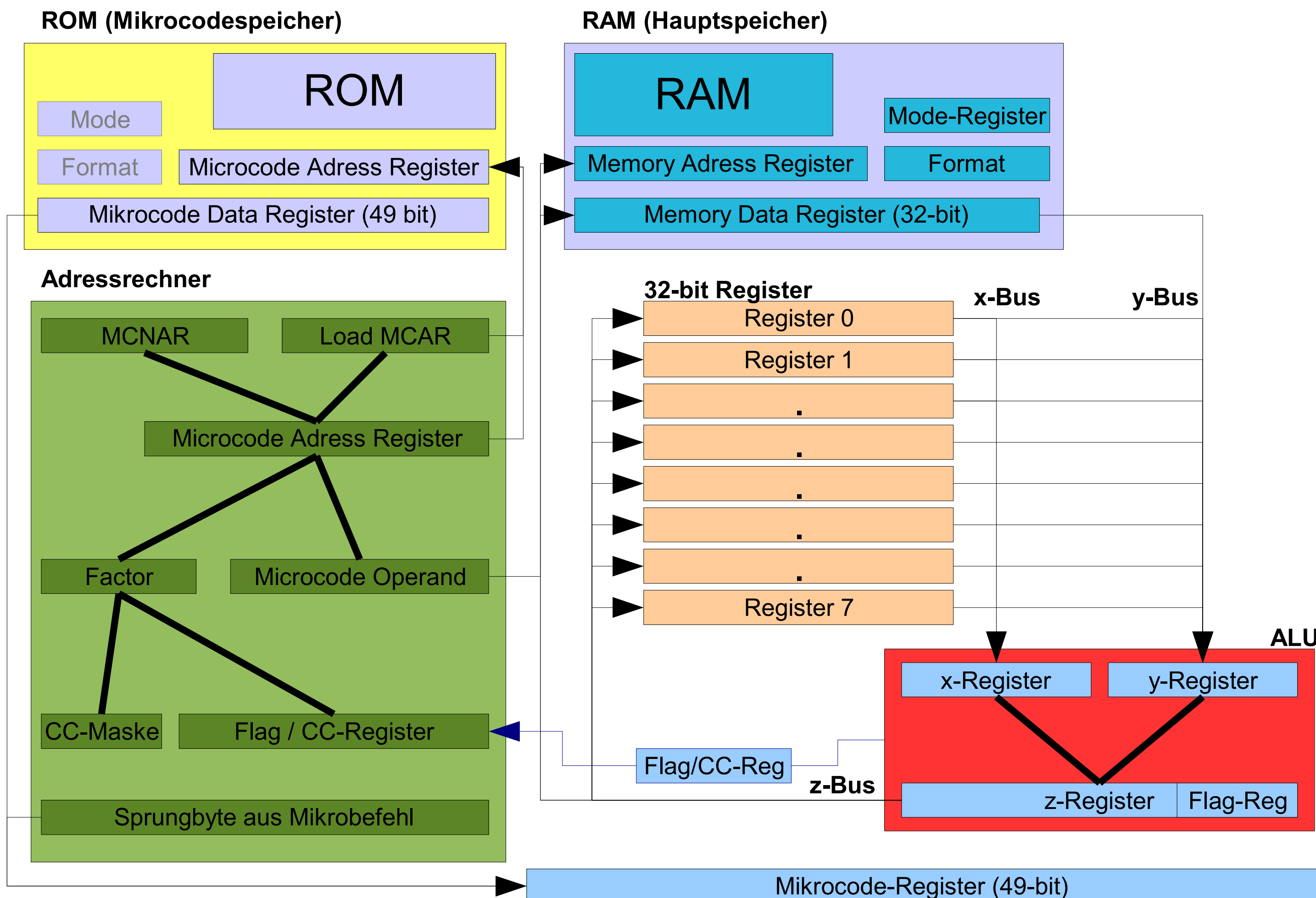
Zusammenfassung der Kapitel 5 und 6

Kapitel 5 - Modell des Rechnerkerns

Dieses Bild am Besten einrahmen und (vorläufig) über das Foto der Freundin / des Freundes kleben!!



Modell des Rechnerkerns nach Mikrosim / Daniel

Dreh- und Angelpunkt des Rechnerkerns ist die Arithmetische Logikeinheit (ALU). Diese nimmt zwei Werte entgegen und generiert daraus einen dritten Wert. Passiert dabei etwas außergewöhnliches, wird dies durch das Setzen eines Bits im Flag-Register signalisiert. Jedes Bit im Flag-Register hat damit eine feste Bedeutung!

Die ALU nimmt die zu verarbeitenden Werte über ihre internen x- und y-Register (je eins) entgegen. Der neue Wert wird im internen z-Register gespeichert. Nun können diese Daten nicht einfach irgendwo her kommen und sie sollten auch nicht einfach im Nirvana verschwinden. Dazu besitzt jede CPU außerhalb der ALU einen unterschiedlich großen Vorrat Register, welche in der realen Welt Namen haben, hier aber zur Vereinfachung einfach von 0…7 durch nummeriert werden. (8 Register sind auch viel zu wenig für sinnvolles Arbeiten).

Im Mikrosim sind die Allzweckregister 32-bit breit, so dass es sich also um eine simulierte 32-bit CPU handelt.

Über den x-Bus und den y-Bus sind diese Allzweckregister also direkt mit dem x- und dem y-Register der ALU in Einbahnstraßenmanier verbunden, so dass Werte aus den Allzweckregistern in die ALU gelangen können. Der Wert, den die ALU damit produziert, wird über den z-Bus wieder zurück in eines der Allzweckregister geführt.

y- Bus und z-Bus sind auch direkt mit dem Hauptspeicher des Computers verbunden, da ja von sich aus, auch nichts sinnvolles in den Allzweckregistern steht. Es ist also möglich, Werte aus dem RAM direkt in das y-Register der ALU zu laden, und das Operationsergebnis wieder direkt in den RAM zu speichern.

Es besteht übrigens keine direkte Verbindung zwischen Allzweckregistern und RAM, so dass die gesamte Kommunikation zwischen Allzuweckregistern und RAM nur über die ALU von statten geht!

RAM (Random Access Memory, Speicher mit Wahlfreiem Zugriff) und ROM (Read only memory, Speicher mit Nur-Lese-Zugriff, im Gegensatz zum WOM – Write only memory) stellen externen Speicher zur Verfügung, welcher sich zunächst einmal außerhalb des Rechnerkerns befindet. Allerdings ist der ROM immer direkt in der CPU eingebaut und so nicht ganz so extern wie der RAM.

Beide Speicher sind sequenziell aufgebaut als eine lange Liste von Speicherwörtern. (Die Vorstellung einer Tabelle ist hier nicht ganz korrekt, da die Adressierung im Speicher linear erfolgt, eben wie bei einer langen, durchnummerierten Liste). Ein Speicherwort ist im ROM 49-bit breit (zumindest im Mikrosim, in der realen Welt nicht unbedingt), im RAM ist ein Speicherwort 8-bit breit. (Nicht nur im Mikrosim sondern auch bei allen 8088-kompatiblen oder dem Zilog 80. Beim Motorla 68k und beim Power PC liegt der Teufel im Detail, aber dies ein ander mal).

Um nun auf ein Speicherwort zugreifen zu können, müssen folgende Daten vor liegen.

- Wo steht das Wort auf der Liste?
- Was soll damit geschehen?
- Wie viele Wörter sollen verwendet werden?

Also:

- Die Adresse des Wortes
- Lese- oder Schreibzugriff?
- Die Anzahl der zu verwendenden Wörter

Dies wird im RAM sowie im ROM durch folgende Register abgebildet:

- Memory Adress Register (11-bit im Mikrosim, glaube ich. Zwischen 16-bit [8088] und 64-bit bei modernen CPUs)
- Memory Mode Register (2 bit im Mikrosim)
- Memory Format Register (2-bit im Mikrosim)

Um also einen Wert aus dem RAM in das y-Register der ALU zu lesen müssen (jeweils mit Umweg über die ALU!) die Adresse des Wortes, die Anweisung zu lesen, und die Anzahl 1 Wort in die entsprechenden RAM-Register gebracht werden. Ist dies geschehen, wird ein Byte (8-bit) aus dem RAM in das **Memory Data Register** gelesen. Von dort kann es über den y-Bus in die ALU (und von dort in die Allzweckregister) geleitet werden.

Um einen Wert im RAM zu speichern, muss man einfach umgekehrt gehen. Ein Wert wird (von der ALU) in das Memory Data Register gebracht. Zusätzlich werden die Angaben über die Position im Speicher, der Schreibbefehl und die Anzahl 1 (von der ALU) in die Memory Register gebracht. Der Schreibzugriff erfolgt.

Wenn der Hauptspeicher byte-weise segmentiert ist, wozu ist dann die Angabe über die Anzahl der Wörter nötig? Byte-weißer Zugriff heißt doch byte-weißer Zugriff.

Dies ist zwar grundsätzlich richtig, allerdings haben wir es hier nicht mit einer 8-bit CPU (Z80, PIC 8xx …) zu tun, sondern mit einer 32-bit CPU, derren Allzuweckregister immer genau 32-bit aufnehmen. Die Vorteile dessen sind bekannt. Also wäre es verschwendete Zeit, 4 Speicherzugriffe zu investieren, um ein Allzweckregister (oder ein Register der ALU) zu füllen. Allerdings will man nicht immer mit den vollen 32-bit Rechnen. Manche Zahlen benötigen eben nur 16-bit oder 24-bit (True Color Farben zum Beispiel). Also gibt man einfach an, wieviele Bytes (bis zu vier) man tatsächlich auslesen oder speichern will. Die Register werden dazu automatisch mit führenden Nullen aufgefüllt, um z.B. aus 16-bit wieder 32-bit zu machen.

Obwohl es nur Lese- und Schreibzugriff gibt, ist das **Memory Mode Register** 2-bit breit, wo nur eines gebügen würde. Hier kommt das „Schalterprinzip" zum Tragen. Wenn das erste Bit den Schalter für den Lese-Zugriff setzt, setzt das zweite Bit den Schalter für dne Schreibzugriff (weiß nicht, ob es gerade so rum ist). Ist kein Schalter gesetzt, passiert halt nichts. Sind beide gesetzt, kann logischer Weiße auch nichts geschehen. Im weiteren Sinne (außerhalb des Kapitels) ist dies wichtig, um den RAM so aus dem Systembus auszuhängen und in einen Warte-Status zu setzen.

Bis jetzt haben wir nur den rechten Teil der obigen Grafik besprochen und den ROM völlig außer Acht gelassen. (Den WOM gibt es nicht, falls das bis hier her jemand geglaubt hat). Wenn man allerdings das Prinzip des RAMs verstanden hat, funktioniert der ROM äquivalent. Der Aufbau ist der Selbe wie beim RAM, nur dass ein Speicherwort eben 49-bit breit ist.

Da die einzige Aufgabe des ROMs darin besteht, der CPU Mikrobefehle anzuliefern, ist der ROM-Zugriff einfacher als der RAM-Zugriff. Ein Mikrobefehl besteht immer aus 49-bit, so dass auch das Memory Data Register 49-bit breit ist. Durch diesen Umstand entfällt auch das Format-Register (oder ist zwar symbolisch vorhanden, aber mit einer Konstante belegt), weil ja immer nur ein Mikrobefehl = 49-bit gelesen werden soll.

Da es auch keine Schreibzugriffe gibt, entfällt auch das Mode-Register. Beide sind in der Grafik oben daher ausgegraut.

Was übrig bleibt sind die lange Liste, welche den eigentlichen Speicher darstellt, das Memory Adress Register, über das die Zugriffsadresse definiert wird, und das Memory Data Register, in das ein ausgelesener Wert erst einmal landet. Die Vorgabe, von welcher Adresse gelesen werden soll kommt vom Adressgenerator, welcher für unsere Klausur nicht von Belang ist. Alle gelesenen Werte landen im Microcode Register der CPU (49-bit breit, tataa!). Was dort einmal drin ist, wird dann auch ausgeführt.

Da das so ist, folgt hier eine Kurze Zusammenfassung über den Aufbau eines Micorcode-Befehls.

Grundsätzlich arbeitet jede CPU getaktet. (So kommt die Mhz oder neuerdigns auch die Ghz-Zahl zu stande. Es gab übrigens auch CPUs, die nur mit kHz getaktet waren!!). In jedem Takt wird genau ein Microcode-Befehl ausgeführt.

Dieser Takt lässt sich in drei Zyklen unterteilen:

- Holen
- Rechnen
- Bringen

Im ersten Zyklus werden alle Daten bewegt, die für die Rechenoperation notwendig sind. Im zweiten Zyklus wird die Operation ausgeführt. Im dirtten Taktzyklus wird das Ergebniss abtransportiert.

Da die Allzweckregister nur über die ALU mit dem RAM verbunden sind, bedarf es mindestens eines Extra-Taktes (und damit mindestens eines Extra-Microcodes) nur um Daten vom RAM in die Register zu speichern.

Ein Micorcode setzt sich wie folgt zusammen:

8-bit Sprungbyte | 6-bit ALU-Befehl | 1-bit CC-Flag | 8-bit x-Register Bewegung | 8-bit y-Register Bewegung | 8-bit z-Register Bewegung | 6-bit I/O-Schalter | 2-bit Mode | 2-bit Format

Die einzelnen Bits fungieren als Schalter im obigen, bunten Diagramm!

x-Register Bewegung	Je ein Bit für R0 nach x | R1 nach x | …
y-Register Bewegung	Je ein Bit für R0 nach y | R1 nach y | …
z-Register Bewegung	Je ein Bit für z nach R0 | z nach R1 | …
I/O-Schalter	Alle anderen Transporte (z.B. RAM -> y; z -> RAM …)
Mode / Format	Werden direkt in Mode / Format vom RAM übernommen
Mode:	'01' Lesen, '10' Schreiben, '00' Warten, '11' Warten
Format	Anzahl der zu verwendenden Bytes des MDR von hinten – 1
ALU-Befehl	Spezifiziert die Verknüpfung, welche die ALU ausführt.
CC-Flag	Wenn gesetzt, wird das Flag-Register in das CC-Register kopiert, (um hinterher Flags zu prüfen)
Sprungbyte	Nur wichtig für Adressgenerator. Definiert z.B., ob nach dem Microbefehl ein weiter folgt oder nicht.

Die Kurznotation, (R0 -> x) mit der wir in der Klausur kleine Mikroprogramme schreiben sollen, ist nur eine andere Darstellung für den binären Aufbau eines Microbefehls. Jede Zeile spiegelt sich also 1:1 als ein gesetztes oder nicht gesetzte Bit im Microbefehl wieder.

Kapitel 6 – Von den höheren Sprachebenen zum Microcode (und damit der Bezug zum Microsim :|)

Es gibt mehrere Sprachebenen, auf denen Programmiersprachen anzusiedeln sind. Dabei handelt es sich nur um eine einfache Skala, wie sehr eine Programmiersprache von der Hardware abstrahiert ist. Generell gilt: Je höher die Abstraktion, desto hardware-unabhängiger ist eine Sprache.

Somit ergibt sich eine einfache Skala in der folgenden Reihenfolge (von unten nach oben)

Microcode, Assembler, „höhere Programmiersprache“ (z.B. C/C++, Pascal, Java), Sprachen der 4. Generation (sog. 4GL-Sprachen, Beschreibungssprachen wie z.B. HTML, XML), u.a. Sprachen der „logischen Programmierung“ (also aus dem Bereich der KI, wie z.B. Prolog), …

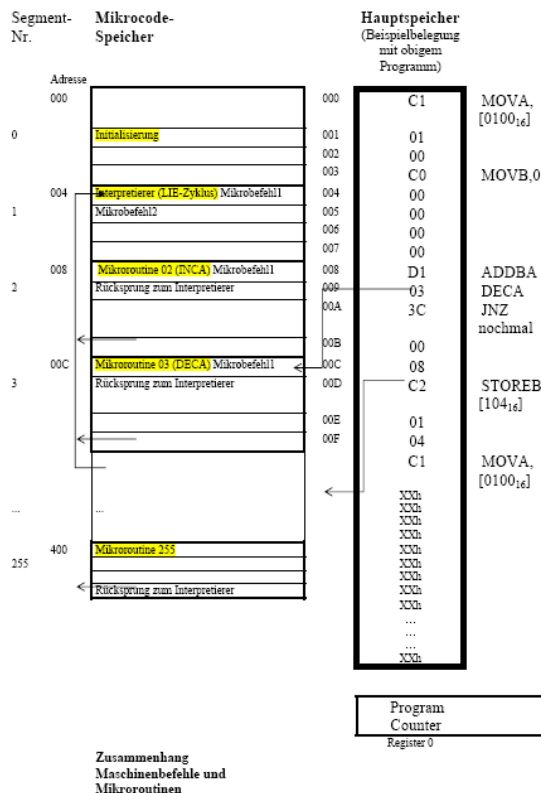
Die Beziehung, welche zwischen diesen Ebenen gilt lässt sich wie folgt ausdrücken: **Jeder Befehl einer Programmiersprache ist ein ganzes Programm in der Ebene darunter!**

Die Interpretation eines HTML-Befehls benötigt ein ganzes C++ Programm. Ein C++ Befehl stellt ein ganzes Asemblerprogramm dar. Ein Assemblerbefehl verkörpert ein ganzes Mikrocode-Programm.

Folglich lassen sich größere Probleme mit den höher angesiedelten Sprachen schneller und effizienter Lösen, da für das Problem unwichtige Maschinendetails verborgen werden. Der Multi-Platform-Support ist auch schon im Preis inbegriffen. (Zumindest in der Theorie). Im Gegenzug dazu erhöht sich der Overhead (unnötige Ausführschritte) und die Ablaufgeschwindigkeit nimmt immer mehr ab. Aus diesem Grund wird oft eine Mischung gewählt, so dass 95% einer Software in höheren Sprachen realisiert sind, und besonders zeitkritische Programmteile (sehr wichtig für Digital Signal Processing, DSP, im Audio- und Video-Bereich) in Assembler handcodiert werden.

Mikroprogramme werden von Mikroprogrammieren bei der Erstellung einer neuen CPU programmiert und sind somit für die meisten aller Programmierer unzugänglich, ja die wenigsten wissen überhaupt von ihrer Existenz. Das „Unterste“ was es also für den normalsterblichen Programmierer gibt ist Assembler, darum auch Maschinensprache genannt. Auf diese Ebene müssen wir uns nun herablassen.

Folgende (aller letzte) Seite aus dem Skript von Herrn Daniel erscheint hilfreich:



Zusammenhang Maschinenbefehle und Mikroroutinen

Zunächst einmal, da ich schon sehr oft danach gefragt wurde, es gibt für die CPU keinerlei Unterscheidung zwischen „Daten“ und „Befehlen“. Alles, wirklich alles, egal ob Maschinenbefehl, ASCII-Zeichen, Teil einer MP3, es sind alles nur Einsen und Nullen

irgendwo in den unendlichen Weiten des Hauptspeichers, wo noch nie zuvor ein Mensch je gewesen ist.

Das Allzweckregister 0 (aus der bunten Grafik ganz oben) wird auch **Program counter** genannt. Die Zahl, die darin steht wird als Adresse im RAM aufgefasst und sagt der CPU, wo der nächste, auszuführene Assemblerbefehl steht. Dies ist der allereinzigste Hinweis, den es gibt, ob eine Speicherstelle ausführbaren Code enthält, oder nicht.

Programmabstürze rühren eben u.a. genau daher, dass irgendwo im Speicher noch alte Nutzdaten herumliegen, die nicht als Code gedacht sind. Aus ominösen Gründen zeigt aber der Program counter auf eben genau diese Speicherstelle und die doofe CPU hat nichts besseres zu tun, als das was da steht auch noch auszuführen. Irgendwann geht's dann halt schief und das System kackt mehr oder minder schwer ab.

Im Folgenden gehe ich immer von der obigen Grafik aus Daniels Skript aus.

Zunächst zeigt der Program counter auf die Speicherstelle 0. Das Byte, welches dort zu finden ist (hier als Hex-Zahl geschrieben) steht symbolisch für den Assemblerbefehl **MOVA [0100_{16}]**.

Das Byte wird also geladen und der fest in der CPU integrierte Interpreter schaut es sich an. Das Erste, was dier Interpreter macht, ist den Program counter um eins hochzuzählen, so dass er auf die Stelle 1 zeigt.

Nun überlegt sich der Interpreter (oder besser der Adressgenerator in der bunten Grafik), wo im ROM steht das Mikroprogramm, das für den MOVA-Befehl ausgeführt werden muss. Das Memory Adress Register des ROMs wird beschrieben, der erste Mikrobefehl wandert ins Microbefehlsregister, die CPU tuckelt. Da der MOVA-Befehl noch zwei Bytes benötigt, werden diese aus dem Speicher gelesen, und zwar von dort, worauf der Program Counter zeigt! Dieser wird vom Mikroprogramm dann soweit hochgezählt, so dass der Counter schließlich auf den nächsten Opcode im Speicher zeigt.

D.h. Es gibt auch keinen Hinweis im Speicher, wieviele „Parameter“ (oder „Argumente“ würde man bei eine Java-Methode sagen) ein Opcode benötigt. Das Mikrprogramm, welches den Opcode bearbeitet weis das ganz für sich alleine und holt sich die Daten einfach, egal was da steht.

Ist alles gut gelafuen, zeigt der Program Counter also auf den nächsten Opcode im Speicher, und das Spiel beginnt von vorne, indem der Interpreter wieder aktiv wird.

Diese drei Zyklen des Interpreters werden auch LIE-Zyklus genannt.

- Load Lade den zu bearbeitenden Opcode
- Increment Zähle den Programm Counter eins hoch
- Execute Führe das entsprechende Mikroprogramm aus. Werden noch Parameter benötigt, werden diese gelesen, von der Speicherstelle, auf die der Programm Counter zeigt. Der Programm Counter wird so hoch gezählt, dass er auf die Stelle **hinter** den gelesenen Daten zeigt. => Am Ende eines LIE-Zyklus zeigt der Program Counter **immer** auf den nächsten Opcode.

Deutlich wird die Parametergeschichte am **ADDBA**-Befehl an der Speicherstelle 8 im Bild. Dieser Befehl hat keine Argumente. Was geschieht also?

Programm Counter = 8, Interpreter tue dein Werk:

- Load Lade Opcode aus Speicherstelle 8
- Increment Zähle Program Counter eins hoch, er zeigt also auf den nächsten Opcode im Speicher !!!
- Execute Führe Mikroprogramm aus. Dieses benutzt den Programm Counter nicht, da keine Parameter nötig.

Im Gegenzug dazu der **MOVB 0**-Befehl an Stelle 3:

Programm Counter = 3, Arbeit für den Interpeter

- Load Lade Opcode aus Speicherstelle 3
- Increment Erhöhe Program Counter um eins, er zeigt also auf die Parameter des Befehls !!!
- Execute Go! Da noch Parameter benötigt werden, lese sie an der Stelle, die der Counter angibt. Erhöhe den Counter entsprechend. Nach Ausführung des Mikroprogramms zeigt er also auf den nächsten Opcode im Speicher !!!

Nach der Ausführung des Mikroprogramms wird wieder in den Interpreter gesprungen. Dieser hat nichts zu tun und beginnt seine Arbeit von Vorne. Lade nächsten Opcode …

Als Abschluss noch der Schritt von einem ausgeschriebenem Assemblerprogramm zum Opcode im Speicher. Dieser Vorgang wird Assemblierung genannt. (Im Gegensatz zur Compilierung wo ein Hochsprachenprogramm in ein Assemblerprogram gewandelt wird, welches anschließend auch assembliert werden muss.)

Gegeben sie folgendes einfaches Programm:

```
MOVB, 0
MOVA, [0023₁₆]
ADDBA
```

In Worten:

- Speichere die Konstante 0 ins Register B.
- Speichere den Inhalt an der Stelle 0023_{16} aus dem RAM ins Register A.
- Addiere Register A und B, Speichere das Ergebniss in B.

Der Assembler (ein ansich simples Programm, darum gibt es auch so viele davon) geht diesen Quellcode also in mehreren Durchläufen durch, am Ende kommen dabei die Opcodes heraus, welche im Speicher geschrieben von der CPU (und ihrem Interpreter) direkt ausgeführt werden können.

Zunächst werden alle Zahlendarstellungen (hier dezimal 0 und hex '0023') ins Binärformat gewandelt. Falls es „Variablen" gibt, werden sie in Speicheradressen relativ zum Programmanfang umgewandelt. Das selbe geschieht mit den „Sprungmarken". (Was beides ist, folgt gleich). Soweit fertig wird jeder Befehl (auch Mnemonic genannt) an Hand einer simplen Tabelle in ein simples Binärmuster umgewandelt. Fertig ist das assemblierte Programm, welches z.B. So aussehen könnte:

```
00000110 00000000 00000000 00000000 00000000
00000011 00000000 00000000 00000000 00100011
00110000
```

Jede Zeile entspricht dabei einer der obigen Zeilen, wenn der Assembler mit seiner Arbeit fertig ist. Es handelt sich um die Binärdarstellung von 8-bit-Werten.

Bei den nicht unterstrichenen Werten handelt es sich einfach nur im die binäre Kurzform der obigen Assemblerbefehle, welche Anhand einer Tabelle ermittelt wurden. (Hier sind sie natürlich nur erfunden). Die Unterstrichenen Teile sind die 32-bit Darstellung der im Programm angegebenen Zahlen.

Wie man sieht, steckt also keinerlei Zauberei hinter der Assemblierung, nur ein primitiver Algorithmus. Die Frage ob ein Wert als Befehl oder als Datum verstanden werden soll hängt, wie gesagt, vom Kontext ab. Nur das Mikroprogramm allein, welches einen Opcode verarbeitet, weiß aber auf den Opcode folgende Daten als Parameter verwendet werden müssen oder nicht. Entsprechend wird der Program Counter hoch gezählt. Würde hier ein Fehler unterlaufen, würde z.B. der Wert 00100011 als Befehl verstanden werden und die CPU irgend einen Blödsinn machen.

Was sind Variablen und Sprungmarken in Assemblerprogrammen? Nichts besonderes, nämlich einfach Namen für irgendwelche Stellen im Speicher. Man gibt den Adressen einfach Namen. Bei „Variablen" handelt es sich dabei um Speicherstellen, in denen Daten zum Rechnen stehen, Sprungmarken zeigen auf Code im eigenen Assemblerprogramm. Bei der Assemblierung (also der Umwandlung von „Menschen"sprache in Binärmuster) werden alle Variablennamen und Spruingmarken zu Speicheradressen umgewandelt und tauchen somit nur noch indirekt im assemblierten Programm auf.

Das ist alles zu den Kapiteln 5 und 6 im Schnelldurchlauf. Bei Fragen entweder ICQ (113-810-966), eMail (dennis@windows3.de) oder Telefon: 07245/916804.

Nochmal dringend: Montag und Mittwoch 9:30 BA-Casino zum Wiederholen von ERS und JAVA, wer Lust hat. Ich werde da sein.

Dennis