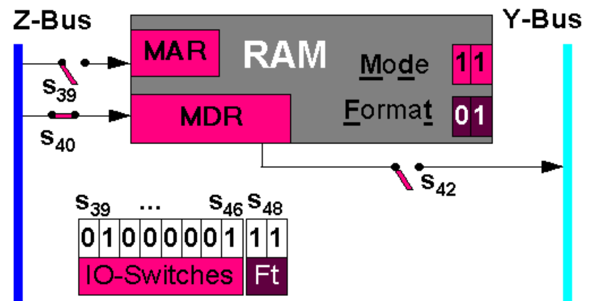


Abb. 4: Der Hauptspeicher mit der entsprechenden Erweiterung des Maschinenwortes. Einige Schalter stellen den Betriebsmodus dieser CPU-Komponente ein.

studieren. Bereits hier können Designentscheidungen diskutiert werden: Welche Buszugänge sind für die Speicherregister nötig oder sinnvoll, welcher Schalter sollte in welcher Phase aktiv sein, so daß sich häufig benötigte Operationen in möglichst wenigen Takten ausführen lassen? Mit einem RAM-Editor (siehe Abb. 8) läßt sich der Speicherinhalt extern einstellen, oder die Veränderung als Folge einer Maschinenoperation überprüfen.

3. Lernschritt

Im dritten Schritt wird der ROM-Speicher hinzugefügt, in dem sich Folgen von Maschinenworten ablegen lassen (siehe Abb. 5). Die Abarbeitungsreihenfolge ist zunächst sequentiell, dann werden Sprünge zu festen Zielen bzw. bedingte Sprünge, abhängig von dem Flag-Register der CPU behandelt. Da das ROM nur Maschinenworte enthält, ist der ROM-Editor auf solche

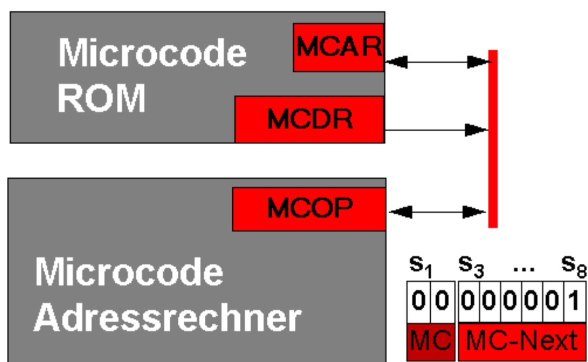


Abb. 5: Das Mikrocode ROM mit dem wichtigen MCOP-Register des Mikrocodeadressrechners.

spezialisiert (siehe Abb. 9). Ein Maschinenwort erscheint im ROM-Editor nicht nur als eine Folge von Bits, sondern als eine Kette von Bitgruppen, von denen jede einzelne Gruppe zusammen mit ihrer Bedeutung erscheint. So läßt sich auf dieser Ebene ein Maschinenbefehl menügesteuert zusammenstellen. Das Erstellen und Austesten von Mikroprogrammen ist einfacher kaum vorstellbar.

4. Lernschritt

Im vierten und letzten Schritt geht es um das Design einer Assemblersprache und um deren Implementierung. Im ROM wird jeder Assemblerbefehl durch ein kleines Mikroprogramm implementiert. Jedem Befehl wird eine Nummer (OpCode) zugeordnet, welche gleichzeitig den Ort des entsprechenden Mikroprogramms angibt (4*OpCode). Die Ausführung eines Assemblerprogramm im RAM führt dann zu der Ausführung der den Befehlen entsprechenden Mikroroutinen. Die Steuerung wird von einem am Anfang des ROM plazierten Interpreter ausgeführt. Der Entwurf der Assemblersprache, ihre Implementierung sowie der zugehörige Interpreter ist offen. Da der ROM-Inhalt als Datei des Simulators geladen und gespeichert werden kann, kann der Benutzer verschiedene Alternativen austesten oder aber eine vorhandene Implementierung laden. Auch auf dieser Ebene wird der Benutzer durch geeignete Maschinensprache-Editoren unterstützt, mit denen man den Überblick über bereits implementierte Befehle und deren Wirkungsweise behält (siehe Abb. 10).

Arbeitsweise und Aufbau des CPU-Simulators

Register, ALU und Datenbusse

Wir gehen nun näher auf die oben skizzierten Lernschritte ein und deuten an, wie sie sich in dem Simulator darstellen. Im ersten Schritt sind nur die folgenden Bauteile sichtbar (siehe Abb. 3):

- die **ALU** mit den Buszugangsregistern X und Y, dem Ergebnisregister Z, dem Flag-Register und dem Funktionscoderegister. Die eingestellte ALU-Operation erscheint in Pseudocode auf der Vorderseite. Die in den X- und Y-Registern liegenden Werte werden verknüpft und das Ergebnis im Z-Register bereitgestellt.
- die **Busse: X-Bus, Y-Bus und Z-Bus** mit den Anschlüssen an die gleichnamigen Register der ALU.
- eine Gruppe von **8 Registern R0, ..., R7** mit Eingang vom Z-Bus und Ausgängen zu den X- und Y-Bussen. Register und Busse sind jeweils 32 Bit breit. Die Register besitzen anfangs einen zufälligen Inhalt. Über einen "Button" öffnet man das Register, um einen beliebigen Inhalt einzu-

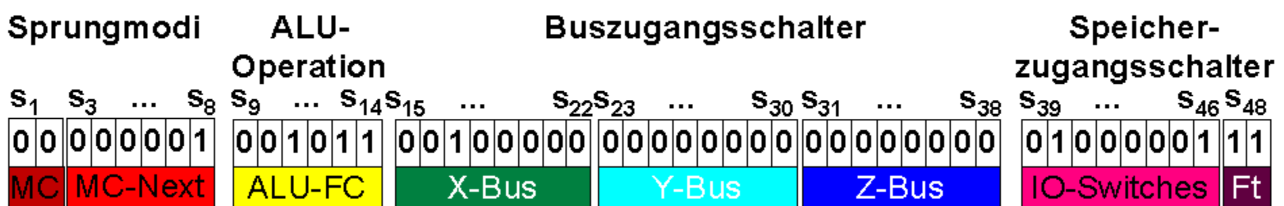


Abb. 6: Dieser komplette 48-Bit Mikrocode führt den letzten Teil der Anweisung [R1]:=[R1]+R2 aus. Dabei bestimmt der ALU-Funktionscode die Operation Z := X+Y. Das Y-Register der ALU hat bereits den Wert [R1] und X erhält die Konstante aus R2 über den X-Bus. Das Ergebnis wird in das RAM als 4Byte-Wert zurückgeschrieben, da die beiden letzten IO-Switches das Schreiben ins RAM und Ft=11 das Speicher-Format angibt.

tragen (siehe Abb. 7).

Am unteren Rand des Bildschirm ist das gegenwärtige Maschinenwort zu sehen. Die einzelnen Bitgruppen, stellen die Zugänge der Register R0 bis R7 zu den Bussen X, Y bzw. von dem Bus Z dar. Die Entsprechung wird durch die einheitliche Farbgebung unterstützt. Die ersten 6 Bit, die mit ALU-FC bezeichnet sind, kodieren eine der 64 ALU-Funktionen.

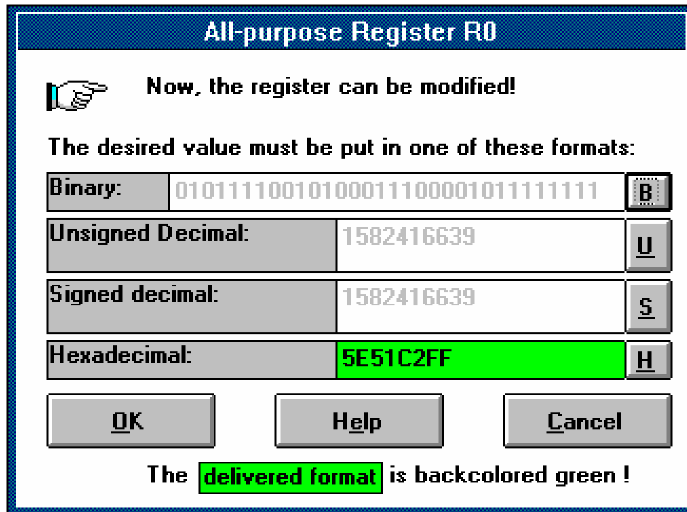


Abb. 7: Das Editierfenster ermöglicht das Eingeben und Ablesen des Registerinhalts des gewählten Registers (hier R0) in verschiedenen Zahlenformaten.

Um beispielsweise die Assembleroperation ADD R1, R2 auszuführen, muß die ALU-Funktion ADD eingestellt werden (Code 001011). Außerdem werden die Schalter von R1 zum X-Bus und von R2 zum Y-Bus geöffnet. Damit das Ergebnis der Addition wieder nach R1 gelangen kann wird der Schalter vom Z-Bus zu R1 geöffnet. Der Mikrocode 001011 01000000 00100000 01000000 codiert damit die Operation ADD R1, R2. Die Ausführung geschieht in einem Takt. Jeder Takt ist wiederum in drei Phasen unterteilt:

- In der **Holphase** sind die gewünschten Schalter von den Registern zu X- bzw. Y-Bus geöffnet
- In der **Rechenphase** sind diese Schalter wieder geschlossen. Die ALU berechnet die eingestellte Operation und liefert das Ergebnis im Z-Register ab.
- In der dritten, der **Bringphase**, sind die Schalter vom Z-Bus zu den Registern aktiv, das Ergebnis der Berechnung kann wieder in die Register gelangen.

Die Simulation kann durch Knopfdruck Phase für Phase oder in kompletten Takten ablaufen.

Der Hauptspeicher - RAM

Im zweiten Schritt fügen wir den RAM-Speicher hinzu. Das Interface zum Speicher bilden die Register MAR (memory address register) und MDR (memory data register). Das MAR enthält also die Adresse des aktuellen Speicherplatzes, während das MDR als Durchgangsregister für die zu speichernden bzw. im RAM gelesenen Daten dient. Die Anbindung des MDR an die Busse geschieht wie bei den Registern Ri: Über den Z-Bus wird MDR mit den (zu speichernden Daten versorgt), über den Y-Bus werden die (gelesenen) Daten an die ALU weitergegeben (siehe Abb. 4). Auch die zeitliche Aufteilung der Speicheroperationen auf die Phasen eines Taktes - Lesen in Phase 1 und Schreiben in Phase 3 - entspricht der Vorgehensweise bei den Registern. Da das Adressregister MAR nur Adressen über den Z-Bus entgegennehmen muß, ist eine Verbindung zu den Operandenregistern der ALU überflüssig. Die Funktion des Speichers (Schreiben, Lesen oder Warten) sowie das Datenformat (1-, 2-, 4-Byte) erfordern jeweils zwei Bit im nun längeren Maschinenwort. Hinzu kommen noch die Schalter die den Zugang von MAR und MDR zu den Bussen regeln.

Als Beispiel betrachten wir die Aufgabe, zu einer Speicherzelle eine Konstante zu addieren. Die Adresse der Speicherzelle befinde sich in R1 und die Konstante in R2, also [R1]:=[R1]+R2. Wir benötigen zwei Takte. Zunächst muß der Inhalt von R1 nach MAR transportiert werden. Dazu stellen wir die ALU Operation Z := X (OpCode 000010) ein und öffnen den Schalter von R1 zum X-Bus und vom Z-Bus nach MAR. Der Speichermodus ist "Wartend", also inaktiv. Für den nächsten Takt stellen wir den Speichermodus

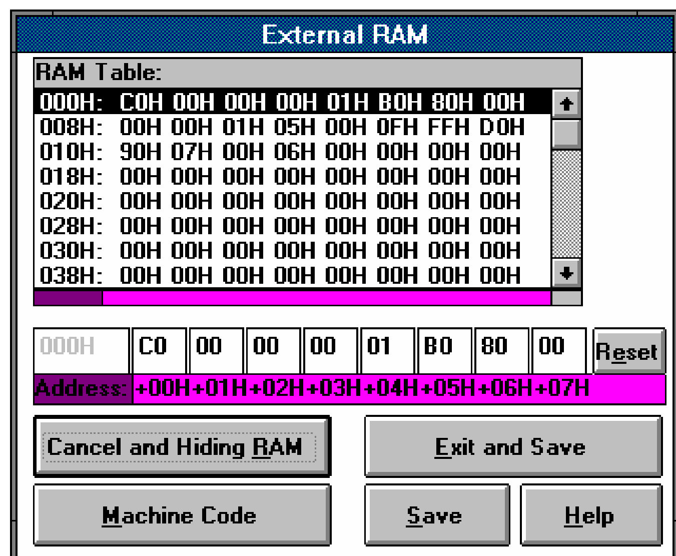


Abb. 8: Mit dem RAM-Editor kann man das externe RAM betrachten und modifizieren.

"Lesen" ein, so daß nach Phase 1 bereits der gelesene Wert in MDR vorliegt. Gleichzeitig sind die Wege MDR→Y und R2→X offen und die ALU-Funktion Z := X+Y eingestellt. In Phase 3 desselben Taktes fließt das Ergebnis bereits über den Z-Bus nach MDR und kann im Abschluß der Phase 3 gespeichert werden. Ist das Adreßregister bereits richtig gesetzt, so benötigt eine einfache arithmetische Operation im Speicher also nur einen Takt. Abb. 6 zeigt den dazugehörigen Mikrocode, der diese Operationen durchführt.

Der Festwertspeicher - ROM

In der nächsten Simulationsebene fügen wir das ROM hinzu. Es ist ein Speicher für Mikrocode, auf den über MCAR (micro code address register) und MCDR (micro code data register) zugegriffen werden kann. Wie das Akronym (ROM = read only memory) schon sagt, kann der Inhalt nur gelesen werden. Daher ist der Modus immer "Lesend" und das Format immer 48 Bit, nämlich die Größe eines Maschinenwortes. Auf Knopfdruck öffnet sich das ROM und ein spezieller ROM-Editor (siehe Abb. 9) gestattet die Modifikation später im Normalfall gerade einen Assemblerbefehl enthalten.

Die Ausführung der Mikrobefehle

Die im ROM abgelegten Mikrobefehle werden üblicherweise sequentiell ausgeführt, doch muß die Möglichkeit von Sprüngen vorgesehen werden. In den ersten beiden Bits eines Mikrobefehls wird die Art des Sprunges codiert:

- an eine **absolute Adresse**,
- um einen **festen Betrag vor- oder rückwärts**
- oder **an eine Adresse**, die in einem bisher noch nicht erwähnten Register MCOP steht. Letzterer Sprung kann auch bedingt (abhängig vom Inhalt des Flag-Registers der ALU) ausgeführt werden.

6 Bits des Mikrobefehls stellen, je nachdem ob es sich um einen absoluten oder um einen bedingten Sprung handelt, entweder eine Segmentadresse oder eine Maske dar, die über das Flag-Register gelegt wird.

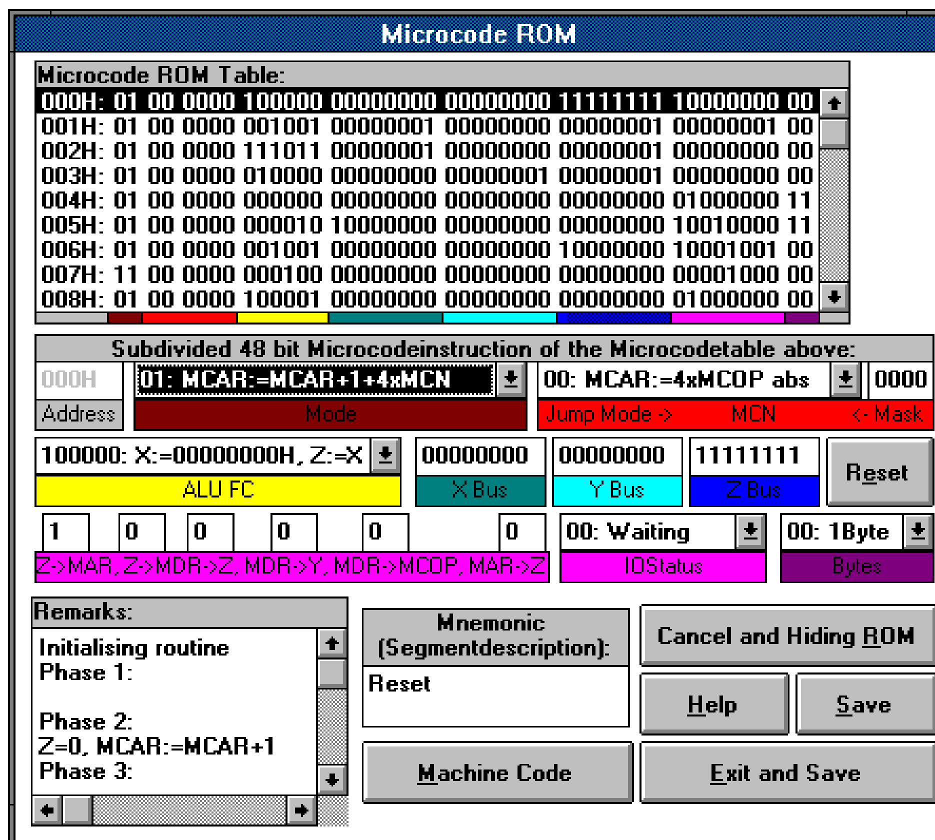


Abb. 9: Mit dem Mikrocode-ROM-Editor kann man spielend einfach 1024 Mikrobefehle verwalten und bitgruppenweise bearbeiten.

des Inhaltes, wobei jeder Mikrobefehl aus den konstituierenden Bitgruppen menü-gesteuert zusammengesetzt werden kann. Je 4 Mikrobefehle faßt man zu einem Segment zusammen. Ein Segment wird

Insgesamt ist ein Mikrobefehl damit bereits auf 48 Bit angewachsen. Die Sprungberechnung ist schon hinreichend kompliziert, so daß eine gesonderte Adressberechnungs-ALU damit befaßt ist, aus der aktuellen Mikrocodeadresse die Adresse des nächsten

Mikrocodes zu berechnen. Dies Berechnungseinheit ist schematisch in Abb. 5 und detailiert in Abb. 1 und 2 zu sehen. Abb. 6 faßt das Format eines kompletten Mikrobefehls zusammen. In dieser Simulationsstufe lassen sich jetzt also bereits beliebige Programme im ROM ablegen und ausführen.

Mit der Assembler- oder Maschinensprache schafft man schließlich ein Repertoire von abstrakteren Befehlen. Diese Befehle sind bereits stärker an Problemlösungen orientiert als an den Eigenheiten einer CPU. Der Programmierer soll sich nicht mehr um Datenwege, Phasen oder Speichermodi kümmern müssen. Interne Register wie MAR, MDR, MCOP etc. bleiben ihm verborgen. Aus seiner Sicht gibt es lediglich einen Speicher, Allzweckregister, Programmzähler, Stackpointer und einige wenige zusätzliche Register. Programme und Daten werden im RAM abgelegt. Der Benutzer des Simulators kann sich seine eigene Maschinensprache entwerfen und implementieren. Die Verwendung der Register R0, ..., R7, wie in Abb. 2 dargestellt ist, ist lediglich ein Vorschlag und bezieht sich auf eine als ROM-Datei beigelegte Implementierung mehrerer Maschinensprachebefehle.

Wir vereinbaren daher, daß das Register R0 als Programmzähler (PC) dienen soll, R1 und R2 als Akkumulatoren A und B. Weitere Register wie Stack Pointer, Data Pointer, Loop Index etc. werden auf R3 bis R7 verteilt. Jeder Befehl der Maschinensprache wird nun als kurze Mikrobefehlsroutine im ROM implementiert. Üblicherweise reicht ein Segment für einen Befehl aus. Dessen Nummer ist gleich dem OpCode des Befehls. Der Simulator verwendet ein ROM mit 1024 Adressen, also 256 Segmenten. Die ersten beiden Segmente enthalten dabei den Interpreter für Maschinencode die folgenden können zur Implementierung einer Maschinensprache mit maximal 254 Befehlen dienen. Der Interpreter ist erstaunlich einfach und kurz. Vier Mikrobefehle in Segment 0 dienen zur Initialisierung verschiedener Register und des Programmzählers. Der eigentliche Interpreter, auch als Load-Increment-Execute Zyklus (L-I-E) bekannt, befindet sich in Segment 1 und besteht aus den folgenden Schritten:

1. **Load**: Lese im RAM den OpCode, auf den der Programmzähler PC zeigt. Befördere den OpCode in MCOP
2. **Increment**: Erhöhe PC
3. Execute: Springe an die in MCOP befindliche Adresse des ROM, dort befindet sich das Mikroprogramm für den aktuellen Assemblerbefehl.

Dabei müssen die folgenden Eigenschaften (Invarianten) gewährleistet bleiben :

- Zu Beginn des L-I-E Zyklus zeigt der Programmzähler PC auf den nächsten Befehl im RAM
- Das einen Assemblerbefehl implementierende Mikroprogramm schließt mit einem Sprung an den Beginn des L-I-E Zyklus, also an Segment 1.

Betrachten wir beispielsweise den Assembler-Befehl ADD [A], B. Weisen wir ihm willkürlich OpCode 57 zu, so werden wir im ROM in Segment 57 die Mikro-Befehlsfolge ablegen, die den Inhalt von R1 in MAR befördert, den Speicher liest, zum Inhalt von MDR den Inhalt von R2 addiert, das Ergebnis nach MDR bringt, den Speicher auf "Schreiben" stellt und zu guter Letzt an Segment 1 springt. Für Assemblerbefehle, die ein Argument erwarten (etwa ADD A, <constant>) muß dieses direkt nach dem OpCode im RAM stehen. Im Mikrocode für den Befehl darf man aufgrund der erwähnten Invarianten davon ausgehen, daß PC bereits auf das Argument zeigt. Es muß allerdings dafür gesorgt werden, daß danach PC erhöht wird, damit die Invariante wieder erfüllt ist. Auch Sprünge (JMP) lassen sich leicht implementieren, es muß lediglich PC entsprechend gesetzt werden. Eine Komplikation zeigt sich allerdings bei der Implementierung bedingter Sprünge, etwa JNE (Jump if not equal zero). Das Flag-Register, wie es sich am Ende des letzten Befehles darstellte muß ausgewertet werden. Seither hat jedoch wieder der L-I-E Zyklus stattgefunden, so daß zu Beginn der JNE-Routine das

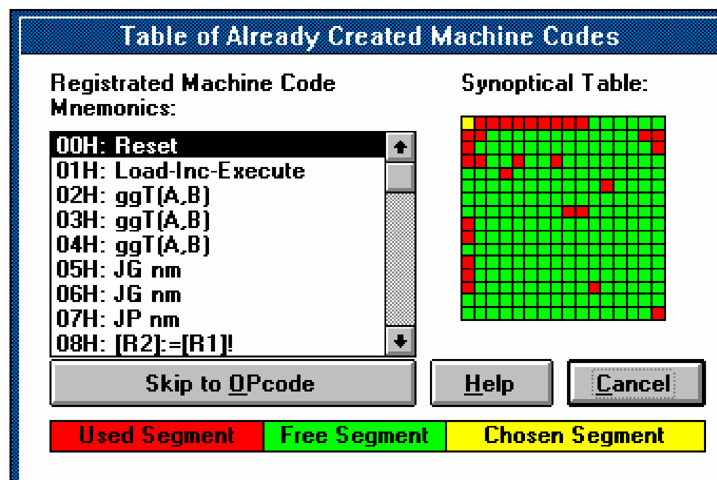


Abb. 10: Mit dem Editor für Maschinensprache hat man schnell den Überblick über die Verwendung der 256 Segmente.

ursprüngliche Flag-Register nicht mehr vorliegt. Verschiedene Lösungen dieses Problems sind denkbar, die entweder zu einer leichten Verkomplizierung der L-I-E-Zyklus führen, oder andererseits zu einer Implementierung dieses kurzen Zyklus allein durch ALU-Operationen, die keine Flags verändern.

Der Simulator wird mit großem Erfolg in den Anfängervorlesungen an der Universität Marburg eingesetzt. Studenten implementieren Mikroprogramme, entwerfen und implementieren ihre Maschinensprache. Im Vergleich mit früheren Vorlesungen, in denen der Stoff nur abstrakt diskutiert wurde, stellt sich ein deutlich tieferes Verständnis für die Problematik ein. Designentscheidungen werden versteh- und kritisierbar, Verbesserungsvorschläge geäußert und diskutiert. Die oben angeführte Schwierigkeit der Implementierung von bedingten Sprungbefehlen im Assembler wurde erst anhand der Arbeit mit dem Simulator erkannt. Das Programm MikroSim wurde in der Kategorie "Computer Science" mit dem European Academic Software Award 1994 ausgezeichnet.

Literatur

[GS] H. P. Gumm, M. Sommer : *Einführung in die Informatik*, Addison Wesley, 1994.

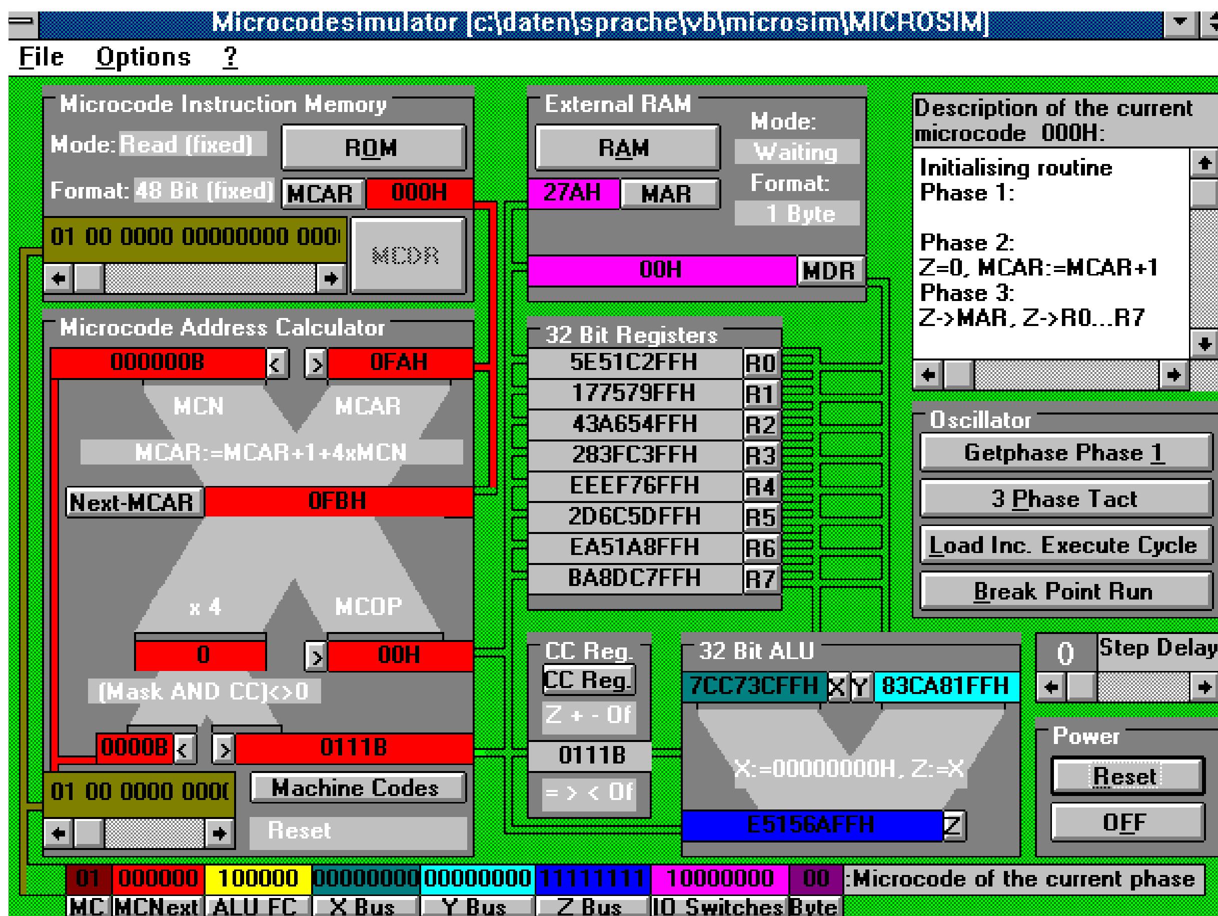


Abb. 11: MikroSim hat die Datei "Microsim" geladen, wurde initialisiert und befindet sich jetzt im Simulationsmodus. Die Abarbeitung der Maschinensprachebefehle im RAM kann nun entweder im Einzeltakt, 3-Phasentakt, LIE-Zyklus oder als Breakpoint-Ablauf betrachtet werden.