Institut für Zuverlässiges Rechnen

Prof. Dr. Thomas Teufel

Bachelorarbeit

im Studiengang „Informatik Ingenieurwesen“

Eine tracing-fähige Z80 System-on-a-Chip Implementierung

An implementation of a traceable Z80 system-on-a-chip

Betreuer: Prof. Dr. Thomas Teufel

Dipl.-Ing. Malte Baesler

Vorgelegt von:

Martin Ringwelski

Matrikel-Nr.: 20624206

E-mail: martin.ringwelski@tuhh.de

Hamburg, Juli 2009

Erklärung

Ich versichere, diese Arbeit im Rahmen der im Arbeitsbereich üblichen Betreuung selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Hamburg, den 7. Juli 2009

Inhaltsverzeichnis

1 Einleitung 1

2 Aufgabenstellung und Herangehensweise 4

3 Werkzeuge 6

3.1 Xilinx ISE 10.1 ... 6

3.2 Z80 Assembler ... 7

3.3 Dev-C++ ... 7

3.4 Asm2vhdl Skript ... 7

4 Der Zilog Z80 9

4.1 Der undokumentierte Befehlssatz ... 9

4.2 Register ... 10

4.3 Interrupt Modi ... 11

5 Dokumentation 13

5.1 Schnittstellen des Z80 SoC von OpenCores ... 15

5.1.1 LEDs ... 15

5.1.2 Buttons, Schalter und Drehknopf ... 15

5.1.3 LC-Display ... 15

5.1.4 Video Schnittstelle ... 16

5.1.5 PS/2 Keyboard ... 17

5.2 Erweiterte Module des SoC ... 18

5.2.1 Der Interrupt Controller ... 19

5.2.2 Der programmierbare Timer ... 20

5.2.3 Die Tracing Schnittstelle ... 21

5.3 Das Tracing-Programm ... 24

6 Probleme 25

6.1 Interrupt Annahme des Z80 ... 25

6.2 Zählen der Instruktionen in dem Tracing-Modul ... 26

6.3 Video RAM Timing ... 27

6.4 Shift-, Caps- und Alt-Gr-Erkennung der Tastatur ... 29

6.4.1 Caps-Erkennung....29
6.4.2 Shift- und Alt-Gr-Erkennung....29

7 Programme **32**
7.1 Beispielprogramm....32
7.2 Ping-Pong-Spiel....33

8 Bedienungsanleitung **35**
8.1 Laden des Designs....35
8.2 Benutzen des Tracing-Programms....36
8.3 Erstellen eines ROM Images....37

9 Zusammenfassung und Ausblick **39**

A Literatur Verzeichnis **40**

B Abkürzungen **41**

C Befehlssatz des Z80 **42**

D Quellcode des Beispielprogramms **46**

1 Einleitung

Diese Bachelorarbeit behandelt die Implementierung und Weiterentwicklung eines Z80 System-on-a-Chip (SoC) auf dem *Spartan-3E Starter Kit Board* von Xilinx (siehe *Abbildung 1*).

Abbildung 1: Spartan-3E Board[1]

Bei einem SoC sind die wichtigsten Systemelemente auf einem Chip integriert. Dazu gehören z. B. die CPU (Central Processing Unit = Zentrale Recheneinheit), Peripherie und Verbindungsstrukturen. Vorteile solcher Systeme sind unter anderem der geringe Strom- und Platz-Verbrauch sowie Geschwindigkeitsvorteile durch kürzere Verbindungen zwischen den Modulen. *Abbildung 2* zeigt den groben Aufbau des in dieser Arbeit entwickelten SoC.

Als Plattform dient der *Xilinx XC3S500E Spartan-3E FPGA*, mit 232 I/O Pins, 360KBit Block RAM und über 10.000 Logik-Einheiten. Diese Logik-Einheiten sind nicht fest verdrahtet, sondern frei programmierbar. Dadurch ist es möglich durch

Rekonfiguration verschiedene Systeme auf einem Chip zu realisieren und zu testen.

Der Chip ist auf dem Spartan-3E Board eingebettet. Das Board beinhaltet darüber hinaus folgende Komponenten:

- eine VGA-Schnittstelle
- eine PS/2-Schnittstelle
- zwei RS-232 Schnittstellen
- einen Digital/Analog- (D/A) und einen Analog/Digital- (A/D) Wandler
- eine 10/100 Ethernet Buchse
- vier Buttons, Schalter und einen Drehknopf
- acht LEDs
- ein LC-Display
- 64 MByte DDR RAM
- 16 MByte paralleler NOR Flash-Speicher
- 16 MByte serieller Flash-Speicher
- einen Oszillator mit 50 MHz.

Die Konfiguration des FPGA erfolgt über USB und einem auf dem Board integrierten Programmieradapter.

Grundlage dieser Arbeit bildet das Z80 SoC Design von *OpenCores*[2], einem über das Internet koordinierten Projekt, das von vielen Entwicklern weiter entwickelt wird und dessen Quellcode frei zur Verfügung steht. Ziel dieser Arbeit ist die Analyse und Erweiterung des Designs.

Die folgenden Kapitel beschreiben zunächst den Z80 und die Herangehensweise an die Aufgabenstellung. Diese Arbeit dokumentiert das hierfür entwickelte System und gibt dem Leser einen Überblick über die verschiedenen Schnittstellen und seine Möglichkeiten. Nach einer Bedienungsanleitung zum schnellen Einstieg in die Benutzung des SoC, sollen zum Schluss Probleme bei der Entwicklung des Systems aufgezeigt und einige Punkte genannt werden, an denen das System noch verbessert werden kann.

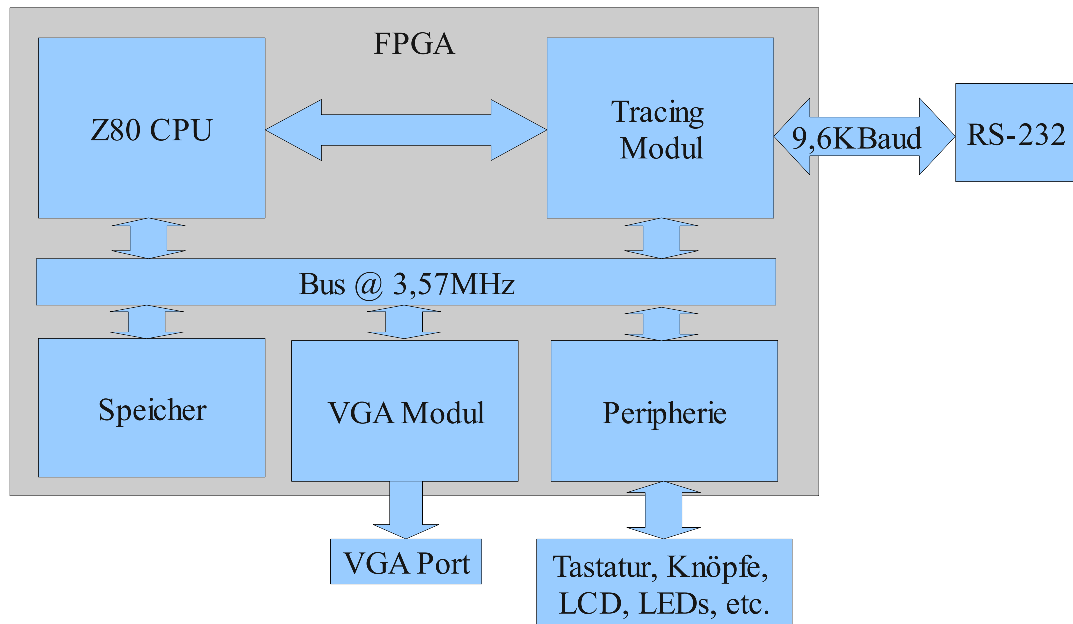

Abbildung 2: Aufbau des Z80 System-on-Chip

Zahlen, die in dieser Arbeit mit einem nachstehenden *h* gekennzeichnet sind, stellen hexadezimal codierte Zahlen dar. Signale sowie Ein- und Ausgänge, die überstrichen sind, stellen negierte Signale dar.

2 Aufgabenstellung und Herangehensweise

Als Basis dieser Arbeit wurde das Z80SoC Projekt von *OpenCores* in der letzten Version vom 24. Mai 2008 verwendet. Das Projekt ist vollständig in der Hardwarebeschreibungssprache VHDL verfasst. Trotz des stabilen Status und der langen Zeit, in der keine Änderungen am Projekt erfolgten, ist dieses in einigen Punkten noch unvollständig.

Beim Design war ein Bash-Skript zum Umwandeln von Binär-Dateien in VHDL-ROM-Dateien enthalten, welches aber nur unter *UNIX*-artigen Betriebssystemen lauffähig ist. Es wurde daher im Rahmen dieser Arbeit ein eigenes Skript geschrieben, welches unter *MS Windows* diese Aufgabe übernehmen soll.

Das Design wurde zunächst in seiner ursprünglichen Form getestet. Es besitzt ein Beispiel-ROM, welches die Funktionen des SoC demonstrieren soll. Folgende Funktionen wurden von dem SoC bereitgestellt:

- Ausgabe auf das LC-Display,
- Ansprechen der LEDs,
- Abfrage der Buttons, Schalter und des Drehknopfes,
- Ausgabe über die VGA Schnittstelle,
- Eingabe von Zeichen über die Tastatur.

Hier zeigten sich bereits einige Grenzen:

- Der Bildschirm konnte nur 40 Spalten und 30 Zeilen mit Zeichen füllen.
- Die Bildschirmausgabe war auf blaue Schrift auf schwarzen Hintergrund begrenzt.
- Die Tastatur unterstütze keine Shift-Taste und die Capslock-Erkennung funktionierte nicht richtig.
- Interrupts wurden nicht unterstützt.

Da der Z80 nur einen maskierbaren Interrupt-Pin zur Verfügung stellt, sollte die Hauptaufgabe zunächst die Implementierung eines Interrupt Controllers sein, der mehrere Interruptquellen verwaltet. Zum Test verschiedener Interrupts wurde ein

Timer-Modul erstellt, welches nach einer konfigurierbaren Zeit einen Interrupt auslöst. Des Weiteren wurden zwei Buttons und der PS/2 Controller an den Interrupt Controller angeschlossen.

Danach wurde ein Tracing-Modul erstellt, welches es ermöglicht, die Ausführung des Programms anzuhalten und über die serielle RS-232 Schnittstelle die Inhalte der Register und des Speichers auszulesen. Hierfür wurde ein PC Programm geschrieben, welches die Kommunikation mit dem Tracing-Modul übernahm. Mit Hilfe des Tracing-Programms ist es möglich die Ausführung nach einer beliebigen Anzahl an Befehlen anzuhalten. Auch ein Stopp beim Auftreten eines Interrupts ist möglich.

Zusätzlich wurden einige andere Elemente des Designs verbessert. Dazu gehören die Bildschirmausgabe und die Tastatureingabe. Des Weiteren wurde ein Beispielprogramm zum Testen der Module erstellt.

Zuletzt wurde ein Ping-Pong-Spiel für das SoC entwickelt.

3 Werkzeuge

In diesem Kapitel werden die Programme und Werkzeuge vorgestellt, die beim Bearbeiten dieser Bachelorarbeit verwendet wurden. Weiterhin wird auf die Besonderheiten eingegangen und die Gründe für den Einsatz dieser Werkzeuge werden erläutert.

3.1 Xilinx ISE 10.1

Zur Entwicklung des SoC wurde das *Integrated Software Environment* (ISE) von der Firma Xilinx in der Version 10.1.03 verwendet. Mit dem *Project Navigator* werden das Design und die einzelnen Module verwaltet. Es stellt unter anderem einen Editor und Synthetisierungs-Tool für die Hardware Beschreibungssprache VHDL zur Verfügung.

Die Module können synthetisiert, implementiert und schließlich zur Konfiguration des Boards mit dem integrierten *iMPACT* Programm auf das Board übertragen werden. *iMPACT* erkennt automatisch das über USB angeschlossene Board.

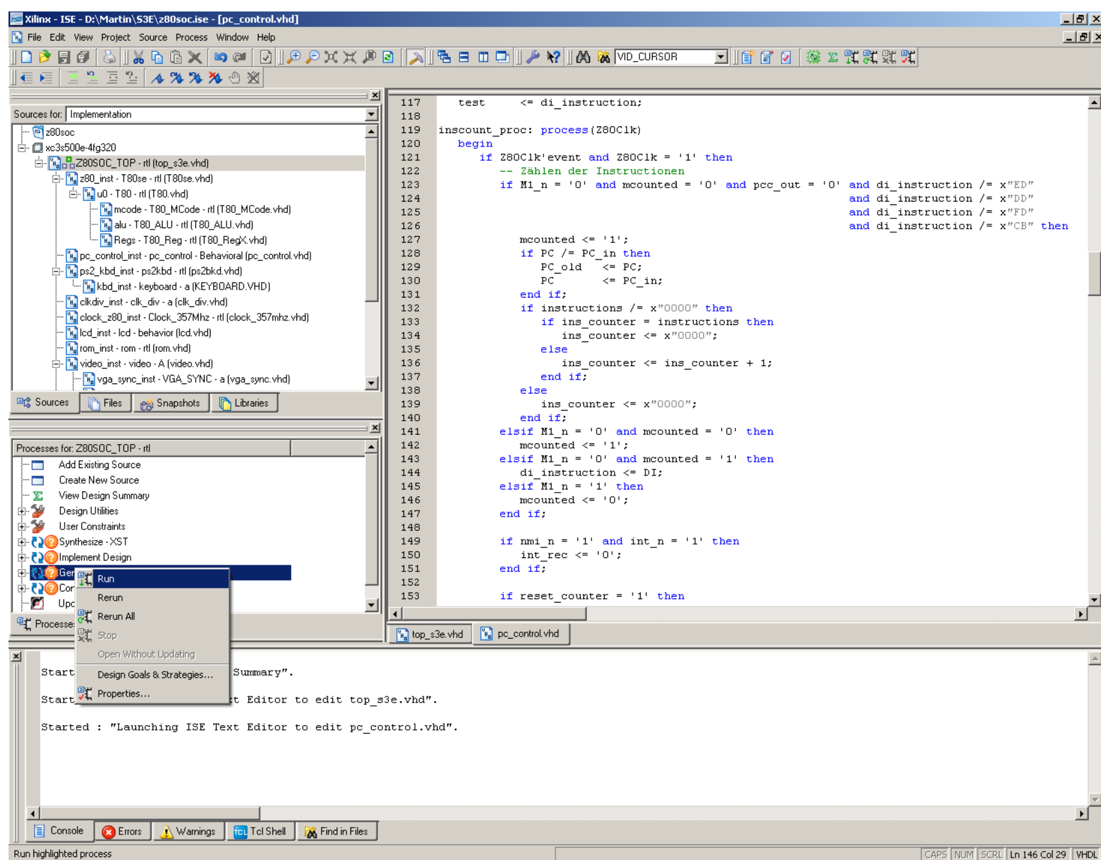

Abbildung 3: Xilinx - ISE

3.2 Z80 Assembler

Zum Erstellen von Programmen für den Z80 diente das Open Source Projekt des Z80 Assembler unter der Leitung von Bas Wijnen[3]. Mit dem Assembler können die Befehle der CPU von einer für Menschen lesbaren Form in Maschinensprache übersetzt werden.

Dieser Assembler war portierbar, musste nur unter *MS Windows* kompiliert werden und erwies sich als leicht bedienbar. Er unterstützt den gesamten dokumentierten Z80 Befehlssatz sowie weitere undokumentierte Befehle, die in dieser Arbeit jedoch nicht verwendet wurden. Auf den undokumentierten Befehlssatz wird im Kapitel 4.1 genauer eingegangen.

3.3 Dev-C++

Dev-C++ ist eine freie Entwicklungsumgebung von Bloodshed Software für die Programmiersprachen C und C++. Es wurde die letzte Version 4.9.9.2 verwendet, die seit 2005 nicht mehr aktualisiert wurde. *Dev-C++* zeichnet sich durch seine leichte Bedienbarkeit aus.

Dev-C++ wurde für die Entwicklung aller Bestandteile des *Asm2vhdl* Skriptes und zum Erstellen des Tracing-Programms verwendet. Der im *Dev-C++* Paket enthaltene *MinGW* Compiler, der eine Windows Variante der *GNU Compiler Collection* (GCC) ist, diente zum Kompilieren des *Z80 Assemblers*.

3.4 Asm2vhdl Skript

Das Asm2vhdl (sprich: *Assembler-to-VHDL*) Skript wurde im Zuge dieser Arbeit geschrieben und übernimmt einige aufeinander folgende Schritte beim Erzeugen des ROM-Image-Moduls für das SoC. Im Design Paket des Z80 SoC von *OpenCores* war zwar ein Skript für diese Umwandlung enthalten, allerdings benötigt dieses eine *UNIX*-ähnliche Umgebung. Dieses sollte aber unter *MS Windows* verwendet werden. Das Skript konnte nicht einfach angepasst, sondern musste komplett neu geschrieben werden.

Als Parameter wird eine Assembler Datei angegeben. Diese wird zunächst mit Hilfe des Z80 Assemblers assembliert und in eine HEX-Datei umgewandelt. Hierfür wurde

ein Programm geschrieben, welches die binär-codierten Zeichen der assemblierten Datei einliest, hexadezimal codiert und in eine Datei schreibt. Anschließend wird daraus durch ein weiteres Programm eine VHDL-ROM-Datei erstellt, in der jeder HEX-Wert nacheinander einer Speicheradresse zugeordnet wird. Zum Schluss wird diese Datei in das Verzeichnis des Z80 SoC kopiert.

Das ROM wird auf dem FPGA mit den vorhandenen Logik-Einheiten umgesetzt und wie alle anderen Module des SoC über das Konfigurationsbitstream des FPGA initialisiert. Daher muss das Projekt nach Erstellung einer neuen VHDL-ROM-Datei neu synthetisiert, implementiert und übertragen werden, damit das System mit dem angegebenen Programm im ROM starten kann. Je nach Größe des Programms kann das Synthetisieren bis zu 15 Minuten dauern.

4 Der Zilog Z80

Der Zilog Z80 ist ein 8-Bit-Prozessor mit einer CISC (Complex Instruction Set Computing) Architektur und einem zunächst fest verdrahteten, in späteren Versionen jedoch mikroprogrammierten Steuerwerk. Er wurde 1976 von der Firma Zilog auf den Markt gebracht und ist abwärtskompatibel zum zwei Jahre früher erschienenen 8080 von Intel. Jedoch besitzt der Z80 einige Vorteile gegenüber dem 8080. Dazu gehören eine bessere Stromversorgung, eine eingebaute Refresh-Steuerung für dynamischen RAM (DRAM), 16-Bit-Blockkopier- und Vergleichsbefehle, zwei echte 16-Bit-Register, ein doppelter Registersatz, Ein- und Ausgabe (I/O)-Operationen und komplexe Interrupt-Funktionen.

Für die I/O-Operationen werden acht Bit des 16 Bit breiten Adressbusses genutzt. Die Geräte werden direkt über diese I/O Ports und spezielle I/O Befehle angesprochen (Isolated I/O). Sie müssen also nicht in den Adressraum der CPU abgebildet werden (Memory Mapped I/O), wie es z.B. beim 8080 der Fall war.

Die bessere Stromversorgung und die Refresh-Steuerung sind bei dieser Arbeit nicht von Bedeutung, da die Stromversorgung im FPGA geregelt ist und kein externer DRAM benutzt wird.

Der Z80 hatte ursprünglich eine Taktrate von 1 MHz und keinen Multiplikator, der es erlaubt die CPU mit einem Vielfachen des Bus-Taktes zu betreiben. In diesem Design werden der Z80 und der Datenbus mit 3,57 MHz betrieben.

4.1 Der undokumentierte Befehlssatz

Der ursprüngliche Z80 verfügte über einen undokumentierten Befehlssatz, der den eigentlichen Befehlssatz in etwa verdoppelte und zur Popularität des Z80 beigetragen hat. In späteren Varianten des Z80 war dieser Befehlssatz nicht mehr vorhanden. Er geht vermutlich auf das fest verdrahtete Steuerwerk des Z80 zurück, welches so verschiedene Kombinationen von Operationsbytes zulässt, die nicht vorgesehen waren. In den späteren mikroprogammierten Varianten des Z80 waren nur noch die programmierten Befehle, also der dokumentierte Befehlssatz, verfügbar. Der dokumentierte Befehlssatz wird in *Anhang C* aufgeführt.

4.2 Register

Für arithmetische und logische Operationen besitzt der Z80 acht 8-Bit-Register (A, B, C, D, E, H, L, F), die zusätzlich als alternatives Registerset zur Verfügung stehen (A', B', C', D', E', H', L', F'). Das Flag-Register F kann allerdings nicht für arithmetische oder logische Operationen benutzt werden (siehe *Tabelle 1*). Es kann jeweils nur ein Registerset zur Zeit verwendet werden. Zum Zugriff auf das andere Registerset muss das aktive Set geändert werden. Die Registerpaare BC, DE und HL können zusätzlich als 16-Bit-Register verwendet werden.

Bit	Flag
0	Carry Flag: Zeigt an, ob bei der vorherigen Operation ein Überlauf stattgefunden hat.
1	Substract Flag: Ist gesetzt, wenn die letzte Aktion eine Subtraktion war.
2	Parity/Overflow Flag: Bei einer geraden Anzahl von Bit oder bei einer Operation im 2er-Complement mit Überlauf gesetzt.
3	undokumentiert
4	Half Carry Flag: Gesetzt bei einem Überlauf von Bit drei zu Bit vier.
5	undokumentiert
6	Zero Flag: Zeigt an, ob das Ergebnis einer Operation negativ ist.
7	Signed Flag: Gesetzt, wenn das höchste Bit im Ergebnis gesetzt ist.

Tabelle 1: Aufbau des Flag-Register des Z80

Die Index Register IX und IY sind echte 16-Bit-Register, die für Speicheroperationen konzipiert wurden. Durch die undokumentierten Befehle können mit diesen Registern auch viele andere Operationen durchgeführt werden.

Das Interrupt Register I erlaubt es, Interrupt Routinen beliebig im Speicher zu platzieren, die mit Hilfe dieses Registers indirekt adressiert werden können. Die höherwertigen acht Bit der Adresse werden vom I-Register gebildet, die niederwertigen acht Bit werden vom Interrupt erzeugenden Gerät geliefert und vom Datenbus gelesen (siehe *4.3 Interrupt Modi*).

Der Stackpointer SP sowie der Program Counter PC sind 16-Bit-Register. SP verweist auf die aktuelle Speicheradresse des obersten Stackelements. Auf dem Stack werden insbesondere die Rücksprungadressen bei Funktionsaufrufen und Interrupts gespeichert. Daher ist es wichtig, dass der SP zu Beginn eines Programms gesetzt wird. Es bietet sich an, ihn auf die höchst mögliche Speicheradresse zu initialisieren, da beim Speichern auf dem Stack der SP Wert verringert wird. PC enthält immer die

Adresse der nächsten auszuführenden Instruktion. Dieses Register kann der Programmierer nur durch Sprünge und Funktionsaufrufe manipulieren.

Das Refreshregister R wird von der CPU automatisch mit jedem Taktzyklus erhöht und kann zum Auffrischen von dynamischen Speichern verwendet werden. Damit ist es für das SoC, welches keinen externen DRAM nutzt ohne Bedeutung. Möglich wäre es jedoch, es als Seed oder Quelle für Pseudo-Zufallszahlen zu nutzen. Diese Zahlen sind zwar determiniert, da es aber sehr unwahrscheinlich ist, dass die Benutzereingabe immer nach der gleichen Anzahl von Takten geschieht, ist ungewiss, auf welchem Stand das R Register gerade steht. Diese Methode wird bei dem Ping-Pong-Spiel für das SoC verwendet (siehe *Kapitel Fehler: Referenz nicht gefunden*).

4.3 Interrupt Modi

Der Z80 hat zwei Interrupt Eingänge, einen für maskierbare Interrupts (MI) und einen für nicht-maskierbare Interrupts (NMI). Ein Interrupt unterbricht die aktuelle Programmausführung und lässt die CPU zu einer Interrupt Behandlungsroutine springen. Der NMI ist von höchster Priorität und kann nicht deaktiviert werden. Er wird von der Hardware in der Regel für Sonderfälle, wie z.B. Speicherfehler, verwendet. Damit soll sichergestellt werden, dass die CPU in bestimmten Fehlerfällen in einen definierten Zustand kommt.

Während auf einen NMI immer mit einem Sprung zur Adresse 0066h reagiert wird, kann die Reaktion auf einen MI durch drei verschiedene Modi bestimmt werden. Ein Interrupt-Enable-Flipflop bestimmt, ob ein MI überhaupt angenommen werden soll.

Im **Mode 0** wird nach einem Interrupt auf dem Datenbus ein 8-Bit-Befehl von dem Gerät bereit gestellt, welches den Interrupt ausgelöst hat. Dieser wird anstelle des folgenden Befehls im Programmcode ausgeführt. Da die Möglichkeiten einer 8-Bit-Instruktion sehr begrenzt sind, wird hier in der Regel ein Reset zur Speicherstelle 0 angegeben.
Dieser Modus entspricht der Interrupt Behandlung des 8080 und ist standardmäßig aktiviert.

Im **Mode 1** reagiert die CPU auf einen MI ähnlich wie auf einen NMI, mit dem Unterschied, dass anstelle der Speicherstelle 0066h auf die Speicherstelle 0038h

gesprungen wird. Die verschiedenen maskierbaren Interrupts sind in diesem Modus nicht zu unterscheiden.

Mode 2 ist der einflussreichste und am meisten verwendete Interrupt Modus. Dieser Interrupt Modus wird auch von dem in diesem SoC verwendeten Interrupt Controller vorausgesetzt. Das Interrupt-auslösende Gerät legt ein 8-Bit-Datum auf den Datenbus, welches zusammen mit dem Inhalt des Interrupt Registers eine 16-Bit-Adresse ergibt, an der sich die Adresse einer Interrupt Routine befindet (siehe *Abbildung 4*).
Da die Adressen im Speicher 16 Bit breit sind, sind so 128 verschiedene Interrupt Routinen möglich. Interrupts, die in dieser Arbeit mit einer Nummer versehen sind, weisen auf die Nummer der Routine hin, welche durch diesen Interrupt aufgerufen wird.

Das Setzen des Interrupt Modus erfolgt über die Befehle IM 0, IM 1, bzw. IM2. Durch die Befehle EI und DI wird das Interrupt-Enable-Flipflop gesetzt bzw. gelöscht.

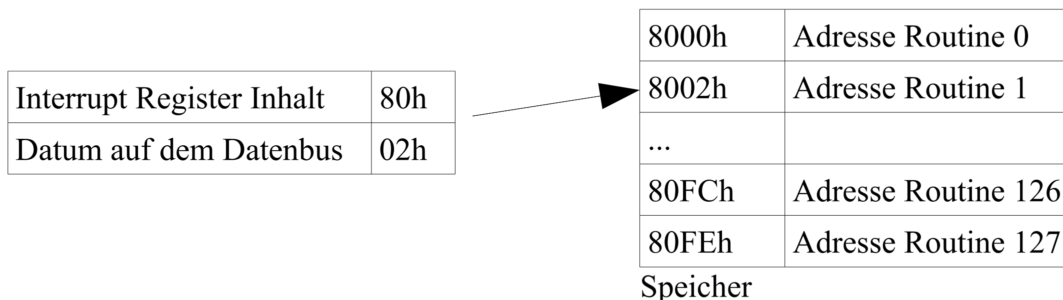

Abbildung 4: Adressierung von Interrupt Routinen im Mode 2

5 Dokumentation

In diesem Kapitel sollen die nutzbaren Ein- und Ausgabe Schnittstellen (I/O Ports) und Speicherbereiche des Z80 SoC erklärt werden. Zuletzt wird auf die Funktionsweise des Tracing-Programms eingegangen.

Tabelle 2 zeigt die möglichen I/O Port Adressen und die dadurch erreichbaren Geräte bzw. Funktionen. *Tabelle 3* beschreibt den Speicheraufbau und die durch die verschiedenen Adressen zu erreichenden Geräte. Einige Geräte sind sowohl über Isolated I/O, also Port-Adressen, als auch über Memory Mapped I/O, also Speicheradressen, zu erreichen.

Port Adresse	Gerät
01h	LEDs
20h	Schalter
30h	Buttons
70h	Drehknopf
80h	Tastatur auslesen
81h	Tastatur Status auslesen*
90h	Bildschirm schreiben/lesen
91h	X Position auf dem Bildschirm
92h	Y Position auf dem Bildschirm
93h	Video Farbe*
94h	Video Modus*
A0h	Timer Interrupt*

*Tabelle 2: I/O Ports des Z80 SoC. Mit * markierte Geräte wurden im Zuge dieser Arbeit erstellt.*

Speicheradresse	Gerät
0000h - 3FFFh	ROM Speicher
4000h - 5FFFh	Video-RAM
7FDBh	Keyboard Scancode*
7FDDh	BFFEh zum Setzen des Stackpointers
7FE0h - 7FFFh	LCD RAM
8000h - BFFFh	RAM

*Tabelle 3: Speicheradressen des Z80 SoC. Mit * markierte Geräte wurden im Zuge dieser Arbeit erstellt.*

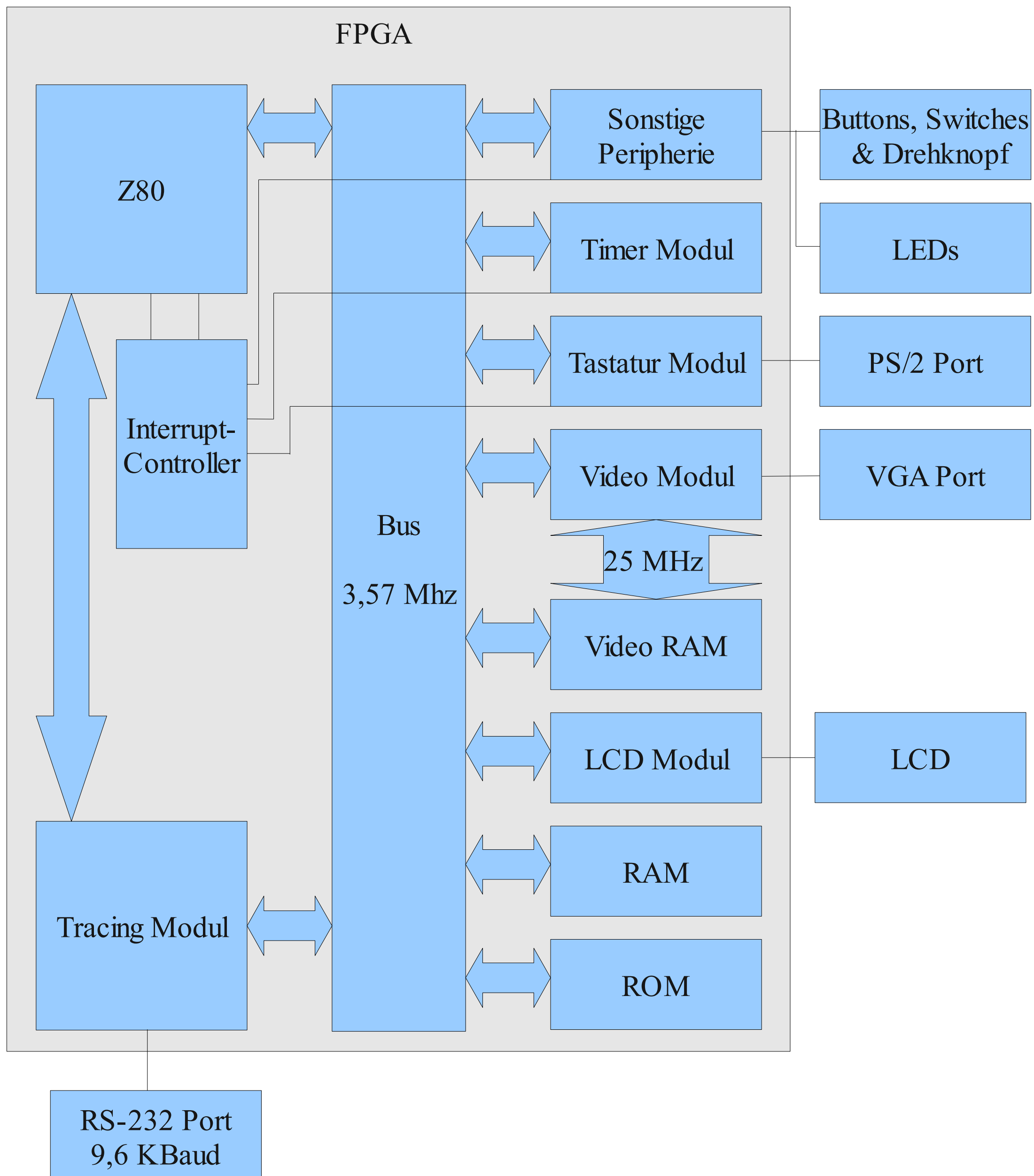

Abbildung 5: Kompletter Aufbau des Z80 SoC

5.1 Schnittstellen des Z80 SoC von OpenCores

Das Z80 SoC Design von *OpenCores* ist bereits im originären Zustand mit einigen Modulen und Schnittstellen für die Ein- und Ausgabe ausgestattet. Einige dieser Schnittstellen wurden im Zuge dieser Arbeit erweitert und verbessert.

Abbildung 6: Komponenten des Spartan 3E Boards

5.1.1 LEDs

Auf dem *Spartan 3E* Board befinden sich acht grüne LEDs (siehe *Abbildung 6* A). Diese können mit Hilfe des I/O Ports 01h beschrieben und ausgelesen werden. Jedes Bit repräsentiert eine LED.

5.1.2 Buttons, Schalter und Drehknopf

Der Zustand der Schalter (*Abbildung 6* B) kann über Port 20h ausgelesen werden. Die niederwertigen vier Bit repräsentieren die Zustände der vier Schalter, die höherwertigen vier Bit sind nicht belegt und daher auf Null gesetzt.

Die vier Buttons (*Abbildung 6* C), die mit North, South, East und West gekennzeichnet sind, lassen sich über den Port 30h auslesen.
Im Zuge dieser Arbeit wurde dem North-Button der Interrupt 0 und dem East-Button der NMI zugewiesen. Der South-Button führt zu einem Fortsetzen der Ausführung nach Anhalten durch das Tracing-Modul.

Der Drehknopf (*Abbildung 6* D) kann über den Port 70h ausgelesen werden. Linksdrehen setzt das Bit 0, Rechtsdrehen setzt das Bit 1. Ein Drücken des Drehknopfes löst einen Reset des Z80 zur Speicherstelle 0 aus.

5.1.3 LC-Display

Das LC-Display (*Abbildung 6* E) besteht aus zwei Zeilen zu je 16 Zeichen. Auf dem Display werden Zeichen dargestellt, die in die Adressen 7FE0h bis 7FFFh

geschrieben werden. 7FE0h bis 7FEF beinhalten die erste Zeile und 7FF0h bis 7FFFh die zweite. Zur Ausgabe dient der um die Codepage 437 erweiterte ASCII Zeichensatz.

5.1.4 Video Schnittstelle

Das Video-Modul gibt den Inhalt des Video-RAM auf der VGA-Schnittstelle aus. Beim Video-RAM handelt es sich um einen asynchronen dual ported RAM. Dieser ermöglicht dem Video-Modul eine direkte Anbindung an den Video-RAM mit 25 MHz, während die CPU über den BUS mit 3,57 MHz auf den RAM schreiben kann (*Abbildung 7*). Der Video-RAM ist auf den Adressen 4000h bis 5FFFh abgebildet. Jedes Byte wird als Zeichen nach Codepage 437 ausgegeben. Die Auflösung beträgt 640x480 Pixel. Jedes Zeichen besteht aus 8x8 Pixel.

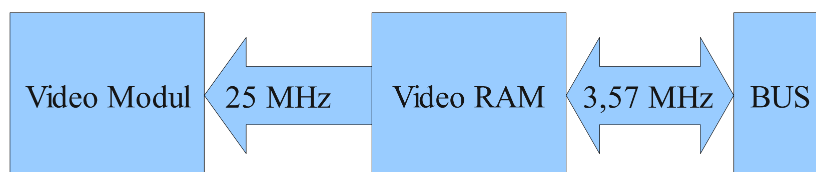

Abbildung 7: Video-RAM: Asynchroner dual ported RAM

Zum Schreiben auf den Bildschirm gibt es zwei Möglichkeiten. Zum einen kann direkt in den Speicher geschrieben werden. Dabei muss das Programm jedoch selbst die aktuelle Cursor-Position abspeichern. Zum anderen werden drei Ports bereit gestellt, über die man in den Video-RAM schreiben kann. Zeichen, die auf dem Port 90h ausgegeben werden, werden an die aktuelle Cursorposition geschrieben und der Cursor wird automatisch erhöht. Mit den Ports 91h und 92h können die X- und Y-Position des Cursors ausgelesen und manuell gesetzt werden.

Im Zuge dieser Arbeit wurde das Video-Modul erweitert. Mit Port 93h lässt sich so die Vorder- und Hintergrundfarbe setzen. Die höherwertigen vier Bit setzen die Hintergrundfarbe, die vier geringwertigen Bit die Vordergrundfarbe. Die VGA-Ausgabe ist ein analoges Signal, das heißt die Intensität einer Farbe wird durch die Spannung am Ausgang bestimmt. Da das FPGA jedoch nur zwei Zustände unterscheiden kann – Spannung oder keine Spannung – können jeweils nur die geringwertigen drei Bit genutzt und nicht zwischen hellen und dunklen Farben unterschieden werden. Somit ergeben sich acht mögliche Farben.

Der folgende VHDL-Code Ausschnitt zeigt wie die Signal-Zuweisung funktioniert. *VGA_R_sig* ist das Signal für Rot, *VGA_G_sig* das Signal für Grün und *VGA_B_sig* das Signal für Blau. *COLOR* ist der Wert, der über Port 93h gesetzt wurde und *pixel_sig* das zu zeichnende Pixel des aktuellen Zeichens.

```
VGA_R_sig <= (pixel_sig and COLOR(2)) or ((not pixel_sig) and COLOR(6));
VGA_G_sig <= (pixel_sig and COLOR(1)) or ((not pixel_sig) and COLOR(5));
VGA_B_sig <= (pixel_sig and COLOR(0)) or ((not pixel_sig) and COLOR(4));
```

Weiterhin wurde das Modul um einen Modus-Port (94h) erweitert. Im Modus 0 (Bit 0 ist nicht gesetzt) werden die Pixel der Zeichen viermal auf den Bildschirm geschrieben. Damit ergibt sich im Gegensatz zur tatsächlichen Auflösung von 640x480 Pixel ein nutzbare Auflösung von 320x240 Pixel und ein Zeichenraster von 40 Spalten und 30 Zeilen. Dies ist die ursprüngliche Darstellungsweise der Video-Schnittstelle. Dieser ursprüngliche Darstellungsmodus wurde so erweitert, dass sich mit den höchstwertigen sechs Bit (Bit 2 bis 7) des Modus-Ports der Bildschirm horizontal scrollen lässt.

Im Modus 1 (Bit 0 ist gesetzt) werden die Pixel der Zeichen einzeln auf den Bildschirm gebracht. Es ergibt sich ein Raster von 80 Spalten und 60 Zeilen. In diesem Modus ist kein horizontales Scrollen möglich.

Es ist möglich von beiden Modi hin und her zu schalten. Im Modus 0 ist nur ein Ausschnitt der oberen Hälfte des Bildschirms von Modus 1 sichtbar.

5.1.5 PS/2-Keyboard

Das PS/2-Keyboard-Modul wandelt die eingehenden Daten der Tastatur automatisch in ASCII Zeichen um und stellt diese über den Port 80h zur Verfügung. Ursprünglich setzte das Modul eine englische Tastatur um, dieses wurde im Rahmen dieser Arbeit jedoch auf eine deutsche Tastatur geändert.

Durch eine Erweiterung des Moduls lässt sich der Scancode der Tastatur über die Adresse 7FDBh als 2-Byte-Word auslesen. Damit sollen andere Tastatur-Layouts von Seiten der Software unterstützt werden können.

Sowohl das Betätigen als auch das Loslassen einer Taste löst den Interrupt 2 aus.

Das Einrasten des Capslock wurde verbessert und ein weiterer Port hinzugefügt. Mit diesem kann ein Tastatur-Status-Byte ausgelesen werden, welches den Capslock

Status sowie gedrückte Steuertasten enthält (Port 81h). Den Aufbau des Tastatur-Status-Byte zeigt *Tabelle 4*.

Bit	**Taste**
0	Capslock
1	Numlock
2	Strg (links)
3	Alt
4	Strg (rechts)
5	Alt Gr
6	Shift (links)
7	Shift (rechts)

Tabelle 4: Bedeutung der Bit im Tastatur-Status-Byte

Der folgende Code zeigt die Erkennung des Capslock. Von der Tastatur wird das Signal *scan_code_sig* empfangen. 58h ist der Scancode der Caps-Taste. Der Status des Capslock ist in c*aps(0)* enthalten, *caps(1)* speichert ob der Scancode bereits registriert wurde.

```
if scan_code_sig = x"58" then
        if caps(1) = '0' then
                caps(1) <= '1';
                caps(0) <= not caps(0);
        end if;
else
        caps(1)      <= '0';
end if;
```

Es ist auch möglich mit den Shift-Tasten große Buchstaben und mit der Alt-Gr-Taste weitere Zeichen einzugeben. Bei diesen Tasten zeigen sich jedoch noch ein paar Probleme, die im Kapitel 6.4 genauer beschrieben werden. Der Numlock-Status hat bisher keine Auswirkungen. Der Nummernblock, so wie die Pfeil- und Funktionstasten werden nicht vom Modul unterstützt, ihr entsprechender Scancode kann aber über die Speicheradresse 7FDBh ausgelesen werden.

5.2 Erweiterte Module des SoC

Die folgenden sollen Module vorgestellt werden, welche nicht im Design von *OpenCores* nicht enthalten sind. Diese wurden Laufe der vorliegenden Arbeit neu

erstellt.

5.2.1 Der Interrupt Controller

Der Interrupt Controller kann neun verschiedene Interrupts verwalten. Einen nicht-maskierbaren (NMI) und acht maskierbare Interrupts (MI). Der Controller setzt voraus, dass sich der Z80 im Interrupt Mode 2 befindet.

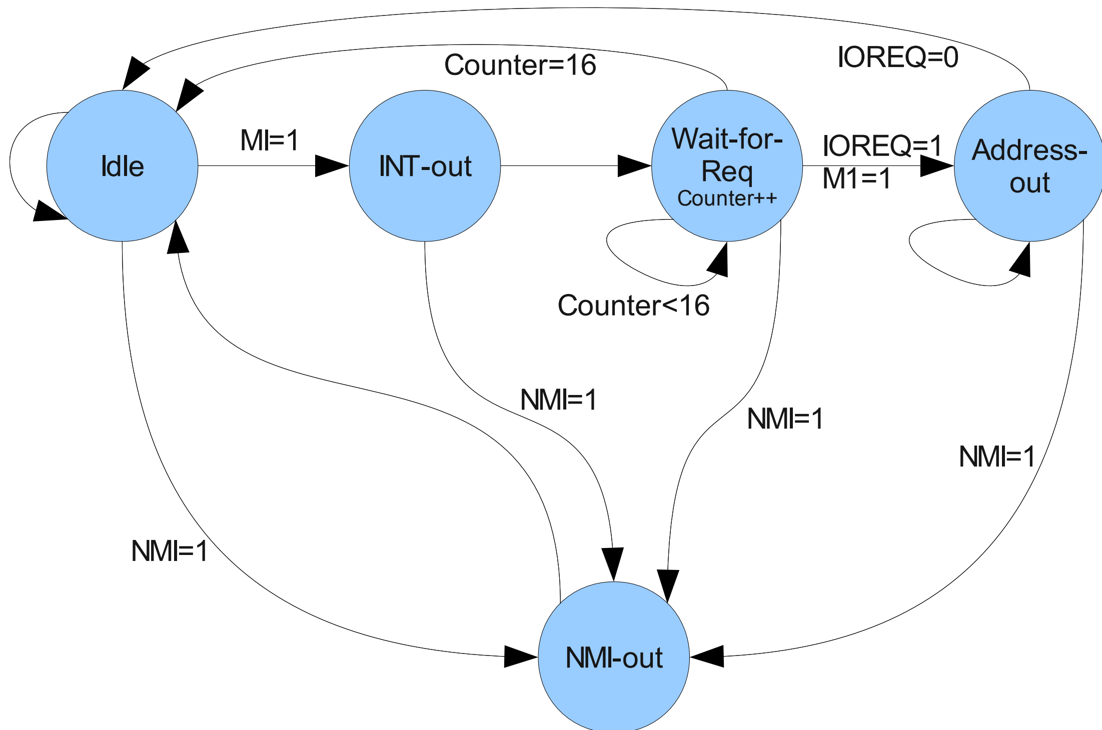

Abbildung 8: Zustandsdiagramm des Interrupt Controllers

Zustand	$\overline{\text{INT}}$	$\overline{\text{NMI}}$	Adresse
Idle	1	1	Nicht angelegt
MI-out	0	1	Nicht angelegt
Wait-for-Req	1	1	Nicht angelegt
Address-out	1	1	angelegt
NMI-out	1	0	Nicht angelegt

Tabelle 5: Zustände und Ausgänge des Interrupt Controllers

Das Zustandsdiagramm des Interrupt Controllers (*Abbildung 8*) zeigt die verschiedenen Zustände des Controllers und deren Ablauf in Abhängigkeit der Eingänge. Die dazu gehörige Belegung der Ausgänge ist in *Tabelle 5* beschrieben.

Wird ein NMI erkannt wird dieser unabhängig vom aktuellen Zustand des Controllers sofort an die CPU weitergeleitet und der Zustand des Interrupt Controllers auf *NMI-out* gesetzt.

Maskierbare Interrupts werden nur im *Idle* Zustand bearbeitet. Tritt ein Interrupt auf, obwohl der Controller in einem anderen Zustand ist, wird der Interrupt gespeichert und bearbeitet sobald der Controller wieder im *Idle* Zustand ist. Treten verschiedene Interrupts gleichzeitig auf oder sind verschiedene Interrupts gespeichert, wird der Interrupt mit der höchsten Priorität zu erst bearbeitet.

Beim Bearbeiten eines Interrupts wird der Zustand zunächst auf *INT-out* gesetzt und damit die MI-Leitung auf „0“ gelegt. Im *INT-out* Zustand wird der Status ohne Bedingung auf den *Wait-for-Req* Zustand gesetzt. Setzt die CPU im *Wait-for-Req* Zustand innerhalb von 16 Takten sowohl das M1, als auch das IOREQ Bit, wodurch sie signalisiert, dass sie die Interrupt Adresse auf dem Datenbus erwartet, wird der Zustand auf *Address-out* gesetzt und die Adresse je nach Interrupt ausgegeben. Kommt keine Anfrage der CPU, fällt der Controller in den *Idle* Zustand zurück. Wurde im Zustand *Address-out* die Adresse von der CPU gelesen, wird sie vom Datenbus genommen und der Controller springt wieder in den *Idle* Zustand.

Tabelle 6 zeigt die Geräte-Belegung der verschiedenen Interrupts und ihre Prioritäten. Der Controller besitzt noch fünf unbelegte Interrupt-Leitungen.

Interrupt	**Priorität**	**Gerät**
NMI	0	East Button
MI 0	1	North Button
MI 1	2	Timer
MI 2	3	Tastatur
MI 3	4	Nicht belegt
MI 4	5	Nicht belegt
MI 5	6	Nicht belegt
MI 6	7	Nicht belegt
MI 7	8	Nicht belegt

Tabelle 6: Interrupts des Z80 SoC (Je niedriger die Prioritätszahl, um so höher ist die Priorität.)

5.2.2 Der programmierbare Timer

Der Timer besteht aus einem 24-Bit-Zähler, der mit dem Systemtakt von 3,57 MHz arbeitet. Über den Port A0h lässt sich der Timer konfigurieren. Wird auf diesen der Wert „0“ geschrieben, wird der Timer deaktiviert. Jeder andere Wert aktiviert den Timer und wird gespeichert. Beim aktivierten Timer wird der Zähler so lange inkrementiert bis die höherwertigen acht Bit den gespeicherten acht Bit auf dem Port entsprechen. In diesem Fall löst das Modul den Interrupt 1 aus und setzt den Zähler zurück. Durch den Systemtakt ergeben sich mögliche Zeiten für den Timer von ca. 0,02 bis ca. 4,68 Sekunden.

5.2.3 Die Tracing Schnittstelle

Die Tracing Schnittstelle kann den Programmablauf anhalten und die Register, den aktuellen Speicherausschnitt und den Stack an einen über die serielle RS-232 Schnittstelle angeschlossenen Computer übertragen. Das Modul ist standardmäßig aktiviert und die Schrittweite auf „1“ gestellt, d.h. bereits beim ersten Befehl wird die Ausführung angehalten. Die Schrittweite gibt an, wie viele Instruktionen ausgeführt werden, bis das Tracing-Modul den Ablauf anhält. Ein Wert von „0“ deaktiviert das Tracing-Modul.

Das Modul zählt die Instruktionen der CPU anhand der M1 Leitung des Z80. Zusammen mit der MREQ Leitung wird mit der M1 Leitung das Laden einer Instruktion signalisiert. Dabei ist zu beachten, dass Instruktionen von mehr als einem Byte mehrere M1 Zyklen hervorrufen (siehe *Kapitel 6.2*).

Es werden 128 Byte des Speichers übertragen. Dabei wird darauf geachtet, dass dieses Speicherabbild etwa die 64 Byte vor und die 64 Byte hinter der aktuellen Progamm Counter Position enthält. Eine Ausnahme bilden Werte des Programm Counters unter 64. In diesem Fall werden die ersten 128 Byte des Speichers übertragen. Vom Stack werden die letzten 64 Byte übertragen.

Ist die Ausführung angehalten, führt ein Empfangen eines Wertes von *63h* über die serielle Schnittstelle oder das Drücken des South Button zur Fortsetzung des Programms.

Ein Halt-on-Interrupt (HOI)-Bit des Moduls gibt an, ob beim Auftreten eines Interrupts die Ausführung angehalten werden soll. Dieses Bit kann über die serielle

Schnittstelle durch Senden eines Wertes von *69h* aktiviert bzw. deaktiviert werden.

Tabelle 7 enthält die weiteren Befehle des Tracing-Moduls, die über die RS-232-Schnittstelle empfangen werden können. Die Befehle des Tracing-Moduls entsprechen nicht den Befehlen des Tracing-Programms (vgl. *Tabelle 9*). *Tabelle 8* zeigt, in welchem Format die Daten übertragen werden.

Übertragung	**Reaktion des Moduls**
63h	Fortfahren mit dem nächsten Schritt.
66h	Setzt die Ausführung fort und hebt die Schrittweite auf.
88h + zwei Byte	Setzt die Schrittweite. Die zwei Byte enthalten die Schrittweite im Big Endian Format (das höchstwertige Bit am Ende).
73h	Hält die Ausführung an.
69h	Setzt bei Übertragung das HOI Bit des Tracing-Moduls auf Eins. Eine erneute Übertragung setzt es wieder auf Null.
AAh	Sendet die Daten erneut.

Tabelle 7: Mögliche Befehle des Tracing-Moduls

Die Übertragungrate der RS-232 Schnittstelle beträgt 9.6 KBaud. Eine andere Übertragungsgeschwindigkeit ist nicht einstellbar.

Da die RS-232-Schnittstelle beim Übertragen keinen eigenen Takt mitsendet, arbeitet das Modul beim Empfang von Daten mit einer Taktrate von 154 KHz. Dies entspricht ungefähr dem 16fachen der Übertragungsrate. Wie *Abbildung 9* zeigt startet jede Übertragung eines Byte über die RS-232 Schnittstelle mit einer logischen

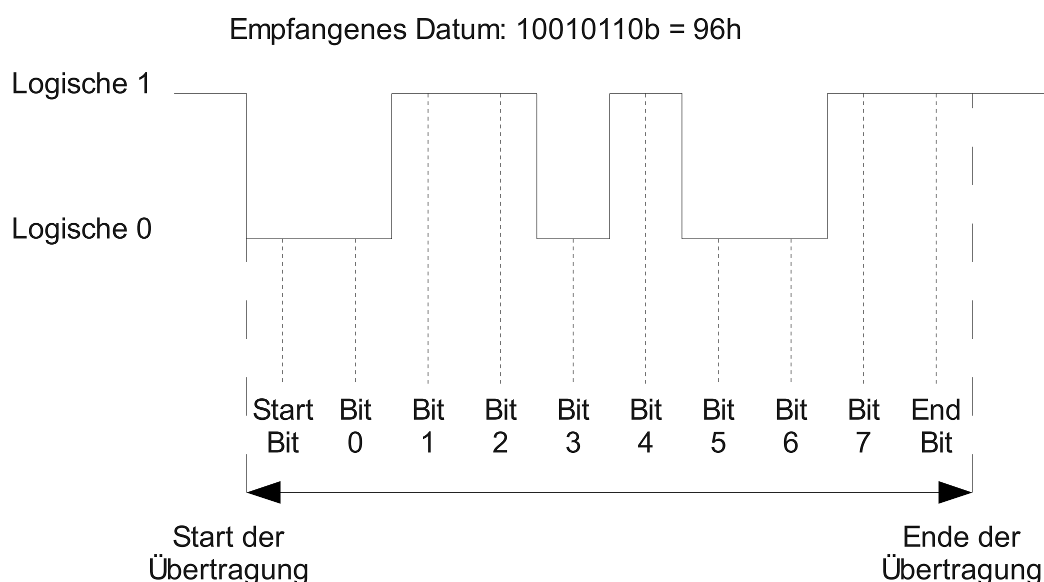

Abbildung 9: Empfangen eines Byte über die RS-232 Schnittstelle

„0“ und endet mit einer logischen „1“. Es werden also immer zehn Bit übertragen. Der Start einer Übertragung kann an einem Wechsel des Signals von einer logischen „1“ zu einer logischen „0“ erkannt werden. Mit dem Start einer Übertragung (logische „0“), wird ein 4-Bit-Zähler aktiviert. Bei einem Zählerwert von „8“, d.h. der Hälfte eines Bit-Signals, wird der Signalpegel der Rx-Leitung abgetastet und gespeichert. Wenn das letzte Bit keine logische „1“ ist, so wird die Übertragung als fehlgeschlagen bewertet und das Datum verworfen.

Beim Übertragen kann eine Taktrate von 9.6 KHz verwendet und die zu übertragenden Bit mit diesem Takt auf die Leitung gelegt werden. Hier muss ebenfalls das Protokoll mit dem Start- und End-Bit eingehalten werden.

Datum	Inhalt
00h	Register B
01h	Register C
02h	Register D
03h	Register E
04h	Register H
05h	Register L
06h - 07h	Register IX
08h	Register B'
09h	Register C'
0Ah	Register D'
0Bh	Register E'
0Ch	Register H'
0Dh	Register L'
0Eh - 0Fh	Register IY
10h	Akkumulator Register (A)
11h	Flag Register (F)
12h	Register A'
13h	Register F'
14h	Interrupt Register (I)
15h	Refresh Register (R)
16h - 17h	Program Counter (PC)
18h - 19h	Stack Pointer (SP)
1Ah - 1Bh	Speicherabbild Adresse
1Ch - 1Dh	Vorheriger Programm Counter
1Eh	Bit 0 – Interrupt, Bit 1 – NMI, Bit 2 – HOI
1Fh - 9Eh	Speicherabbild
9Fh - DEh	Stackabbild

Tabelle 8: Aufbau der gesendeten Daten des Tracing-Moduls

5.3 Das Tracing-Programm

Das Tracing-Programm stellt die über die serielle RS-232-Schnittstelle empfangenen Daten des SoC dar und ermöglicht die Kontrolle des Tracing-Moduls.

Die empfangen Daten werden in einem Array gespeichert. War die Übertragung erfolgreich, wird das Array analysiert und auf dem Bildschirm angezeigt. Wird eine angefangene Übertragung abgebrochen oder auf eine Anfrage an das SoC nicht innerhalb einer Sekunde geantwortet, sendet das Programm automatisch eine *resend* Anweisung an das SoC.

Auf dem Bildschirm wird ein Speicherbereich von 48 Byte, die letzten 14 Byte des Stacks sowie alle Register angezeigt.

Weiter enthält das Tracing-Programm einen integrierten Disassembler, der die Befehle im Speicherausschnitt in einen lesbaren Assembler-Code umwandeln und so für Menschen verständlicher darstellen kann. Dabei versucht das Programm Zeichenketten als solche zu erkennen und nicht zu disassemblieren. Da der aktuelle und der letzte Programm Counter Wert übertragen werden, werden die Instruktionen an diesen Adressen grundsätzlich disassembliert. Bei allen weiteren Adressen gilt, dass wenn die Werte vier aufeinander folgender Byte zwischen 20h und 80h liegen, diese den Beginn einer Zeichenkette markieren. Das Ende der Zeichenkette ist durch einen Wert kleiner 20h markiert. Dabei ist es möglich, dass es zu Fehlinterpretationen kommen kann.

Eine weitere Erklärung zum Umgang mit dem Tracing-Programm wird in Kapitel 8.2 gegeben.

6 Probleme

In diesem Kapitel sollen Probleme und Schwierigkeiten beschrieben werden, die sich während der Arbeit ergaben. Soweit die Probleme gelöst wurden, wird die Lösung angegeben.

6.1 Interrupt Annahme des Z80

Das Timing des Interrupt Controllers erwies sich als sehr schwierig. Während das NMI Signal vom Z80 immer angenommen und bis zur Behandlung gehalten wurde, wurden Maskierbare Interrupts nicht gespeichert und nur am Ende eines Befehls angenommen. Da die Ausführung einer CPU Instruktion keine feste Anzahl an Taktzyklen benötigt, konnte nicht im Voraus gesagt werden, wie lange die MI-Leitung aktiv gehalten werden muss.

Lag das Interrupt Signal genau für einen Takt an der CPU an, führte dies nur zu einer Behandlung ca. jeden dritten Interrupts, bei zwei Takten führte dies zu einer Behandlung etwa jedes zweiten Interrupts. Wurde allerdings das Interrupt Signal drei Takte gehalten, führte dies häufig zu einem Reset der CPU zur Speicherstelle 0. Aus dem Z80 User Manual war dieses Verhalten nicht nachzuvollziehen (siehe *Abbildung 10*).

Der Z80 gibt kein Acknowledge Signal bei erfolgreicher Annahme eines Interrupts.

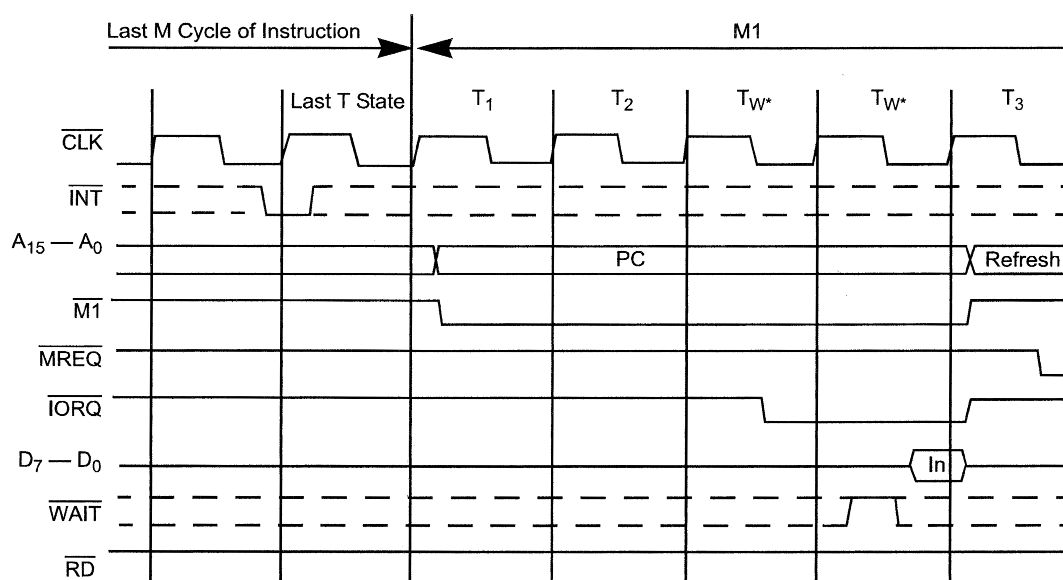

Abbildung 10: Timing-Diagramm einer maskierten Interrupt-Annahme[4]

Es gibt lediglich ein Signal, das anzeigt, ob die Adresse der Interrupt Routine erwartet wird. Dieses Signal kommt jedoch erst sehr spät. Versuche, das Interrupt Signal so lange aktiviert zu halten, bis die Anfrage der Adresse kommt, scheiterten, da hier bereits ein Reset stattgefunden hatte. Für den Interrupt Controller war nicht erkennbar, ob die CPU einen Interrupt angenommen hatte oder nicht, daher konnte auch keine Lösung realisiert werden, in der der Controller den Interrupt anlegte, bis er angenommen wurde.

Das Problem wurde gelöst, indem der Z80 Kern von *OpenCores* bearbeitet wurde, sodass ein Interrupt Signal bei gesetzten Interrupt-Enable-Flipflop gespeichert und gehalten wird, bis es bearbeitet wurde. Damit ist sichergestellt, dass jeder Interrupt bearbeitet wird, so lange das Interrupt-Enable-Flipflop eingeschaltet ist. Dieses Vorgehen entspricht, bis auf die Berücksichtigung des Interrupt-Enable-Flipflops, dem Vorgehen des Z80 Kerns bei einem NMI. Der Interrupt Controller kann jetzt das Interrupt Signal für genau einen Takt anlegen und sicher gehen, dass der MI detektiert wird.

Zum Verdeutlichen dieses Vorgehens ist hier der entsprechende Ausschnitt des VHDL Quellcodes aufgeführt. *IntE_FF1* ist der Status des Interrupt-Enable-Flipflops. *IntCycle* wird auf „1“ gesetzt, wenn die Interrupt-Behandlung beginnt. *INT_n* ist der negierte Interrupt-Eingang, *INT_s* ist das gehaltene Signal. Analog gilt gleiches für *NMICycle*, *NMI_n* und *NMI_s*.

```
if IntCycle = '1' or IntE_FF1 = '0' then
        INT_s  <= '0';
elsif INT_n = '0' and OldINT_n = '1' then
        INT_s <= '1';
end if;
OldINT_n := INT_n;
if NMICycle = '1' then
        NMI_s <= '0';
elsif NMI_n = '0' and OldNMI_n = '1' then
        NMI_s <= '1';
end if;
OldNMI_n := NMI_n;
```

6.2 Zählen der Instruktionen in dem Tracing-Modul

Das Tracing-Modul sollte in der Lage sein, eine bestimmte Anzahl an Instruktionen zu zählen bis es die weitere Ausführung anhält. Die Takte zu zählen ist nicht ausreichend, da Instruktionen nicht immer die gleiche Anzahl an Taktzyklen zur Bearbeitung benötigen. Hierfür kann das Signal M1 verwendet werden, das negiert als Ausgang des Z80 vorliegt.

Dieses Signal wird zusammen mit dem MREQ Signal gesetzt, wenn eine Instruktion aus dem Speicher gelesen wird. Durch Zählen dieser M1-Zyklen ist es möglich die Instruktionen zu zählen.

Beim Testen des Tracing-Moduls kam es jedoch vor, dass der übertragene Programm Counter einen ungültigen Wert enthielt. Er zeigte eine Adresse an, die innerhalb eines 16-, 24- oder 32-Bit Befehls lag.

Es stellte sich heraus, dass längere Instruktionen, wie das Setzen des Stack Pointers und Operationen mit den 16 Bit Registern IX und IY mehr als einen M1-Zyklus erzeugen. Diese Befehle erkennt man am ersten Instruktionsbyte. Diese sind entweder CBh, EDh, DDh oder FDh. Das einfache Zählen der M1-Zyklen war daher nicht ausreichend, sondern die Art der Befehle musste zusätzlich berücksichtigt werden.

Indem das Tracing-Modul die Instruktionen speichert, die während eines M1-Zyklus auf dem Datenbus liegen, kann es nun erkennen, ob ein M1-Zyklus zur vorherigen Instruktion gehört und der Zähler daher nicht inkrementiert werden soll. Hierbei ist das Timing des Speichers zu beachten. Die Instruktion liegt nicht mit der Flanke des M1-Signals zeitgleich auf dem Datenbus, sondern erst nach der nächsten fallenden Taktflanke.

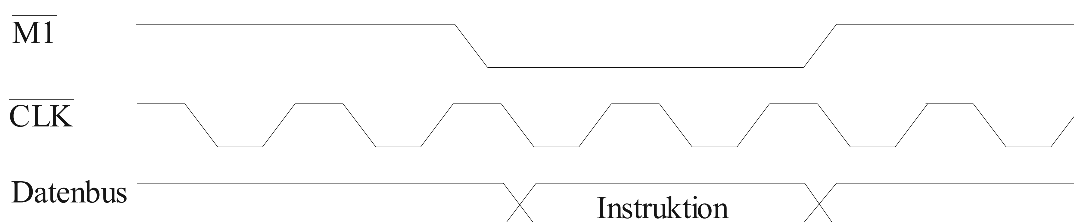

Abbildung 11: Zeitlicher Ablauf eines M1-Zyklus zum Laden einer Instruktion

6.3 Video-RAM Timing

Im Zuge dieser Arbeit wurde die Darstellung der Schrift auf dem VGA Bildschirm verkleinert bzw. die Zeichendarstellung der echten Auflösung von 640x480 Pixel angepasst (siehe *Kapitel 5.1.4, Mode 1*). Bei den ersten Versuchen zeigte sich ein Fehler in der Zeichendarstellung. Zunächst blieben die ersten beiden Pixelspalten eines Zeichens leer und wurden erst in den ersten beiden Pixelspalten des folgenden Zeichens angezeigt (siehe *Abbildung 12*).

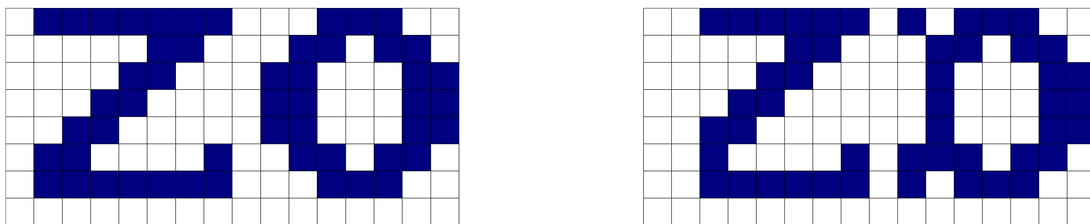

Abbildung 12: Falsche Darstellung der Zeichen. Links: Die Zeichen, wie sie dargestellt werden sollten. Rechts: Die Zeichen, wie sie dargestellt wurden.

Das ursprüngliche VGA-Modul von *OpenCores* hatte nicht berücksichtigt, dass das Video-RAM einen Taktzyklus benötigt, um die Adresse anzunehmen und einen weiteren Taktzyklus um das Datum zurück zu liefern. Da die erste Spalte eines Zeichens immer leer ist und das ursprüngliche Design die Auflösung künstlich halbierte – jedes Pixel eines Zeichens wurde viermal gezeichnet – fiel dieser Fehler nicht auf.

Nach einigen Veränderungen sah es zunächst so aus, als wenn das Problem gelöst wäre. Der Zähler, durch den die Pixelspalte und die Zeichenspalte bestimmt wird, wurde dahingehend geändert, dass der Zählerwert an der Pixelspalte um zwei dekrementiert wurde. Allerdings zeigte sich nun in der ersten Pixel-Spalte des Bildschirms die letzte Pixel-Spalte der Zeile.

Durch die Änderungen am Zähler wurden die Zeichen zwar alle richtig dargestellt, allerdings nicht an den richtigen Pixel-Positionen, sondern um zwei Pixel verschoben. Da heutige Bildschirme sich nicht genau an die Synchronisierungs-Signale der VGA-Schnittstelle halten, sondern sich flexibel den übertragenen Pixeln anpassen können, ist diese Verschiebung zunächst nicht aufgefallen. Bei einem Zeichen in der letzten Spalte des Bildschirms fiel aber schließlich auf, dass die

letzten beiden Pixelspalten des letzten Zeichen erst in den ersten Spalten gezeigt wurden.

Es mussten einige Änderungen beim Timing des Schreibens auf die VGA Schnittstelle geändert werden. So wurden das Synchronisierungssignal und die Pixelsignale um zwei Takte verschoben. Die letzten beiden Pixelspalten des letzten Zeichens werden nun in den letzten Spalten des Bildschirms angezeigt und die ersten beiden Spalten, die ebenfalls die letzten beiden Pixelspalten des letzten Zeichens enthielten wurden deaktiviert.

6.4 Shift-, Caps- und Alt-Gr-Erkennung der Tastatur

6.4.1 Caps-Erkennung

Im ursprünglichen Zustand des PS/2 Controller Moduls wurde die Shift-Taste gar nicht und die Caps-Taste nur sehr schlecht unterstützt. Das heißt das Drücken der Caps-Taste führte augenscheinlich nicht immer zum Aktivieren bzw. Deaktivieren des Capslock. Zudem wurde das Capslock-Lämpchen auf der Tastatur nicht angesteuert. Es war also für den Benutzer des Boards nicht zu erkennen, ob das nächste Zeichen als großes oder kleines Zeichen erkannt werden würde.

Der Controller negierte bei jedem Eintreffen des Scancodes der Caps-Taste ein Signal, welches den Capslock Zustand beinhaltete. Da jedoch bei längerem Halten so wie beim Loslassen der Scancode erneut von der Tastatur gesendet wird, war es eine Frage des Zeitlichen, ob der Capslock aktiviert oder deaktiviert wurde.

Die Erkennung des Capslock wurde dahingehend geändert, dass eine Detektierung des Caps-Tasten-Scancodes den aktuellen Capslock-Wert negiert und ein Signal im PS/2-Controller positiv gesetzt wird, damit ein weiteres Übertragen des Caps-Tasten-Scancodes nicht erneut zum Negieren des aktuellen Capslock-Wertes führt. Erst wenn zwischenzeitig ein anderer Scancode übertragen wird, führt dies zum Rücksetzen dies Signals und lässt damit beim erneuten Drücken der Caps-Taste wieder ein Negieren des Capslock-Wertes zu.

Durch die gleiche Methode konnte auch ein Einrasten der Num-Taste realisiert werden. Allerdings wird dieser Wert bisher noch nicht vom Tastatur Modul benutzt und der Nummernblock wird nicht unterstützt.

6.4.2 Shift- und Alt-Gr-Erkennung

Das ursprüngliche Design des PS/2-Tastatur-Controllers unterstützte das Protokoll der MF2-Tastaturen, welche heute den Standard bilden, nicht vollständig. So wurden Breakcodes und erweiterte Scancodes nicht erkannt. Es ist außerdem mit diesem Modul nicht möglich Daten an die Tastatur zu senden, um z.B. das Capslock-Lämpchen einzuschalten.

Das Protokoll der MF2-Tastatur ist wie folgt aufgebaut:
Sobald eine Taste gedrückt wird, wird ihr Scancode übertragen. Bei einem längeren Halten der Taste wird der Scancode wiederholt übertragen. Beim Loslassen der Taste wird der Scancode erneut mit vorhergehenden Breakcode (F0h) gesendet.

Mit Hilfe des Breakcodes ist es möglich zu erkennen, ob eine Taste, in diesem Fall die Shift- und Alt-Gr-Taste, gedrückt oder losgelassen wurde. Wird über die PS/2-Schnittstelle ein Wert von *F0h* empfangen, wird ein Signal gesetzt, dass der folgende Scancode ein Breakcode ist. Das Shift-Signal wird also gesetzt, wenn der Scancode für die Shift-Taste empfangen wird, beim Empfangen des Breakcodes wird es wieder auf „0“ gesetzt.

Nachdem durch diese Implementierung die Shift-Taste unterstützt wurde, war es nicht weiter schwer auch eine Unterstützung für die Alt-Gr-Taste umzusetzen. Hierfür wird jedoch der erweiterte Scancode benötigt, da die Alt-Gr-Taste sonst den gleichen Scancode besitzen würde wie die Alt-Taste. Der erweiterte Scancode setzt vor den normalen 1-Byte-Scancode ein weiteres Byte (E0h) zur Unterscheidung.

Der Folgende VHDL Programm Ausschnitt zeigt, wie die Breakcodes (*breakcode*) und erweiterten Scancodes (*Ecode*) erkannt werden.

```
if scan_code_sig(7 downto 5) = "111" then
	if scan_code_sig = x"F0" then
		breakcode	<= '1';
	end if;
	if scan_code_sig = x"E0" then
		Ecode		<= '1';
	end if;
else
	Ecode		<= '0';
	breakcode	<= '0';
end if;
```

Damit erkennbar gemacht werden kann, welche Steuertasten gedrückt bzw. welche Signale eingerastet sind, können diese Werte über den Port 81h ausgelesen werden.

Ein Problem konnte nicht gelöst werden. So führt ein Drücken der Steuer- und der Shift-Tasten dazu, dass die nächste Taste nicht beim Drücken erkannt wird, sondern erst, bei längerem Drücken – also wenn der Scancode zum wiederholten Male gesendet wird. Das gleiche gilt, wenn eine Steuer- bzw. Shift-Taste losgelassen wird. Diesen Fehler beinhaltete schon das ursprüngliche Modul. In dieser Arbeit konnte nicht geklärt werden, wie dieser Fehler zustande kam.

7 Programme

7.1 Beispielprogramm

Das Beispielprogramm soll einige Möglichkeiten des SoC zeigen und gleichzeitig mit Hilfe der Tracing Schnittstelle die Funktion des Stack und den Ablauf von Sprüngen und Interrupts verdeutlichen.

Zunächst wird von dem Programm der Stack eingerichtet, indem der Stackpointer gesetzt wird. Ohne einen gesetzten Stackpointer kann die CPU nach einem Interrupt oder einem anderen Funktionsaufruf nicht wieder zur richtigen Adresse zurück springen, da auf diesem die Rücksprungadresse gespeichert wird. Zum Setzten des Stack wird eine Funktion des SoC verwendet: In der Speicherstelle 7FDEh befindet sich die höchste RAM Adresse des SoC.

```
LD      SP,     (7FDEh)         ; Stack Einrichten
```

Als nächstes wird das Interrupt Register gesetzt und die Adressen für die Interrupt Routinen eingetragen. Zum Schluss wird der Interrupt Modus auf 2 gestellt und Interrupts werden eingeschaltet. Es folgt ein Sprung zum Hauptprogramm.

```
DI                              ; Interrupts ausschalten
LD      A,      80h             ; Setzen des Interrupt Vektors auf 80h
LD      I,      A
LD      BC,     taster ; Setzen der Interrupt Routinen für den Taster,
LD      (8000h), BC
LD      BC,     timer           ; den Timer
LD      (8002h), BC
LD      BC,     keyboard        ; und das Keyboard.
LD      (8004h), BC
LD      A,      0
OUT     (0A0h),       A         ; Setze den Timer auf 0
IM      2                       ; Setzt den Interrupt Mode 2
EI                              ; Interrupts zulassen
JP      begin                   ; Sprung zum Hauptprogramm
```

An der Speicherstelle 0066h befindet sich die Routine für nicht-maskierbare Interrupts. Sie dient dem Umschalten der Bildschirmausgabe von 40x30 auf 80x60 Zeichen und zurück. Die Funktion *taster* dient zum Ändern der Bildschirmfarben.

Im Hauptprogramm wird zunächst eine Ausgabe auf den Bildschirm gemacht. Darauf befindet sich das Programm in einer Schleife. Mit dem Drehknopf kann die Bildschirmausgabe im 40x30 Modus nach links und nach rechts verschoben werden.

Mit der Tastatur kann man auf dem Bildschirm schreiben. Das Drücken der Tastatur ruft durch den ausgelösten Interrupt die Funktion *keyboard* auf. Diese gibt das gedrückte Zeichen auf dem Bildschirm aus und aktiviert den Timer. Dieser ruft durch einen Interrupt die Funktion *timer* auf, welche das Zeichen wiederholt auf den Bildschirm schreibt. Erkennt die Funktion *keyboard* ein Loslassen der Taste, wird der Timer deaktiviert.

Auf dem Board wird mit Hilfe der LEDs angezeigt, welche Steuertasten gerade gedrückt werden und ob Capslock und Numlock aktiviert sind.

Der komplette Quellcode des Beispielprogramms kann im Anhang *D - Quellcode des Beispielprogramms* (Seite 47 ff.) eingesehen werden. Es befindet sich im S3E Unterverzeichnis „rom“ unter dem Namen „keyint.asm“. Eine vorgefertigte Bit-Datei zur Konfiguration des FPGA befindet sich im Unterverzeichnis „bitfiles“ unter dem Namen „Z80Beispiel.bit“.

Mit Hilfe des Tracing-Programms lässt sich der Programmablauf nachvollziehen. Bei Vergleichsoperationen der CPU ist zu erkennen, wie die verschiedenen Flags im Flag-Register gesetzt werden. Bei Funktionsaufrufen kann man z. B. auf dem Stack die Rücksprungadresse sehen. So kann man die Funktionsweise des Systems besser verstehen.

7.2 Ping-Pong-Spiel

Das entwickelte Ping-Pong-Spiel ist mit 783 Byte das größte Programm für das SoC und geht damit an die Grenzen des FPGAs. Mit der VHDL-ROM-Datei dieses Spiels werden 92 Prozent der Slices des FPGAs genutzt. Die Synthetisierung, Implementierung und anschließende Konfiguration dauern ca. zehn Minuten.

Mit *b* kann ein Spiel gestartet bzw. neu gestartet werden. Der linke Schläger wird mit *w* nach oben und mit *s* nach unten navigiert. Es wird bis zehn gespielt.

Da das R Register als Quelle für Zufallszahlen genutzt wird, ist es möglich, dass das Spiel in eine Schleife gerät und ohne Eingabe des Benutzers ewig weiter läuft. Allerdings wird bereit durch den Druck einer Taste diese Schleife wieder verlassen.

Das Programm befindet sich im S3E Unterverzeichnis „rom“ unter dem Namen „pong.asm“. Eine vorgefertigte Bit-Datei zur Konfiguration des FPGA befindet sich im Unterverzeichnis „bitfiles“ unter dem Namen „Z80pong.bit“.

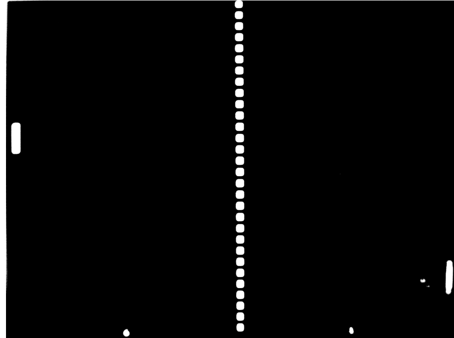

Abbildung 13: Ping-Pong-Spiel auf dem Z80 SoC

8 Bedienungsanleitung

8.1 Laden des Designs

Das Board muss zum Laden des Designs über ein USB-Kabel an den PC angeschlossen und eingeschaltet sein. Zuerst muss das Programm *iMPACT* gestartet werden (Start – Ausführen – impact). Gleich zu Beginn wird gefragt, welches Projekt geladen werden soll. Falls noch nicht geschehen wird hier über den Browser die Datei *S3E.ipf* im Verzeichnis des Designs gewählt.

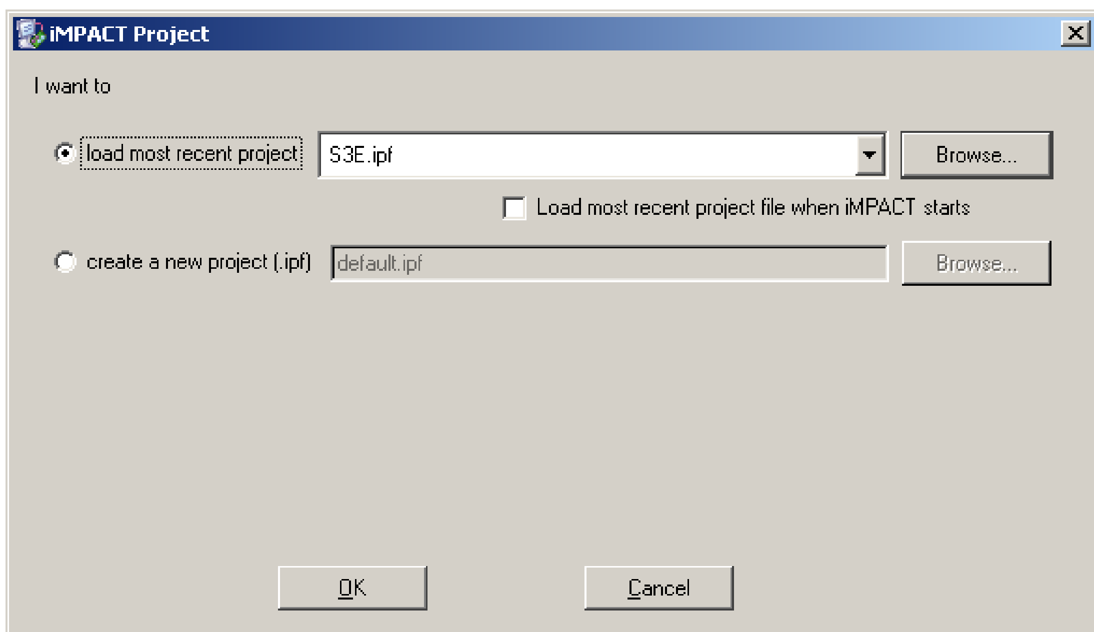

Abbildung 14: Auswählen des Projektes beim Start von iMPACT.

Im *Boundary Scan* Fenster sieht man drei Chips welche auf dem Board programmiert werden können. Es wird der erste Chip *xc3s500e* gewählt. Wenn dort noch nicht *z80soc_top.bit* als zu ladende Datei ausgewählt ist, kann diese über einen Rechtsklick und *Assign New Configuration File...* ausgewählt werden. In dieser Bit-Datei ist das zuletzt implementierte Design. Im Unterordner „bitfiles“ des Designs befinden sich die vorgefertigsten Bit-Dateien für das Beispielprogramm und das Ping-Pong-Spiel.

Ist die richtige Datei gewählt, wird mit der rechten Maustaste auf den Chip geklickt und *Program* gewählt. Das Design wird nun auf den Chip geladen und die Ausführung beginnt.

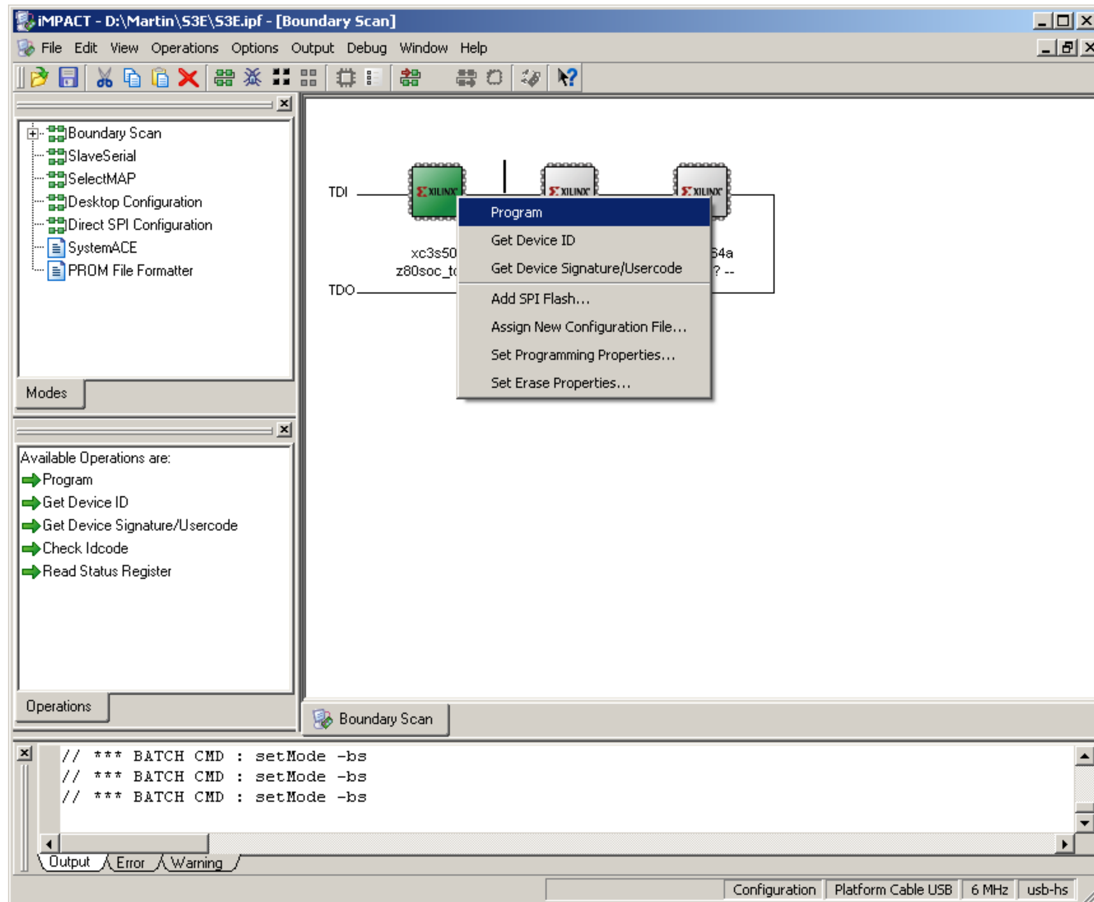

Abbildung 15: Laden des Designs in iMPACT

8.2 Benutzen des Tracing-Programms

Das Tracing-Programm ist eine Konsolen-Anwendung für Windows. Es befindet sich im Verzeichnis „tracing“ des SoC Designs und trägt den Namen *z80soc-control.exe*.

Beim Starten muss zunächst die Nummer der Schnittstelle angegeben werden. Anschließend befindet man sich in der Anzeigemaske.

Bei fehlgeschlagenen Übertragungen oder wenn das SoC läuft bevor das Tracing-Programm gestartet wurde, muss mit *r* die erneute Übertragung angefordert werden. In der Regel stellt das Programm automatisch eine *resend* Anfrage, wenn eine begonnene Übertragung mehr als eine Sekunde benötigt. Alle Befehle sind in der ersten Zeile des Programms sichtbar (siehe *Abbildung 16*) und in *Tabelle 9* nachzusehen.

In der Ausgabemaske sieht man oben links die Register und darunter das alternative Register Set. Rechts ist der Stack aufgelistet. Es folgt ein Speicherausschnitt und der

letzte sowie der aktuelle Befehl (gelb markiert), gefolgt von den nächsten Befehlen im Speicher.

Taste	Beschreibung
q	beendet das Programm
c	setzt die Ausführung fort
h	hält den laufenden Betrieb des SoC an
s	fordert Sie auf ,die neue Schrittweite einzugeben
l	löscht die Schrittweite und setzt die Ausführung fort
i	stellt ein, ob bei Interrupts angehalten werden soll oder nicht
r	fordert das SoC auf die Daten noch einmal zu senden

Tabelle 9: Befehle des Tracing-Programms

Der integrierte Disassembler versucht Zeichenketten automatisch zu erkennen (siehe *Abbildung 16, letzte Zeile*). Dabei kann es jedoch zu Fehlinterpretationen der Maschinensprache kommen. Werden bei vier aufeinander folgenden Byte Werte erkannt, die Buchstaben oder Zahlen repräsentieren, wird angenommen, dass es sich um eine Zeichenkette handelt und sie wird bis zum Auftreten eines Wertes unter 20h als solche ausgegeben. Der aktuelle und der letzte Befehl sind jedoch immer bekannt und werden daher stets immer als konkrete Assembler Befehle dargestellt.

```
D:\Martin\z80socc\z80soc-control.exe
Quit   Continue   Halt   Set Steps   Clear Steps   Interrupt   Resend
Register:                                                   Stack:
A  = 80     B  = 01     C  = 17     D  = 00     E  = 00     BFFE: 00   C005: FF
H  = 00     L  = 00     I  = 80     R  = 0D                 BFFF: 00   C006: FF
IX = 0000   IY = 0000   PC = 0017   SP = BFFE               C000: FF   C007: FF
Flags:  CNPXHYZS        INT      NMI      HOI               C001: FF   C008: FF
Alternate Registers:                                        C002: FF   C009: FF
A' = FF     B' = 00     C' = 00     D' = 00     E' = 00     C003: FF   C00A: FF
H' = 00     L' = 00     F' = CNPXHYZS                       C004: FF   C00B: FF
Memory: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
000.    ED 7B DE 7F F3 3E 80 ED 47 01 0C 01 ED 43 00 80
001.    01 17 01 ED 43 02 80 01 3C 01 ED 43 04 80 3E 00
002.    D3 A0 ED 5E FB C3 8A 00 5A 38 30 20 53 6F 43 20     .Z80 SoC

0013    LD      (8002h), BC
0017    LD      BC, 13Ch
001A    LD      (8004h), BC
001E    LD      A, 0h
0020    OUT     (A0h), A
0022    IM 2
0024    EI
0025    JP      8Ah
0028    db      "Z80 SoC - Martware 1.0", Dh
```

Abbildung 16: Ansicht des Tracing-Programms

8.3 Erstellen eines ROM Images

Das Programm eines ROM Images muss in Assembler geschrieben werden. Es kann ein Editor nach Wahl benutzt werden. Die Datei sollte am besten im Unterverzeichnis

„rom“ des Z80 SoC Designs gespeichert werden.
In diesem Verzeichnis befindet sich auch der Z80 Assembler, so wie das *asm2vhdl* Skript.

Um aus dem Assembler Code eine VHDL-ROM-Datei zu erstellen, wechselt man in der Konsole (Start – Ausführen... - cmd) in das *rom* Verzeichnis und gibt dort

```
asm2vhdl.bat datei.asm
```

ein. Das Skript assembliert die Datei und erstellt eine VHDL Datei, die in das übergeordnete Verzeichnis kopiert wird.

Das Design muss neu synthetisiert und implementiert werden und kann dann zur Konfiguration des FPGA übertragen werden. Zum Synthetisieren muss die Datei *z80soc.ise* geöffnet werden.

Im *Sources*-Rahmen muss das *Z80SOC_TOP* Modul ausgewählt sein, dann kann im *Processes*-Rahmen mit einem Doppelklick auf *Configure Targes Device* der Synthetisierungsvorgang gestartet werden.

Wenn das Design fertig implementiert ist, wird automatisch *iMPACT* geladen und die Konfiguration kann wie in Kapitel 8.1 beschrieben übertragen werden.

9 Zusammenfassung und Ausblick

Im Zuge dieser Arbeit wurde ein lauffähiges Z80 System-on-a-Chip entwickelt, welches durch ein integriertes Tracing-Modul zur Demonstration von Abläufen in einem Computer benutzt werden kann. Durch einen Interrupt Controller werden bis zu neun verschiedene Interrupts unterstützt. Ein Beispielprogramm nutzt die entwickelten Komponenten dieses Systems und kann mit Hilfe des Tracing-Moduls und des Tracing-Programms dazu verwendet werden, um die Funktionsweise des Stack, der Interrupts und verschiedener Sprünge zu demonstrieren.

Es gibt jedoch noch Komponenten am und im System, die ergänzt und verbessert werden können. So könnte z.B. eine Anbindung von externen Flash-Speicher realisiert werden, damit für neue Programme auf dem Board nicht das komplette Design neu implementiert werden muss, was bis zu 15 Minuten dauern kann. Hierfür wäre es möglich ein UART-Modul für die zweite RS-232-Schnittstelle des Boards zu implementieren und für Programme zur Verfügung zu stellen. So könnte ein Standard-Programm im ROM Programme von der Schnittstelle empfangen, im Flash-Speicher ablegen und starten.

Außerdem wäre es möglich das Video-Modul so zu erweitern, sodass verschiedene Schriftsätze möglich wären und jedes Zeichen einen eigenen Farbwert erhalten könnte. Schließlich könnte das PS/2 Modul erweitert werden, sodass auch Daten an die Tastatur gesendet werden könnten. Auch das bekannte Problem mit den Steuertasten könnte durch eine Erweiterung behoben werden.

A Literatur Verzeichnis

[1] Spartan-3E FPGA Starter Kit Board User Guide, 20. Juni 2008

[2] OpenCores Website: http://www.opencores.org

[3] Z80 Assember Website: http://savannah.nongnu.org/projects/z80asm

[4] Z80 User Manual, 4. Dezember 2004

B Abkürzungen

ASCII	American Standard for Information Interchange
BRAM	Block Random Access Memory
CISC	Complex Instruction Set Computing
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
Hex	Hexadezimal
ISE	Integrated Software Environment
I/O	Input/Output
LCD	Liquid Crystal Display
LED	Light Emitting Diode
MI	Maskable Interrupt
NMI	Non-maskable-Interrupt
RAM	Random Access Memory
ROM	Read Only Memory
SoC	System-on-Chip
UART	Universal Asynchronous Receiver Transmitter
VGA	Video Graphic Adapter
VHDL	Very High Speed Integrated Circuit Hardware Description Language

C Befehlssatz des Z80

ADC X, Y	Addiert X und Y und das Carry Bit des Flag-Registers
ADD X, Y	Addiert X und Y
AND X	AND Verknüpfung von X mit dem Register A
BIT X, Y	Testet das Bit X, des Registers Y
CALL X	Springt zur Speicherstelle X und Speichert die Adresse des nächsten Befehls auf dem Stack.
CALL Y, X	Springt zur Speicherstelle Y, wenn Bedingung X erfüllt ist, und Speichert die Adresse des nächsten Befehls auf dem Stack.
CCF	Negieren des Carry Flag
CP X	Vergleicht X mit dem Register A
CPD	Vergleicht den Speicherbereich, der durch das HL Registerpaar indiziert ist mit dem Register A
CPDR	Vergleicht den Speicherbereich, der durch das HL Registerpaar indiziert ist. Ist der Speicherbereich und A nicht gleich sind oder BC gleich Null ist, werden HL und BC decrementiert.
CPI	Wie CPDR nur, dass das HL Registerpaar inkrementiert wird
CPIR	Wir CPIR. Wenn der Speicherbereich ungleich den Register A ist wird der Befehl wiederholt
CPL	Der Inhalt des Registers A wird invertiert
DAA	Wenn das Register A größer als 09h ist, wird 06h addiert, um eine Binär Codierte Dezimalzahl zu erhalten
DEC X	Decrementiert das Register X
DI	Schaltet Interrupts aus
DJNZ X	B wird decrementiert und ein relativer Sprung um X vollzogen, wenn B ungleich Null ist

EI	Schaltet Interrupts ein
EX X, Y	Tauscht die Werte der Register X und Y. Bei EX AF werden die Inhalte von A und F mit ihrem alternativen Registerset vertauscht
EXX	Tauscht die Registersätze
HALT	Hält die weiter Programmausführung an, bis ein Interrupt oder Reset auftritt
IM X	Stellt den Interrupt Mode X ein
IN X, Y	Speichert den Wert auf dem Port Y im Register X
INC X	Inkrementiert das Register X
IND	Liest den durch C adressierten Port, speichert den Wert im Register A und decrementiert B.
INDR	Wie IND mit Wiederholung, bis B gleich Null
INI	Liest den durch C adressierten Port, speichert den Wert im Register A und inkrementiert B
INIR	Wie INI mit Wiederholung, bis B gleich Null
JP X	Springt zur Speicherstelle X
JP X, Y	Springt zur Speicherstelle Y, wenn X erfüllt ist. X steht für C (Carry Flag ist gesetzt), NC (Carry Flag ist nicht gesetzt), Z (Zero Flag ist gesetzt), NZ (Zero Flag ist nicht gesetzt), P (Positives Vorzeichen), M (Negatives Vorzeichen), PO (Parity Bit ist nicht gesetzt), PE (Parity Bitt ist gesetzt)
JR X	Relativer Sprung um X
JR X, Y	Relativer Sprung um Y, wenn X erfüllt ist. Für X sieht JP X, Y, allerdings sind nur C, NC, Z und NZ erlaubt
LD X, Y	Speichert den Wert Y in X
LDD	Speichere den Inhalt der durch DE adressierten Speicherzelle in der durch HL adressierten Speicherzelle und decrementiere beide Registerpaare und das Registerpaar BC um eins

LDDR	Wie LDD so lange wiederholt bis BC gleich Null
LDI	Speichere den Inhalt der durch DE adressierten Speicherzelle in der durch HL adressierten Speicherzelle, inkrementiere beide Registerpaare und decrementiere das Registerpaar BC
LDIR	Wie LDI so lange wiederholt bis BC gleich Null
NEG	Der Wert des Registers A wird negiert. (2er Komplement)
NOP	Keine Operation
OR X	OR Verknüpfung des Wertes X mit dem Register A
OTDR	Schreibt den durch das HL Registerpaar adressierten Speicherbereich auf den durch das Register C adressierten Port und dekrementiert HL und B. Wiederholung bis B gleich Null
OTIR	Wie OTDR, nur dass HL inkrementiert wird
OUT X, Y	Schreibt den Wert Y auf den Port X
OUTD	Schreibt den durch das HL Registerpaar adressierten Speicherbereich auf den durch das Register C adressierten Port und dekrementiert HL und B
OUTI	Wie OUTD, nur das HL inkrementiert wird
POP X	Holt ein Word vom Stack und speichert es im Registerpaar X
PUSH X	Speichert das Registerpaar X auf dem Stack
RES X, Y	Setzt das Bit Y der Registers X auf Null
RET	Springt zu der Adresse auf dem Stack
RETI	Springt zu der Adresse auf dem Stack und setzt ein Signal auf den I/O Bus, dass die Interrupt Routine beendet wurde.
RETN	Springt zu der Adresse auf dem Stack und setzt das Interrupt-Enable-Flipflop auf seinen vorherigen Wert.
RL X	Verschiebt die Bit von X um eins nach links. Bit 7 rückt in das Carry Flag, das Carry Flag rückt in das Bit 0
RLA	Wie RL mit dem Register A

RLC X	Verschiebt die Bit von X um eins nach links. Bit 7 rückt in das Carry Flag und in Bit 0.
RLCA	Wie RLC mit dem Register A.
RLD	Die unteren vier Bit des durch HL adressierten Speichers werden in die oberen vier Bit der gleichen Adresse kopiert. Die oberen vier Bit der Adresse werden in die unteren vier Bit des Registers A und diese in die unteren vier Bit des adressierten Speichers kopiert.
RR X	Wie RL, nur wird nach rechts verschoben.
RRA	Wie RLA, nur wird nach rechts verschoben.
RRC X	Wie RLC, nur wird nach rechts verschoben.
RRCA	Wie RLCA, nur wird nach rechts verschoben.
RRD	Wie RLD, nur wird nach rechts verschoben.
RST X	Führt einen Reset zur Speicherstelle X aus. Diese muss ein vielfaches von acht sein und zwischen 0 und 38h liegen.
SBC X, Y	Subtrahiert Y mit dem Carry Bit des Flag-Registers von X
SCF	Setzt das Carry Flag.
SET X, Y	Setzt das Bit Y des Registers X auf eins.
SLA X	Verschiebt den Inhalt X um ein nach links. Bit 0 wird 0, Bit 7 kommt in das Carry Flag.
SRA X	Wie SLA X, nur wird nach rechts verschoben.
SRL X	Verschiebt den Inhalt von X um eins nach rechts. Bit 7 wird 0, Bit 0 kommt in das Carry Flag.
SUB X, Y	Subtrahiert Y von X
XOR X	XOR Verknüpfung von X mit dem Register A

D Quellcode des Beispielprogramms

```
1       LD    sp,   (7FDEh)     ; Stack Einrichten
2       DI
3       LD    A,    80h         ; Setzen des Interrupt Vectors auf 80h
4       LD    I,    A
5       LD    BC,   taster      ; Setzen der Interrupt Routinen
6       LD    (8000h), BC
7       LD    BC, timer
8       LD    (8002h), BC
9       LD    BC, keyboard
10      LD    (8004h), BC
11      LD    A,    0
12      OUT   (0A0h),    A
13      IM    2
14      EI                      ; Interrupts zulassen
15      JP    begin
16
17    BildText:
18      db    "Z80 SoC - Martware 1.0", 13
19      db    "=======================", 13
20      db    "> ", 0
21
22      db    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
23
24    nmi:                            ; Routine für den Nichtmaskierbaren
25      DI                            ; Interrupt. Muss immer an der Speicher-
26      PUSH  AF                      ; Stelle 0x66 sein.
27      IN    A,    (94h)
28      CP    0
29      JR    Z,    nmi_one
30      LD    A,    0
31      OUT   (94h),     A
32      JR    nmi_ende
33    nmi_one:
34      LD    A,    1
35      OUT   (94h),     A
36    nmi_ende:
37      POP   AF
38      EI
39      RETI
40
41    db "Hauptprogramm", 0
```

```
42
43  begin:
44     LD   A,    0
45     OUT  (91h),     A
46     OUT  (92h),     A
47     OUT  (94h),     A
48     LD   A,    0Fh
49     OUT  (93h),     A
50     LD   BC,   BildText
51     CALL VGAPrint
52     LD   BC,   0
53     LD   (8100h),   BC
54     IN   A,    (70h)
55     LD   B,    A
56     LD   A,    0
57     OUT  (94h),     A
58  ende:
59     EI
60     IN   A,    (81h)
61     OUT  (01h),     A
62     IN   A,    (70h)
63     CP   B
64     JR   Z,    ende
65     LD   B,    A
66     CP   1
67     JR   Z,    shiftright
68     CP   2
69     JR   Z,    shiftleft
70     JR   ende
71
72  shiftright:
73     IN   A,    (94h)
74     ADD  A,    4
75     OUT  (94h), A
76     JP   ende
77
78  shiftleft:
79     IN   A,    (94h)
80     SUB  A,    4
81     OUT  (94h), A
82     JP   ende
83
84  ; Funktion zum schreiben auf den Bildschirm
85  ; Startadresse des String muss in BC angegeben werden.
```

```
 86  VGAPrint:
 87     PUSH AF         ; Sichern der Register, die verändert werden.
 88     LD   A,    (BC) ; Erstes Zeichen in das Register laden.
 89  VGAPLoop:
 90     CP   0                ; Beim Null Byte ist der String zu Ende und wir
 91     JR   Z,    VGAPEnd    ; springen zum Ende der Funktion.
 92     CP   10
 93     JR   Z,    VGAPBack
 94     CP   13
 95     JR   Z,    VGAPNL
 96     OUT  (90h),     A     ; Zeichen an das Display schicken.
 97  VGAPNext:
 98     INC  C                ; Nächstes Zeichen laden
 99     JR   Z,    VGAPIncB
100  VGAPLDNX:
101     LD   A, (BC)
102     JP   VGAPLoop
103  VGAPIncB:
104     INC  B
105     JP   VGAPLDNX
106  VGAPNL:            ; Neue Zeile
107     LD   A,    0
108     OUT  (91h),     A
109     IN   A,    (92h)
110     INC  A
111     CP   30
112     JR   Z,    VGAPFL
113     OUT  (92h),     A
114     JP   VGAPNext
115  VGAPFL:            ; Erste Zeile
116     LD   A,    0
117     OUT  (92h),     A
118     JP   VGAPNext
119  VGAPBack:                ; Zum Anfang der Zeile
120     LD   A,    0
121     OUT  (91h),     A
122     JP   VGAPNext
123  VGAPEnd:
124     POP  AF         ; Gesicherte Register zurück holen.
125     RET             ; Rücksprung
126
127  taster:            ; Ändert die Farbe
128     DI
129     PUSH AF
```

```
130     IN   A,    (93h)
131     INC  A
132     OUT  (93h),     A
133     POP  AF
134     EI
135     RETI
136
137  timer:                    ; Wiederholt die Tasteneingabe
138     DI
139     PUSH AF
140     IN   A,    (80h)
141     CP   0
142     JP   Z,    keyboard_end
143     CP   13
144     JR   Z,    timer_newline
145     CP   8
146     JR   Z,    timer_back
147     OUT  (90h),     A
148  timer_weiter:
149     LD   A,    1
150     OUT  (0A0h),    A
151     POP  AF
152     EI
153     RETI
154  timer_back:
155     call back
156     JR   timer_weiter
157  timer_newline:
158     call newline
159     JR   timer_weiter
160
161  keyboard:           ; Wenn eine Taste gedrückt oder los gelassen wird.
162     DI
163     PUSH AF
164     IN   A,    (80h)
165     CP   0
166     JR   Z,    keyboard_end
167     CP   13
168     JR   Z,    keyboard_newline
169     CP   8
170     JR   Z,    keyboard_back
171     OUT  (90h),     A
172  keyboard_weiter:
173     LD   A,    11
```

```
174     OUT  (0A0h),    A
175     POP  AF
176     EI
177     RETI
178  keyboard_end:
179     LD   A,    0
180     OUT  (0A0h),    A
181     POP  AF
182     EI
183     RETI
184  keyboard_back:
185     call back
186     JR   keyboard_weiter
187  keyboard_newline:
188     call newline
189     JR   keyboard_weiter
190
191  newline:
192     LD   A,    0
193     OUT  (91h),     A
194     IN   A,    (92h)
195     INC  A
196     OUT  (92h),     A
197     ret
198  back:
199     IN   A,    (91h)
200     cp   0
201     jr   Z,    lineback
202     DEC  A
203     OUT  (91h),     A
204     LD   A,    0
205     OUT  (90h),     A
206     IN   A,    (91h)
207     DEC  A
208     OUT  (91h),     A
209     ret
210  lineback:
211     IN   A,    (94h)
212     CP   0
213     JR   NZ,   biglineback
214     ld   A,    39
215     OUT  (91h),     A
216     IN   A,    (92h)
217     DEC  A
```

```
218     OUT   (92h),     A
219     LD    A,    0
220     OUT   (90h),     A
221     ld    A,    39
222     OUT   (91h),     A
223     IN    A,    (92h)
224     DEC   A
225     OUT   (92h),     A
226     ret
227  biglineback:
228     ld    A,    79
229     OUT   (91h),     A
230     IN    A,    (92h)
231     DEC   A
232     OUT   (92h),     A
233     LD    A,    0
234     OUT   (90h),     A
235     ld    A,    79
236     OUT   (91h),     A
237     IN    A,    (92h)
238     DEC   A
239     OUT   (92h),     A
240     ret
241
```