Institut für Parallele und Verteilte Systeme

Abteilung für Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Bachelorarbeit

Entwicklung einer Programmierhilfe für die Diehl Combitron

Felix Queißner

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer:	Dipl.-Inf. Christian Corti
begonnen am:	15.12.2016
beendet am:	14.06.2017
CR-Klassifikation:	C.1.1, D.3.4

Abstract

Die Diehl Combitron ist ein Tischrechner aus den späten sechziger Jahren, der mit einem turingvollständigen Computer ausgestattet ist. Dieser Computer besitzt eine exotische Architektur, welche ohne Befehlszähler oder Speicheradressiereinheit auskommt. Da dieser Computer nur sehr wenig Dokumentation in Form von zwei Servicehandbüchern besitzt, wird in dieser Arbeit nun eine ausführlichere, zusammenhängende Dokumentation erstellt.

Zudem werden für den Computer Programmierwerkzeuge entwickelt, unter anderem ein Assembler und eine Programmanalyse-Werkzeug. Diese Werkzeuge werden in Entwicklung, Benutzung sowie Funktionsweise dokumentiert.

Inhaltsverzeichnis

Informationen:
Hellblaue Blöcke wie dieser kennzeichnen eine Information, welche das Lesen dieser Arbeit erleichtern soll und auf spezielle Formatierungen bzw. Notationen hinweist.

Wichtige Hinweise:
Hellrote Blöcke wie dieser weisen auf wichtige Erkenntnisse der Arbeit oder Widersprüche zum Wartungshandbuch hin.

Kapitel 1

Einführung

Die Diehl Combitron ist ein im Jahre 1967 produzierter Tischrechner. Neben der vorzeichenkorrekten Ausführung der vier Grundrechenarten beherrscht die Combitron auch Wurzelziehen und besitzt 10 Wertespeicher, in denen Zwischenergebnisse einer Rechnung abgelegt werden können. Zudem erlaubt der Rechner auch das Anlegen von Makros und das Ausführen von gespeicherten Prozeduren, die von einem Lochstreifen eingelesen werden.

Eine Besonderheit des Tischrechners ist, dass die Logik und das Rechenwerk nicht fest verdrahtet wurden, sondern über einen frei programmierbaren Computer implementiert sind. Dadurch wurde es möglich, kostengünstig mehrere Varianten des Rechners herzustellen, unter anderem die *Decitron*, die *Combitron* und die *Combitron S*.

Dieser Computer wurde von Stanley Frankel entworfen, dem Entwickler des LGP-30 [6]. Hierbei hat er die Architektur des LGP-30 noch weiter vereinfacht und alle nicht zwingend notwendigen Komponenten weggelassen. Dies ergab, dass der Computer einige markante Merkmale besitzt, die heute sehr ungewöhnlich sind.

Zu diesen Merkmalen gehört das Fehlen eines Befehlszählers, die Nutzung eines Ultraschall-Laufzeitspeichers sowie eine sehr geringe Anzahl an verwendeten Transistoren.

Diese Architektur wurde von Frankel am 1. Oktober 1965 zum Patent angemeldet [3], ein Jahr, nachdem er die Patentanmeldung für einen Tischrechner einreichte [4]. Dieser Rechner basierte ebenfalls schon auf einem Ultraschall-Laufzeitspeicher, war aber im Gegensatz zur Combitron noch nicht frei programmierbar, sondern nutzt eine fest verbaute Logik.

Nun, nach mehr als 60 Jahren, existiert quasi keine Dokumentation mehr über die Combitron. Die wenigen Dokumente, die verblieben sind, ist das Wartungshandbuch des Computers sowie die beiden Patentschriften über die Architektur des Computers und den Tischrechner auf Laufzeitspeicher-Basis.

In dieser Arbeit soll nun eine Dokumentation entstehen, welche die Hardware in einem Guss dokumentiert, die Patentschriften und das Wartungshandbuch zusammenfasst und mit neuen Erkenntnissen und Informationen ergänzt. Zudem sollen Konzepte zur Programmierung der Maschine entwickelt werden, sowie Werkzeuge, um das Programmieren zu erleichtern.

Kapitel 2

Hardware

Die Combitron besitzt eine einzigartige Architektur, welche nur mit einer sehr geringen Menge an Hardware auskommt:
Für den gesamten Computer ohne Peripherie-Geräte wurden nur 124 Transistoren benötigt. 24 davon wurden zu insgesamt 12 Flip-Flops verschaltet. Die restlichen Transistoren dienen als logische Inverter oder werden im Taktgenerator verwendet.

Dies wird erreicht, indem der Speicher auf einem akustischen Effekt basiert, anstatt, wie moderne Speichermedien, die Daten durch eine elektrische Ladung oder eine Magnetisierung zu speichern. Zudem verarbeitet die Maschine alle Daten bitseriell, wodurch eine Minimalisierung der arithmetisch-logischen Einheit ermöglicht wird.

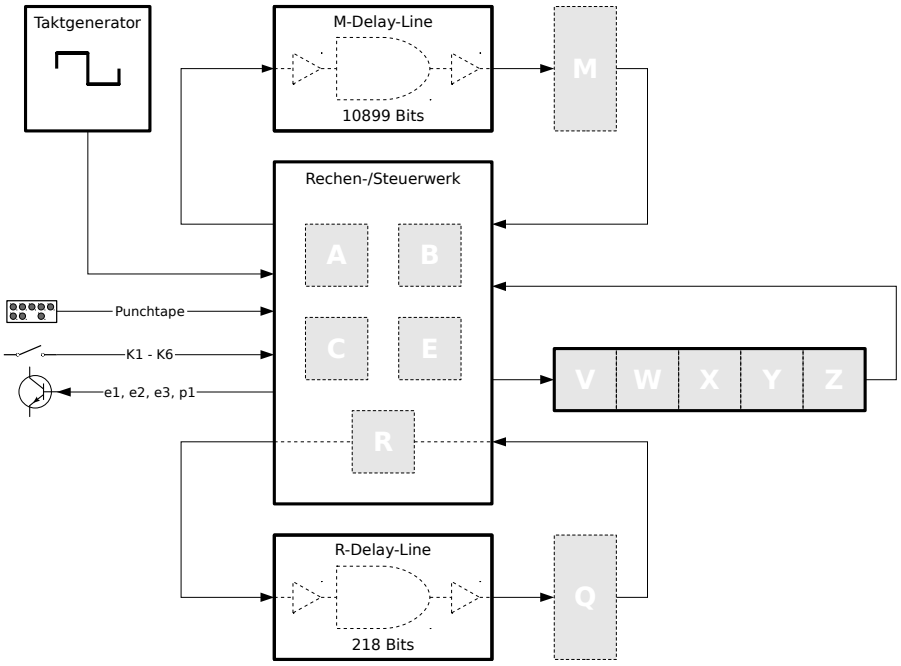


Abbildung 2.1: Blockdiagramm des Computers

Abbildung 2.1 zeigt das Blockschaltbild des Computers. Das *Rechen-/Steuerwerk* ist hierbei die zentrale Komponente. Es ist mit allen anderen Komponenten verbunden und steuert den Datenfluss zwischen diesen. Zudem enthält es die Flip-Flops A, B, C, E und R. Die *M-Delay-Line* sowie die *R-Delay-Line* sind zwei Laufzeitspeicher, die den Haupt- sowie Registerspeicher

bereitstellen. Diese werden durch die Flip-Flops `M` sowie `Q` erweitert. Der Taktgenerator stellt Taktsignale für die Verarbeitung einzelner Bits sowie Worte bereit. *V*, *W*, *X*, *Y* und *Z* bilden zusammen das *VZ-Register*, ein Schieberegister, welches den aktuell ausgeführten Befehl bzw. Ladezustand enthält.

Als Eingabe-Schnittstellen stehen die sechs digitalen Eingangsleitungen *K1* bis *K6* zur Verfügung, sowie ein Lochstreifenleser, welcher zum Laden von Programmcode verwendet wird.

Eine digitale Ausgabe wird über die Leitungen *e2*, *e3* sowie *p1* ermöglicht, die Leitung *e1* hingegen stößt einen Ladevorgang von Band an (siehe Abschnitt 2.6).

2.1 Zeitkette und Timings

Um Daten, die in einem Laufzeitspeicher liegen, zu verarbeiten, wird ein präzises Timing und ein deterministischer Programmfluss benötigt. Hierzu muss sowohl die Dauer als auch der exakte Ausführungszeitpunkt eines Befehls bekannt sein.

Die Combitron organisiert ihren Speicher in Zyklen von jeweils 4 Worten, wobei immer zwei Worte ineinander verschränkt, also interleaved, gespeichert werden. Diese Organisation wird durch den Taktgenerator des Computers ermöglicht. Der Generator erzeugt hierbei zwei relevante Takte: *P* und *I*.

Jeder Flankenwechsel von *P* zeigt den Beginn eines neues Bits an. Hierbei werden alle *HIGH*-Phasen *P* (notiert als P), als Daten-Bits verwendet, alle *LOW*-Phasen von *P* (notiert als $\overline{\text{P}}$) stellen Code-Bits dar. Hierdurch wird eine Trennung zwischen Code und Daten erreicht.

Ein Flankenwechsel von *I* zeigt eine Wortgrenze an, zudem wird das Taktsignal als Zustand des Steuerwerkes verwendet.

Da die Periodendauer des *P*-Taktes $2\mu s$, sowie die des *I*-Taktes $220\mu s$ beträgt, ergibt sich eine Wortgröße von 55 Bit.

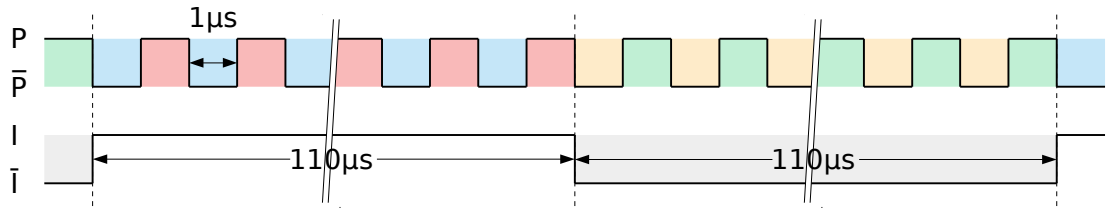


Abbildung 2.2: Taktdiagramm

Zusammen bilden die Taktsignale *I* und *P*, wie in Abb. 2.2 dargestellt, vier voneinander abgetrennte Zeiträume, welche in Abbildung 2.2 farbig markiert sind. Diese vier Zeiträume markieren die in der *R-Delay-Line* gespeicherten Worte. Jedes dieser Worte besitzt eine spezielle Funktion und kommt einem klassischen Register gleich.

Ein vollständiges Taktdiagramm der Maschine kann im Service-Manual auf Seite 143 eingesehen werden.

Um die Arbeit mit der Combitron zu vereinfachen und Missverständnissen vorzubeugen, werden die vier Zeiträume mit einem eindeutigen, leicht zu merkenden Namen versehen:

Zeitraum	Name	Beschreibung
$\overline{\text{I}}$P	`ACC`	Der Akkumulator, auf welchem Rechen- und Speicheroperationen ausgeführt werden.
$\overline{\text{IP}}$	`BAK`	Ein Hilfsregister, dessen Inhalt mit dem Akkumulator getauscht werden kann.
IP	`EXTRA`	Ein Hilfsregister, welches für die Multiplikation bzw. Division benötigt wird.
I$\overline{\text{P}}$	`CCA`	Enthält den aktuell ausgeführten Programmcode (siehe Abschnitt 2.4).

Das Register `ACC` ist der Akkumulator und wird für die meisten arithmetischen Operationen verwendet. Dieses Register kann seinen Inhalt mit dem Backup-Register `BAK` tauschen. Dies erlaubt es, die Anzahl an Speicherzugriffen zu reduzieren. Zudem kann das Wort in `BAK` auch in ein $\overline{\text{P}}$-Wort der M-Delay-Line gespeichert werden (siehe Abschnitt 3.4, Befehl `/r`).

Das Register `EXTRA` speichert ein zusätzliches Wort, welches für Multiplikation und Division sowie für einen langen Shift-Befehl verwendet wird. Dies erlaubt das Arbeiten mit Worten von 110 Bit Länge.

Im Register `CCA` (Control Code Assembly[3]) wird Programmcode abgelegt, welcher nach dem Laden des Registers ausgeführt wird. Dieses Verhalten wird im Abschnitt Maschinensprache (2.4) näher erläutert.

2.2 Speicher

Die Maschine enthält zwei Ultraschall-Laufzeitspeicher. Der erste Speicher, die *M-Delay-Line*, im folgenden M-Speicher genannt, ist der Hauptspeicher und enthält 198 Worte. Der zweite Speicher, die *R-Delay-Line*, im folgenden R-Speicher, wird als CPU-Zustand verwendet und speichert die vier im vorigen Kapitel definierten Register.

Laufzeitspeicher ist eine Speicherarchitektur, ähnlich einem Schieberegister, bei dem nur Zugriff auf das herauszuschiebende Bit existiert. Man kann Daten im Speicher ablegen und erhält nach einer gewissen Zeitspanne wieder Zugriff auf diese Daten. Anschließend „verfallen“ diese. Eine ausführlichere Beschreibung des verwendeten Ultraschall-Laufzeitspeichers findet sich in Anhang J.

Um nun einen persistenten Speicher zu erzeugen, werden die aus dem Speicher gelesenen Bits nach der Verarbeitung durch das Rechenwerk wieder einen der beiden Speicher eingespeist. Hierdurch bildet sich ein Ringpuffer.

Der R-Speicher besitzt einen entscheidenden Unterschied zum M-Speicher:
Von den 4 gespeicherten Worten werden nur 218 Bits im Laufzeitspeicher abgelegt. Das letzte Bit des Speichers wird im R-Flip-Flop abgelegt, das vorletzte Bit im Q-Flip-Flop. Dies erlaubt es, die im R-Speicher abgelegten Worte zwischen der P- und $\overline{\text{P}}$-Phase zu vertauschen. Das wird erreicht, in dem beiden Flip-Flops vom Steuerwerk aus modifiziert werden können und hierbei beim Vertauschen-Befehl die Reihenfolge beim zurückschreiben getauscht wird.

2.3 Rechen- und Steuerwerk

Das Rechenwerk der Combitron besteht aus 70 miteinander verknüpften Logikgleichungen sowie 12 Flip-Flops, welche zusammen das Steuerwerk sowie das komplette Rechenwerk des Computers darstellen.

5 der Flip-Flops bilden das *VZ*-Register, welches den aktuell ausgeführten Befehl enthält. Während der Fetch-Phase wird das *VZ*-Register mit der *R-Delay-Line* zu einem großen Schieberegister zusammengeschaltet und es werden 5 Bit in das Register rotiert. Dies geschieht während der $\overline{\text{IP}}$-Phase für 5 P-Takte.

Ein solches Befehlswort aus 5 Bit wird auch „Silbe“ [2, Seite 127] genannt. Ein Wort der Maschine enthält daher 11 Silben.

Das Flip-Flop *R* ist das 220igste Bit der *R-Delay-Line* und hält das aktuell verarbeitete Bit. Dieses Bit wird zu Beginn eines Rechenschrittes aus der *R-Delay-Line* gelesen und im Normalfall am Ende des Rechenschritts wieder zurück in die *R-Delay-Line* geschrieben.

Das Flip-Flop *B* wird dazu verwendet, eine „Verzögerung“ um ein Bit einzuführen. Hierbei wird das Flip-Flop in Reihe mit der R-Delay-Line geschaltet und somit der Speicher auf 221 Bit „verlängert“. Dieses Flip-Flop wird in den Shift-Befehlen verwendet.

Das Flip-Flop *C* ist das Carry-Bit bei arithmetischen Operationen und das Eingabe-Bit, welches bei den Tastenprüfbefehlen gesetzt wird. Zudem wird das Flip-Flop dafür verwendet, bedingte Programmverzweigungen zu steuern.

Das Flip-Flop *E* wird zur Kontrolle der einzelnen Steuerwerk-Phasen verwendet und zeigt an, ob gerade ein Befehl ausgeführt wird oder ob das *VZ*-Register neu geladen wird.

Das Flip-Flop *A* wird dafür verwendet, die arithmetischen Operationen zu steuern und auszuführen.

Das Flip-Flop *Q* enthält das vorletzte Bit des R-Speichers.

Das Flip-Flop *M* enthält das vorletzte Bit des M-Speichers.

2.4 Maschinensprache

Die Befehle der Maschine werden in drei Gruppen eingeteilt: Kurzbefehle, Doppelbefehle und Langbefehle. Kurz- und Doppelbefehle benötigten zur Kodierung eine Silbe, Langbefehle benötigen eine Silbe für den Befehl sowie eine zweite Silbe für einen Zählwert, der die Ausführungsdauer bestimmt.

Diese Befehle werden in einem Wort zu einer Control-Code-Assembly (CCA) zusammengesetzt, einer Art Unterprogramm. Diese Unterprogramme sind immer genau ein Wort groß und können von daher nur an einem Stück in das Register `CCA` geladen werden. Dies kommt einem Sprung im Codefluss gleich. Hierbei besteht die Einschränkung, dass nur die höchstwertigste Silbe, also der logisch erste Befehl angesprungen werden kann. Ein Sprung zu einer anderen Stelle im Unterprogramm ist nicht möglich.

Die Maschine benötigt keinen Befehlszähler, da die Auswahl des als nächstes auszuführenden Befehls über eine Schiebeoperation geschieht:
Wie oben beschrieben, wird während der I-Phase der R-Speicher für fünf volle P-Takte mit dem *VZ*-Register zusammengeschaltet. Anschließend wird eine Schiebeoperation ausgeführt, welche 5 Bit durch das sich nun ergebende ringförmige Schieberegister schiebt.

Da die Schiebeoperation nur in die höherwertige Richtung ausgeführt werden kann, wird die höchstwertigste Silbe der CCA in das *VWXYZ*-Register geschoben, der Inhalt des Registers wird wieder zurück in die `CCA` geschoben. Hierdurch ergibt sich, dass die Befehle von der höchstwertigsten zur niederwertigsten Silbe ausgeführt werden, also von „hinten“ nach „vorne“.

Die drei Gruppen an Befehlen haben jeweils eine unterschiedliche Ausführungszeit. Kurzbefehle benötigen einen R-Zyklus zur Ausführung, Doppelbefehle zwei R-Zyklen. Die Ausführungszeit von Langbefehlen wird über die nachfolgende Argument-Silbe bestimmt:

$$t(\text{cmd}, \text{arg}) = 66 - 32 \cdot (\text{cmd} \bmod 2) - \text{arg} \tag{2.1}$$

Hierbei ist *cmd* der Wert der Binärkodierung des aktuell ausgeführten Befehls, *arg* der Wert der Argumentsilbe. Der Wert der Funktion t gibt die Anzahl der R-Zyklen an, die der Langbefehl zur Ausführung benötigt.

Anders ausgedrückt sind die Opcodes der Langbefehle nur vier Bit breit, das niederwertigste Bit des Opcodes ist das sechste und damit höchstwertigste Bit des Zählwertes. Dieser Zählwert ist im Dualkomplement kodiert und beschreibt die Ausführungsdauer des Befehls.

Der Zählwert gibt aber nur die Anzahl an Rechenschritten an, die der Befehl ausführt. Zusätzlich zu diesen Rechenschritten werden zwei weitere R-Umläufe benötigt, um die Befehlsausführung anzustoßen. Dies führt dazu, dass die Ausführungszeit Zählwert + 2 beträgt.

2.5 Input / Output

Der Combitron stehen nur sehr stark eingeschränkte I/O-Möglichkeiten zur Verfügung, trotzdem wurden einige Befehle für die Ein- bzw. Ausgabegabe von Daten bereitgestellt.

Um in einem Programm Daten auszugeben, stehen die Befehle e2, e3 sowie p1 zu Verfügung. Diese Befehle erzeugen für die Dauer der Befehlsausführung einen Impuls auf der ihnen zugeordneten Steuerleitung. Dieser Impuls ist $110\mu s$ lang.

Die Eingabe von Daten erfolgt über die 6 Befehle T1 - T6, welche bei Ausführung den Zustand der korrespondierenden Leitungen K1 - K6 mit dem Flip-Flop C verodern.

Die Grundlagen der Ansteuerung des Druckers sowie das Auslesen der Tastatur werden im Anhang Druckersteuerung bzw. Tastaturmatrix näher erläutert.

2.6 Bootstrap

Da der Ultraschall-Laufzeitspeicher ein flüchtiger Speicher ist, muss die Maschine mit einem speziellen Ladevorgang gestartet werden. Dieser Startvorgang füllt den R-Speicher mit Daten, die von einem Metall-Lochstreifen eingelesen werden. Anschließend führt die Maschine die in den R-Speicher geladene `CCA` aus.

Durch diese Ausführung können der Inhalt der 3 Datenregister `ACC`, `BAK` und `EXTRA` im M-Speicher abgelegt werden. Anschließend können durch den Befehl e1 weiterer Daten vom Lochstreifen geladen oder aber nach Abschluss des Ladevorgangs das Programm ausgeführt werden. Dies geschieht durch die passende Ausführung eines Füll-Befehls, der den Einstiegspunkt des Programmes anspringt.

Auf dem Lochstreifen wird die Bootsequenz in zwei Spuren abgelegt. Die eine Spur enthält den Takt des Signals, die andere Spur die einzelnen Datenbits. Die Taktspur enthält zu Beginn einen Vortakt, welcher die Initialisierung der Maschine anstößt, und ist regelmäßig unterbrochen, um ausreichend Pausen für die Ausführung der geladenen CPU-Zustände zu ermöglichen und einem Ladefehler vorzubeugen. Hierbei ist eine Mindestpause von der Ausführungszeit des vorher ausgeführten Codes sowie einem gesamten M-Speicher-Umlauf (also ca. 11 ms) nötig. In der Praxis hat sich eine Wartezeit von ca. 50 ms als praktisch erwiesen.

Zudem beträgt die Mindest-Taktrate ca. 250 μs, da für jedes übertragene Bit ein R-Speicher-Umlauf sowie einige Zustandsänderungen der Maschine erfolgen. [2, Seite 178ff]

In Abbildung 2.3 wird eine mögliche Datenübertragung dargestellt. $t + g$ stellt hierbei das Taktsignal dar, t_- das Datensignal.

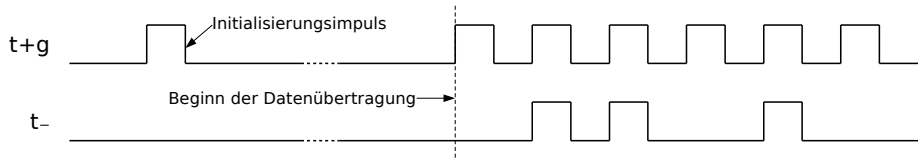


Abbildung 2.3: Zeitverlauf der Signale vom Lochstreifen

Die einzelnen CPU-Zustände, die auf dem Lochstreifen kodiert sind, werden während der Ladephase vom hochwertigsten zum niederwertigsten Bit geladen. Hierbei ist die Reihenfolge genau umgekehrt der Verarbeitungsreihenfolge in der CPU selbst.

Das erste Bit einer Datenübertragung ist `0`, anschließend werden 219 Bit übertragen. Hierbei werden zuerst die beiden I-Worte übertragen, anschließend die beiden $\overline{\text{I}}$-Worte, wobei das niederwertigste Bit des $\overline{\text{IP}}$-Wort verworfen wird. Während der Übertragung eines Bit-Paares wird zuerst das P-Bit übertragen, anschließend das $\overline{\text{P}}$-Bit.

Dieses Verhalten ergibt sich aus der Notwendigkeit eines Marker-Bits im R-Speicher, welches den „Beginn“ bzw. das „Ende“ des Speichers anzeigt. Sobald dieses Bit aus dem Speicher herausgeschoben wird, endet die Ladephase. Hierbei verfällt dieses Bit und wird als 0 in den Speicher geschrieben.

Vorsicht: Durch den Verfall des letzten Bits kann der letzte Befehl einer Ladesequenz nicht beliebig sein. Auch der Befehl zum Nachladen einer weiteren Bandsequenz fällt unter diese Restriktion. Daher wird es empfohlen, in Ladesequenzen maximal 10 Befehle bzw. Silben zu verwenden.

2.7 Initialisierungsphase

Nach dem Zurücksetzen der Maschine über den *m6*-Impuls befindet sich die CPU in der Initialisierungsphase und wartet auf den Initialisierungsimpuls. Dieser wird durch einen Taktimpuls des Lochstreifens erzeugt.

Der *m6*-Impuls wird nach dem Einschalten der Maschine erzeugt, kann aber auch während der Ausführung durch ein externes Gerät ausgelöst werden.

Nachdem der Initialisierungsimpuls empfangen wurde, beginnt die Maschine 220 gesetzte Bit in den M-Speicher zu schreiben, um hiermit den sogenannten M-Marker zu erzeugen. Dieser

markiert den Beginn bzw. das Ende des Speichers und erlaubt hiermit eine Bestimmung der Startzeit der Programmausführung.

Anschließend werden 220 Bit vom Lochstreifen in den R-Speicher geschoben, wie in Kapitel 2.6 beschrieben.

Um nun die Ausführung zu Beginn des Speichers zu gewährleisten, sucht die Maschine nun nach einem I-$\bar{\text{I}}$-Phasen-Paar, in dem alle Bits gesetzt sind. Dies ist bei normaler Nutzung der Maschine gewährleistet, da ein Code-Wort mit ausschließlich gesetzten Bits keinen sinnvollen Code darstellt.

Wenn eine solche Sequenz an Bits gefunden wurde, wird zu Ende dieser Sequenz mit der Ausführung begonnen.

Der M-Marker belegt die letzten vier Worte des Speichers und darf nur überschrieben werden, wenn keine weiteren Nachladebefehle ausgeführt werden müssen, da diese den Marker als Einstiegspunkt benötigen. Zudem dürfen keine weiteren Wort-Quadrupel existieren, in denen alle Bits gesetzt sind.

Kapitel 3

Programmierung der Combitron

In diesem Kapitel werden Programmierkonzepte sowie die einzelnen Befehle der Maschine (siehe auch Anhang A Befehlssatz) vorgestellt. Zudem werden Hilfsmittel und Konventionen eingeführt, die ein einfacheres Arbeiten mit der Combitron ermöglichen.

3.1 Program Timer

Der Program Timer (`PT`) ist ein gedachtes Hilfskonstrukt, welches das Programmieren mit der Combitron erleichtern soll.

Er ist kein physisch existentes Register, sondern muss vom Programmierer oder einem Programmier-Werkzeug beim Schreiben eines Programmes mitberechnet werden. Dies ist nötig, um Speicherzugriffe einfach zu gestalten.

Er mimt ein selbstinkrementierendes Register, welches bei jedem R-Umlauf um 2 erhöht wird. Der maximale Wert, den der Program Timer annehmen kann, ist 99. Danach läuft er über und springt zurück auf die 1. Da der `PT` nach dem ersten Überlauf auf 2 steht, nach dem zweiten Überlauf aber wieder auf 1, wird nach jedem Umlauf der Zugriff auf die Worte der I- bzw. $\overline{\text{I}}$-Phase getauscht. Hiermit indexiert der `PT` alle 198 Worte (99 Wortpaare bestehend aus einem P-Wort sowie einem $\overline{\text{P}}$-Wort) des M-Speichers und erlaubt damit auch jeden möglichen Zugriff auf jedes Wort im Speicher. Eine Formel, welche die Inkrementierung des `PT` beschreibt, ist in Gleichung 3.1 beschrieben.

$$\text{PT}^{t+1} = ((\text{PT}^t + 1) \bmod 99) + 1 \tag{3.1}$$

Zu Beginn der Programmausführung, nach der Initialisierungsphase bzw. einer Ladephase über `e1`, wird der Timer auf 1 zurückgesetzt.

3.2 Speicherzugriff

Der Speicherzugriff der Combitron kann immer nur auf das Wort erfolgen, welches gerade den Laufzeitspeicher verlässt. Hierdurch benötigt die Maschine keine Adressierungseinheit, sondern verwendet einen Wartebefehl, um auf ein bestimmtes Daten- oder Codewort zu warten.

Die einzelnen Befehle können dadurch immer nur auf ein Wort zugreifen, welches in einem bestimmten, zeitlichen Offset zu ihnen liegt. Dieses Offset wird immer relativ zum `PT` angegeben.

Ein Füllbefehl greift beispielsweise immer auf das zweite Wort in $\overline{\text{P}}$ nach Beginn der Ausführung zu (PT + 2) und lädt dieses. Hierdurch wird garantiert, dass die Ausführung eines Code-Wortes aus dem M-Speicher immer zum selben Zeitpunkt in `PT` geschieht. Dies erleichtert die Berechnung der Adresszeiten, da diese immer vom Beginn des ausgeführten Code-Wortes abhängen.

3.3 Adressnotation

Um während der Programmierung eine Unterscheidung zwischen P-Adressen und $\overline{\text{P}}$-Adressen im M-Speicher vorzunehmen, wird folgende Adressnotation eingeführt:

```
1 <Zahl>P
2 <Zahl>N
```

Hierbei ist *Zahl* ein Wert zwischen 1 und 99. Dieser gibt den `PT` -Wert an, zu welchem das Wort zugreifbar ist. Das `P` zeigt eine P-Adresse an, das `N` eine $\overline{\text{P}}$-Adresse.

Da einige Befehle der Combitron in unterschiedlichen zeitlichen Abschnitten lesend oder schreibend auf den M-Speicher zugreifen, zudem auch einige Flip-Flops modifizieren, wird dieses über folgende Tabelle beschrieben:

A	B	+0				+1				+2				+3			
C	E	r$\overline{\text{P}}$	w$\overline{\text{P}}$	rP	wP	r$\overline{\text{P}}$	w$\overline{\text{P}}$	rP	wP	r$\overline{\text{P}}$	w$\overline{\text{P}}$	rP	wP	r$\overline{\text{P}}$	w$\overline{\text{P}}$	rP	wP

Die vier Felder links beschreiben die vier relevanten Flip-Flops, welche bei der Befehlsausführung modifiziert werden. `A` wird in jedem Fall modifiziert, daher ist es nicht angegeben. Die vier Spalten rechts davon geben die vier relevanten Zeitabschnitte während der Befehlsausführung an. Der Spaltenkopf gibt das zeitliche Offset zum `PT` an, die vier Felder darunter, in welche Phase des M-Speichers ($\overline{\text{P}}$ oder P) geschrieben (w) oder gelesen (r) wird.
Wenn ein Feld in den folgenden Tabellen eingefärbt ist, gilt dieses als benutzt.
Für Kurzbefehle ist die rechte Hälfte der Tabelle leer, da diese nicht verwendet wird und somit eine schnelle visuelle Orientierung bietet.

3.4 Kurzbefehle

Kurzbefehle benötigen zur Ausführung einen R-Umlauf Zeit, schreiten also 2 Worte im M-Speicher voran.

`NUL` (Keine Operation)

A	B	+0				+1				+2	+3
C	E	r$\overline{\text{P}}$	w$\overline{\text{P}}$	rP	wP	r$\overline{\text{P}}$	w$\overline{\text{P}}$	rP	wP		

Der `NUL`-Befehl entspricht einer NOP und verändert den Zustand der Maschine nicht.

T1, T2, T3, T4, T5, T6 (Prüfe K-Leitung)

A	B	+0				+1				+2	+3
C	E	rP̄	wP̄	rP	wP	rP̄	wP̄	rP	wP		

Jeder dieser Befehle prüft die ihm zugeordnete Leitung *K1* bis *K6*. Wenn die Leitung aktiv ist, wird das Carry-Bit gesetzt, ansonsten bleibt es unverändert.

e1 (Lade von Band)

A	B	+0				+1				+2	+3
C	E	rP̄	wP̄	rP	wP	rP̄	wP̄	rP	wP		

Dieser Befehl stößt die Ladephase, wie in Abschnitt 2.6 beschrieben, aus dem Programm an. Dies wird benötigt, um ein komplettes Programm in den Speicher zu laden, da sonst nur einmalig der R-Speicher vom Lochstreifen geladen werden würde.

e2, e3, p1 (Ausgabe an Druckersteuerung)

A	B	+0				+1				+2	+3
C	E	rP̄	wP̄	rP	wP	rP̄	wP̄	rP	wP		

Diese Befehle sind als Ausgabebefehle gedacht und geben einen $110\mu s$ langen Impuls auf der jeweils korrespondierende Leitung aus.

+, - (Addieren, Subtrahieren)

A	B	+0				+1				+2	+3
C	E	rP̄	wP̄	rP	wP	rP̄	wP̄	rP	wP		

Dies sind die arithmetischen Befehle der Combitron. `+` addiert den aus dem Speicher gelesenen Wert auf den Akkumulator, `-` zieht den gelesenen Wert vom Akkumulator ab. Zusätzlich dazu wird noch der Wert des Carry-Flip-Flops auf den Akkumulator addiert bzw. davon abgezogen. Der Übertrag der Addition/Subtraktion ist anschließend im Carry-Flip-Flop abgelegt.

b (Bringe P)

A	B	+0				+1				+2	+3
C	E	rP̄	wP̄	rP	wP	rP̄	wP̄	rP	wP		

Hiermit wird ein Wort aus der Datenphase des M-Speichers in den Akkumulator `ACC` gelesen.

r (Abspeichern P)

A	B	+0				+1				+2	+3
C	E	rP̄	wP̄	rP	wP	rP̄	wP̄	rP	wP		

Mit diesem Befehl wird der Akkumulator `ACC` in den M-Speicher geschrieben.

/r (Abspeichern $\overline{P}$)

A	B	+0				+1				+2	+3
C	E	r$\overline{P}$	w$\overline{P}$	rP	wP	r$\overline{P}$	w$\overline{P}$	rP	wP		

Dieser Befehl erlaubt die Modifikation des Codes der Maschine. Er schreibt das Backup-Register `BAK` in den M-Speicher.

c (Lösche $\overline{IP}$ in R-Delay-Line)

A	B	+0				+1				+2	+3
C	E	r$\overline{P}$	w$\overline{P}$	rP	wP	r$\overline{P}$	w$\overline{P}$	rP	wP		

`c` setzt den Inhalt des Akkumulators `ACC` auf 0.

x (Vertauschen)

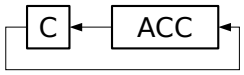
A	B	+0				+1				+2	+3
C	E	r$\overline{P}$	w$\overline{P}$	rP	wP	r$\overline{P}$	w$\overline{P}$	rP	wP		

Dieser Befehl tauscht den Inhalt des Akkumulators `ACC` mit dem Backup-Register `BAK`.

s (Schieben)

A	B	+0				+1				+2	+3
C	E	r$\overline{P}$	w$\overline{P}$	rP	wP	r$\overline{P}$	w$\overline{P}$	rP	wP		

Schiebt das Register `ACC` um eine Stelle nach links. Hierbei wird der Inhalt des Carry-Flip-Flops in die niederwertigste Stelle geschoben. Der Wert des herausgeschobenen Bits wird anschließend in das Carry-Flip-Flop übertragen.



d (Herunterzählen)

A	B	+0				+1				+2	+3
C	E	r$\overline{P}$	w$\overline{P}$	rP	wP	r$\overline{P}$	w$\overline{P}$	rP	wP		

Dekrementiert das aktuelle Datenwort aus dem M-Speicher und schreibt es anschließend an die selbe Stelle zurück. Hierbei wird Carry gesetzt, falls **kein** Underflow aufgetreten ist. Hiermit kann ein Zähler realisiert werden, ohne dass es nötig ist, ein Wort in den Akkumulator zu laden, zu dekrementieren und anschließend wieder an die selbe Stelle abzuspeichern.

Das Dekrementieren kann verhindert werden, in dem zu Beginn der Befehlsausführung Carry gesetzt ist.

3.5 Doppelbefehle

Doppelbefehle benötigen zur Ausführung doppelte so viel Zeit wie ein Kurzbefehl, also 4 Wortzeiten. Dies erlaubt es, auf mehr als ein Datenwort gleichzeitig oder aber auf ein Wort in der I-Phase des R-Speichers zuzugreifen.

`F` (Unbedingter Füllbefehl)

A	B	+0				+1				+2				+3			
C	E	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP

Lädt ein Wort aus dem Speicher in die `CCA`. Dies entspricht ungefähr einer Sprung-Operation, da hiermit der Codefluss gesteuert werden kann.

`C` (Bedingter Füllbefehl)

A	B	+0				+1				+2				+3			
C	E	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP

Verhält sich analog zu `F`, wobei das Laden nur ausgeführt wird, wenn das Carry-Flip-Flop gesetzt ist. Dies entspricht einem bedingten Sprung. Anschließend wird das Carry-Flip-Flop zurückgesetzt.

`B` (Bringe P- doppelt)

A	B	+0				+1				+2				+3			
C	E	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP

Lädt zwei konsekutive Worte. Das erste Wort wird nach `ACC` geladen, das zweite Wort nach `EXTRA`.

`R` (Abspeichern P- doppelt)

A	B	+0				+1				+2				+3			
C	E	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP	$rP\overline{}$	$wP\overline{}$	rP	wP

Speichert zwei konsekutive Wörter ab. `ACC` wird in das erste Wort gespeichert, `EXTRA` in das zweite Wort.

3.6 Langbefehle

Langbefehle benötigten eine vom Programmierer festgelegte Zeit an R-Zyklen. Diese wird durch die nachfolgende Befehlssilbe, das Befehls-Argument, sowie das niederwertigste Bit der Befehlssilbe bestimmt.

Die Berechnung der Ausführungszeit wird im Kapitel Maschinensprache beschrieben.

W0, W1 (Warten)

A	B	+0				+1				+2	+3
C	E	rP̄	wP̄	rP	wP	rP̄	wP̄	rP	wP		

Dieser Befehl ist der wichtigste Befehl der Combitron, da hiermit effektiv Zeit „verbraucht“ werden kann. Er verhält sich wie `NUL` (Keine Operation), nur dass die Ausführungszeit variabel und auch wesentlich höher ist.

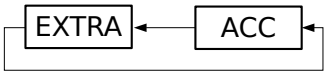
Hiermit kann auf die gewünschten Daten im Speicher gewartet werden, da mit zwei Wartebefehlen schon auf mehr als einen gesamten M-Speicher-Umlauf gewartet werden kann.

S0, S1 (Schieben)

A	B	+0				+1				+2	+3
C	E	rP̄	wP̄	rP	wP	rP̄	wP̄	rP	wP		

Der lange Shift-Befehl schiebt den Inhalt der Register `ACC` sowie `EXTRA` um den angegeben Zählwert nach links. Die herausgeschobenen Bits werden in das jeweils andere Register geschoben. Dies kann auch als eine Rotation eines großen Registers gesehen werden, welches durch die Verknüpfung von `ACC` mit `EXTRA` gebildet wird.

Für diese Schiebe-Operation wird das Flip-Flip C nicht verwendet, das heißt, es wird eine reine Rotation, durchgeführt, das Carry-Bit wird nicht modifiziert.



M0, M1 (Multiplizieren)

A	B	+0				+1				+2	+3
C	E	rP̄	wP̄	rP	wP	rP̄	wP̄	rP	wP		

Während der Erstellung dieser Arbeit konnte die Funktionsweise der Multiplikations- sowie Divisionsbefehle nicht verifiziert werden. In der Theorie multiplizieren die beiden Befehle `M0` sowie `M1` den Wert im Akkumulator mit dem aus dem M-Speicher gelesenen Wert. Welche Operationen aber tatsächlich ausgeführt wird, wurde nicht festgestellt.
Abhängig vom Zählwert verhielt sich die Multiplikation wie ein Schiebebefehl oder setzte aber den Inhalt des Akkumulators auf null.

D0, D1 (Dividieren)

A	B	+0				+1				+2	+3
C	E	rP̄	wP̄	rP	wP	rP̄	wP̄	rP	wP		

Analog zu den Multiplikations-Befehlen wurde die Funktionsweise dieser beiden Befehle nicht festgestellt.

Kapitel 4

Erkenntnisse und deren Umsetzung

Dieses Kapitel berichtet über die während der Entwicklungszeit gewonnene Erkenntnisse bezüglich der Entwicklung von Software für die Combitron.

4.1 Hardwarenaher Assembler

Die erste, und relativ offensichtliche Erkenntnis war, dass man die Maschine nicht in Maschinensprache bzw. einer reinen Zahlendarstellung programmieren möchte.

Hierbei wurden zuerst einige syntaktische Prototypen erstellt und nach ihrer Lesbarkeit bewertet.

Prototyp 1

```
1  .nphase
2
3  .org 27
4  ; Increment ACC by two
5  CCA {
6          add
7          wait1
8          4
9          F
10 }
11
12 .org 58
13 ; Decrement ACC by one
14 CCA {
15         sub
16         wait0
17         3
18         F
19 }
20
21 .pphase
22
```

```
; Two for incrementing
.org 30
.data 2

; One for decrementing
.org 61
.data 1
```

Der erste Prototyp war sehr stark an einem klassischen Assembler orientiert. Hierbei gab es die typischen Direktiven wie `.org` sowie zwei spezielle Direktiven `.pphase` sowie `.nphase`. Zudem gab es einfache Makros für die Langbefehle, welche ein Argument in Klammern annehmen konnten und damit in einen passenden Befehl übersetzt wurden.

Dabei ergaben sich einige Probleme:

Die Programmausführung der Combitron folgt nicht linear den einzelnen Speicheradressen. Der Assembler vermittelt aber das Gefühl einer linearen Programmierung.

Zudem war durch die Trennung der beiden Phasen P sowie $\overline{\text{P}}$ in eine blockorientierte Form sowie das automatische Inkrementieren von Wort-Adressen sehr unübersichtlich und erschwerte das Nachvollziehen der Position eines Worts im Speicher.

Prototyp 2

```
Addr |  P-Phase | /P-Phase
-----|----------|-------------------
  27 |        0 | [ + W1 4 F]
  30 |        2 | 0
  58 |        0 | [ - W0 3 F]
  61 |        1 | 0
```

Der zweite Prototyp griff die Probleme des ersten Prototypen auf und führte eine direkte Adresszuweisung ein. Die erste Spalte der Tabelle enthielt die konkrete Adresse des M-Speichers, die zweite und dritte Spalte entweder ein Zahlenwert oder eine CCA.

Hiermit konnte das Programm sehr viel übersichtlicher gestaltet werden, was das Nachvollziehen von Speicherpositionen und Phasen anging, die Syntax des Assemblers war aber sehr kompliziert und auch oft schwer zu lesen.

Prototyp 3

```
# Increment by 2
27N: [ + W1 4 F]
30P: 2

# Decrement by 1
58N: [ - W0 3 F ]
61P: 1
```

Dieser Prototyp griff die Ideen des zweiten Prototypens auf, änderte aber die Zuweisung an eine Adresse. Hierdurch entstand eine Syntax, welche direkte Wertzuweisungen ermöglicht, die Phase korrekt ausdrückt und gleichzeitig eine hohe Lesbarkeit bietet.

Aus diesem Prototypen wurde anschließend der Direkt-Assembler(siehe Abschnitt 5) entwickelt, welcher nun in der Toolchain (siehe Abschnitt 7) implementiert ist.

4.2 Analyse-Werkzeuge

Schon während der Entwicklung des zweiten Assembler-Prototypen wurde klar, dass das Programmieren der Combitron ohne entsprechende Hilfswerkzeuge sehr mühsam ist, da für jeden Befehl die genaue Wortzeit bekannt sein muss.

Daraus folgte die Konzeption und Entwicklung von verschiedenen Analysewerkzeugen, welche am Ende zu einem einzigen Analysewerkzeug zusammengefasst wurden.

Lokale Zeitkette

Beim Schreiben einer `CCA` ist es notwendig, für jeden Befehl die exakte Ausführungszeit zu kennen. Da dies von Hand eine mühsame Arbeit ist, wurde ein Programm entwickelt, welches die Ausführungszeit von Befehlen bestimmt.

Dieses Programm gab eine Tabelle aus, welche die Liste der ausgeführten Befehle enthielt sowie eine Annotation für jeden Befehl, zu welcher Wortzeiten dieser ausgeführt wird.

Daraus ließen sich dann einfach die passenden Zugriffszeiten ableiten.

Codeflow-Analyse

Bei einem experimentellen Versuch, die original Firmware der Combitron zu analysieren und zu verstehen, fiel auf, dass für das Verständnis des Programms der Codefluss zwischen den einzelnen Befehlsworten sehr wichtig war.

Um den Befehlsfluss zu analysieren, wurde ein weiteres Programm, unabhängig der Zeitketten-Analyse, entwickelt. Dieses Programm hatte als Eingabe einen Speicherdump der Firmware und berechnete daraus einen Graphen mit allen Verzweigungen und Sprüngen.

Die Ausgabe des Programms erfolgt im Format für GraphViz, mit welchem anschließend der generierte Graph dargestellt werden konnte.

Durch die graphische Darstellung konnten einige Ungereimtheiten in der Ausgabe der Zeitketten-Analyse festgestellt werden, welche nachträglich korrigiert wurden.

Einer der hier festgestellten Fehler war, dass die Zeit, welche für das Laden einer `CCA` in den R-Speicher benötigt wird, nicht mit einbezogen wurde und damit alle absoluten Zeitwerte um 2 Worte daneben lagen.

Später wurde die Analyse noch mit einer Sprungvorhersage ausgestattet, welche die Pfeile des Graphen unterschiedlich eingefärbt darstellte. Die Sprungvorhersage basiert auf dem Wissen, dass einige Befehle das Carry-Flip-Flip modifizieren und damit garantiert setzen, löschen oder auch in einem unvorhergesehenen Zustand hinterlassen.

4.3 Toolchain

Während der Entwicklung der bisher beschriebenen Werkzeuge stellte sich irgendwann die Frage, wie man die Artefakte der Werkzeuge zwischen den Werkzeugen austauschen könnte.

Die erste Idee hierfür war, eine programmierbare Toolchain zu erstellen. Diese sollte alle Werkzeuge enthalten sowie den Austausch von Artefakten zwischen den Werkzeugen über Objekte zu regeln. Diese Objekte werden zu Laufzeit erstellt und dann in die einzelnen Werkzeuge gegeben.

Zu Beginn wurde für jedes neue Experimente die Hauptfunktion der Toolchain modifiziert und das ganze System neu compiliert. Dies wurde aber aufgrund von immer komplexer werdenden Anforderungen, sowie der Existenz von mehreren parallelen Experimenten immer unhandlicher.

Als Lösung hierfür wurde dann eine Art Makefile erstellt, welches die einzelnen Schritte der Toolchain steuern kann und auch mit späteren Versionen noch ausführbar ist.

Hierbei wurde eine einfache, nicht zeilenbasierte Syntax verwendet, da die einzelnen Aufrufe der Toolchain-Komponenten öfter lange Zeilen produzieren und ein Zeilenumbruch innerhalb eines Befehls erlaubt werden sollte.

Da der Begriff „Makefile“ schon stark durch das Werkzeug `make` der GNU Buildtools geprägt ist, wurde sich später dafür entschieden, die Datei „Recipe“ zu nennen.

4.4 Intuitiver Assembler

Um das Programmieren der Combitron auch für einen weniger erfahrenen Programmierer zu erlauben, oder aber auch, um das Programmieren zu vereinfachen, kam der Gedanke an einen „linearen“ Assembler auf, welcher sich wie ein gewohnter Assembler verhält.

Dieser sollte die internen Vorgänge wie Speicheradressierung durch Warten oder aber auch die sich ergebenden Programmlayouts vor dem Programmierer verstecken.

Die Anforderungen an den Assembler waren folgende:

- Erlaube eine intuitive Programmierung
- Sehe aus wie ein typischer Assembler
- Ermögliche direkte Adressierung des Speichers
- Verwende Labels für Sprünge
- Verstecke die komplexen Layouts des Speichers

Um diesen Anforderungen gerecht zu werden, wurde zuerst wieder ein syntaktischer Prototyp des Assemblers erstellt:

```
1 ;
2 ; Bit-Catcher 1.0
3 ;
4 ; 02-03-2017,FQ: Erste Version
5 ;
6
7 .equ address 10
8 .equ addr 12
```

```
9
10 label: ; p-Phase
11     shl       ; s
12     shl 10    ; S0, S1
13     mul 20    ; M0, M1
14     div 30    ; D0, D1
15     test 3    ; T1,…,T6
16     load      ; e1
17     pulse p1
18     dec       ; d, ACC--
19     clear     ; c, ACC=0
20     add addr ; +, ACC += *addr
21     sub addr ; -, ACC -= *addr
22     ld  addr  ; b, Load ACC
23     st  addr  ; r, Store ACC
24     swp       ; x, Swap ACC/BAL
25     stc label ; /r, Store BAK to code, but how?
26
27     jmp label ; F, Jump to
28     jpc label ; C, Jump when carry
29
30     ld2 addr  ; B, Load ACC/EXTRA
31     st2 addr  ; R, Store ACC/EXTRA
```

Hierbei wurde das Augenmerk besonders auf die Lesbarkeit des Codes gelegt. Dies führte zu längeren Mnemonics sowie der Zusammenfassung von mehreren Befehlen in ein einzelnes Mnemonic.

Nach der Definition der Syntax wurde eine Liste der notwendigen Funktionen des Assemblers erstellt:

- Automatischer Sprung von einem Label zum nächsten Label
- Automatisches Aufteilen des Codes in mehrere `CCAs`
- Automatisches Einfügen von Wartebefehlen vor Speicherzugriffen
- Automatische Adresszuweisung für Labels

Nach der Implementierung des Parsers für den Assembler-Code wurde die Codegeneration für die Speicherzugriffe umgesetzt. Hierbei wurden noch feste, vom Programmierer vergebene Adressen für die einzelnen Labels verwendet.

Anschließend wurden das Aufteilen von Code in `CCAs` sowie der automatische Sprung implementiert. Hierbei wurde langsam klar, dass der letzte Punkt der Liste, die automatische Adresszuweisung, ein nicht-triviales Problem darstellt, welches anscheinend NP-hart ist.

Um trotzdem eine effiziente Lösung zu erhalten, wurde ein randomisierter Algorithmus implementiert, der die verwendeten Speicherstellen während der Übersetzung zufällig aus einem Pool an Adressen bezieht.

Hiermit wird nach einigen hundert Versuchen eine relativ gute Strategie gefunden, auch wenn diese nicht optimal ist. Die genaue Implementierung dieses Algorithmus sowie des Assemblers an sich ist in Kapitel 8.2 näher erläutert.

4.5 Ladewerkzeug

Parallel zu der Entwicklung der oben genannten Werkzeuge wurde an einem Gerät gearbeitet, welches einem erlaubt, die Combitron auch von einer beschreibbaren Programmquelle zu starten.

Dieses Gerät muss in der Lage sein, den in der Combitron verbauten Lochstreifen zu ersetzen und dessen Verhalten nachzuahmen.

Hierfür wurde ein LAB8/E von Digital Equipment Corporation als Hardware-Interface verwendet, da dieser bereits im Computermuseum zur Verfügung steht und eine ausreichend große Anzahl an I/O-Pins bietet.

Das Programm, welches auf dem LAB8/E läuft, empfängt Daten in einem einfachen ASCII-Protokoll (siehe Abschnitt 8.4), mit welchem die Daten eines virtuellen Lochbands beschrieben werden können.

Dieses Protokoll nutzt eine USB-Seriell-Schnittstelle, welche zwischen einem beliebigen PC mit USB-Anschluss sowie dem LAB8/E verbunden werden kann.

Anschließend können die Daten in Form einer Lochstreifenübertragung (siehe Kapitel 2.6) an die Maschine übertragen und somit beliebige Programme ausgeführt werden.

In den folgenden Abschnitten wird die Syntax von einigen Programmier- oder Beschreibungssprachen beschrieben. Da viele der syntaktischen Konstrukte sehr einfach sind, wird oft nur ein Beispiel für diese Konstrukte geben, wobei Platzhalter verwendet werden.
Der Name dieser Platzhalter ist in spitzen Klammern eingeschlossen im Beispiel eingefügt und wird im anschließenden Absatz erläutert.

Beispiel: `'magicnumber = <eigenschaft>'`
Hierbei darf der Platzhalter *eigenschaft* durch einen beliebigen, passenden Wert ersetzt werden:
`'magicnumber = 42'`

Kapitel 5

Direkt-Assembler

Der Direkt-Assembler stellt eine direkte Abbildung des Speicherinhaltes in Textform dar. Während der Übersetzung wird keine Modifikation des angegebenen Codes durchgeführt, zudem werden nicht angegebene Addressen im Zielspeicher nicht modifiziert.

5.1 Syntax und Semantik

```
1 # Kommentare werden mit einem Doppelkreuz eingeleitet
2 23P: 10020300
3 99N: [ NUL NUL 31 ]
4 CCA: [ p1 e1 ]
```

Befehle im Direkt-Assembler werden in der Zeilen-Form `<Addresse> ':' <Wert>` notiert. Jede Zeile ist hierbei eine Zuweisung eines Wertes an eine Speicheradresse. Spätere Zuweisungen an die selbe Adresse überschreiben frühere Zuweisungen.

Eine *Adresse* kann hierbei eine Adresse im M-Speicher oder R-Speicher sein. Eine M-Speicher-Adresse verwendet die Standard-Adressnotation der Form `<Zahl>(N|P)`, eine R-Speicher-Adresse wird über den Name des zu belegenden Registers angegeben, also `CCA`, `ACC`, `BAK` oder `EXTRA`.

Ein *Wert* ist entweder eine Dezimalzahl, welche im 55-Bit-Wertebereich liegt, oder aber eine Control-Code-Assembly. Die CCA wird mit einer eckigen Klammer geöffnet, anschließend können 0 bis 11 Befehle oder Parameter angegeben werden. Die CCA schließt ebenfalls mit einer eckigen Klammer. Die Befehle in einer CCA werden im Little-Endian-Format angeben, das heißt, der erste logische Befehl belegt die höchstwertigen 5 Bit, der letzte Befehl die Niederwertigsten, dies entspricht der Ausführungsreihenfolge (Siehe Kapitel 2.4). Für einen Befehl können sowohl die Mnemonics (siehe Anhang A) als auch eine Dezimalzahl zwischen 0 und 31 angegeben werden.

5.2 Vorgehen bei der Programmierung

Um ein Programm für die Combitron zu entwickeln, ist es wichtig, verstanden zu haben, wie die Adressierung funktioniert, da diese für das Layout des Programms relevant ist.

Viele Paradigmen und Vorgehensweisen der klassischen Programmierung von Register-Systemen können auch bei der Combitron angewandt werden, doch der Speicherzugriff, und damit auch

der Codefluss, benötigen eine besondere Aufmerksamkeit und das Verständnis des Programmierers.

Bei der Programmierung einzelner Codeabschnitte sollte darauf geachtet werden, möglichst wenig Speicherzugriffe zu benötigen, da diese in vielen Fällen zu Wartezeiten führen und damit auch die Code-Länge erhöhen. Zudem kann durch ein schlechtes Speicher-Layout ebenfalls die Menge an benötigtem Code erhöht werden.

Zudem ist es notwendig, alle benötigten Variablen sowie Konstanten zu Beginn der Programmierung zu kennen, da nur 97 Werte im Speicher abgelegt werden können (siehe Abschnitt 2.7). Zwar können Speicherstellen von mehr als einem Programmteil geteilt werden, Konstanten belegen aber ebenfalls Datenworte und dürfen nicht versehentlich überschrieben werden.

Ein geschicktes Layout eines Programms im Speicher kann die Ausführungsgeschwindigkeit und den Platzbedarf eines Programmes minimieren und damit indirekt auch zu mehr Funktionalität. Da aber das Wählen eines geeigneten Layouts viel Erfahrung benötigt, werden im Folgenden zwei Vorgehensweisen mit Vor- und Nachteilen vorgestellt.

5.2.1 Entwicklungsorientiertes Vorgehen

Das erste Vorgehen orientiert sich am Programmierer und erfolgt *greedy*. Hierbei wird ein beliebiger Einstiegspunkt festgelegt, von diesem aus dann das Programm geschrieben wird. Bei jedem Speicherzugriff wird der Variable oder nachfolgendem Code-Wort bei erstmaliger Verwendung die Adresse zugewiesen, welche ohne Wartebefehl adressiert werden würde. Ist diese Adresse schon durch eine vorherige Zuweisung belegt, wird die nächstbeste Speicherstelle benutzt. Hierdurch werden zu Beginn der Programmierung nur sehr wenige Wartebefehle benötigt, führt aber bei größeren Programmen bei fortführender Länge zu einem immer höheren Bedarf an Wartebefehlen und damit zu einer schlechten Performance.

5.2.2 Performance-orientiertes Vorgehen

Das zweite Vorgehen greift eben dieses Problem auf: Zuerst werden die performancekritischen Stellen des Programms ausprogrammiert. Hierbei wird besonders darauf geachtet, dass in den innersten Schleifen möglichst wenig Warte-Befehle benötigt werden. Anschließend werden die weniger performance-kritischen Elemente programmiert, bis man schließlich bei der Programminitialisierung und dem Einstiegspunkt angelangt ist. Diese Methode ermöglicht es, performante Programme zu schreiben, welche durch die innere Optimierung auch die globale Anzahl an Wartebefehlen gering hält. Der große Nachteil ist aber, dass das Programm sehr sauber geplant werden muss und die performance-kritischen Abschnitte schon zu Beginn der Programmierung klar identifizierbar sind.

5.3 Mnemonics

Der Befehlssatz des Direktassemblers entspricht direkt dem Befehlssatz der Maschine, wie er in Kapitel 3.4 und Folgenden sowie Appendix A dokumentiert ist. Hierbei werden noch einige Befehle, welche ein im ASCII-Format nicht direkt darstellbares Symbol enthalten, mit einem Alias versehen. Dies ermöglicht eine einfache Eingabe des Programmcodes.

Kapitel 6

Linear-Assembler

Der Linear-Assembler ist ein Compiler einer Hochsprache für die Combitron. Er gibt vor, dass der Code der Combitron beliebig lange Codeblöcke erlaubt, direkte Adressierung ermöglicht und versteckt die nötigen Wartebefehle.

Hierbei geht das Verhalten weit über das eines Makro-Assemblers hinaus, da nicht nur Befehle expandiert werden, sondern auch ein Layout für das Programm bestimmt wird. Zudem werden aus dem geschriebenen Code automatisch `CCAs` compiliert, welche durch geeignete Füllbefehle zu einer linearen Codekette verknüpft werden.

6.1 Syntax und Semantik

Die Syntax des Linear-Assemblers ist im Gegensatz zum Direkt-Assembler stark an die Syntax eines klassischen Assemblers angelehnt.

```
1  ; Equations legen Daten-Adress-Aliase fest
2  .equ d0 10
3  .equ d1 11
4
5  ; Direkte Wortzuweisungen an Daten-Worte
6  .data d0 := 12345
7  .data d1 := 25635
8
9  ; Einstiegspunkt an Speicheradresse 1N
10 .location init := 1
11
12 init:
13         ld2 d0 ; Lade d1 nach EXTRA
14 loop:
15         ; Schiebe Bitmuster im Kreis
16         shl 1
17         jmp loop
```

Hierbei werden die syntaktischen Elemente in vier Gruppen aufgeteilt: *Mnemonics*, *Labels*, *Direktiven* und *Kommentare*.

Mnemonics sind Instruktionen, welche während des Compilierens in die nativen Befehle übersetzt werden, hierbei werden automatisiert die passenden Wartebefehle eingefügt. Mnemonics können kein, ein oder zwei Argumente besitzen. Das erste Argument, falls gegeben, ist im Falle von Sprunginstruktionen sowie `stc` *Labels*, für alle weiteren speicherbezogenen Instruktionen sind es Datenwort-Adressen. Alle Instruktionen, welche in Langbefehle übersetzt werden, benötigen ein zusätzliches Argument, welches die Anzahl an auszuführenden Schritten beschreibt.

Die Notation für Mnemonics ist hierbei wiefolgt:

```
1        <mnemonic> <arg1> <arg2>
```

mnemonic ist hierbei eine der in Kapitel 6.2 beschrieben Instruktionen. *arg1* sowie *arg2* sind Instruktionsargumente und je nach Instruktion optional.

Die Argumente können hierbei ein Label, eine Adress-Angabe oder eine Equation sein.

Labels stellen einen anspringbaren Zielpunkt dar und trennen `CCA`s logisch voneinander. Jedes Label definiert den Beginn einer neuen `CCA`.

Labels werden hierbei von einem Wort eingeleitet und mit einem einfachen Doppelpunkt beendet. Hierbei darf nach dem Doppelpunkt noch eine Instruktion stehen.

```
1 <name>:
```

Hierbei ist *name* der Name des Labels.

Direktiven bezeichnen Anweisungen an den Compiler, die nicht direkt in Code übersetzt werden können. Sie werden mit einem einfachen Punkt, direkt gefolgt von einem Wort eingeleitet. Anschließend folgen ein oder mehrere, direktivenabhängige Argumente oder Symbole:

```
1 .equ      <name> <addr>
2 .wild     <name>
3 .data     <addr>  := <value>
4 .location <label> := <addr>
```

`.equ` legt hierbei eine Equation an, also einen benannten Alias für die Adresse eines Datenwortes. *name* ist hierbei das Alias, *addr* die Addresse, welche in der Standard-Adressnotation angegeben werden.

`.wild` legt eine wilde Equation an. Diese verhält sich analog zu einer einfachen Equation, es wird aber nur der Alias benötigt. Der Compiler versucht hierbei, die Adresse selbst optimal festzulegen. Wilde Equations können nur Daten-Worte adressieren, Code-Worte müssen über eine feste Equation festgelegt werden.

`.data` ist eine Wertezuweisung für die angegeben Addresse *addr*. Diese kann hierbei ein vorher definiertes Alias oder aber direkt eine Datenwort-Adresse sein. *value* ist eine vorzeichenlose 55-Bit-Ganzzahl oder aber eine `CCA` in der Notation des Direkt-Assemblers, welche in das Wort an der angegebenen Adresse gelegt wird. Dies kommt der Zuweisung einer Konstante oder vorinitialisierten Variable gleich.

`.location` weißt einem Label *label* gezielt eine feste Adresse im Speicher zu. Dies kann praktisch sein, um selbstmodifizierenden Code zu ermöglichen oder aber Teile eines Programms mit dem Linear-Assembler sowie dem Direkt-Assembler zu programmieren.

Kommentare Zeilenkommentare im Linear-Assembler werden mit einem Semikolon eingeleitet, alles danach bis zum Ende der Zeile wird vom Assembler ignoriert, Blockkommentare sind nicht möglich.

```
1 <Befehl, Direktive oder Label> ; <dies ist der Kommentar>
```

6.2 Kompletter Befehlssatz

`nop`

`nop` wird direkt in den Befehl `NUL` übersetzt, hierbei werden keine zusätzlichen Befehle eingefügt.

`clr`

Die `clr`-Instruktion löscht das Register `ACC` und setzt dessen Inhalt auf 0. Hierbei wird das Carry-Flip-Flip nicht verändert. Diese Instruktion ruft intern den c-Befehl auf.

`add <addr>`

`add` addiert den Wert an der Daten-Speicherstelle *addr* auf den Akkumulator. Es werden hierbei die passenden Wartebefehle generiert sowie anschließend der Befehl + angehängt. Der Übertrag der Addition wird im Carry-Flip-Flop abgelegt. Zudem wird der Wert des Carry-Flip-Flops addiert.

`sub <addr>`

`sub` verhält sich analog zu `add`, wobei hier der Wert der Speicherstelle vom Akkumulator abgezogen wird.

`ld <addr>`

`ld` lädt das Wort an der Speicherstelle *addr* in den Akkumulator. Hierbei kann nur aus einem Daten-Wort gelesen werden.

`st <addr>`

`st` speichert den Akkumulator an die angegebene Speicherstelle *addr* ab, falls *addr* ein Datenwort adressiert. In dem Fall, dass *addr* ein Code-Wort adressiert, wird `BAK` an diese Stelle geschrieben.

`swp`

Tauscht den Inhalt von `ACC` und `BAK`.

`jmp <label>`

Springt an das angegebene Label *label*. Dies kann entweder ein Label eines Code-Blocks sein oder aber eine Code-Wort-Adresse.

`jpc <label>`

Analog zu `jmp`, der Sprung wird aber nur ausgeführt, falls das Carry-Flip-Flop gesetzt ist.

`shl`

Schiebt den Inhalt des Akkumulators um ein Bit nach links. Hierbei wird wie beim Befehl `c` der Inhalt Carry-Flip-Flops in das niederwertigste Bit, sowie das höchstwertigste Bit in das Carry-Flip-Flop geschoben.

`shl <cnt>`

Schiebt den Inhalt des Akkumulators um *cnt* Stellen nach links. Das Carry-Flip-Flop wird hierbei nicht modifiziert.

`test <line>`

Prüft, ob eine der Leitungen `K1` - `K6` gesetzt ist. Falls ja, wird das Carry-Flip-Flop gesetzt. Die zu prüfende Leitung wird über das Argument *line* als eine Zahl zwischen 1 und 6 angegeben.

`wait <cycles>`

Wartet für die angegebene Anzahl an R-Zyklen. Hiermit können längere Verzögerungen erzeugt werden.

`load`

Schaltet die Maschine in die Ladephase. Hiermit können nachträglich Programmfragmente geladen werden.

`pulse <line>`

Gibt einen 110μs langen Impuls auf der angegebenen Leitung aus. Hierbei kann *line* die Werte `e2`, `e3` oder `p1` annehmen.

`ld2 <addr>`

Lädt das Wort an Adresse *addr* in den Akkumulator, das Wort an der Stelle $addr + 1$ in das Register `EXTRA` .

`st2 <addr>`

Speichert das Wort im Akkumulator an die Adresse *addr*. Das Wort im Register `EXTRA` wird an die Adresse $addr + 1$ geschrieben.

`dec <addr>`

Dieser Befehl zählt das Wort im M-Speicher an der Adresse *addr* um eines herunter. Falls hierbei **kein** Überlauf stattfindet, wird das Carry-Flip-Flop gesetzt, der Befehl kann mit einem vorher gesetzten Carry deaktiviert werden.

`mul <addr> <cnt>` (Funktion unbekannt)

Dieser Befehl wird in einen `M0`- oder `M1`-Befehl übersetzt. Hierbei wird die Ausführung an der Adresse *addr* ausgerichtet, sodass diese in der $\bar{\text{I}}$-Phase bereit steht. *cnt* gibt die Anzahl an Iterationsschritten an.

Die Funktion der Multiplikationsbefehle ist nicht bekannt (siehe $\texttt{M}_0$, $\texttt{M}_1$ (Multiplizieren)).

`div <addr> <cnt>` (Funktion unbekannt)

Dieser Befehl wird in einen `D0`- oder `D1`-Befehl übersetzt. Hierbei wird die Ausführung an der Adresse *addr* ausgerichtet, sodass diese in der $\bar{\text{I}}$-Phase bereit steht. *cnt* gibt die Anzahl an Iterationsschritten an.

Die Funktion der Divisionsbefehle ist nicht bekannt (siehe $\texttt{D}_0$, $\texttt{D}_1$ (Dividieren)).

6.3 Automatismen

Wie schon weiter oben im Text angemerkt, verfügt der Linear-Assembler über einige Automatismen. Diese übersetzen unter anderem die Instruktionen in tatsächliche Maschinenbefehle, fügen automatisiert Wartezeiten ein und organisieren den Code in CCAs. Zudem wird versucht, ein gutes Layout für das Programm zu finden.

Befehlsersetzung

Der Linear-Assembler verwendet 20 Mnemonics, welche bei der Übersetzung den passenden Maschinenbefehlen zugeordnet werden. Hierbei wird je nach Argumentliste eines Mnemonics der passende Befehl gewählt und gegebenenfalls mit einem passenden Befehlsargument versehen.

Einige der Befehle werden direkt übersetzt, wie Beispielsweise `clr`, Befehle wie `st` werden aber zum Beispiel mit Wartezeiten versehen und durch entweder `r` oder `/r` ersetzt, je nach Argument.

In Anhang B befindet sich eine Tabelle, welche alle möglichen Ersetzungen auflistet.

Wartezeiten

Um einen bequemen Speicherzugriff zu ermöglichen und dem Programmierer das Berechnen der korrekten Wartezeiten zu ersparen, fügt der Linear-Assembler diese passend ein.

Hierbei werden die Wartezeiten so eingefügt, dass dem Befehl zum Zeitpunkt `PT` $+1$ die Daten zur Verfügung stehen. Die einzige Ausnahme bilden die beiden Sprungbefehle: Hier wird die Wartezeit so angepasst, dass die gewünschten Daten zum Zeitpunkt `PT` $+2$ bereit stehen.

Eine Sequenz an Wartebefehlen benötigt maximal 4 Silben Platz, da mit einem W_0- und einem W_1-Befehl bereits auf jedes beliebige Wort gewartet werden kann.

Übersetzung in CCAs

Jedes Label bezeichnet den Beginn einer neuen Sektion, welche endet, wenn ein weiteres Label im Quelltext erscheint.

Eine Sektion wird vom Linear-Assembler in ein oder mehrere `CCAs` übersetzt. Da diese maximal 11 Befehle fassen können, werden automatisch, falls ein Befehl nicht mehr genug Platz in einer `CCA` hat, eine neue `CCA` angelegt und in der vorherigen `CCA` ein passender Füllbefehl angefügt. Zudem wird zum Ende einer Sektion ein Sprung in die nächste Sektion vollzogen.

Hiermit ergibt sich eine lineare Code-Kette, welche das Verhalten eines Prozessors mit klassischem Befehlszähler nachahmt.

Layout-Findung

Da, wie in Abschnitt 5.2 beschrieben, das Layout eines Programmes über die Anzahl der benötigten Wartebefehle sowie die Performance des gesamten Programmes einen großen Einfluss hat, versucht der Linear-Assembler hier eine gute Lösung zu finden.

Dies geschieht über die Annahme, dass ein Programm-Layout mit einer geringen Anzahl an benötigten Code-Worten auch eine bessere Performance besitzt als ein Layout des gleichen Programmes mit einer höheren Anzahl an Code-Worten.

6.4 Vorgehen bei der Programmierung

Der Linear-Assembler vereinfacht das Programmieren der Combitron: Er übernimmt das Layouting des Programms, die Zuweisungen von Speicherstellen für die einzelnen `CCAs`, sowie das Aufteilen längerer Codepassagen in mehrere Worte.

Dies ermöglicht es, auch weniger erfahrenen Programmierern, die Combitron zu programmieren und ein Gefühl für die Architektur zu entwickeln.

Um ein Programm zu entwickeln, kann progressiv vorgegangen werden:
Man beginnt beim Einstiegspunkt des Programmes und schreibt den Code von dort ausgehend. Wenn eine Variable benötigt wird, kann diese im Code als wilde oder feste Equation definiert werden. Es ist kein spezielles Augenmerk auf die Position der Variable nötig.

Da der Compiler nach einer für das gesamte Programm optimalen Lösung sucht, ist es nicht nötig, auf performancekritische Stellen zu achten.

Kapitel 7

Toolchain

Die *Toolchain* ist ein Werkzeug, welches das einfache Entwickeln für die Combitron erlaubt. Hierbei bietet sie eine Menge an kleineren Werkzeugen und Befehlen, welche im Zusammenspiel eine Arbeitsumgebung bieten.

7.1 Benutzung / Interface

Die Toolchain bietet ein einfaches Kommandozeilen-Interface. Hierbei wird ein Recipe (siehe Kapitel 7.2) über die Standardeingabe eingelesen und anschließend ausgeführt. Zudem ist es möglich, die Recipe-Datei als ersten Kommandozeilenparameter zu übergeben.

Wenn das Programm ohne eine Weiterleitung auf die Standardeingabe oder Eingabedatei gestartet wird, befindet sich die Toolchain im interaktiven Modus. Hier können Befehle von Hand eingegeben werden. Dabei wird zeilenweise vorgegangen: Der Benutzer gibt eine Zeile eines Recipes ein und bestätigt diese anschließend mit `Enter`. Der eingegebene Befehl wird nun sofort ausgeführt. Nach Abschluss des Befehls steht wieder die Eingabemöglichkeit bereit.

7.2 Recipe

Die einzelnen Unterprogramme werden über ein Recipe gesteuert. Dies ist eine Datei, welche einzelne, sequenziell abgearbeitete Befehle enthält. Die Befehle folgen einer einfachen Syntax, welche eine Art Satzform bilden:

`<Befehl> <Argument> <Argument> … ';'`

Hierbei wird der *Befehl* sowie die *Argumente* mit ein oder mehreren Leerzeichen voneinander getrennt und mit einem Semikolon beendet.

Ein Befehl oder Argument kann entweder ein Wort, eine Zahl oder ein Text sein. Texte werden mit Anführungszeichen eingeschlossen.

Beispiel: Dateianalyse

```
1 declare testcode as rdelay;
2 declare report as analysis;
3 assemble "testroutine.dct" into testcode;
4 analyse testcode into report;
```

```
5 generate table from report into "scratchpad.md";
6 exit;
```

7.3 Objektverwaltung

Die Toolchain verwendet zur internen Speicherung vier Typen an Objekten, welche mit der folgenden Syntax deklariert werden:

```
declare <obj> as <type>;
```

Hierbei ist *obj* der Name des Objektes, *type* gibt den Typ des Objektes an. Die folgenden Objekttypen sind möglich:

Typ	Beschreibung
`rdelay`	R-Speicher
`mdelay`	M-Speicher
`analysis`	Analyse
`punchtape`	Bandspeicher

M-Speicher
Ein M-Speicher-Objekt enthält den Zustand der M-Delay-Line, also 198 Worte zu jeweils 99 Worten in der P-Phase und der $\overline{P}$-Phase.

Zudem enthält es eine Liste an textuellen Symbolen, welche mit einer Addresse versehen sind. Diese können als Einstiegspunkt in ein Programm verwendet oder aber vom Linear-Assembler referenziert werden.

R-Speicher
Ein R-Speicher-Objekt enthält den Zustand der R-Delay-Line, also einen CPU-Zustand. Es werden die vier Worte `CCA`, `ACC`, `BAK` oder `EXTRA` gespeichert.

Analyse
Eine Analyse enthält mehrere Teilanalysen zu R-Speichern sowie maximal eine Teilanalyse eines M-Speichers.

Bandspeicher
Ein Bandspeicher enthält eine geordnete Sequenz an R-Speichern, welche an ein Gerät übertragen werden kann.

7.4 Direkt-Assembler

```
assemble <file> into <object>;
```

Hierbei ist *file* der Dateiname der zu assemblierenden Datei, *object* beschreibt den Namen eines vorher deklarierten Speicher-Objekts.

Mehr Informationen zum Aufbau der Assembler-Dateien findet sich in Abschnitt 5.

7.5 Linear-Assembler

```
compile <file> into <object>;
```

Hierbei ist *file* der Dateiname der zu assemblierenden Datei, *object* beschreibt den Namen eines vorher deklarierten M-Speicher-Objekts. Hierbei werden neben der Code-Übersetzung auch die definierten Labels des Assembler-Programmes in *object* abgelegt.

Mehr Informationen zum Aufbau der Assembler-Dateien findet sich in Abschnitt 6.

7.6 Analyse-Werkzeug

```
analyse <memory> into <analysis>;
```

Dieser Befehl erstellt eine Auswertung des Speichers *memory* und fügt diese der Auswertungssammlung *analysis* an. Hierbei können pro Auswertungsobjekt nur ein M-Speicher, aber beliebig viele R-Speicher angefügt werden.

```
generate table from <analysis> into <file>;
generate diagram from <analysis> into <file>;
```

Dieser Befehl erstellt entweder eine tabellarische Form der Auswertung (ausgegeben als GitHub-Markdown-Datei) oder aber ein Code-Fluss-Diagramm zwischen den einzelnen Speicherzellen (Graphviz-Dot-Datei). Hierbei wird die darzustellende Analyse über den Parameter *analysis* festgelegt, die Zieldatei der Analyse via *file*.

7.7 Bandspeicher-Generator

```
append <memory> to <punchtape>;
```

Mit diesem Befehl wird ein Speicher-Objekt an einen Bandspeicher angehängt. Wenn *memory* ein R-Speicher ist, wird dieser direkt an des Ende des Bandspeichers angehängt. Für einen M-Speicher wird eine Sequenz an R-Speichern generiert, welche die nicht-leeren Speicherzellen an die passende Stelle im M-Speicher der Maschine überträgt.

7.8 Gerätesteuerung

Der `device`-Befehl unterstützt Unterbefehle, mit welchen man eine Lade-Schaltung steuern kann und damit ein Programm an die Combitron übertragen und ausführen.

`device open <serialPort>;`
Dieser Befehl öffnet eine serielle Schnittstelle, hierbei gibt *serialPort* den Namen der Geräte-Datei unter Linux oder den Namen der seriellen Schnittstelle unter Windows an.

`device close;`
Hiermit wird die serielle Schnittstelle geschlossen und der Port zur Verwendung durch andere Programme freigegeben.

`device reset;`
Mit diesem Befehl wird die Rücksetzung der Combitron durchgeführt. Anschließend sind die Speicher der Combitron leer und das Gerät befindet sich in der Lade-Phase.

`device load <memory>;`
memory gibt den Namen eines R-Speicher-Objektes an, welches an die Combitron übertragen wird. Das Gerät sollte sich hierzu in der Lade-Phase befinden.

`device boot <punchtape>;`
Dieser Befehl ist ähnlich zu `load`, wobei hier kein einzelner R-Speicher übertragen wird, sondern eine Liste an Befehlen, welche in dem Bandspeicher *punchtape* abgelegt sind.

`device start <address>;`
Überträgt ebenfalls einen Befehlssequenz an die Combitron, wobei diese automatisch generiert wird. Diese Sequenz wartet auf die angegebene Addresse in *address*, wobei diese in der Code-Phase ($\overline{\text{P}}$) liegen muss.

`device start <label> from <memory>;`
Dieser Befehl ist ähnlich dem vorherigen, wobei hier aber die Startaddresse über ein im M-Speicher *memory* definiertes Label *label* ist.

Eine Adressangabe im Standard-Adressformat ist ebenfalls ein erlaubter Einstiegspunkt für *label*.

`device transfer <bitstream>;` Mit diesem Befehl kann eine Übertragung von „Rohdaten“ stattfinden. Diese Daten liegen in Form eines Strings *bitstream* vor, welcher nur die Zeichen `'0'`, `'1'` sowie `' '` erlaubt.

Die Kodierung des Streams ist wie folgt: Eine `'0'` übertägt ein 0-Bit, eine `'1'` übertragt analog dazu ein 1-Bit. `' '` wird ignoriert und dient zur Steigerung der Lesbarkeit. Die Daten müssen in Form einer Boot-Sequenz vorliegen, diese wird in Abschnitt 2.6 näher erläutert.

Der Stream wird von links nach rechts übertragen, zwischen jedem Bit befindet sich eine kurze Pause.

7.9 Steuerbefehle

`sleep <time>;`
Wartet *time* Sekunden und fährt dann mit der Ausführung fort.

`sleep <time> <unit>;`
Wartet die angegebene Zeit. Hierbei bestimmt *time* die Menge und *unit* die Einheit. *unit* kann *s* (Sekunden) oder *ms* (Millisekunden) sein.

`echo <text>;`
Gibt den Text *text* auf der Standardausgabe aus.

`exit;`
Beendet die Interpretation der Datei und beendet die Toolchain.

`label <name>;`
Definiert ein Sprungziel *name*. Dieses kann mit dem `goto`-Befehl angesprungen werden. Dies erlaubt das Erstellen von Schleifen in der Ausführung der Toolchain.

`goto <label>;`
Springt an ein Label. Hierbei gibt *label* den Namen des Labels an.

`set <option> <value>;`
Setzt einen Konfigurationseintrag der Toolchain. Hiermit können während der Ausführung ver-

schiedene Eigenschaften angepasst werden. *option* gibt hierbei den Namen der Option an, welche auf *value* gesetzt wird.

Eine Liste aller möglichen Optionen befindet sich im Anhang D.

7.10 Interaktive Befehle

`list <what>;`
Gibt eine Liste aller deklarierten Objekten vom Typ *what* aus. Hierbei kann *what* einen der folgenden Werte annehmen:

`<what>`	**Ausgabe**
`all`	Alle deklarierten Objekte werden ausgegeben
`memory`	Alle R- und M-Speicher-Objekte
`rmemory`	Alle R-Speicher-Objekte
`mmemory`	Alle M-Speicher-Objekte
`analysis`	Alle Analyse-Objekte
`punchtape`	Alle Bandspeicher-Objekte

Kapitel 8

Toolchain - Implementierungsdetails

Im folgenden Abschnitt wird näher auf die internen Funktionsweise der Toolchain eingegangen. Der Fokus liegt hierbei auf implementieren Algorithmen und weniger auf der Benutzung der Software selbst.

Die komplette Toolchain wurde in C# entwickelt, da die Sprache sowohl einen großen Abstraktionslevel bietet, also auch genug Flexibilität und Funktionalität, wenn es um die Arbeit mit vorzeichenlosen Integer-Typen, regulären Ausdrücken sowie Hardware-Schnittstellen geht.

Der hierbei verwendete Dialekt für reguläre Ausdrücke wird in [7] näher beschrieben.

8.1 Direkt-Assembler

Der Direkt-Assembler ist sehr einfach implementiert. Er benutzt einen einfachen Parser, um die Assembler-Dateien einzulesen. Dieser übersetzt den Code direkt in die gewünschte Form und ist zusammengesetzt aus einem generierten Lexer sowie der händisch programmierten semantischen Auswertung der erkannten Tokens.

Der Lexer verwendet ein sehr einfaches Vorgehen:

```
1  lexer(text, patterns):
2          i := 0
3          ergebnis := [ ]
4          while i < länge(text):
5                  foreach pat in patterns:
6                          match := regex(pat, text, i)
7                          if nicht_passend(match):
8                                  continue
9                          ergebnis += (pat, text(match), i)
10                         i += länge(text(match))
11                         break
12                 if kein_match:
13                         fehler("Unbekannter token!")
14         return ergebnis
```

Hierbei ist *text* der Eingabetext der Assembler-Datei, *patterns* eine Liste von regulären Ausdrücken. *ergebnis* ist eine Liste an erkannten Token.

Die Funktion *regex* führt einen regulären Ausdruck auf der Eingabe ab einer bestimmten Position aus, *nicht_passend* prüft, ob das Ergebnis der Ausführung erkannt wurde oder nicht.

Die für den Direkt-Assembler verwendeten Token-Kategorien sind wie folgt:

Typ	Regulärer Ausdruck [7]
Adresse	`\d{1,2}[NP]|ACC|BAK|CCA|EXTRA`
Ganzzahl	`\d+`
Befehl	`\/?[A-Za-z0-9_+\-]{1,3}`
Doppelpunkt	`\:`
AnfangCCA	`\[`
EndeCCA	`\]`

Zudem gibt es noch zwei nicht ausgegebene Token-Kategorien:

- Kommentar
- Whitespace

Diese sind für die Semantik des Codes nicht erforderlich und können daher bei der semantischen Auswertung ignoriert werden.

Diese Auswertung erfolgt nach erfolgreichem Tokenisieren der Eingabedatei. Hierbei wird die folgende Grammatik erkannt:

```
1 Programm  = { Zuweisung } ;
2 Zuweisung = Adresse, Doppelpunkt, ( Ganzzahl | CCA ) ;
3 CCA       = AnfangCCA, { Befehl | Ganzzahl }, EndeCCA ;
```

Da die Grammatik hierfür sehr einfach ist und keine Rekursion enthält, wurde der Parser hierfür händisch programmiert und nicht auf einen Parser-Generator zurückgegriffen.

Während des Einlesens des Programmes wird jede Zuweisung sofort ausgeführt. Falls einer Adresse eine Ganzzahl zugewiesen wird, wird diese direkt als dezimaler Zahlenwert für das Wort übernommen. Im Falle einer CCA-Zuweisung wird zuerst die CCA aus den einzelnen Befehlen bzw. Silbenwerten zusammengesetzt und anschließend ebenfalls dem Speicher an der gegebenen Adresse zugewiesen.

8.2 Linear-Assembler

Der Linear-Assembler besitzt einen mehrstufigen Aufbau. Zuerst wird der Text, wie beim Direkt-Assembler, mit einem Lexer gelesen. Anschließend wird die tokenisierte Version des Quellcodes in eine abstrakte Struktur übersetzt, diese wird anschließend mehrere Male in Maschinensprache übersetzt, um ein gutes Layout zu finden.

Parser

Der Parser verwendet die selbe Implementierung der Lexer-Logik wie der Direkt-Assembler (siehe Abschnitt 8.1). Hierbei werden andere Token-Typen verwendet, da die Syntax der beiden Assembler-Dialekte nur wenig Gemeinsamkeiten besitzen:

Typ	Regulärer Ausdruck [7]
Equation	`\.equ`
Data	`\.data`
Wild	`\.wild`
Location	`\.location`
Zuweisung	`\:\=`
Label	`\w\:`
Adresse	`\d?\d[NP]`
Ganzzahl	`\d+`
Wort	`\w+`

Die Tokens werden ebenfalls über einen handgeschriebenen Parser für folgende Grammatik (in EBNF nach ISO-14977[5]) erkannt:

```
1  Programm = {
2             DirEqu
3           | DirData
4           | DirWild
5           | DirLoc
6           | Label
7           | Mnemonic
8  } ;
9  DirEqu   = Equation, Wort, Zuweisung, Adresse ;
10 DirData  = Data, ( Adresse | Wort ), Zuweisung, Ganzzahl ;
11 DirWild  = Wild, Wort ;
12 DirLoc   = Location, Wort, Zuweisung, Adresse ;
13 Mnemonic = Wort, { Wort | Ganzzahl | Adresse } ;
```

Der Parser übersetzt diese Grammatik in eine Liste an Code-Sequenzen, welche mit einem *Label* beginnen und anschließend von einem oder mehreren *Mnemonic*s gefolgt werden. Hierbei wird jede Sequenz mit ihrem Label benannt. Hierbei wird die Mehrdeutigkeit der *Mnemonic*-Regel aufgelöst, in dem der Parser die Liste der möglichen Parameter für jeden Mnemonic kennt und diese entsprechend anwendet.

Die Assembler-Direktiven *DirEqu*, *DirData*, *DirWild* sowie *DirLoc* werden beim Parsen in mehreren Listen abgelegt, um später ausgewertet werden zu können.

Das Ergebnis des Parsens ist zum Einen eine Menge an benannten Code-Sequenzen, zum anderen die Listen, die diese Direktiven enthalten.

Codegeneration

Auf Basis der vom Parser ausgegebenen Daten findet anschließend die Code-Generation statt. Hierbei werden die einzelnen Mnemonics in den Code-Sequenzen in die passenden Maschinenbefehle übersetzt.

Um dies zu ermöglichen, wird der sogenannte *Code-Emitter* verwendet. Dieser abstrahiert die Übersetzung einzelner Befehle und bietet die Möglichkeit, vor dem eigentlichen Befehl die passenden Wartebefehle auszugeben.

Zudem sorgt der *Code-Emitter* für die Aufteilung des generierten Codes in einzelne `CCAs`. Dies geschieht über einen internen Befehlszähler, welcher einen Sprungbefehl in eine neue `CCA` einfügt, wenn weniger als 4 Befehle in der aktuell generierten `CCA` zur Verfügung stehen.

Die Zieladresse der neu erzeugten CCA erhält der *Code-Emitter* über einen Adress-Pool, welcher im Abschnitt 8.2 näher erläutert wird.

Zu Beginn der Übersetzung einer Code-Sequenz wird die Position der initialen `CCA` ebenfalls über den selben Adress-Pool bestimmt. Die hier bestimmte Adresse entspricht der Adresse des Labels, welches die Sequenz einleitet.

Um die passende Wartezeit auf eine bestimmte Speicheradresse zu generieren, wird zuerst bestimmt, ob es sich um einen Füllbefehl (bedingt oder unbedingt) handelt. Wenn dies der Fall ist, wird die Adresse um 2 reduziert, da die Füllbefehle auf das nachfolgende Wort zugreifen.

Anschließend wird die Zeit berechnet, welche von der aktuellen Program Timer-Zeit bis zu dieser Adresse vergeht. Für diese Zeitspanne wird dann die Silbenanzahl der benötigten Wartebefehle berechnet. Falls die Anzahl der verbleibenden Silben in der aktuellen `CCA` nicht mehr ausreichend sind, um den Befehl zu generieren, wird, wie oben beschrieben, eine neue `CCA` bereitgestellt. Falls dies geschieht, wird die Zeitspanne neu berechnet und anschließend der passende Code in die `CCA` emittiert.

Randomisierte Layout-Findung

Da der Linear-Assembler ein optimiertes Ergebnis liefern soll, muss die Auswahl der Nachfolge-CCAs passend erfolgen. Je weniger auf eine bestimmte Speicherstelle gewartet wird, desto effizienter ist der Code.

Um diese Optimierung zu lösen, müssen die Abhängigkeiten zwischen den Befehlen und ihren Zielworten aufgelöst werden. Wenn hierbei die Adresse eine logischen Wortes so angepasst wird, dass die globalen Wartezeiten auf dieses Wort reduziert werden, kann es geschehen, dass es an einer Stelle im Code zu einem Blow-Up kommt und hiermit neue Code-Worte alloziert werden müssen. Dies kann zu Ringabhängigkeiten in der Optimierung führen.

Diese Optimierung wird in der konkreten Implementierung mit einem randomisierten Ansatz angegangen. Hierbei werden die verwendeten Wort-Adressen aus einem zufällig gemischten Adresspool gezogen. Hierdurch entsteht eine willkürliche Anordnung des Programmcodes im Speicher.

Das dabei entstandene Programm besitzt nun eine gewisse Qualität, welche durch die Anzahl der benötigten Worte gemessen wird. Je geringer diese Anzahl ist, desto weniger Wartebefehle werden benötigt und damit ist die geschätzte Qualität höher.

Um nun ein gutes Layout zu finden, wird die Code-Generation mehrere hundert mal mit unterschiedlichen Pseudo-Zufallswerten ausgeführt. Am Ende wird die Variante behalten, welche höchste Qualität bietet.

Der Pseudo-Zufallsgenerator wird zu Beginn eines Übersetzungsvorgangs mit immer dem selben Seed initialisiert, um eine deterministische Übersetzung zu gewährleisten.

Kompatibilität mit Direkt-Assembler

Um die Kompatibilität mit in Direkt-Assembler erstellten Programmen zu ermöglichen, erlaubt die Direktive `.location` sowie `.equ` das manuelle Festlegen von Speicheradressen für einzelne Code- bzw. Datenworte.

Zudem werden bei der Übersetzung des Programms die schon verwendeten Worte aus dem Adress-Pool entfernt, um eine doppelten Verwendung zu verhindern.

Hiermit kann mit dem Linear-Assembler ein Programm um Komponenten erweitert werden, welche mit dem Direkt-Assembler oder einem anderen Werkzeug erstellt wurden.

8.3 Programm-Analyse

Um den Kontrollfluss eines Programmes besser zu verstehen und Programmierfehler zu finden, wurde eine Code-Analyse implementiert.

Diese nimmt als Eingabewert ein M-Speicher-Objekt und analysiert dessen Inhalte. Hierbei wird ein Graph angelegt, dessen Knoten den einzelnen Code- und Datenworten des Speichers entsprechen.

Die Kanten des Graphen stellen Speicherzugriffe und Sprünge dar. Jeder Befehl in einem Wort, der auf ein anderes Wort lesend oder schreibend zugreift, erzeugt eine Daten-Kante vom „ausführenden“ zum referenzierten Wort. Jeder bedingte oder unbedingte Sprung erzeugt eine Kontrollfluss-Kante zum angesprungenen Wort.

Zudem werden bei der Analyse für jeden Knoten eine Liste der enthaltenen Befehle erstellt, wobei jeder Knoten mit der konkreten Ausführungszeit, sowie Dauer annotiert wird. Weiterhin wird, falls der Befehl eine Kante erzeugt, diese Kante ebenfalls mit dem Befehl assoziiert.

Die Ergebnisse einer solchen Analyse können nun vom Programmierer entweder in tabellarischer Form oder aber als graphische Darstellung des Graphen analysiert werden.

Address 38P̄

Numeric Value: 28999345158914048

Disassembly: [W0 24 b W1 0 W0 14 F]

Jumped At From: 12P̄, 38P̄, 58P̄

Index	Opcode	Command	Time	Duration	Info
1	11001	W0	40	10	
2	11000				
3	00110	b	60	1	61P → ACC
4	11000	W1	62	66	
5	00000				
6	11001	W0	95	20	
7	01110				
8	10101	F	36	2	↳ 38P̄

Abbildung 8.1: Ausschnitt aus der tabellarische Analyse

Die tabellarische Darstellung ermöglicht es, gezielt nach Programmierfehlern zu suchen und die Binärdarstellung der Daten zu kontrollieren. Zudem enthält die Tabelle, wie in Abb. 8.1 dargestellt, eine weitere Spalte, welche eine einfache Interpretation des Codes darstellt. Hierbei bestehen Hyperlinks zwischen den referenzierten Worten. Dies erlaubt es, dem Programmfluss leicht zu folgen.

Um die Tabellen darzustellen, wird ein GitHub-Markdown-Dokument erstellt, welches anschließend vom Benutzer mit dem von GitHub bereitgestellten Renderer in ein HTML-Dokument übersetzt werden kann. Zudem wird eine Cascading-Style-Sheet-Datei benötigt, welche eine angenehme Formatierung des Dokuments bereitstellt.

Eine visuelle Darstellung des Graphen wird über das `dot`-Format des Graphviz-Projektes ausgeben. Diese kann mit dem Werkzeug `dot` in eine Bilddatei gerendert werden. Diese Darstellung ermöglicht es, schnell eine Übersicht über den Kontrollfluss eines Programmes zu gewinnen.

Zudem zeigt sich durch die Anordnung des Graphen, welche „Module“ bzw. Gruppen das Programm bildet. Eine solche Gruppenbildung ist in Abb. 8.2 dargestellt.

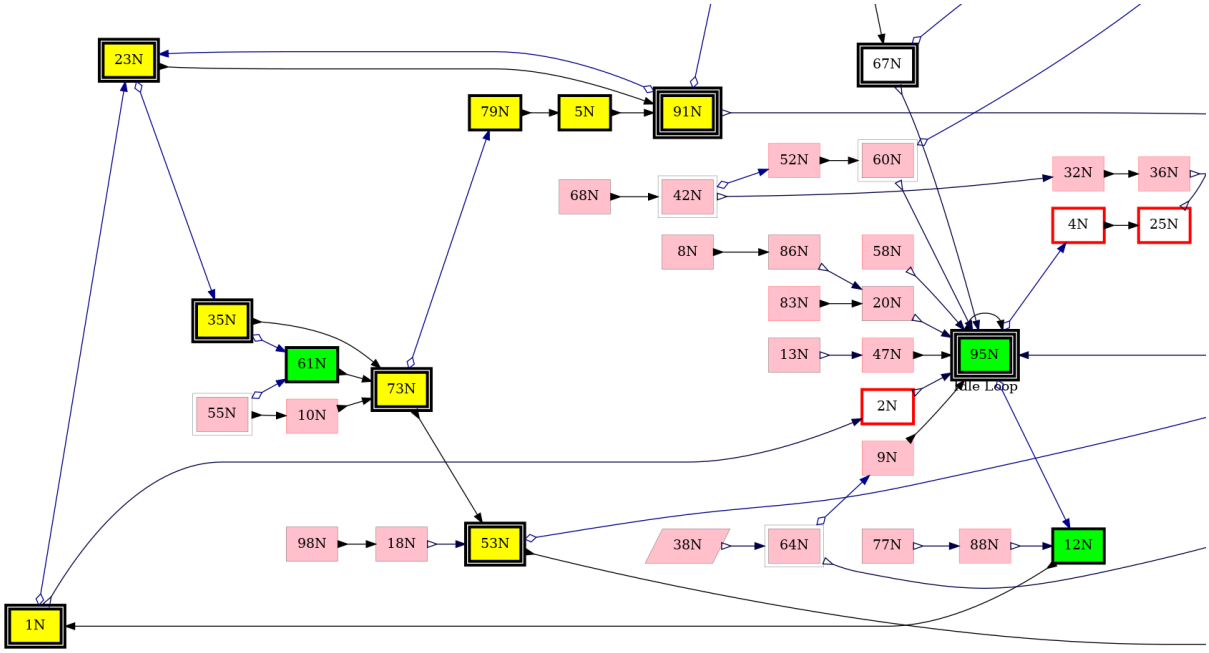


Abbildung 8.2: Ausschnitt aus der graphischen Analyse

8.4 Ladeschaltungs- und Übertragungsprotokoll

Das mit dem LAB8/E entwickelte Ladewerkzeug verwendet ein allgemein funktionsfähiges Protokoll, welches es auch ermöglicht, andere Ladewerkzeuge zu entwerfen.

Das Werkzeug enthält intern einen Puffer, welcher einen R-Speicher enthält. Dieser kann mithilfe des Protokolls verändert werden und kann anschließend an das Gerät übertragen.

Der Zugriff auf diesen Speicher erfolgt in Silbenpaaren, also immer gleichzeitig auf die Silben der $\overline{\text{P}}$-Phase sowie der P-Phase. Hierbei stellen die Silbenpaare 0 bis 10 die Silben des Wortpaares in der $\overline{\text{I}}$-Phase dar, die Silbenpaare 11 bis 21 enthalten die Silben des Wortpaares der I-Phase.

Das Protokoll benötigt eine ASCII-fähige Halb-Duplex-Schnittstelle. Über diese werden vom Host Befehle an das Ladewerkzeug übertragen. Ein Befehl besteht immer aus einem einzelnen ASCII-Zeichen. Die einzige Ausnahme bildet der Silben-Befehl `'S'`. Nach der Übertragung des Befehls sendet der Host ein Line Feed, um die Ausführung des Befehls zu starten.

Das Ladewerkzeug quittiert die Ausführung des Befehls entweder mit einem `'O'` oder einem `'E'`. Das große O stellt die Bestätigung der erfolgreichen Ausführung des Befehls dar, ein großes E zeigt einen Fehler an.

Für die Übertragung numerischer Werte wird das oktale Stellenwertsystem verwendet, da dies eine einfache Interpretation der Daten auf dem LAB8/E ermöglicht.

Das Protokoll unterstützt folgende Befehle, welche vom Host gesendet werden können:

Befehl	Bedeutung
`'R'`	Löst den *m6*-Impuls aus und überträgt ein einzelnes Taktsignal. Reset der Combitron.
`'T'`	Transfer des gepufferten R-Speichers an die Combitron.
`'S XX YYZZ'`	Setzt das Silbenpaar XX auf den Wert YY für $\overline{\mathrm{P}}$ und ZZ für P.
`'L'`	Fordert die Übertragung eines Listings des gepufferten R-Speichers an.
`'N'`	Setzt alle Silben im gepufferten R-Speicher auf 0.

Zusätzlich zu der Befehlsbestätigung mit `'O'` und `'E'` muss das Ladegerät auf den Befehl `'L'` mit einer Übertragung des R-Speichers antworten. Hierbei wird der `'S'`-Befehl verwendet, der vom Host zum Setzen einzelner Silben genutzt wird. Es müssen dabei nur die Silbenpaare übertragen werden, welche nicht null sind.

Kapitel 9

Fazit & Ausblick

Die Arbeit zeigt, dass es möglich ist, einen Turing-vollständigen Computer zu bauen, der nur eine geringe Anzahl an Transistoren und Flip-Flops benötigt. Dies führt aber dazu, dass sehr viel Arbeit auf den Programmierer abgewälzt wird und beispielsweise die Adressierung des Speichers durch den Programmierer erfolgt.

Es wird ebenfalls gezeigt, dass mit der Hilfe moderner Computer diese Probleme zu einem gewissen Teil auch automatisiert werden können und dem Programmierer zur Hand gehen.

Eine weitere Erkenntnis der Arbeit ist, dass ein Assembler-Dialekt nicht zwingendermaßen eine gewohnte Syntax besitzen muss, um damit gut arbeiten zu können. Hier hat sich gezeigt, dass eine reine Adress-Wert-Zuweisung besser dazu geeignet ist, die Maschinensprache zu beschreiben, als ein typisch „linearer“ Assembler.

Auch ist die Entwicklungs- und Forschungsarbeit bezüglich der Diehl Combitron noch nicht erschöpft. Der Linearassembler findet im Moment nur „zufälligerweise“ ein gutes Programmlayout. Der Algorithmus hierfür ist nur provisorisch und könnte in Zukunft durch eine nicht randomisierte Variante ausgetauscht werden.

Die Toolchain in ihrer aktuellen Form ist ebenfalls eher ein Behelf als eine tatsächlich produktive Arbeitsumgebung. Sie erfüllt ihren Zweck, ist aber ebenfalls weit von einem Optimum entfernt.

Weiterhin könnte die Toolchain noch um einen Simulator der Maschine erweitert werden, welcher auf Basis der Logikgleichungen eine logisch exakte Simulation ausführt. Dies würde die Entwicklung von Programmen beschleunigen, da hiermit ein möglicher Debugger für die Programme bereitsteht.

Anhang A

Befehlssatz

Die folgende Tabelle enthält eine Auflistung aller 32 Maschinenbefehle mit ihrer Bezeichnung aus dem Handbuch, dem Mnemonic, welches im Handbuch sowie dem Direktassembler verwendet wird, zudem die Binärcodierung der Befehlssilbe.

Handbuch	**Mnemonic**	**Binärcodierung**
Null	`0`	00000
Prüfe K1	`T1`	00001
Prüfe K2	`T2`	01000
Prüfe K3	`T3`	01001
Prüfe K4	`T4`	10000
Prüfe K5	`T5`	10001
Prüfe K6	`T6`	00010
Wiederladen durch Band	`e1`	00011
Ausgabe an Druckersteuerung	`e2`	01010
Ausgabe an Druckersteuerung	`e3`	01011
Herunterzählen	`d`	10010
Löschen $\overline{IP}$ in R-Delay-Line	`c`	10011
Addieren	`+`	00100
Subtrahieren	`-`	00101
Bringe P	`b`	00110
Abspeichern P	`r`	00111
Vertauschen	`x`	01100
Schieben	`s`	01101
Ausgabe an Druckersteuerung	`p1`	01110
Abspeichern $\overline{P}$	`r` `/r`	01111
Bedingter Füllbefehl	`C`	10100
Unbedingter Füllbefehl	`F`	10101
Bringe P – doppelt	`B`	10110
Abspeichern P – doppelt	`R`	10111
Warten (33-64 R-Zyklen)	W_1 `W1`	11000
Warten (1-32 R-Zyklen)	W_0 `W0`	11001
Schieben (33-64 R-Zyklen)	S_1 `S1`	11010
Schieben (1-32 R-Zyklen)	S_0 `S0`	11011
Dividieren	D_1 `D1`	11100
Dividieren	D_0 `D0`	11101
Multiplizieren	M_1 `M1`	11110
Multiplizieren	M_0 `M0`	11111

Anhang B

Befehlsersetzung im Linear-Assembler

Die folgende Tabelle enthält alle Mnemonics, die der Linear-Assembler unterstützt und listet die möglichen Ersetzungen der Befehle auf.

Mnemonic	**Mögliche Befehle**
`nop`	`NUL`
`clr`	`c`
`add <addr>`	`+`
`sub <addr>`	`-`
`ld <addr>`	`b`
`st <addr>`	`r`, `/r`
`swp`	`x`
`jmp <label>`	`F`
`jpc <label>`	`C`
`shl`	`s`
`shl <cnt>`	`S0`, `S1`
`test <line>`	`T1`, `T2`, `T3`, `T4`, `T5`, `T6`
`wait <cycles>`	`W0`, `W1`
`load`	`e1`
`pulse <line>`	`e2`, `e3`, `p1`
`ld2 <addr>`	`B`
`st2 <addr>`	`R`
`dec`	`d`
`div <addr> <cnt>`	`D0`, `D1`
`mul <addr> <cnt>`	`M0`, `M1`

Anhang C

Recipe-Befehle

Befehlssyntax	**Argument**	**Argumentbeschreibung**
`declare <name> as <type>`	`<name>`	Der Name des zu deklarierenden Objektes
	`<type>`	`mdelay`, `rdelay`, `analysis` oder `punchtape`
`assemble <src> into <target>`	`<src>`	Eingabedatei für Direktassembler
	`<target>`	Zielobjekt (R-Speicher oder M-Speicher)
`analyse <src> into <target>`	`<src>`	Zu analysierendes Speicher-Objekt
	`<target>`	Ziel-Analyse
`list <what>`	`<what>`	`memory`, `analysis`, `rmemory` ,`mmemory`, `punchtape` ,`all`
`sleep <time>`	`<time>`	Zeit in Sekunden
`sleep <time> <unit>`	`<time>`	Zeit in `<unit>`
	`<unit>`	`ms` oder `s`
`device load <obj>`	`<obj>`	R-Speicher-Objekt
`device open <dev>`	`<dev>`	Name bzw. Datei des Seriellports
`device boot <obj>`	`<obj>`	Bandspeicher-Objekt
`device transfer <str>`	`<str>`	Bit-Sequenz
`device start <addr>`	`<addr>`	Einstiegspunkt (Adresse)
`device start <label> from <obj>`	`<label>`	Name des Labels
	`<obj>`	Referenz-M-Speicher für Label
`device reset`		
`device close`		
`echo <text>`	`<text>`	Ausgabe-Text
`generate <type> from <src> into <target>`	`<type>`	`diagram` oder `table`
	`<src>`	Analyse-Objekt
	`<target>`	Ausgabe-Datei
`exit`		
`label <name>`	`<name>`	Name des Labels
`goto <label>`	`<label>`	Sprungziel

`compile <src> into <target>`	`<src>`	Eingabedatei für Linear-Assembler
	`<target>`	M-Speicher-Objekt
`set <option> <value>`	`<option>`	Option
	`<value>`	Neuer Wert
`append <src> to <target>`	`<src>`	Speicher-Objekt
	`<target>`	Punchtape

Anhang D

Toolchain-Optionen

Option	**Beschreibung**
`compiler.rounds`	Bestimmt die Anzahl an Optimierungsversuchen, die der Linear-Assembler vornimmt.
`compiler.stats`	`true` oder `false`. Wenn `true`, gibt der Linear-Assembler Informationen über das Programm auf der Standardausgabe aus.
`punchtape.delay`	Die Zeit in Millisekunden zwischen der Übertragung einzelner Ladesequenzen über den Lochstreifen.
`device.verbose`	`true` oder `false`. Wenn `true`, wird die Kommunikation zwischen PC und Lochstreifenemulator auf der Standardausgabe ausgegeben.
`device.delay`	Zeit in Millisekunden, die nach dem Öffnen der Seriellschnittstelle gewartet wird.

Anhang E

Programmbeispiele für Direktassembler

E.1 Beispiel: Zyklische Ausführung

Dieses einfache Beispiel beschreibt ein Programm, welches die Ausführung immer zwischen zwei Codeblöcken hin und her reicht. Jeder Codeblock gibt hierbei ein „Ping“ (`e2`) oder aber ein „Pong“ (`e3`) aus.

```
# Ping
27N: [ e2 W1 4 F]

# Pong
58N: [ e3 W0 3 F ]
```

E.2 Beispiel: Starten eines Programmes

In diesem Beispiel wird eine Sequenz an Programmen angegeben, mit welcher das vorherige Beispiel in den Speicher der Maschine übertragen werden kann.

```
# 01: Lade Ping
CCA: [ W1 4 /r e1 ]
BAK: [ e2 W1 4 F]
```

```
# 02: Lade Pong
CCA: [ W0 6 /r e1 ]
BAK: [ e3 W0 3 F ]
```

```
# 03: Starte mit Ping
CCA: [ W0 22 F ]
```

Anhang F

Programmbeispiele für Linearassembler

F.1 Beispiel: Lauflicht der Stellenanzeige

Dieses Programmbeispiel ist eine einfache Endlos-Schleife, welche über den `p1`-Befehl die Stellenanzeige um eine Stelle nach links schiebt. Anschließend wird eine Programmverzögerung ausgeführt, welche aktiv ca. 500 Millisekunden wartet.

```
1  ;
2  ; Lauflicht 1.0
3  ;
4  ; 02-03-2017,FQ: Erste Version
5  ;
6  ; Dieses Programm soll ein einfaches Lauflicht darstellen,
7  ; welches mit einer Verzögerung von ca. 500ms die
8  ; Stellenanzeige weiter schaltet.
9
10 loop:
11         pulse p1
12         wait 2700 ; ~500ms
13         jmp loop
```

F.2 Beispiel: Eingabe einer Ziffer

In diesem Beispiel wird ein Programm implementiert, welches bei Tastendruck den Zahlenwert der gedrückten Ziffer bestimmt und im Akkumulator ablegt. Zudem wird beim Loslassen der Taste die Stellenanzeige um eine Stelle weiter geschaltet.

```
1  ;
2  ; Key Reader 1.0
3  ;
4  ; 30-05-2017,FQ: Erste Version
5  ;
6  ; Liest bei Tastendruck den Zustand
7  ; einer Zahlentaste in den Akkumulator
8  ; und schiebt die Stellenanzeige eins
9  ; weiter.
10
11 init:
12         ; Aktiviere die Stellenanzeige für Stelle 1
13         pulse p1
14 loop_wait:
15
16         ; Prüfe, ob Taste gedrückt wurde
17         test 1
18         jpc shift
19
20         jmp loop_wait
21
22 shift:
23         ; Lösche und schiebe 4 Bit in den Akkumulator
24         clr
25         test 3
26         shl
27         test 4
28         shl
29         test 5
30         shl
31         test 6
32         shl
33
34         ; Warte auf Loslassen der Taste
35 loop_release:
36         test 1
37         jpc loop_release
38
39         ; Schalte Stellenanzeige eine Stelle weiter
40         pulse p1
41         jmp loop_wait
```

Anhang G

Tastaturmatrix

Das Einlesen einer Tastatur-Eingabe erfolgt über die Befehle `T1` bis `T6`. Hierbei wird der Zustand der Tastaturmatrix abgefragt.

Falls eine Taste gedrückt wurde, ist die Leitung K1 gesetzt. Falls K1 nicht gesetzt ist und mindestens eine andere Leitung aktiv ist, wurde eine Änderung am Schieberegler vorgenommen.

Eine vollständige Dokumentation der Tastaturmatrix sowie eine Tabelle befinden sich im Wartungshandbuch „Serviceanleitung“ ab Seite 87. [1]

Anhang H

Druckersteuerung

Die Druckersteuerung erfolgt über die Ausgabe-Befehle `e2`, `e3` sowie `p1`. Zudem werden anscheinend noch weitere Signale über den Zustand des Flip-Flips B ausgegeben.

Die Dokumentation zu der Druckerplatine befindet sich ebenfalls in der Serviceanleitung ab Seite 41. [1]

Anhang I

Beispielübertragung des Ladegeräts

Dieses Beispiel zeigt, wie eine Übertragung eines R-Speichers mit dem Ladeschaltungsprotokoll geschieht.

Hierbei wird folgendes Programm als Übertragungsbeispiel verwendet:

```
# Verwendet wird Direkt-Assembler
CCA: [W1 16 /r e1 ]
BAK: [W1 0 W0 5 T2 NUL F ]
```

Die Übertragung sieht dann wie folgt aus. Hierbei leitet → eine Übertragung von Host zu Gerät ein, ← eine Übertragung von Gerät zu Host. ↓ stellt ein Line Feed dar.

```
→ 'N↓'
← 'O'
→ 'S 00 0030↓'
← 'O'
→ 'S 02 0031↓'
← 'O'
→ 'S 03 0005↓'
← 'O'
→ 'S 04 0010↓'
← 'O'
→ 'S 06 0025↓'
← 'O'
→ 'S 13 0030↓'
← 'O'
→ 'S 14 0020↓'
← 'O'
→ 'S 15 0017↓'
← 'O'
→ 'S 16 0003↓'
← 'O'
→ 'T↓'
← 'O'
```

Anhang J

Ultraschall-Laufzeitspeicher

Die Combitron verwendet einen Laufzeitspeicher auf Ultraschall-Basis, welcher Bits durch eine Schallwelle in einem Metalldraht speichert. Diese Wellen werden durch eine Piezoelement am einen Ende des Drahtes emittiert, wandern nun mit Schallgeschwindigkeit durch den Draht und werden am anderen Ende des Drahtes durch ein weiteres Piezoelement wieder in elektrische Signale umgewandelt. Durch die hierbei entstehende Verzögerung des Signals wird der Speicher gebildet:
Mehrere hintereinander abgebildete Impulse werden für die Traversaldauer im Draht gespeichert.

Persistenz in einem Laufzeitspeicher wird erreicht, in dem der Ausgang des Speichers wieder mit dem Eingang verbunden wird. Hierbei kann eine Logik die Daten verarbeiten und neue Daten in den Speicher einspeisen.

Literaturverzeichnis

[1] DIEHL Rechenmaschinen: *Serviceanleitung*

[2] DIEHL Rechenmaschinen: *Wartungshandbuch.* `ftp://ftp.informatik.uni-stuttgart.de/pub/cm/diehl/Diehl.pdf`

[3] Frankel, Stanley: *General Purpose Digital Computer.* Oktober 1968

[4] Frankel, Stanley: *Recirculating Memory Timing.* Juni 1970

[5] ISO/IEC: *ISO/IEC 14977*

[6] Krause, Klemens: *LGP-30.* `http://computermuseum.informatik.uni-stuttgart.de/dev/lgp30/`. Version: Mai 2017

[7] Microsoft: *Sprachelemente für reguläre Ausdrücke – Kurzübersicht.* `https://msdn.microsoft.com/de-de/library/az24scfc(v=vs.110).aspx`. Version: Mai 2017

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Stuttgart, den 8. Juni 2017