UNIVERSITÄT
LEIPZIG

Der Minirechner 2a

Handbuch und
Bedienungsanleitung
Version 1.32

Max Braungardt
Thomas Schmid

Abteilung Neuromorphe
Informationsverarbeitung
Institut für Informatik

Kontakt

Max Braungardt
Augustusplatz 10 | Raum P522
04109 Leipzig

E-Mail: braungardt@informatik.uni-leipzig.de

Thomas Schmid
Augustusplatz 10 | Raum P533
04109 Leipzig

E-Mail: schmid@informatik.uni-leipzig.de

Der Minirechner 2a wurde an der Universität Tübingen von Werner Dreher entwickelt und ist seit ca. 2010 Bestandteil des Praktikums im Modul „Grundlagen der Technischen Informatik 2“ der Universität Leipzig. Diese Bedienungsanleitung wurde von der Abteilung Neuromorphe Informationsverarbeitung der Universität Leipzig erstellt.

Inhaltsverzeichnis

1 Überblick

Der Minirechner 2a ist ein einfacher, aber vollständiger 8-Bit-Rechner. Er realisiert eine von-Neumann-Architektur, weshalb sich im grundsätzlichen Aufbau insbesondere Datenpfad und Steuerwerk unterscheiden lassen. Funktionen und Berechnung lassen sich auf dem Minirechner 2a durch binär kodierte Befehle (Befehlsbytes) realisieren. Zusätzlich kann ein Assembler verwendet werden, um die Programme (Abfolgen von Befehlsbytes) zu erstellen.

Physikalisch besteht die Minirechner-Plattform aus drei Platinen:

1. einer Logikplatine, die im Wesentlichen ein Field Programmable Gate Array (FPGA) und ein Random Access Memory (RAM) für Daten (kurz: Daten-RAM) enthält;
2. einer Display-Platine, auf der sich ca. 300 LEDs befinden;
3. einer Erweiterungsplatine mit 2 Digital-Analog-Wandlern, einem Temperatursensor und einem Lüfter.

Die CPU mit Datenpfad und Steuerwerk ist im FPGA realisiert. Der Daten-RAM ist über den Memory-Bus an die CPU angekoppelt. Dieser Bus besteht aus folgenden Leitungen:

- 8 Datenleitungen von der CPU zum Daten-RAM (Memory Data Out: MEMDO)
- 8 Datenleitungen vom Daten-RAM zur CPU (Memory Data In: MEMDI)
- 8 Adressleitungen (Memory Address: MEMA)
- 3 Steuerleitungen (Chip Enable: CE, Output Enable: OE, Write Enable: WE)

Außer dem Daten-RAM hängen an diesem Bus noch vier Input-Register („in FC“ bis „in FF“ auf Adresse FC bis FF), zwei Output-Register („out FE“ und „out FF“ auf Adresse FE bzw. FF) und eine serielle Schnittstelle (Universal Asynchronous Receiver and Transmitter: UART, Adressen FA und FB). Die Input-Register können mit den Tastern bitweise beschrieben werden und dienen zur Eingabe von Daten an die CPU; sie können von der CPU nur gelesen werden. Die Output-Register können durch die CPU beschrieben werden und dienen zur Visualisierung von Ergebnissen durch LEDs. Über den UART kann die CPU z.B. mit einem PC kommunizieren. Die Input- und Output-Register und der UART befinden sich innerhalb des FPGAs, der Daten-RAM ist ein eigener Baustein. Der Daten-RAM, die darin enthaltenen Input-Register sowie die Output-Register sind im Blockschaltbild (Abbildung 1.1) nicht enthalten, da sie nicht Teil der eigentlichen CPU sind, sondern als zusätzliche Bauelemente zur Verfügung stehen.

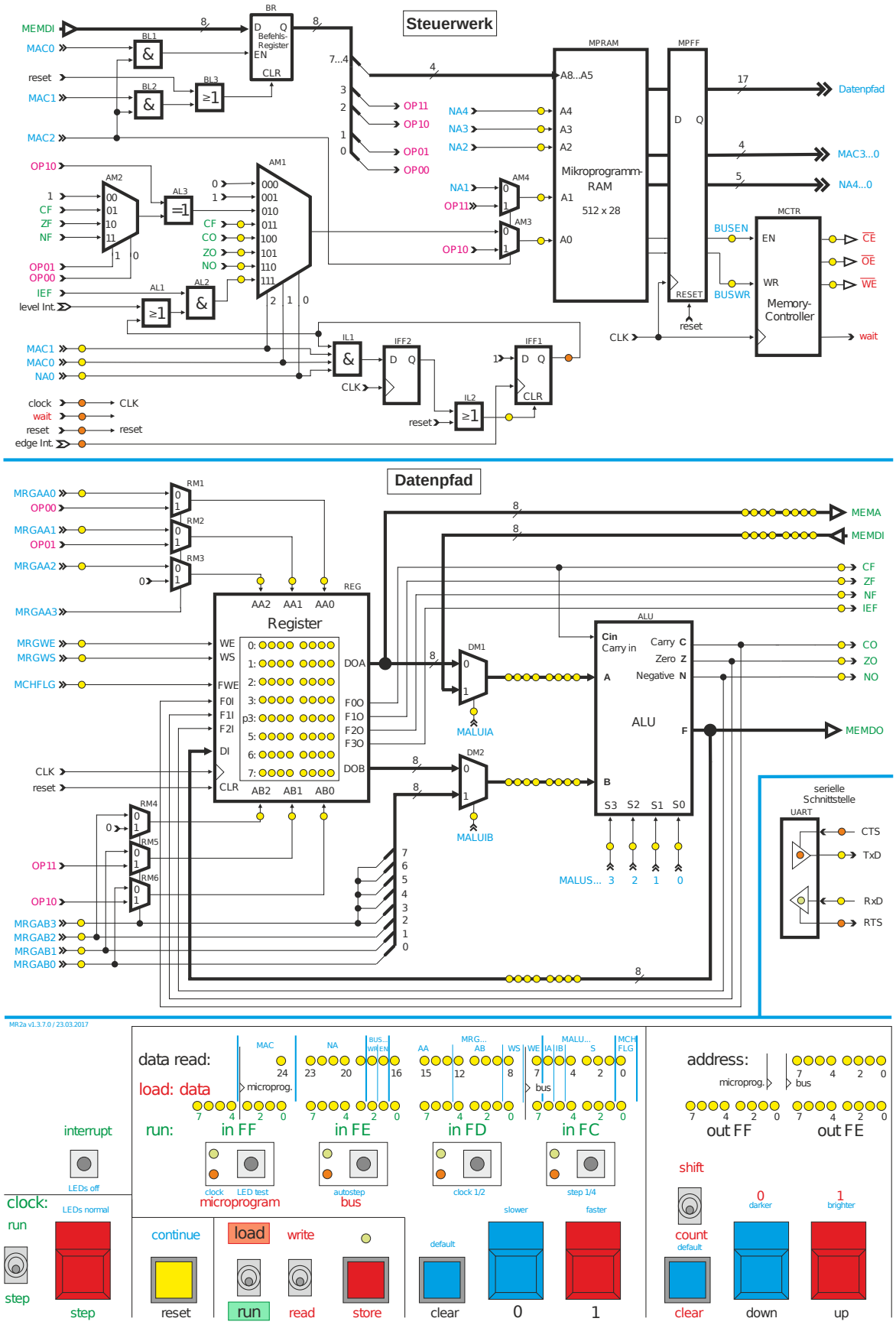

Abbildung 1.1: Blockschaltbild des Minirechner 2a.

2 Datenpfad

Der Datenpfad kann 8-Bit-breite Daten verarbeiten und besteht im Wesentlichen aus einem Register-Block und einer arithmetisch-logischen Einheit. Der Register-Block beinhaltet acht universelle Register zu je acht Bit, in denen sowohl Daten (z.B. Zwischenergebnisse) als auch Daten-RAM-Adressen gespeichert werden können. Die ALU stellt 16 Funktionen zur arithmetischen und logischen Verknüpfung der Eingänge A und B zu Verfügung. Vor diesen beiden Eingängen befindet sich jeweils ein 2-zu-1-Multiplexer (Abbildung 2.3), mit denen die für die ALU bestimmten Daten aus je zwei Quellen ausgewählt werden können. Als Quellen kommen der Register-Block, der Daten-RAM sowie Befehlsbytes (Eingabe von Konstanten) in Frage. Die Abbildung 2.1 zeigt eine Übersicht über den Datenpfad.

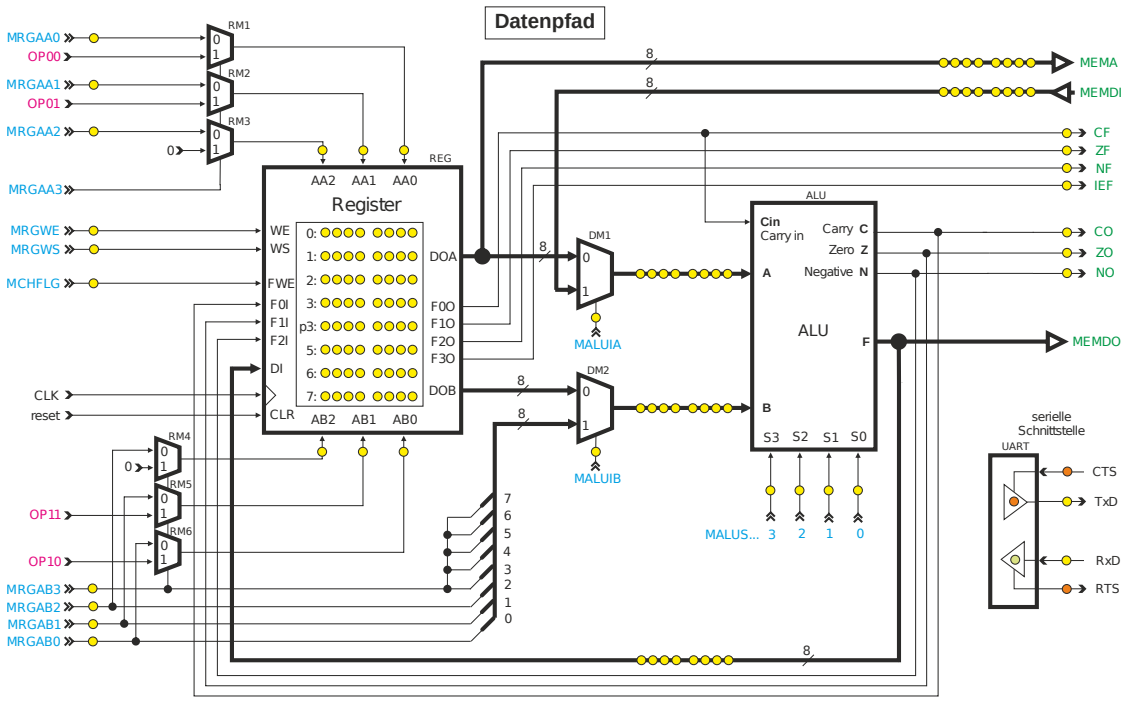

Abbildung 2.1: Schematische Darstellung des Datenpfads.

2.1 Registerblock

Der Register-Block (Abb. 2.2) kann 8-Bit-breite Daten, wie (Zwischen-) Ergebnisse und Adressen speichern. Er besitzt einen 8-Bit-breiten Daten-Eingang (Data In: DI) und die zwei 8-Bit-breiten Daten-Ausgänge Data Out A (DOA) und Data Out B (DOB). Mit Hilfe der zwei 3-Bit-breiten Adresseingänge Address A (AA0, AA1, AA2) und Address B (AB0, AB1, AB2) kann ausgewählt werden, welche Register an die Ausgänge DOA und DOB angelegt werden sollen. Sollen Daten in eines der Register geschrieben werden, so muss mit Write Select (WS) ausgewählt werden, welche der beiden Adressen (AA oder AB) für die Auswahl des zu beschreibenden Registers verwendet werden soll. Wird der Eingang Write Enable (WE) aktiviert, so werden die am 8-Bit-breiten Eingang DI anliegenden Daten bei der nächsten aktiven Taktflanke in das ausgewählte Register geschrieben.

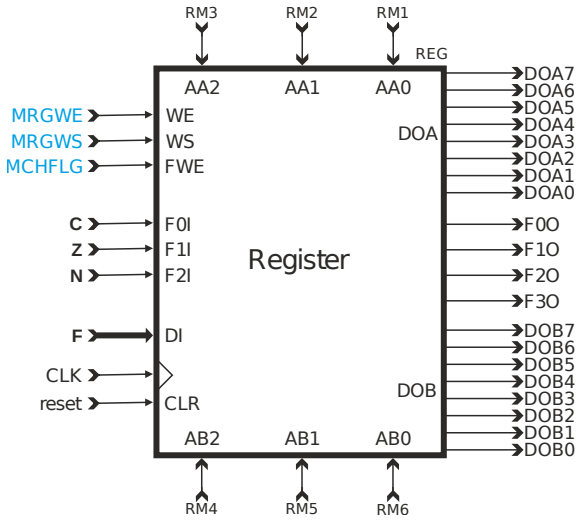

Abbildung 2.2: Schematische Darstellung des Registerblocks.

2.2 Arithmetisch-logische Einheit (ALU)

Die ALU ist für die arithmetische bzw. logische Verknüpfung zweier Zahlen zuständig. Als Datenquelle für den Eingang A der ALU kann entweder der Ausgang DOA des Registerblocks oder der Daten-RAM-Bus (Memory Data In: MEMDI) dienen (Auswahl durch den entsprechenden Steuereingang (Microprogram-Bit ALU Input A select: MALUIA) des Multiplexers vor dem Eingang A). Als Datenquelle für den Eingang B der ALU kann der Ausgang DOB des Registerblocks oder eine Konstante zwischen -8 und 7 (8-Bit-Zweierkomplement) dienen (Auswahl durch MALUIB, die Konstante wird durch die Steuereinheit bestimmt). Die Adressen für den Daten-RAM kommen immer aus dem Ausgang DOA.

Die ALU enthält drei arithmetische bzw. logische Funktionseinheiten (Abbildung 2.3), mit denen man binäre und unäre Operationen durchführen kann:

- das 8-Bit-breite NOR U1 (bestehend aus 8 NORs mit je 2 Eingängen)
- den 8-Bit-Volladierer U2 (mit Carry-Eingang Cin und Carry-Ausgang C)
- den Shifter U3, der den Wert um ein Bit nach rechts versetzt durchreicht.

Mit Hilfe des Multiplexers U4 wird der gewünschte Wert ausgewählt und an den Ausgang F gelegt. Der Shifter ist in Wirklichkeit keine eigene Schaltung, sondern der Eingang A um ein Bit versetzt an den Multiplexer U4 angeschlossen. Der Multiplexer U5 wählt das Carry-Bit

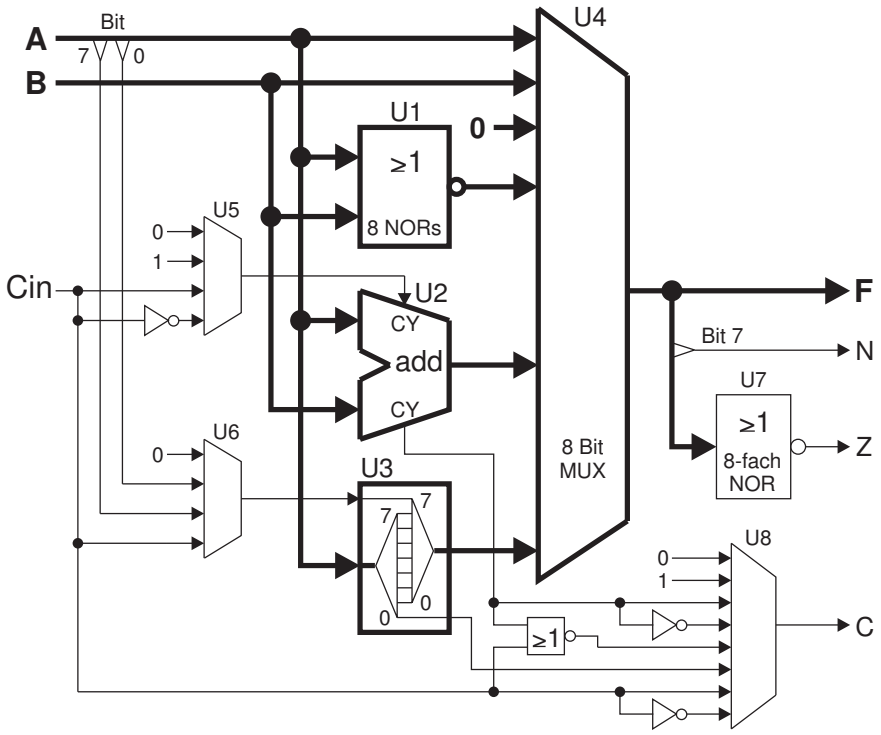

Abbildung 2.3: Blockschaltbild der ALU (dicke Leitungen enthalten acht Bit).

Steuer-eingänge	Befehl allg.	Befehl bei $B = A$	Funktion	C	N	Z	Bemerkung
00 00	ADDH	LSLH	$F = A + B$	OR	*	*	**add** and **h**old carry: $C = Cin \vee C$
00 01	A	-	$F = A$	0	*	*	Eingang **A** durchreichen
00 10	NOR	COM	$F = A$ NOR B	0	*	*	bei $B = A$: **com**plement
00 11	0	-	$F = 0$	0	0	1	Ergebnis immer 0
01 00	ADD	LSL	$F = A + B$	Ca	*	*	bei $B = A$: **l**ogical **s**hift **l**eft
01 01	ADDS	(SL1)	$F = A + B + 1$	$\overline{Ca}$	*	*	**add** for **s**ubtraction bei $B = A$: **s**hift **l**eft, rechts **1** einschieben
01 10	ADC	RLC	$F = A + B + Cin$	Ca	*	*	**ad**d with **c**arry bei $B = A$: **r**otate **l**eft through **c**arry
01 11	ADCS	-	$F = A + B + \overline{Cin}$	$\overline{Ca}$	*	*	**ad**d with **c**arry for **s**ubtraction
10 00	LSR	-	$F(n) = A(n+1), F(7) = 0$	$A(0)$	*	*	**l**ogical **s**hift **r**ight, links 0 einschieben
10 01	RR	-	$F(n) = A(n+1), F(7) = A(0)$	$A(0)$	*	*	**r**otate **r**ight
10 10	RRC	-	$F(n) = A(n+1), F(7) =$Cin	$A(0)$	*	*	**r**otate **r**ight through **c**arry
10 11	ASR	-	$F(n) = A(n+1), F(7) = A(7)$	$A(0)$	*	*	**a**rithmetic **s**hift **r**ight
11 00	B CLC	-	$F = B$	0	*	*	Eingang **B** durchreichen **cl**ear **c**arry flag
11 01	SETC	-	$F = B$	1	*	*	**set c**arry flag
11 10	BH	-	$F = B$	Cin	*	*	**B** and **h**old carry flag
11 11	INVC	-	$F = B$	$\overline{Cin}$	*	*	**inv**ert carry flag

A, B = Dateneingänge, F = Ergebnis, C = carry out, N = negative out, Z = zero out, * = entsprechend dem Ergebnis F; Cin = Carry input in ALU, Ca = Carry aus Addierer, $\overline{x}$ = Signal x invertiert

Tabelle 2.1: Funktionen der ALU.

für den Volladdierer, während die Einheit U6 entscheidet, ob und welches Bit beim Rechtsshift links eingeschoben wird. Das achtfach-NOR U7 analysiert den Ausgang F und liefert eine 1, falls dieser 0 ist. Der Multiplexer U8 bestimmt das Carry-Flag.

Die ALU ist ein reines Schaltnetz, weshalb der berechnete Wert innerhalb des gleichen Taktes am Ausgang F zur Verfügung steht. Die ALU besitzt außerdem drei Flag-Ausgänge:

- C = Carry = arithmetischer Überlauf oder rausgeschobenes Bit beim Shiften
- Z = Zero = Ergebnis ist 0
- N = Negative = Ergebnis ist eine negative Zahl (= Bit 7 ist gesetzt)

Diese Flag-Ausgänge werden im Register R4 gespeichert. Dieses Register übernimmt die Werte am Eingang D mit steigender Taktflanke, wenn gleichzeitig am Eingang Enable (EN) eine 1 anliegt. Wenn EN = 0 ist, bleibt der Ausgang Q unverändert. Das zwischengespeicherte Carry-Bit ist an den Carry-Eingang der ALU zurückgeführt und kann für Berechnungen mit mehr als acht Bit Datenwortbreite benutzt werden. Die Multiplexer in der ALU werden durch die Eingänge Select 0 bis 3 (S0 bis S3) gesteuert.

2.3 Ein- und Ausgabe

I/O-Interface

Programmierung und Programmausführung werden durch eine Reihe von Tastern gesteuert, die sich im Bedienfeld (Abbildung 2.4) befinden. Rote Beschriftungen gelten nur im Programmiermodus, grüne nur im Run-Modus, schwarze immer. Farbig beschriftete Schalter oder Taster haben im jeweils anderen Modus keine Funktion. Für eine ausführliche Beschreibung der Tasterfunktionen siehe Abschnitte 5.2 und 5.3.

Die 32 LEDs in der Zeile „load: data“ zeigen im Programmiermodus den Inhalt des Eingaberegisters an. Im Run-Modus wird diese Anzeige in die vier Input-Register FC, FD, FE und FF aufgeteilt.

Die 28 LEDs mit der Bezeichnung „data read“ zeigen im Programmiermodus (bei Auswahl „microprogram“) und im Run-Modus den Inhalt des Mikroprogramm-RAMs an der eingestellten Adresse. Ist im Programmiermodus „bus“ ausgewählt, so wird der Inhalt des Daten-RAMs des gewählten Input-Ports, oder das Daten- bzw. Statusregisters des UARTs an der eingestellten Adresse angezeigt.

Die 8 LEDs mit der Bezeichnung „address“ zeigen die eingestellte Adresse und die 16 LEDs „out FE“ und „out FF“ den Inhalt der Output-Register.

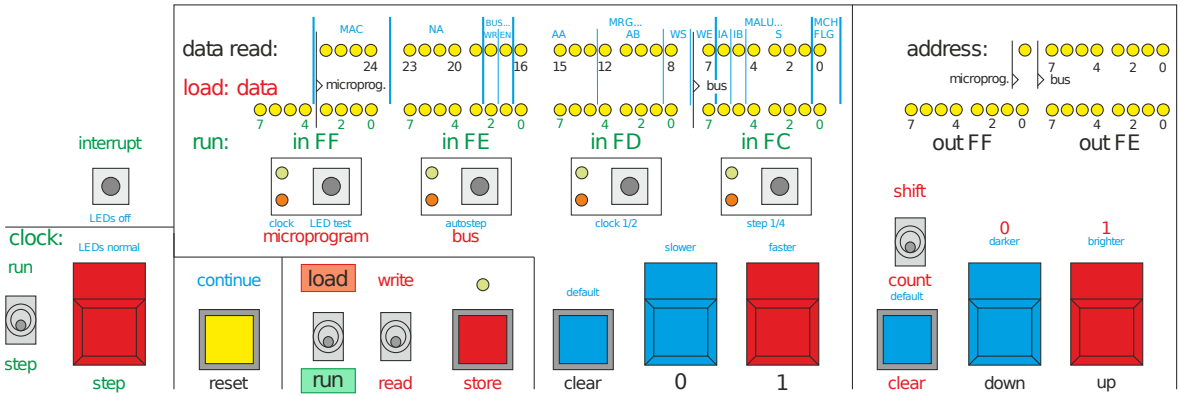

Abbildung 2.4: Schematische Darstellung des Bedienfelds.

Erweiterungsplatine MR2DA2

Die Platine MR2DA2 enthält zwei Digital-Analog-Wandler (kurz: D/A-Wandler), zwei analog-Komparatoren, einen Temperatursensor, zwei digitale 8-Bit-breite Ausgänge und einen digitalen 8-Bit-breiten Eingang (siehe Abb. 2.5). Sie ist über einen 32-poligen Steckverbinder mit dem Memory-Bus des Minirechners verbunden.

Schreibt man ein Byte auf die Adresse F0 des Minirechners, so wird es im 8-Bit-breiten Register RG1 gespeichert und vom Digital-Analog-Converter DAC1 in eine analoge Spannung umgewandelt. Der Binärwert wird am 8-Bit-Port (Pfostenstecker) Digital Out 1 (P-DO1), die analoge Spannung am Messstift Analog Out 1 (P-AO1) ausgegeben. Mit dieser Spannung wird auch die rote LED D-AO1 und ein Lüfter proportional angesteuert. Der analog-Komparator CP1 vergleicht diese Spannung mit der Spannung am Stift Analog In 1 (P-AI1). Er gibt eine logische 1 aus, wenn die Spannung an P-AI1 höher ist als die Spannung vom D/A-Wandler, andernfalls eine 0; dies wird durch die grüne LED D-CP1 signalisiert. Liest man von der Adresse F1, so erscheint an Bit 3 der Zustand des Ausgangs des Komparators.

Der zweite D/A-Wandler DAC2 an Adresse F1 ist ähnlich verschaltet. Am Eingang des zugehörigen Komparators CP2 liegt ein Temperaturfühler, der eine mit der Temperatur steigende Spannung abgibt. Diese wird auch durch die rote LED D-AI2 dargestellt.

Die Ausgänge der Register RG1 und RG2 (Typ 74AC573 in CMOS-Technologie) liefern ziemlich genau 0 Volt bzw. 3,3V. Zusammen mit dem Lastwiderstand von 34 kΩ ergibt sich für die Ausgangsspannung ein Bereich von 0V (bei 0) bis 2,55V (bei FF).

Mit Hilfe eines D/A-Wandlers und des zugehörigen Komparators kann durch Intervallschachtelung (sukzessive Approximation) die Spannung am Eingang AI1 bzw. AI2 ermittelt werden. Die Intervallschachtelung muss durch das Programm ausgeführt werden.

Liest man von der Adresse F0, so erhält man die logischen Pegel der 8 Datenpins des Input-Ports (Pfostensteckers) P-DI1.

Auf der Platine befindet sich zusätzlich ein CPLD XC9572XL, welche weitere Funktionen zur Verfügung stellt. Darin ist unter anderem ein 8-Bit-breiter Zähler implementiert, der die Zeit zwischen zwei Tachosignal-Impulsen des Lüfters misst und der über die Adresse F2 ausgelesen werden kann. Drei Pins des CPLDs sind als universelle Input-/Output-Signale UIO1 bis UIO3 an Meßstifte geführt; sie können getrennt als Eingang oder Ausgang programmiert werden (siehe Tabellen 2.2, 2.3 und 2.4).

Die Abgreifstifte zum Messen der Spannungen liegen alle auf der linken Seite der Plati-

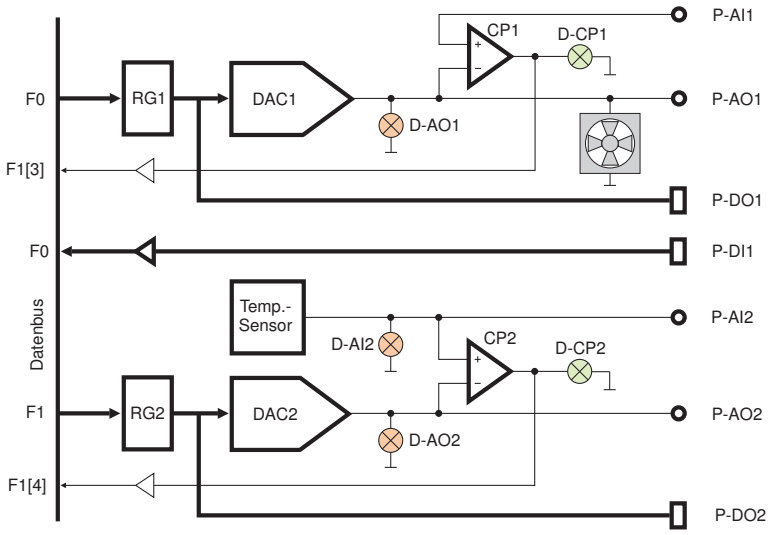

Abbildung 2.5: Blockschaltbild der Platine MR2DA2

ne und sind entsprechend Abbildung 2.5 beschriftet. Die beiden dickeren Stifte neben den Abgreifstiften liegen auf der Referenzspannung und sind somit zum Beispiel für die Referenzklemmen des Oszilloskops geeignet. Die LEDs im Blockschaltbild liegen alle an der Vorderkante der Platine und sind entsprechend beschriftet. Mit dem Trimmpotentiometer bei den LEDs kann der Nullpunkt des Temperatursensors eingestellt werden.

Adresse	read	write
F0 (F4)	8-Bit-Input-Port (Pfostenstecker vorne)	DAC1 (AO1 + CP1 + Lüfter) und Pfostenstecker Mitte
F1 (F5)	Status-Register	DAC2 (AO2 + CP2 [Temp.-Fühler]) und Pfostenst. hinten
F2 (F6)	Umdrehungszeit des Lüfters (in Takten)	Kontrollregister
F3 (F7)	Interrupt Status Register	Interrupt-Flip-Flop zurücksetzen (Wert egal)

Adressen in Klammern, wenn der Adressjumper gesteckt ist (bei 2 Platinen an einem Minirechner)

Tabelle 2.2: Übersicht über die MR2DA2-Adressen.

Bit	Signal
7	Jumper J2 (1 = gesteckt)
6	Jumper J1 (1 = gesteckt)
5	Tachosignal Lüfter
4	Komparator an DAC2 und AI2 (Temperaturfühler): high, wenn Temperatursignal höher als DAC2
3	Komparator an DAC1 und AI1: high, wenn analog-Eingang 1 (AI1) höher als Ausgang von DAC1
2-0	UIO3-UIO1

Tabelle 2.3: Übersicht über die MR2DA2-Status-Register

Wert binär	Funktion für UIO3 - UIO1	n = 0	n = 1
1000.0nnn	output enable	UIOx ist Eingang	UIOx ist Ausgang
0000.0nnn	output value	low ausgeben, wenn Ausgang	high ausgeben, wenn Ausgang

Tabelle 2.4: Übersicht der universellen I/O-Pins (Kontrollregister F2) der MR2DA2-Platine.

Serielle Schnittstelle

Mit dem Universal Asynchronous Receiver and Transmitter (UART, Abbildung 2.6) kann die CPU z.B. mit einem PC kommunizieren. Hierfür sind die Register FA und FB im Daten-RAM als Speicher für empfangene und zu sendende Byte und als Status- und Controll-Register vorgesehen.

Ist das Bit Clear To Send auf 1 (CTS = 1) gesetzt, so ist die Sendeleitung zum angeschlossenen PC frei. Ist das Bit Ready To Send auf 1 (RTS = 1) gesetzt, so signalisiert der Minirechner 2a, dass er bereit ist über die UART-Schnittstelle das nächsten Bit zu empfangen. Das Bit TxD zeigt dann den aktuellen Stand der Sendeleitung an, während das Bit RxD den aktuellen Zustand der Empfangsleitung repräsentiert.

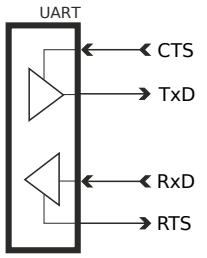

Abbildung 2.6: Schematische Darstellung der UART-Schnittstelle.

3 Steuerwerk

Das Steuerwerk des Minirechner 2a arbeitet die Maschinenbefehle aus dem Befehlsregister ab und steuert andere Funktionseinheiten durch Steuersignale. Es setzt sich im Wesentlichen aus einer Adress-Dekodierung, dem Mikroprogramm-RAM, dem Befehlsregister und einem Memory-Controller zusammen. Darüber hinaus existiert ein Handling für Interrupts. Einen Überblick über das Steuerwerk liefert die Abbildung 3.1.

Um zu verdeutlichen, welche Einheit für die Ansteuerung der einzelnen Funktionseinheiten verantwortlich ist, wird eine Farbcodierung verwendet. In allen Abbildungen stehen blaue Signalnamen für Mikroprogramm-Bits (werden durch das jeweilige Mikroprogramm-Wort gesetzt), grüne kommen aus dem Datenpfad (werden durch Berechnungen oder Multiplexer gesetzt), rote vom Memory-Controller und schwarze von außen. Alle Signale, die aus dem Mikroprogramm kommen, beginnen mit dem Buchstaben „M“. Die Interrupt-Logik im unteren Teil der Abbildung kann den Programmablauf beeinflussen.

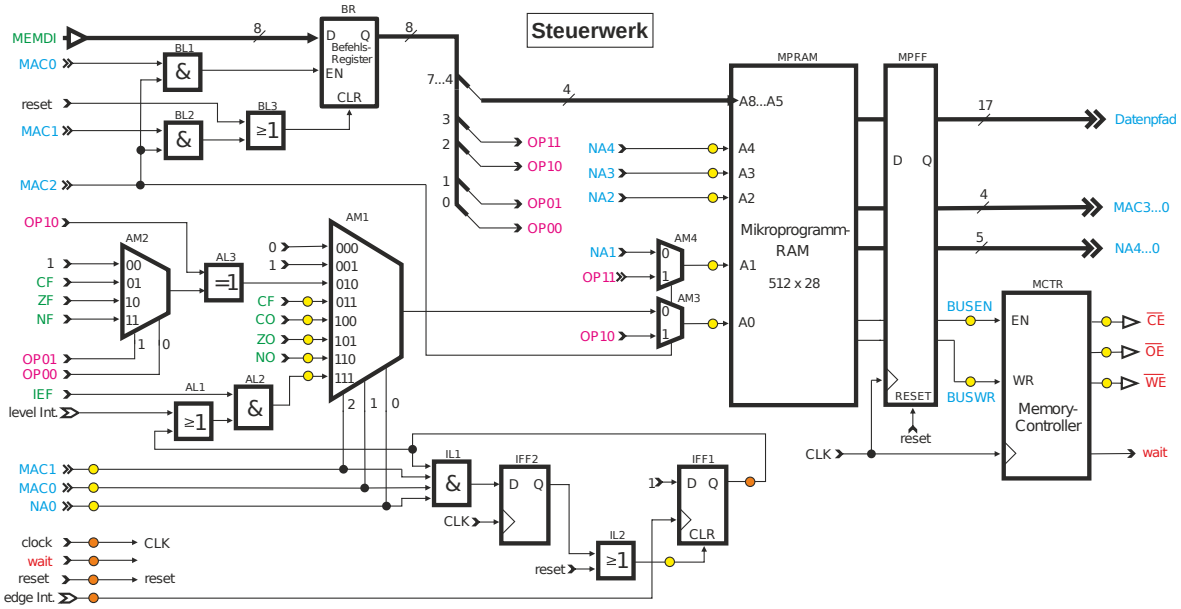

Abbildung 3.1: Schematische Darstellung des Steuerwerks.

3.1 Adress-Decodierung

Der Adress-Decoder (Abb. 3.2) ist ein 8-zu-1-Multiplexer, welcher das Adressbit A0 bestimmt und direkt vor dem Adresseingang geschaltet ist. Er wird durch die Mikroprogramm-Bits Microprogram Address Control (MAC0, MAC1, MAC2), Next Address 0 (NA0) und den Operanden (OP00, OP01, OP10, vgl. Kapitel 4.3) angesteuert. Die dreistelligen Binärzahlen an den Eingängen des Multiplexers geben an, bei welchem Code an den Steuereingängen 0,1,2 der jeweilige Dateneingang an den Ausgang durchgeschaltet wird.

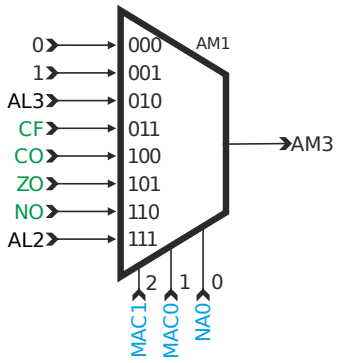

Abbildung 3.2: Schematische Darstellung des Address-Multiplexers.

In Abhängigkeit der Bits MAC2...0, OP10...00 und NA0 sind bedingte Verzweigungen bzw. bedingte Sprünge im Mikroprogramm möglich (siehe Tabelle 3.1). Häufig wird als Verzweigungsbedingung ein Flag-Ausgang verwendet. Soll zum Beispiel bei einer negativen Zahl (NO = 1) eine andere Berechnung ausgeführt werden als bei einer positiven Zahl (NO = 0), so müssen die Adress-Kontroll-Bits auf 011 (MAC2...0 = 011) und das Bit für die nächste Adresse auf 0 (NA0 = 0) gesetzt werden. Der Wert der ersten drei Operanden-Bits (OP10...OP00) ist in diesen Fall nicht relevant.

Zeile	MAC2,1,0	NA0	OP10,01, 00	nächste Adresse Bit 8	7	6	5	4	3	2	1	Bit 0
1	0 0 0	x	x	*	*	*	*	NA4	NA3	NA2	NA1	NA0
2	0 0 1	0	0 0 0	*	*	*	*	NA4	NA3	NA2	NA1	1
3	0 0 1	0	0 0 1	*	*	*	*	NA4	NA3	NA2	NA1	CF
4	0 0 1	0	0 1 0	*	*	*	*	NA4	NA3	NA2	NA1	ZF
5	0 0 1	0	0 1 1	*	*	*	*	NA4	NA3	NA2	NA1	NF
6	0 0 1	0	1 0 0	*	*	*	*	NA4	NA3	NA2	NA1	0
7	0 0 1	0	1 0 1	*	*	*	*	NA4	NA3	NA2	NA1	/CF
8	0 0 1	0	1 1 0	*	*	*	*	NA4	NA3	NA2	NA1	/ZF
9	0 0 1	0	1 1 1	*	*	*	*	NA4	NA3	NA2	NA1	/NF
10	0 0 1	1	x	*	*	*	*	NA4	NA3	NA2	NA1	CF
11	0 1 0	0	x	*	*	*	*	NA4	NA3	NA2	NA1	CO
12	0 1 0	1	x	*	*	*	*	NA4	NA3	NA2	NA1	ZO
13	0 1 1	0	x	*	*	*	*	NA4	NA3	NA2	NA1	NO
14	0 1 1	1	x	*	*	*	*	NA4	NA3	NA2	NA1	Interrupt
15	1 0 0	x	x	*	*	*	*	NA4	NA3	NA2	OP11	OP10
16	1 0 1	x	x	OC3	OC2	OC1	OC0	NA4	NA3	NA2	OP11	OP10
17	1 1 0	x	wird auf 000 gesetzt	0	0	0	0	NA4	NA3	NA2	0	0
18	1 1 1	-	-	-	-	-	-	-	-	-	-	-

Tabelle 3.1: Nächste Adresse (* = unverändert, x = don't care, /y = Signal y invertiert)

In Abhängigkeit des Flag-Werts passt der Adress-Decoder das Bit A0 vor der Ausgabe nach Tabelle 3.1 an. So wird bei einer negativen Zahl der nächste Befehl aus der nächsten ungeraden Adresse geladen, da NO = 1 = A0 gesetzt ist. Bei einer positiven Zahl wird der Wert aus der angegeben Adresse geladen.

Das Signal AL2 ist Bestandteil der Interrupt-Logik (Abschnitt 3.5) und AL3 stammt aus der dem Decoder vorgeschalteten Logik.

3.2 Mikroprogramm-RAM

Der Mikroprogramm-RAM (Abb. 3.3) beinhaltet Mikroprogramme, welche der Minirechner 2a für die Ausführung der verschiedenen Maschinenbefehle verwendet. Er kann insgesamt 512 Mikroprogramm-Worte zu je 28 Bit speichern und ist in 16 Bereiche unterteilt. Jeder einzelne dieser Bereiche beinhaltet ein Mikroprogramm, welches in der Regel einen Maschinenbefehl realisiert. Durch unterschiedliche Ansteuerung kann ein Mikroprogramm allerdings bis zu vier Maschinenbefehle realisieren. Der Mikroprogramm-RAM kann beschrieben werden, jedoch sollte davon abgesehen werde, da sonst die abgelegten Mikroprogramme und somit auch der Minirechner 2a somit nicht mehr funktionsfähig ist.

Die Reihenfolge des Auslesens einzelner Mikroprogramm-Worte wird dabei nicht von einem Zähler vorgegeben, sondern vom Wort selbst bestimmt: die sieben niederwertigsten Bits des Steuerwortes dienen dazu, die nächste auszulesende Adresse festzulegen. Mit den fünf Bits Next Address (NA0 bis NA4) wird die nächste auszulesende Adresse angegeben. Das unterste Adressbit (NA0) kann dabei durch einen der Flag-Ausgänge der ALU, durch die zwischengespeicherten Flags oder durch ein Interrupt-Signal ersetzt werden.

Die oberen vier Bits der Maschinenbefehle (Befehlsbytes, vgl. Kapitel 4.3) werden direkt an die oberen Adressengänge des Mikroprogramm-RAMs gelegt, um so eines der 16 verfügbaren Mikroprogramme zu wählen. Die Operanden-Bits 10 und 11 (OP10, OP11) werden an die Eingänge A0 und A1 gelegt, um so die Startadresse innerhalb des ausgewählten Mikroprogramms festzulegen. Dadurch können bis zu vier Variationen des Maschinenbefehls verwendet werden.

Der nachgeschaltete Flip-Flop MPFF setzt durch Drücken des Tasters **reset** alle Steuereinheiten zurück (vgl. Tabelle 5.2).

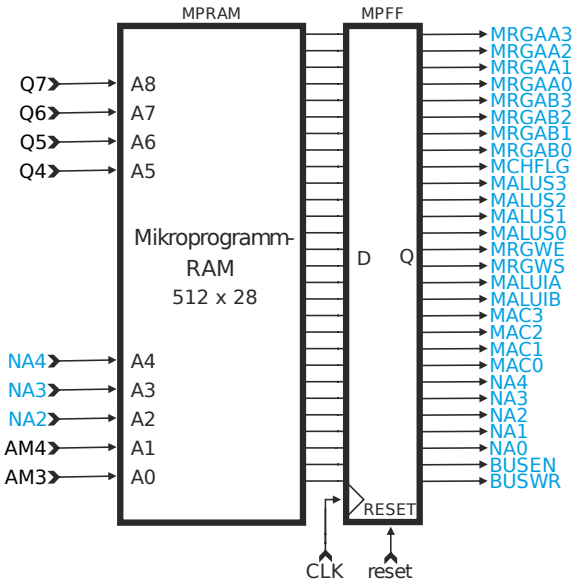

Abbildung 3.3: Schematische Darstellung des Mikroprogramm-RAMs.

3.3 Befehlsregister

Das Befehlsregister (Abb. 3.4) wird durch die Mikroprogramme gesteuert und beinhaltet alle auszuführenden Befehle im Byte-Format. Zum Laden des nächstes Bytes müssen die drei unteren Bits für den Memory Address Control auf 101 (MAC2, MAC1, MAC0 = 101), Bus Enable auf 1 (BUSEN = 1) und Bus Write auf 0 (BUSWR = 0) gesetzt werden. Zusätzlich muss der Inhalt des Programmzählers auf den Daten-Bus (Register R3) gelegt werden.

Die Befehle werden über den Daten-Bus (Memory Data In: MEMDI) in das Befehlsregister geschrieben. Ob das Register beschrieben wird, entscheidet das Logikgatter BL1 vor dem Enable-Eingang (EN). Das Logikgatter BL3 vor dem Clear-Eingang (CLR) entscheidet, ob das Register komplett geleert wird (z.B. beim Drücken der **RESET**-Taste). Der 8-Bit-breite Ausgang Q übergibt die gespeicherten Befehlsbytes schrittweise an den Mikroprogramm-RAM und andere Steuereinheiten.

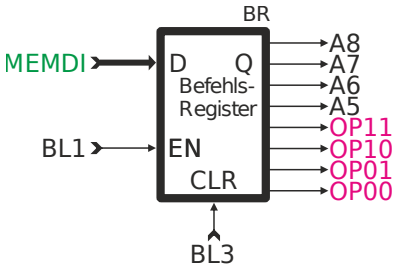

Abbildung 3.4: Schematische Darstellung des Befehlsregisters.

3.4 Memory-Controller

Der Memory-Controller (Abb. 3.5) bestimmt die invertierten Steuersignale Chip Enable ($\overline{CE}$), Output Enable ($\overline{OE}$) und Write Enable ($\overline{WE}$) für den Daten-RAM. Damit der Daten-RAM richtig angesteuert wird, benötigt jeder Zugriff auf den Daten-RAM-Bus zwei Takte, weshalb der Memory-Controller das Signal „wait“ generiert, welches alle Abläufe im Steuerwerk und dem Datenpfad einfriert, so lange es aktiv ist (jeweils für einen Takt aktiviert).

Ist das eingehende Bit Bus Enable auf 1 gesetzt (BUSEN = 1), so wird der Daten-RAM-Bus angesprochen. Das Signal Bus Write (BUSWR) bestimmt dabei ob gelesen (BUSWR = 0) oder geschrieben (BUSWR = 1) wird.

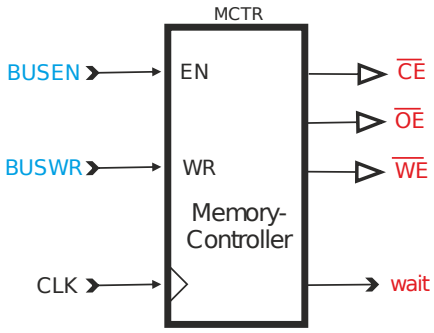

Abbildung 3.5: Schematische Darstellung des Memory-Controllers.

3.5 Interrupts

Interrupt bezeichnet die Unterbrechung des derzeitig laufenden Programms, um ein spezielles Unterprogramm, die Interrupt Service Routine (ISR), auszuführen. Nach dem Beenden der ISR wird das eigentliche Programm an der unterbrochenen Stelle fortgesetzt.

Nach jedem Assemblerbefehl (außer bei den Befehlen Enable Interrupt (EI) und Return from Interrupt (RETI)) wird automatisch der Eingang 111 des Adressmultiplexers 1 (AM1; Ausgang des ANDs AL2) abgefragt. Ist dieser auf High gesetzt, so wird ein Interrupt ausgelöst. Im Falle eines Interrupts werden alle Flags auf den Stack geschrieben, das Interrupt Enable Flag (IEF) auf Low gesetzt und der Befehl „CALL 2“ ausgeführt. Somit beginnt jede ISR an Adresse 2. Da die CPU immer an Adresse 0 beginnt, muss dort ein Jump zum eigentlich Hauptprogramm erfolgen. Hierfür muss der Befehle JR benutzt werden, da der Befehl JMP drei Bytes belegt. An Adresse 2 sollte ein Sprung zur ISR erfolgen.

Wie in Abbildung 3.6 zu erkennen ist, muss IEF auf High gesetzt sein, damit überhaupt ein Interrupt ausgelöst werden kann. Da dieses Flag bei jedem Programmstart auf Low gesetzt wird, muss das IEF durch den EI-Befehle im Programmcode auf High gesetzt werden.

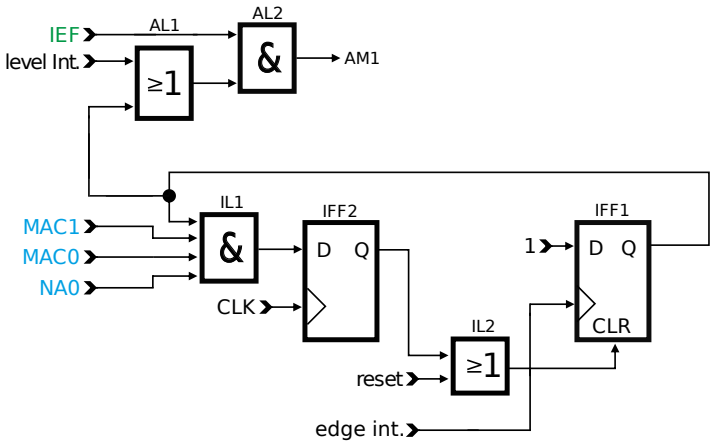

Abbildung 3.6: Schematische Darstellung der Interrupt-Logik.

Die ISR muss mit dem Befehle Return from Interrupt (RETI) beendet werden. Dieser Befehle liest die nächste Adresse des unterbrochenen Programms aus und holt die Flags aus dem Stack zurück. Somit wird automatisch auch das IEF wieder auf High gesetzt. Darf eine Codesequenz im Hauptprogramm nicht unterbrochen werden, da diese z.B. zeitkritisch ist, so kann mit dem Befehle Disable Interrupt (DI) am Anfang dieser Sequenz der Interrupt ausgeschaltet und mittels EI nach der Sequenz wieder aktiviert werden.

Das Interrupt-Signal kann vom Taster „interrupt“, von einem Timer, vom UART oder vom Bus der Erweiterungsplatine MR2DA2 kommen. Welches Signal tatsächlich weitergereicht wird, wird durch das Master Interrupt Control Register (MICR, Adresse F9 im Daten-RAM, vgl. Tab. 3.2) festgelegt. Bei allen vier Arten wird zwischen level Interrupt (pegelgesteuert) und edge Interrupt (flankengesteuert) unterschieden. Es lassen sich mehrere Interrupt-Quellen gleichzeitig freischalten, jedoch wird immer dieselbe ISR ausgeführt.

Ein Interrupt über Timer kann genutzt werden, um in regelmäßigen Abständen einen Interrupt auszuführen. Hierfür kann der Board-Takt von $7,3728\,MHz$ durch drei Taktteiler auf beliebige Werte eingestellt werden. Der erste Taktteiler (D_1) ist ein Binärteiler mit den möglichen Divisoren 1, 16, 256 und 4096. Der zweite Teiler (D_2) ist ein Dezimalteiler mit den möglichen Divisoren 1, 10, 100 und 1000. Der dritte Teiler (D_3) ist beliebig zwischen 1 und 32768 einstellbar. Insgesamt muss die Teiler-Kette mindestens durch zwei teilen. Der

Timer wird gesetzt, wenn man gemäß Tabelle 3.3 und 3.4 in die Register FC und FD schreibt. Wichtig für das Timer Control Register (TCR): Die Bits 14 bis 8 des dritten Teilers (Adresse FD) werden auf 0 gesetzt, wenn man die Bits 7 bis 0 (Adresse FC) beschreibt. Möchte man D_3 also setzen, so muss zuerst Adresse FC beschrieben werden. Die Interrupt-Frequenz ergibt sich insgesamt durch

$$\text{Interrupt-Frequenz} = \frac{7,3728\,MHz}{D_1 \cdot D_2 \cdot (D_3 + 1)}$$

Der Divisor D_3 muss immer um eins niedriger gesetzt werden, als der Wert später tatsächlich sein soll. Für die Werte TCR $= 1001.1011$ und $D_3 = 17$ ergibt sich zum Beispiel:

$$\text{Interrupt-Frequenz} = \frac{7372800\,Hz}{4096 \cdot 100 \cdot (17 + 1)} = 1\,Hz.$$

Bit	Wert	Bedeutung
7,6	xx--.----	nicht verwendet; auf 0 setzen
5	xx1-.----	bus edge interrupt enable
4	xx-1.----	bus level interrupt enable
3	xx--.1---	UART edge interrupt enable (nicht sinnvoll)
2	xx--.-1--	UART level interrupt enable
1	xx--.--1-	timer edge interrupt enable
0	xx--.---1	key edge interrupt enable

- = Bit hat andere Bedeutung (siehe andere Zeile)

x = Bit hat keine Bedeutung, sollte auf 0 gesetzt werden

Tabelle 3.2: Mögliche Belegungen des Master Interrupt Control Register (MICR).

Adresse	Wert	Bedeutung
FC	nnnn.nnnn	D_3[7-0]: Teiler 3, Bit 7 bis Bit 0
FD	0nnn.nnnn	D_3[14-8]: Teiler 3, Bit 14 bis Bit 8
FD	1nnn.nnnn	Timer Control Register (siehe Tab. 3.4)

Tabelle 3.3: Bedeutung der Adressen FC und FD im write-Modus.

Bit	Wert	Bedeutung		
7	1xx-.----	Auswahl des Timer Control Registers (TCR)		
6-5	1xx-.----	nicht verwendet; auf 0 setzen		
4	1xxn.----	Timeraktivierung	n = 0: deaktivieren	n = 1: aktivieren
3-2	1xx-.nn--	Dezimalteiler	nn = 00: D_2 = 1	nn = 10: D_2 = 100
			nn = 01: D_2 = 10	nn = 11: D_2 = 1000
1-0	1xx-.--nn	Binärteiler	nn = 00: D_1 = 1	nn = 10: D_1 = 256
			nn = 10: D_1 = 16	nn = 11: D_1 = 4096

Tabelle 3.4: Mögliche Belegungen des Timer Control Registers (TCR).

3.6 Befehlsformat

Ein Mikroprogramm-Wort des Minirechners 2a besteht aus jeweils 28 Bit. Für die Steuerung der ALU, der Multiplexer und der Register im Datenpfad (vgl. Kapitel 2.1 und 2.2) stehen 17 Bits (Bit 0-16) zur Verfügung. Die entsprechenden Bits sind durch ein „M“ am Anfang gekennzeichnet (z.B. MRGWE, Ausnahme: MAC3...0).

Für die Steuerung des Adress-Decodierers, des Mikroprogramm-RAMs, des Befehlsregisters und des Memory-Controllers (vgl. Kapitel 3.1, 3.2 und 3.4) stehen insgesamt 11 Bits (Bit 17-27) zur Verfügung, welche zum Teil für den Zugriff auf das Daten-RAM zuständig sind.

Die Funktionen der einzelnen Bits sind in Tabelle 3.6 dokumentiert, während Tabelle 3.5 den generellen Aufbau der Mikroprogramm-Worte zeigt.

	Steuerwerk				Datenpfad							
Gruppe:	microprogram control		bus control		register control				ALU control			flag register control
Bedeutung:	microprog address control	next address	bus write	bus enable	register address port A	register address port B	register write port select	register write enable	ALU input A select	ALU input B select	ALU function select	change flags
Signal:	MAC 3 - 0	NA 4 - 0	BUSWR	BUSEN	MRGAA 3 - 0	MRGAB 3 - 0	MRGWS	MRGWE	MALUIA	MALUIB	MALUS 3 - 0	MCHFLG
Bit-Nr.:	27-24	23-19	18	17	16-13	12-9	8	7	6	5	4-1	0

Tabelle 3.5: Aufbau eines Mikroprogramm-Wortes.

Name	Anz. Bits	Bit Nr.	Bedeutung
Datenpfad:			
MCHFLG	1	0	Change Flags: 1 = Ausgänge C,Z,N der ALU in Register übernehmen
MALUS3...MALUS0	4	4...1	Funktion der ALU
MALUIB	1	5	Auswahl Eingang B der ALU: 0 = Register Port B 1 = Konstante
MALUIA	1	6	Auswahl Eingang A der ALU: 0 = Register Port A 1 = Datenbus MEMDI
MRGWE	1	7	Register Write Enable
MRGWS	1	8	Register Write Select: 0 = Write-Adresse ist AA2...AA0 1 = Write-Adresse ist AB2...AB0
MRGAB3...MRGAB0	4	12...9	Register Adresse Port B: 0nnn: Adresse = nnn 1xxx: Adresse = <0, OP11, OP10> und Konstante für Eingang B der ALU: 1000 ... 0111 = –8 ... +7
MRGAA3...MRGAA0	4	16...13	Register Adresse Port A: 0nnn: Adresse = nnn 1xxx: Adresse = <0, OP01, OP00>
BUSEN	1	17	Bus Enable: 1 = Datenbus wird angesprochen (read oder write)
BUSWR	1	18	Bus Write: Datenrichtung: 0 = lesen, 1 = schreiben
NA4...NA0	5	23...19	Next Adress: siehe Tabelle
MAC3...MAC0	4	27...24	Microcode Adress Control: siehe Tabelle MAC3 (Bit 27): Flag: 1 = letzter Befehl des Mikroprogramms (für single step)

Tabelle 3.6: Mikroprogramm-Bits.

4 Befehlssatzarchitektur

Der Minirechner 2a unterstützt insgesamt 48 Befehle in unterschiedlicher Größe (eins bis vier Byte). Er benötigt pro Befehle mehrere Takte und hat 2-Adress-Befehle, weshalb pro Befehle bis zu zwei Operanden geladen werden können. Er realisiert damit eine Complex-Instruction-Set-Computing-Architektur (CISC-Architektur). Die Bitbreite von acht gilt sowohl für die Breite der Daten als auch für die Breite der Register. Zusätzlich lässt sich der Minirechner 2a durch einen Assembler programmieren (siehe Anhang A)

4.1 Register

Der Minirechner 2a besitzt acht universelle Register. Diese werden jedoch zum Teil von der Hardware genutzt und verwaltet, weshalb nur drei zur universellen Nutzung verfügbar sind. Durch die steigende Komplexität durch die Implementierung von Maschinenbefehlen müssen einige Register für interne Zwecke genutzt werden. Nur die Register R0 bis R3 können durch die Befehle angesprochen werden, wobei das Register R3 den Programmzähler (PC) enthält. Das Register R4 ist nur vier Bit breit und enthält die Flag-Werte. Das Register R5 wird für die Verwaltung des Stackpointers (SP) und die Register R6 und R7 intern verwendet. Somit sind nur die Register R0 bis R2 universell nutzbar (vgl. Tabelle 4.1).

Der Minirechner 2a enthält eine im Blockschaltbild (Abbildung 1.1) nicht dargestellte Schaltung, die den Wert des Programmzählers und des Stackpointers überwacht. Verlässt einer der Werte den jeweils erlaubten Bereich, so stoppt die CPU unmittelbar und das jeweilige Register blinkt. Anstelle des Inhalts von R4 wird die Adresse des zuletzt gelesenen Befehls, der den Fehler verursacht hat, angezeigt.

Nach manueller Eingabe eines Programms in den Minirechner wird verhindert, dass der Programmzähler oder der Stackpointer auf den I/O-Bereich (F0 bis FF) zeigt. Dies könnte andernfalls zum Überschreiben von Teilen des Programms durch den Stack führen, falls das

Name	alternativ	Bedeutung	Name	alternativ	Bedeutung
R0		universell	R4	FR	Flags - durch Design festgelegt: 0.0.0.0:IE.N.Z.C
R1		universell	R5	SP	Stackpointer
R2		universell	R6		temporär innerhalb Makroprogramme
R3	PC	Programmzähler	R7		temporär innerhalb Makroprogramme

Tabelle 4.1: Übersicht der Registerinhalte.

Programm oder der Stack ungünstig im Speicher positioniert werden oder das Programm mehr Stackspeicher verwendet als verfügbar ist (z.B. durch rekursive Funktionsaufrufe).

Bei Verwendung des Assemblers wird der gültige Bereich für den Programmzähler dagegen automatisch bestimmt und der Stack auf 16 Byte limitiert. Letzteres kann über einen Befehl (vgl. Kapitel A.1) geändert werden.

4.2 Daten-RAM

Der Daten-RAM ist ein Speicher, welcher zusätzlich zu den Registern im Datenpfad zur Verfügung steht. Für die Nutzung der Befehlsbytes wird dieser RAM in verschiedene Bereiche eingeteilt, welche zusammenfassend in Abbildung 4.2 gezeigt werden. Die Adressen (in hexadezimaler Schreibweise) 00 bis EF sind dabei weiterhin als universeller Speicher nutzbar. Für die Erweiterungsplatine sind die Adressen F0 bis F7 vorgesehen.

Beim Schreiben auf den Daten-Bus wird in der Adresse FA das über die UART-Schnittstelle empfange Byte und in FB die UART-Status-Flags gespeichert. Die Adressen FC bis FF speichern die Registerinhalte der Eingaberegister „in FC“ bis „in FF“.

Beim Lesen steht in FA das über die UART-Schnittstelle zu sendende Byte und FB die UART-Steuerbits. Die Adressen FC und FD werden für den Interrupt-Timer verwendet, während die Speicherzellen FE und FF für die Ausgabe-Register „out FE“ und „out FF“ zur Verfügung stehen.

Bereich hex	Bedeutung bei read	Bedeutung bei write
00 - EF	Memory	
F0 - F7	Minibus (MR2DA2)	
F8	frei	
F9	Master Interrupt Status Register	Master Interrupt Control Register
FA	UART Daten-Register: empfangenes Byte	UART Daten-Register: zu sendendes Byte
FB	UART Status Register	UART Control Register
FC - FD	Input-Ports	Interrupt-Timer
FE - FF	Input-Ports	Output-Ports

Tabelle 4.2: Übersicht der Daten-RAM-Bereiche.

4.3 Befehlsbytes

Mit dem Befehlsbyte wählt man das gewünschte Mikroprogramm aus dem Mikroprogramm-RAM aus, legt die Startadresse innerhalb des Mikroprogramms für die Auswahl einer Funktion fest und übergibt die Adressen der Operanden. Jedes dieser Befehlsbytes ist, wie der Name bereits sagt, acht Bit lang. Es gibt insgesamt sechs verschiedene Formate (siehe Tabelle 4.3), die durch die Hardware unterstützt werden. Welche davon tatsächlich implementiert sind, wird durch die Mikroprogramme im Mikroprogramm-RAM festgelegt. In der Regel sind die acht Bit in zwei Bereiche unterteilt. Der Operation Code (kurz: Opcode), welche die auszuführende Operation auswählt, befinden sich auf Bit sieben bis Bit vier (OC3 bis OC0). Die anderen vier Bits für die Operanden (OP11 bis OP00) dienen grundsätzlich der Bestimmung der Quell- und Zielregister.

Einige der Formate benötigen mehr als nur ein Byte. Dies ist zum Beispiel bei Format 5 der Fall. Hier müssen vier Bytes verwendet werden, um den Befehl ordnungsgemäß ausführen zu können. Der in Format 3 und Format 5 erwähnte Adressmodus beschreibt unterschiedliche Arten der Registerreferenzierung und wird in Kapitel 4.4 näher erläutert. Das Format 6 zeigt den Aufbau des Befehlsbytes für bedingte Sprünge. Die drei Bits für die Bedingung können dabei die in Tabelle 4.4 gezeigten Werte und Bedeutungen annehmen. So müssen

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OC3	OC2	OC1	OC0	OP11	OP10	OP01	OP00

Format 1: Befehle ohne Operanden:

Opcode	Opcode	wird ignoriert

Format 2: Befehle mit einem Operanden, nur Register:

Opcode	Opcode	Register-Nr.

Format 3: Befehle mit einem Operanden, allgemeine Adressierung:

Opcode	Adressmodus	Register-Nr.
(ggf. Konstante oder Adresse)		

Format 4: Befehle mit zwei Operanden, nur Register:

Opcode	Nr. Quellregister	Nr. Zielregister

Format 5: Befehle mit zwei Operanden, allgemeine Adressierung:

1. Opcode	Quelle Adressmodus	Quelle Register-Nr.
(ggf. Konstante oder Quelladresse)		
2. Opcode	Ziel Adressmodus	Ziel Register-Nr.
(ggf. Zieladresse)		

Format 6: Bedingte Befehle:

Opcode	Opcode	Bedingung
(ggf. Konstante oder Adresse)		

Tabelle 4.3: Übersicht der Befehlsformate (Befehlbytes).

zum Beispiel die drei Bits OP10, OP01 und OP00 auf 011 (OP10..00 = 011) gesetzt werden, wenn bei einem negativen Ergebnis gesprungen werden soll (siehe auch Kapitel 3.1).

Die Tabelle 4.5 gibt einen Überblick über alle 48 Befehle. Die Spalten „Byte 1“ bis „Byte 4“ zeigen dabei das jeweilige Befehlsformat und die Anzahl der Bytes, die jeder Befehl benötigt, an. Die Spalte „Befehl“ zeigt einen Mnemonic, also eine sprechende Abkürzung, für die Funktion des Befehls und dient somit als Kurzform für die Spalte „Bedeutung“. Zusätzlich ist diese Spalte die Verbindung zwischen den Befehlsbytes und dem Assembler, welcher die selben Mnemonics verwendet (siehe Tabelle A.1). Zusätzlich gibt es Informationen über die Anzahl der benötigten Takte und welche Flags die Operationen dabei setzt.

Bits für Bedingung	Bedeutung	Bits für Bedingung	Bedeutung
000	immer wahr	100	immer falsch
001	Carry Flag (CF)	101	$\overline{\text{CF}}$
010	Zero Flag (ZF)	110	$\overline{\text{ZF}}$
011	Negative Flag (NF)	111	$\overline{\text{NF}}$

Tabelle 4.4: Übersicht Bedingungen für bedingte Sprünge.

Mikro-progr.-bereich	lfd. Nr.	Byte 1	Byte 2	Byte 3	Byte 4	Befehl	Takte	Flags IE	N	Z	C	Bedeutung
0	1	0000.00.00	-	-	-	STOPE		-	-	-	-	CPU stop (Hardware, Fehlerstop)
	2	0000.00.01	-	-	-	STOP		-	-	-	-	CPU stop (Hardware)
	3	0000.00.10	-	-	-	NOP	3	-	-	-	-	no operation
	4	0000.01.rn	-	-	-	CLR Rn	3	-	-	-	-	clear register
	5	0000.10.xx	-	-	-	EI	4	1	-	-	-	enable interrupt
	6	0000.11.xx	-	-	-	DI	4	0	-	-	-	disable interrupt
1	7	0001.00.rn	-	-	-	PUSH Rn	6	-	-	-	-	push register
	8	0001.01.rn 0001.01.11	-	-	-	POP Rn RET	6	-	-	-	-	pop register POP PC = return
	9	0001.10.00	-	-	-	PUSH F	6	-	-	-	-	push flags
	10	0001.11.00	-	-	-	POP F	5	*	*	*	*	pop flags
2	11	0010.0.ccc	offset	-	-	Jcc offs	4-5	-	-	-	-	conditional jump relative
	12	0010.10.00	adr	-	-	CALL adr	9	-	-	-	-	call subroutine
	13	0010.11.00	-	-	-	RETI	7	1	-	-	-	return from interrupt: POPF; RET
3	14	0011.00.rn	-	-	-	COM Rn	3	-	*	*	0	Einerkomplement
	15	0011.01.rn	-	-	-	NEG Rn	4	-	*	*	*	negate (Zweierkomplement)
	16	0011.10.rn	-	-	-	LSR Rn	3	-	*	*	b0	logical shift right
	17	0011.11.rn	-	-	-	ASR Rn	3	-	*	*	b0	arithmetic shift right
4	18	0100.00.rn	-	-	-	RRC Rn	3	-	*	*	b0	rotate right through carry
	19	0100.01.rn	-	-	-	INC Rn	3	-	*	*	*	increment
	20	0100.10.rn	-	-	-	TST Rn	3	-	*	*	0	Test Register
		0100.11.rn										- frei -
5	21	0101.00.rd 0101.11.11	- adr	-	-	DEC Rd DEC adr	3 10	-	*	*	*	decrement
6	22	0110.rs.rd 0110.rn.rn	-	-	-	ADD Rd,Rs LSL Rn	3	- -	* *	* *	* b7	add (logical) shift left
7	23	0111.rs.rd 0111.rn.rn	-	-	-	ADC Rd,Rs RLC Rn	3	- -	* *	* *	* b7	add with carry rotate left through carry
8	24	1000.rs.rd	-	-	-	SUB Rd,Rs	5	-	*	*	*	subtract
9	25	1001.rs.rd	-	-	-	AND Rd,Rs	8	-	*	*	0	AND
10	26	1010.rs.rd	-	-	-	OR Rd,Rs	6	-	*	*	0	OR
11	27	1011.rs.rd	-	-	-	MUL Rd,Rs	7-37	-	*	*	*	multiplicate
12	28	1100.rs.rd	-	-	-	DIV Rd,Rs	6-519	-	*	*	*	divide
13	29	1101.rs.rd	-	-	-	XOR Rd,Rs	9	-	*	*	0	exclusive OR
14		1110.xx.xx										- frei -
15.1	30	1111.ms.rs	(const/adr)	0001.md.rd	(adr)	MOV dst,src	6-14	-	-	-	-	move source to destination
		1111.10.11	const	0001.00.rd	-	LD Rd,const	8	-	-	-	-	load reg. with constant
		1111.11.11	adr	0001.00.rd	-	LD Rd,(adr)	10	-	-	-	-	load reg. from address
		1111.00.rs	-	0001.11.11	adr	ST (adr),Rs	10	-	-	-	-	store reg. to address
		1111.10.11	adr	0001.00.11	-	JMP adr	8	-	-	-	-	LD PC,(PC+) = jump absolute
		1111.11.rn	adr of adr	0001.00.11	-	JMP (adr)	10	-	-	-	-	LD PC,((PC+)) = jump indirect
15.2	31	1111.ms.rs	(const/adr)	0010.md.rd	(adr)	CMP dst,src	8-16	-	*	*	*	compare (vgl. SUB dst,src)
		1111.00.rs	-	0010.00.rd	-	CMP Rd,Rs	8					
		1111.10.11	const	0010.00.rd	-	CMP Rd,const	10					
		1111.11.11	adr	0010.00.rd	-	CMP Rd,(adr)	12					
15.3	32	1111.ms.rs	(const/adr)	0011.md.rd	(adr)	BITT dst,src	9-17	-	*	*	0	Bit-Test (AND ohne speichern)
15.4	33	1111.ms.rs	(const/adr)	0100.00.00	-	LDSP	5-9	-	-	-	-	load stack pointer
	34			0100.01.00		LDFR	6-10	*	*	*	*	load flag register
15.5	35	1111.ms.rs	(const/adr)	0101.md.rd	(adr)	BITS dst,src	8-17	-	*	*	0	set bits: dst = dst OR src
15.6	36	1111.ms.rs	(const/adr)	0110.md.rd	(adr)	BITC dst,src	9-20	-	*	*	0	clear bits: dst = dst AND(NOT src)

ms = Adressmodus source (2 Bit) ( ) = je nach Adressmodus
md = Adressmodus destination (2 Bit) xx = egal
rs = Registernummer source (2 Bit) Rs = Source-Register
rd = Registernummer destination (2 Bit) Rd = Destination-Register
Flags entsprechend Tabelle A.1.

Tabelle 4.5: Kodierung der Befehle für den Minirechner 2a

4.4 Adressmodi

Die Adressierung der Register kann auf vier Arten geschehen:

1. R = Der Inhalt des Registers wird ausgelesen oder beschrieben.
2. (R) = Der Inhalt des Registers wird als Adresse interpretiert, welche dann ausgelesen oder beschrieben wird.
3. (R+) = Wie (R), zusätzlich wird die Adresse in R danach inkrementiert.
4. ((R+)) = Der Inhalt des Registers wird als Adresse interpretiert. Der Wert in dieser Adresse wird dann ebenfalls als Adresse interpretiert, die dann ausgelesen bzw. beschrieben wird. Zusätzlich wird danach die Adresse in R inkrementiert.

Die Tabelle 4.6 zeigt eine Übersicht über die Adressmodi, wobei die Abkürzung PC für den Programmzähler in Register R3 steht.

Modus	Schreib-weise	Bedeutung	mit R = PC Quelle	mit R = PC Ziel
0 0	R	Register	nicht sinnvoll	Jump
0 1	(R)	Register indirekt	nicht sinnvoll	nicht sinnvoll
1 0	(R+)	Register indirekt postincrement	Konstante	nicht sinnvoll
1 1	((R+))	Register doppelt indirekt postincr.	Adresse	Adresse

Tabelle 4.6: Adressmodi für die Register.

Handelt es sich bei dem Quell-, bzw. Zielregister um den Programmzähler (also R = PC), so gelten folgende Sonderfälle:

- *PC als Ziel:* Jump; z.B. „JMP 55“ entspricht „LD PC,55“ (springe zu Adresse 55)
- *(PC+) als Quelle:* Das Byte, auf das der Programmzähler nach dem Lesen des Opcodes zeigt, also das dem Opcode folgende Byte, wird gelesen. Danach wird der Programmzähler inkrementiert, zeigt dann also auf das nächste zu lesende (Opcode-)Byte. Diese Adressierungsart erlaubt es, eine Konstante direkt in den Befehl (nach dem Opcode) zu schreiben. (Schreibweise: „const“)
- *((PC+)) als Quelle oder Ziel:* Das dem Opcode folgende Byte wird gelesen. Das gelesene Byte wird als Adresse interpretiert, von dieser Adresse wird gelesen oder an diese Adresse wird geschrieben. Danach wird der Programmzähler inkrementiert. Auf diese Weise kann eine konstante Adresse angegeben werden. (Schreibweise: „(adr)“)

4.5 Unterprogramme

Die Ausführung von Unterprogrammen ist beim Minirechner 2a über einen Stack, welcher invertiert (von oben nach unten) angelegt wird, realisiert. Der Stackpointer zeigt dabei auf den zuletzt geschriebenen Wert. Beim Schreiben in den Stack wird zuerst der Stackpointer dekrementiert und dann an diese Adresse geschrieben. Beim Lesen wird zuerst aus der Adresse gelesen und dann der Stackpointer inkrementiert.

Beim Aufruf eines Unterprogramms mit dem Befehl „CALL“ wird der aktuelle Stand des Programmzählers (Adresse des Befehls nach „CALL“) auf den Stack geschrieben und bei der Rückkehr vom Unterprogramm mit dem Befehl „RET“ wieder aus dem Stack gelesen. Mit den Befehlen „PUSH“ und „PUSHF“ kann man die Inhalte von Registern auf den Stack sichern und mit „POP“ und „POPF“ wieder zurücklesen (Reihenfolge beachten).

Die obersten 16 Bytes des Daten-RAMs (Adressen F0 bis FF, vgl. Tabelle 4.2) können nicht verwendet werden, da mit diesen Adressen die Peripherie angesprochen wird. Deshalb legt man den Stack am sinnvollsten ab der Adresse EF abwärts an. Dazu muss man den Stackpointer vor dem ersten „CALL“ bzw. „PUSH“ mit dem Befehl „LDSP 0xEF“ auf den Wert EF setzen (nicht auf F0, vgl. Kapitel 4.2) und darf ihn danach nicht mehr direkt ändern.

Will man Variablen im Daten-RAM anlegen, so kann man ihre Adressen im Programm direkt als Zahl oder als Label angeben (jeweils in Klammern). Am sinnvollsten ist es, wenn man für jede Variable ein Label und nachfolgend die Assembler-Anweisung .BYTE oder .DB verwendet (vgl. Kapitel A.1). Auf diese Weise kann man die Labels als Adressen für die Variablen benutzen:

```
...
LD      R0,(VAR1)
ST      (VAR2),R0
...
VAR1:
.DB     23
VAR2:
.BYTE   1
```

Tabelle 4.7: Ausschnitt eines Programms, das Labels als Variablen verwendet

5 Bedienung

Der Minirechner 2a kennt zwei Betriebsmodi: den Programmiermodus und den Run-Modus. Im Programmiermodus können per Taster Mikroprogramm-Wörter in das Mikroprogramm-RAM und Befehlsbytes in das Daten-RAM geschrieben oder ausgelesen werden. Das Beschreiben des Mikroprogramm-RAMs wird nicht empfohlen, da so die Firmware geändert wird. Im Run-Modus werden die Befehlsbytes ausgelesen und ausgeführt.

5.1 Inbetriebnahme und Einstellungen

Bevor der Minirechner 2a in Betrieb genommen werden kann müssen die Minirechner-Logik und die Taktrate ausgewählt werden. Hierfür müssen auf der Logikplatine (mittlere Platine) der linke Drehschalter auf Position 4 (Auswahl des Minirechner 2a) und der rechte Drehschalter auf Position 4 (Auswahl der Taktquelle; 7.3728 MHz) gestellt werden.

Der Minirechner muss mit einer Spannung von mindestens 12 V versorgt werden. Blinken alle LEDs kurz auf, so ist der Minirechner betriebsbereit. Durch das Drücken des Tasters **LOAD** werden die Einstellungen neu geladen. Das heißt vor allem, dass die Register gelöscht und das Mikroprogramm-RAM neu geladen werden.

Hält man die blaue **CONTROL**-Taste gedrückt, so können mit den Tastern des Display-Boards verschiedene Einstellungen vorgenommen werden. Es gelten dann die blauen Beschriftungen. Alle folgenden Ausführungen gelten bei gedrückter **CONTROL**-Taste.

Die Quelle des Grundtaktes für die CPU kann mittels Drehschalter auf der FPGA-Platine gewählt werden. Nach dem Einschalten bzw. Laden der Minirechner-Konfiguration in das FPGA ist als Quelle immer der Quarz am Mikrocontroller (7,37 MHz) gewählt; die Auswahl am Drehschalter wird erst durch Drücken des Tasters **SET** wirksam. Im FPGA gibt es zwei einstellbare Taktteiler, die diesen Grundtakt durch 2 bis 2^{25} teilen. Mit den Tastern **slower** bzw. **faster** kann der Takt des aktiven Taktteilers in 25 Stufen langsamer bzw. schneller gestellt werden. Mit der Taste **step 1/4** wird zwischen 1-er oder 4-er Schritten gewählt (rote LED leuchtet bei 4-er-Schritten). Mit der Taste **default** wird ein voreingestellter Wert ausgewählt. Die 4 grünen LEDs an den kleinen Tastern zeigen binär die unteren vier Bit der Nummer der eingestellten Taktrate an (00000 bis 11000, größere Nummer = langsamerer Takt). Die rote LED „clock“ blinkt mit dem eingestellten Takt.

Mit den Tastern **darker** bzw. **brighter** passen Sie die Helligkeit aller LEDs des Displays an, mit **default** wird eine voreingestellte Helligkeit ausgewählt.

Der Taster **LED test** schaltet alle LEDs auf dem Display-Board ein. Der Taster **LEDs off** schaltet alle LEDs aus. Der Normalbetrieb wird mit dem Taster **LEDs normal** eingestellt.

5.2 Programmiermodus

Die Programmierung des Minirechners erfolgt über die manuelle Eingabe der Befehlsbytes oder durch die Übertragung des kompilierten Assemblercodes (vgl. Kapitel A.2). Zum Erstellen der Befehlsbytes eignet sich die Tabelle in Anhang C. Die Programme können dann per Taster (siehe Tabelle 5.1) byteweise in das Daten-RAM des Minirechners eingegeben werden. Für diese Eingabe muss jedes Befehlsbyte zuerst in ein Eingaberegister geschrieben und dann der Inhalt dieses Eingaberegisters in das Daten-RAM kopiert werden.

Die Eingabe der Befehlsbytes geschieht im Einzelnen durch folgende Schritte:

1. Schalter **load/run** auf „load“, Schalter **read/write** auf „write“.
2. Mit dem Taster **bus** als Ziel den Daten-Bus (MEMDI) auswählen.
3. Schalter **shift/count** auf „count“ (Adresse hoch- bzw. runterzählen).
4. Mit den Tastern **up** bzw. **down** die Adresse auswählen.
5. Das Befehlsbyte mit den Tastern **0** und **1** eintippen.
6. Durch Drücken von **store** die Eingabe in das Daten-RAM kopieren
7. Nächstes Befehlsbyte: zurück zu 4.

Schalter / Taster	Funktion
run / step	—
step	—
reset	CPU-Register, Output-Register und UART werden gelöscht. Ausgang und Adresseingang des Mikroprogramm-RAMs werden auf 0 gesetzt. Eingabe-Register, Adressregister und Inhalte von Mikroprogramm- und Daten-RAM bleiben unverändert.
load / run	Zwischen Programmier- und Run-Modus umschalten.
write / read	write: Mikroprogramm-RAM bzw. Daten-RAM-Bus sind beschreibbar. read: Schreibschutz (Daten können nur gelesen werden)
store	write (grüne LED über dem Taster leuchtet): Schreiben des Wertes ins Eingaberegister (LEDs „load: data“) in das Mikroprogramm-RAM bzw. auf den Daten-RAM-Bus. Die grüne LED über dem Taster erlischt, wenn der Taster gedrückt ist. read (grüne LED über dem Taster leuchtet nicht): Erzeugen eines read-Strobe (nur von Bedeutung beim Datenregister des UART).
clear	Löschen des gesamten Eingaberegisters (LEDs „data“).
0	0-Bit von rechts in das Eingaberegister einschieben.
1	1-Bit von rechts in das Eingaberegister einschieben.
clear	Adressregister löschen.
shift / count	Funktion der beiden Taster rechts auswählen.
0 / down	Schalter auf „shift“: 0-Bit von rechts in das Adressregister einschieben. Schalter auf „count“: Adressregister um 1 runterzählen.
1 / up	Schalter auf „shift“: 1-Bit von rechts in das Adressregister einschieben. Schalter auf „count“: Adressregister um 1 hochzählen.
continue	Setzt beim Drücken das Flip-Flop IFF1 (siehe Steuerwerk)
in FF / in FE	Auswahl, ob in das Mikroprogramm-RAM oder auf den Bus geschrieben werden soll. Rote LEDs neben den Tastern zeigen die aktuelle Auswahl.
in FD / in FC	—

Tabelle 5.1: Tasterbelegung im Programmiermodus (von links nach rechts, vgl. Abb. 2.4).

5.3 Run-Modus

Die im Programmiermodus gespeicherten Programme (Abfolge von Befehlsbytes) können im Run-Modus ausgeführt werden. Dieser Modus wird durch das Umstellen des Schalters **load/run** auf „run“ eingestellt. Insbesondere können nun auch die Input-Register FC, FD, FE und FF beschrieben werden. Um eines der Register auszuwählen, wird der unter der Beschriftung liegende Taster gedrückt, sodass die entsprechende grüne LED aufleuchtet. Die Register werden dann mit den Tastern **0** und **1** (mittig im Bedienfeld) beschrieben.

Ist der Kippschalter **run/step** auf „step“ gestellt, so kann das Programm mit dem roten **step**-Taster daneben befehlsweise ausgeführt werden. Ist stattdessen „run“ eingestellt, so läuft das Programm mit der eingestellten Taktfrequenz. Ist der letzte Programmbefehl mit einem Sprung auf eine vorherige Programmzeile (z.B. Adresse 0) versehen, so läuft das Programm in einer Schleife.

Im Run-Modus wird das Eingaberegister „load: data“ in die vier einzelnen Input-Register FC, FD, FE und FF aufgeteilt, die von der CPU unter den Daten-RAM-Adressen FC bis FF (hexadezimal) gelesen werden können. In diesem Modus können die vier Bytes unabhängig beschrieben werden (Auswahl eines Bytes durch die kleinen Taster darunter). Alle Tasterbelegungen sind in Tabelle 5.2 zu finden.

Schalter / Taster	Funktion
run / step	step: single step: Abarbeiten des Mikroprogramms im Einzelschritt-Modus; run: Mikroprogramm wird fortlaufend abgearbeitet
step	Wenn Schalter **run/step** auf „step“ steht, wird pro Tastendruck ein einzelner Taktschritt ausgeführt.
reset	CPU-Register, Output-Register und UART werden gelöscht. Ausgang und Adresseingang des Mikroprogramm-RAMs werden auf 0 gesetzt. Eingabe-Register, Adressregister und Inhalte von Mikroprogramm- und Daten-RAM bleiben unverändert. Das Signal „reset“ (siehe rote LED im oberen Teil) bleibt noch 2 Takte nach dem Loslassen des Tasters aktiv. Im Single-Step-Modus muss danach also zweimal **step** gedrückt werden. Wenn gleichzeitig die Taste **CONTROL** auf dem FPGA-Board gedrückt ist: Eingang 111 des A0-Multiplexers liegt auf high, solange reset gedrückt bleibt (**loop**).
load / run	Zwischen Programmier- und Run-Modus umschalten.
write / read	—
store	—
clear	Löschen des aktiven Input-Registers (grüne LED leuchtet).
0	0-Bit von rechts in das aktive Input-Register einschieben.
1	1-Bit von rechts in das aktive Input-Register einschieben.
clear	—
shift / count	Funktion der beiden Taster rechts im Programmiermodus auswählen.
0 / down	Aktives Input-Register um 1 runterzählen.
1 / up	Aktives Input-Register um 1 hochzählen.
continue	Setzt beim Drücken das Flip-Flop IFF1 (siehe Steuerwerk)
in FF / in FE / in FD / in FC	Auswahl, welches der 4 Input-Register mit den Tastern **clear**, **0**, **1**, **down** und **up** angesprochen werden soll. Grüne LEDs neben den Tastern zeigen aktuelle Auswahl an.

Tabelle 5.2: Tasterbelegung im Run-Modus (von links nach rechts, vgl. Abb. 2.4).

Anhang

A Assembler

Der Assembler ist ein Programm, das ein Maschinenprogramm aus der lesbaren Form (z.B. „ADD R0,R1“) in die entsprechenden Befehlsbytes (01100100) übersetzt. Die Maschinenprogramme werden dabei auf einem Desktop-PC geschrieben, übersetzt und danach für die Ausführung an den Minirechner 2a übertragen. Für die Übersetzung und Übertragung auf den Minirechner 2a muss das Kommandozeilenprogramm „mcontrol“ verwendet werden. Diese Software wird durch Werner Dreher (dem Entwickler des Minirechners 2a) zu Verfügung gestellt. Es ist jedoch nicht frei verfügbar.

A.1 Aufbau des Quellcodes

Der Assembler des Minirechners 2a arbeitet zeilenorientiert, das heißt pro Zeile kann immer nur eine Anweisung stehen. Gleichzeitig muss eine Anweisung vollständig in einer Zeile stehen. Es wird empfohlen die Quelldateien mit der Dateiendung „.asm“ zu speichern.

Der Assembler unterscheidet 6 verschiedene Zeilentypen:

1. Kennzeile: In der ersten Zeile muss der Text „#! mrasm“ als Kennzeichnung stehen.
2. Leerzeilen: werden ignoriert.
3. Kommentarzeilen: müssen mit einem Semikolon („ ; “) als erstem sichtbaren Zeichen beginnen und werden vom Assembler ignoriert.
4. Assembler-Anweisungen: Anweisungen an den Assembler selbst; müssen mit einem Punkt beginnen.
5. Labels/Sprungziele: symbolische Namen für Adressen im Programm.
6. Maschinenbefehle (Assemblerbefehle; siehe Tabelle A.1).

Außer der Kennzeile dürfen die Zeilen in beliebiger Reihenfolge vorkommen, soweit dies einen Sinn ergibt. Zwischen Groß- und Kleinbuchstaben wird nicht unterschieden. Leerzeichen und Tabulator-Zeichen am Beginn einer Zeile werden ignoriert (außer bei der Kennzeile). Alles ab einem „;“ innerhalb einer Zeile wird ignoriert (Kommentar).

Assembler-Anweisungen und Assemblerbefehle bestehen aus einem Mnemonic (Abkürzung des Befehlsnamens) und ggf. einem oder mehreren Parametern. Zwischen Mnemonic und Parametern müssen ein oder mehrere Leerzeichen oder ein Tabulatorzeichen stehen,

die Parameter untereinander müssen durch Kommas getrennt werden. Zahlen ohne weitere Angabe werden dezimal interpretiert, mit „0x“ davor hexadezimal und mit „0b“ davor binär (z.B. 14 = 0x0e = 0x0E = 0b1110).

Assembler-Anweisungen

`.ORG n`
set program origin: Adresszähler auf den Wert `n` setzen. Der Assembler-interne Adresszähler gibt an, an welche Adresse im Befehlsregister der nächste Befehl geschrieben wird.
`.BYTE n`
Platz für `n` Byte im Befehlsregister freilassen. Mit dieser Anweisung kann Platz für Variablen reserviert werden. Es wird einfach `n` zum Adresszähler addiert.
`.DB b1,b2,b3,...`
define bytes: Bytes `b1,b2,...` an die aktuelle Adresse im Programmspeicher schreiben. Damit können z.B. Konstanten in das RAM des Minirechners geschrieben werden.
`.DW w1,w2,w3,...`
define words: 16-Bit-Werte `w1,w2,...` an die aktuelle Adresse im Befehlsregister schreiben.
`.EQU name n`
Der Zahl n wird einen Namen zugewiesen. Dieser Name verhält sich wie ein Label. Es gelten die gleichen Regeln und die gleiche Einschränkungen, die auch für Labels gelten.
`*STACKSIZE n`
Manuelle Festlegung der Stackgröße; dabei können für n folgende Werte verwendet werden: 16, 32, 48, 64 Byte und NOSET (unbeschränkt).

Labels

Ein Label wird vom Assembler mit dem Inhalt des Adresszählers an der Stelle des Labels ersetzt. Labels müssen aus einem Wort bestehen und dürfen nicht mit „R“, „PC“ oder „SP“ beginnen. Sie müssen in einer extra Zeile stehen und mit einem Doppelpunkt enden; dahinter darf höchstens ein Kommentar stehen. Labels können als Sprungziel bei Jumps und Calls, als sonstige Memory-Adressen oder anstelle von Konstanten angegeben werden. Bei Sprungzielen müssen Labels verwendet werden (auch bei relativen Sprüngen, der Assembler berechnet dann die Sprungweite). Der Assembler kann maximal 40 Labels verwalten.

A.2 Übersetzen und Übertragung des Assemblers

Das Übersetzen der Assembler-Datei und das Übertragen des Codes auf den Minirechner 2a geschieht durch das Programm „mcontrol“. Die Verbindung zwischen einem Computer und dem Minirechner wird über einen Universal Serial Bus (USB) Typ A (für den PC) zum USB Typ B (für den Minirechner) hergestellt. Der Anschluss am Minirechner 2a befindet sich unter der Spannungsversorgung. Nach dem Aufruf von „mcontrol“ in der Eingabeaufforderung (vorher in das Verzeichnis mit den Source-Files wechseln) wartet das Programm auf Kommandos. Nach jedem Einschalten des Minirechners muss „mcontrol“ mit dem Initialisierungskommando „INIT“ dazu gebracht werden, mit dem Minirechner Verbindung aufzunehmen.

Mit dem Befehl „ASM TEST filename“ wird der Quellcode auf formale Fehler überprüft und dann in die entsprechenden Befehlsbytes übersetzt. Dadurch entsteht eine neue Datei mit der Endung „.lst“, welche die Befehlsbytes in hexadezimaler Schreibweise beinhaltet.

Der Befehl „ASM RUN filename“ durchläuft die selben Schritte wie der Befehle „ASM TEST filename“, überträgt aber zusätzlich die übersetzte Datei an den Minirechner 2a.

Befehl	Operanden	BF	ausgeschrieben	IE	N	Z	C	Bemerkung	
arithmetische Operationen:									
CLR	Rn	2	clear register n	-	-	-	-	Rn = 0	
ADD	Rd, Rs	4	add	-	*	*	*	Rd = Rd + Rs	
ADC	Rd, Rs	4	add with carry	-	*	*	*	Rd = Rd + Rs + CY	
SUB	Rd, Rs	4	subtract	-	*	*	*	Rd = Rd - Rs	
MUL	Rd, Rs	4	multiplicate	-	*	*	*	Rd = Rd * Rs	(obere 8 Ergebnisbits gehen verloren)
DIV	Rd, Rs	4	divide	-	*	*	*	Rd = Rd / Rs	(Divisionsrest geht verloren)
INC	Rn	2	increment	-	*	*	*	Rn = Rn + 1	
DEC	src	3	decrement	-	*	*	*	src = src - 1	
NEG	Rn	2	negate	-	*	*	*	Rn = (NOT Rn) +1	(Zweierkomplement)
logische Operationen:									
AND	Rd, Rs	4	logical AND	-	*	*	0	Rd = Rd AND Rs	(bitweise)
OR	Rd, Rs	4	logical OR	-	*	*	0	Rd = Rd OR Rs	(bitweise)
XOR	Rd, Rs	4	exclusive OR	-	*	*	0	Rd = Rd XOR Rs	(bitweise)
COM	Rn	2	complement	-	*	*	0	Rn = NOT Rn	(bitweise) (Einerkomplement)
BITS	dst, src	5	set bits	-	*	*	0	dst = dst OR src	(bitweise)
BITC	dst, src	5	clear bits	-	*	*	0	dst = dst AND (NOT src)	(in src gesetzte Bits in dst löschen)
Tests und Vergleiche:									
TST	Rn	2	test register	-	*	*	0	Flags setzen entsprechend dem Inhalt von Rn	
CMP	dst, src	5	compare	-	*	*	*	temp = dst - src	Flags werden gesetzt
BITT	dst, src	5	bit test	-	*	*	0	temp = dst AND src	Flags werden gesetzt
Bitschiebe-Operationen:									
LSR	Rn	2	logical shift right	-	*	*	b0	CY = Rn[0]; Rn[x] = Rn[x+1]; Rn[7] = 0	([x] = Bit Nr. x)
ASR	Rn	2	arithmetic shift right	-	*	*	b0	CY = Rn[0]; Rn[x] = Rn[x+1]; Rn[7] = Rn[7]	
LSL	Rn	2	(logical) shift left	-	*	*	b7	CY = Rn[7]; Rn[x] = Rn[x-1]; Rn[0] = 0	(= ADD Rn,Rn)
RRC	Rn	2	rotate right through carry	-	*	*	b0	Rn[7] = CY; CY = Rn[0]; Rn[x] = Rn[x+1]	
RLC	Rn	2	rotate left through carry	-	*	*	b7	Rn[0] = CY; CY = Rn[7]; Rn[x] = Rn[x-1]	
Daten kopieren:									
MOV	dst, src	5	move (copy)	-	-	-	-	dst = src	
Sonderfälle:									
LD	Rn, const		load reg. with constant					MOV Rn, (PC+)	const steht nach erstem Befehlsbyte
LD	Rn, (adr)		load reg. from address					MOV Rn, ((PC+))	adr steht nach erstem Befehlsbyte
ST	(adr), Rn		store reg. to address					MOV ((PC+)), Rn	adr steht nach zweitem Befehlsbyte
PUSH	Rn	2	push register n	-	-	-	-	SP = SP - 1; (SP) = Rn	Register auf Stack sichern
POP	Rn	2	pop register n	-	-	-	-	Rn = (SP); SP = SP + 1	Register von Stack zurücklesen
PUSHF		1	push flags	-	-	-	-	SP = SP - 1; (SP) = FR	Flags auf Stack sichern
POPF		1	pop flags	*	*	*	*	FR = (SP); SP = SP + 1	Flags vom Stack zurücklesen
LDSP	src	5*	load stackpointer	-	-	-	-	SP = src	
LDFR	src	5*	load flag register	*	*	*	*	FR = src	
Programmverzweigung:									
JMP	adr	5	jump absolute	-	-	-	-	PC = adr [Sonderfall von MOV dst,src: MOV PC, (PC+)]	
Jcc	offset	6	conditional jump relative	-	-	-	-	PC = PC + offset (wenn Bedingung erfüllt):	
								JCS = jump if carry flag set	JCC = jump if carry flag cleared
								JZS = jump if zero flag set	JZC = jump if zero flag cleared
								JNS = jump if negative flag set	JNC = jump if neg flag cleared
JR	offset	6	jump relative	-	-	-	-	Sonderfall von Jcc (Bedingung immer wahr)	
CALL	adr	-	call subroutine	-	-	-	-	= PUSH PC; JMP adr	[SP = SP-1; (SP) = PC; PC = adr]
RET		1	return	-	-	-	-	Sonderfall von POP Rn: POP PC	
RETI		1	return from interrupt	1	-	-	-	= POPF; RET	[Interrupt 1 Befehl verzögert]
sonstige:									
STOP		1*	CPU stop	-	-	-	-	CPU stoppt; weiter mit Tasten CONTROL + reset	
NOP		1*	no operation	-	-	-	-	der Befehl bewirkt nichts, auch die Flags bleiben unverändert	
EI		1	enable interrupts	1	-	-	-	FR[IE] = 1	[1 Befehl verzögert]
DI		1	disable interrupts	0	-	-	-	FR[IE] = 0	

Tabelle A.1: Implementierte Befehle im Minirechner 2a

Erläuterungen: BF = Befehlsformat (evt. erweitert*); IE,N,Z,C = Interrupt Enable, Negative, Zero, Carry (- = unverändert, * = entsprechend Ergebnis); Rn = Register n; Rs/Rd = Quell-/Zielregister (R0 – R3); src = allgemeine Quelle (Rs, (Rs), (Rs+), ((Rs+)), const, (adr)); dst = allgemeines Ziel

B Beispielprogramm

Die Adresse 0 ist der Einstiegspunkt für jedes Programm, dort muss der erste Befehl stehen. Um die Ausführung zu vereinfachen sollte die letzte Zeile immer eine Sprung zu einer vorherigen Zeile beinhalten. So kann das Programm in einer Schleife durchlaufen werden. Dies ist vor allem für das Testen verschiedener Werte nützlich. Hierzu lohnt es sich wie in Kapitel 5.3 beschrieben, den Kippschalter **run/step** auf „run“ zustellen (so läuft das Programm in einer Schleife). Fällt während des Testens ein Fehler auf, so kann durch Umschalten auf „step“ das Programm befehlsweise debugged werden. Dabei können die Register R3 und R5 auf eventuelle Fehler hinweise, da diese den Programmzähler und den Stackpointer beinhalten. Blinkt zum Beispiel das Register R5, so handelt es sich möglicherweise um einen Stack-Überlauf oder um einen nicht initialisierten Stackpointer.

Folgendes Beispiel zeigt die Herleitung des Quellcodes für ein Programm, welches einen Register-Wert auf ein Ausgabe-Register (Adresse FF) und in den Daten-RAM (Adresse F0) legt und diesen Wert dann inkrementiert.

1. **Leere das Register R0 (adr. 0)**. Damit man mit dem Wert null startet, soll zu erst das Register R0 geleert werden. Hierzu wird der Befehl „CLR Rn“ (Rn = R0) benutzt. Laut Tabelle 4.5 muss hierfür der Befehle wie folgt aussehen: 0000.01.rn, wobei „rn“ durch die Nummer des Registers R0 ersetzt wird. Somit ergibt sich für den ersten Befehl folgendes Befehlsbyte: 0000.01.00.

2. **Lade den Wert in R0 ins Output-Register FF (adr. 1)**. Der vorherige Befehl ist nur ein Byte lang, weshalb der zweite Befehle in Adresse 01 beginnt. Hier soll der Wert des Registers R0 in das Ausgabe-Register FF geschrieben werden. Hierfür wird der Befehl „ST (adr), Rs“ verwendet. Dabei steht „Rs“ für das Quellregister (in diesem Fall R0) und „(adr)“ für die Adresse, in der der Wert geschrieben werden soll. Laut Tabelle 4.5 ist dieser Befehle drei Byte lang: 1111.00.rs - 0001.11.11 - adr
   Durch die Ersetzung (rs = 00, adr = FF = 1111.1111) ergeben sich folgende Bytes:
   1111.00.00 - 0001.11.11 - 1111.1111

3. **Lade den Wert in R0 in die Daten-RAM-Adresse F0 (adr. 4)**. Da der vorherige Befehle drei Byte lang ist und somit die Adressen 01, 02 und 03 benötigt, startet der dritte Befehl in Adresse 04. Hier soll im wesentlichen das selbe gemacht werden, wie im zweiten Schritt. Es ändert sich lediglich die Adresse auf F0 = 1111.0000. So ergeben sich folgende drei Befehlsbytes: 1111.00.00 - 0001.11.11 - 1111.0000

Zeile	Adr. (hex)	Befehl	Adresse	Byte 1	Byte 2	Byte 3	Byte 4
1	00	CLR R0	0000.0000	0000.01.00			
2	01	ST (FF),R0	0000.0001	1111.00.00	-	0001.11.11	1111.1111
3	04	ST (F0),R0	0000.0100	1111.00.00	-	0001.11.11	1111.0000
4	07	INC R0	0000.0111	0100.01.00			
5	08	JR 01	0000.1000	0010.0.000	1111.0111		

a) Maschinenprogramm

```
#! mrasm

.ORG 0
CLR     R0
LOOP:
ST      (0xFF),R0
ST      (0xF0),R0
INC     R0
JR      LOOP
```

b) Assemblerprogramm

Tabelle B.1: Vergleich Mikroprogramm- und Assemblercode.

4. **Inkrementiere den Wert in R0 (adr. 7)**. Da der vorherige Befehle drei Byte lang ist und somit die Adressen 04, 05 und 06 benötigt, startet der dritte Befehl in Adresse 07. Hier soll nun der Wert im Register R0 um eins erhöht, also inkrementiert werden. Hierfür ist der Befehl „INC Rn“ vorgesehen und äußert sich durch das einzelne Byte 0100.01.rn, wobei „rn“ wieder für die Adresse des Registers R0 vorgesehen ist. Somit ergibt sich für den vierten Befehl das Byte 0100.01.00

5. **Springe zurück zu Adresse 01 (Zeile 2) (adr. 8)**. Der vorherige Befehl ist nur ein Byte lang, weshalb der letzte Befehl in Adresse 08 steht. Dieser dient als Sprung zum zweiten Befehl (Adresse 01), um so ein einer Schleife den neuen Register-Wert wieder auszugeben und zu erhöhen. Für diesen Sprung wird der Befehle „Jcc offs“ verwendet. Dieser wird nach Tabelle 4.5 als relativer bedingter Sprung beschrieben und teilt sich in zwei Bytes auf: 0010.0.ccc - offset
   Hierbei steht „ccc“ für die Sprung-Bedingung und „offset“ für die relative Anzahl der Adressen, die „übersprungen“ werden sollen. Die Tabelle 4.4 zeigt die möglichen Sprung-Bedingungen. Da dieser Sprung immer stattfinden soll, wird somit ccc = 000 gesetzt. Dies ist auch der Grund, warum dieser Befehl auch als „JR offs“ bezeichnet wird. Das „JR“ steht dabei für Jump Relative und bezeichnet einen Sprung, der immer (bedingungslos) ausgeführt wird. Der Offset beträgt 247, da das letzte Byte in Adresse 09 steht und der Sprung von dieser Adresse ausgeht. So ergeben sich die beiden Bytes: 0010.0.000 - 1111.0111

Die Tabelle B.1 zeigt sowohl die oben hergeleitet Befehlsbytes als auch den dazugehörigen Assemblercode. Der Assembler nutzt dabei exakt die selben Befehle, welche jedoch nicht mehr händisch in die einzelnen Bytes übersetzt werden müssen. Lediglich der relative Sprung ist über das Label „LOOP“ realisiert.

C Emulator

Der Emulator für den Minirechner 2a bietet eine Möglichkeit selbst geschriebene Programme zu testen. Der von Malte Tammena entwickelte Emulator ist in der Programmiersprache `Rust` geschrieben. Als reines Kommandozeilenprogramm bietet es dennoch ein Terminal User Interface (TUI) für grafische Darstellungen.

C.1 Installation

Die Installation kann auf zwei Wegen erfolgen. Man kann entweder ein direkt einsetzbares Binary herunterladen, oder den Quellcode downloaden und selbst kompilieren.

Installation der Binaries

Für Windows und Linux sind Binaries, also vorkompilierte Programmdateien, verfügbar[1]. Nach dem Download der entsprechenden Archivdatei kann diese in einen beliebigen Ordner entpackt werden. Der Emulator ist dann ohne weitere Installationsschritte direkt verwendbar und kann über ein Kommandozeilenprogramm ausgeführt werden.

Download und Kompilieren des Quellcodes

Der Quellcode ist auf den Git-Servern der Universität Leipzig[2] zu finden. Hier kann das Projekt heruntergeladen oder geklont werden. Das Klonen des Repositorys ist mit folgender Anweisung möglich:

```
git clone https://git.informatik.uni-leipzig.de/ti/hwprak/2a-emulator
```

Die Dokumentation[3] der Git-Server erklärt die Arbeit mit Git genauer. Vor allem der Unterpunkt „Getting started with GitLab“ liefert einen schnellen Einstieg.

Zusätzlich ist die Installation der Programmiersprache `Rust`[4] nötig. Hierbei muss es sich um mindestens Version 1.47.0 handeln. Der Installationspfad muss dann gegebenenfalls manuell der Systemvariable `PATH` hinzugefügt werden. Ist `Rust` bereits mit einer niedrigeren Version installiert kann mittels `rustup` auf eine neuere Version geupdated werden.

Um das Projekt zu kompilieren wechselt man in einer Kommandozeile (Terminal) in den entsprechenden Ordner und führt den Befehl `cargo build --release` aus.

[1] `https://git.informatik.uni-leipzig.de/ti/hwprak/2a-emulator/-/releases`
[2] `https://git.informatik.uni-leipzig.de/ti/hwprak/2a-emulator`
[3] `https://git.informatik.uni-leipzig.de/help`
[4] `https://www.rust-lang.org/en-US/` (min. Version 1.47.0)

C.2 Bedienung

Dateiformat

Der 2a-Emulator versteht den selben Assembler, mit dem man auch den realen Minirechner 2a programmiert. Wie diese Assembler-Programme erstellt werden wird im Kapitel A erläutert. Man beachte unbedingt die Kompatibilitätsrichtung: Wenn der im Emulator integrierte Parser eine Datei als syntaktisch korrekt einstuft, dann wird dieses auch auf dem Minirechner 2a ausführbar sein. Dies gilt aber nicht umgekehrt.

Befehle und Bedienung

Der Emulator kann unter Linux mittels `./2a-emulator` in einer Kommandozeile gestartet werden. Unter Windows gelingt dies durch Ausführen der `2a-emulator.exe` durch einen Doppelklick oder in einer Kommandozeile. *Hinweis: Vorher mit der Kommandozeile in den entsprechenden Ordner wechseln. Hat man das Programm selbst kompiliert, so befindet sich die ausführbare Datei im Ordner* `2a-emulator/target/release`.

Eine Programmdatei kann auf zwei Arten ausgeführt werden:

1. Man kann den Pfad beim Starten des Emulators als Argument mit angeben:
   `./2a-emulator pfad/zu/programm.asm` unter Linux, oder
   `2a-emulator.exe pfad\zu\programm.asm` unter Windows.

2. Wenn der Emulator bereits läuft, kann man den internen `load`-Befehl verwenden:
   `load pfad/zu/programm.asm`

Das Programm kann nun schrittweise (Befehl für Befehl) durch Drücken der Enter-Taste ausgeführt werden. (*Hinweis:* Da jeder Assemblerbefehl als Makrobefehl anzusehen ist, benötigt man pro Anweisung mehrere Takte.) Um die Input-Register zu füllen weist man dem Register einfach den entsprechenden Wert zu (z.B. `FD = 10010110`, oder `FE = 0x2a`).

Als zusätzliches Feature können die Assemblerprogramme durch automatisierte Tests auch die Korrektheit geprüft werden. Zusätzliche Hinweise und Tastenkürzel werden im Terminalinterface unter dem Punkt `help` auf der rechten Seite angezeigt. Weitere Informationen sind im Git-Repository des Emulators zu finden.

D Programmtabelle

Auf der nächsten Seite ist ein Muster der Programmtabelle für das Schreiben eigener Programme zu finden.

In der Spalte „Adr.“ steht in hexadezimaler Schreibweise die Adresse im Daten-RAM, an die der Befehl geschrieben werden soll. In der Spalte „Befehl“ sollte eine lesbare Abkürzung (Mnemonic), wie zum Beispiel *LD FF,Reg 0* (kurz für: lade die Konstante FF in Register 0) stehen. Hierbei sollte man Bezug auf die Tabelle 4.5 nehmen und die dort verwendeten Mnemonics beibehalten. Die Spalte „Adresse“ beinhaltet die Adresse in binärer Schreibweise. Die vier letzte Spalten („Byte 1“ bis „Byte 4“) sind für die entsprechenden Befehlsbytes vorgesehen. *Hinweis: Nicht jeder Befehl benötigt vier Bytes (vgl. Tabelle 4.3).*

Zeile	Adr. (hex)	Befehl	Adresse	Byte 1	Byte 2	Byte 3	Byte 4
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							
23							
24							
25							
26							
27							
28							
29							
30							
31							
32							
33							
34							
35							
36							
37							
38							
39							
40							
41							
42							
43							
44							
45							