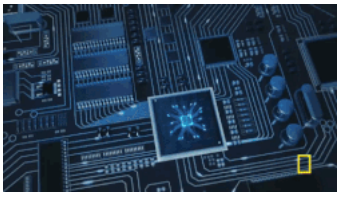
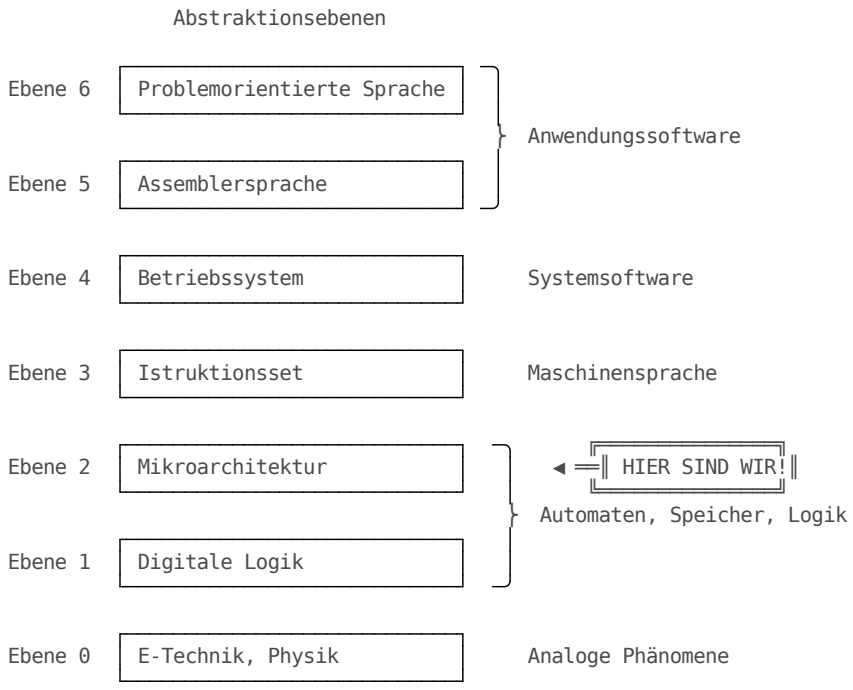
Modell CPU

Parameter	Kursinformationen
Veranstaltung:	`Digitale Systeme / Eingebettete Systeme`
Semester:	`Wintersemester 2025/26`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Modell-CPU-Implementierung, Instruktionszyklus, praktische CPU-Simulation`
Link auf GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_EingebetteteSysteme/blob/master/11_Modell_CPU.md
Autoren:	Sebastian Zug & André Dietrich & Fabian Bär

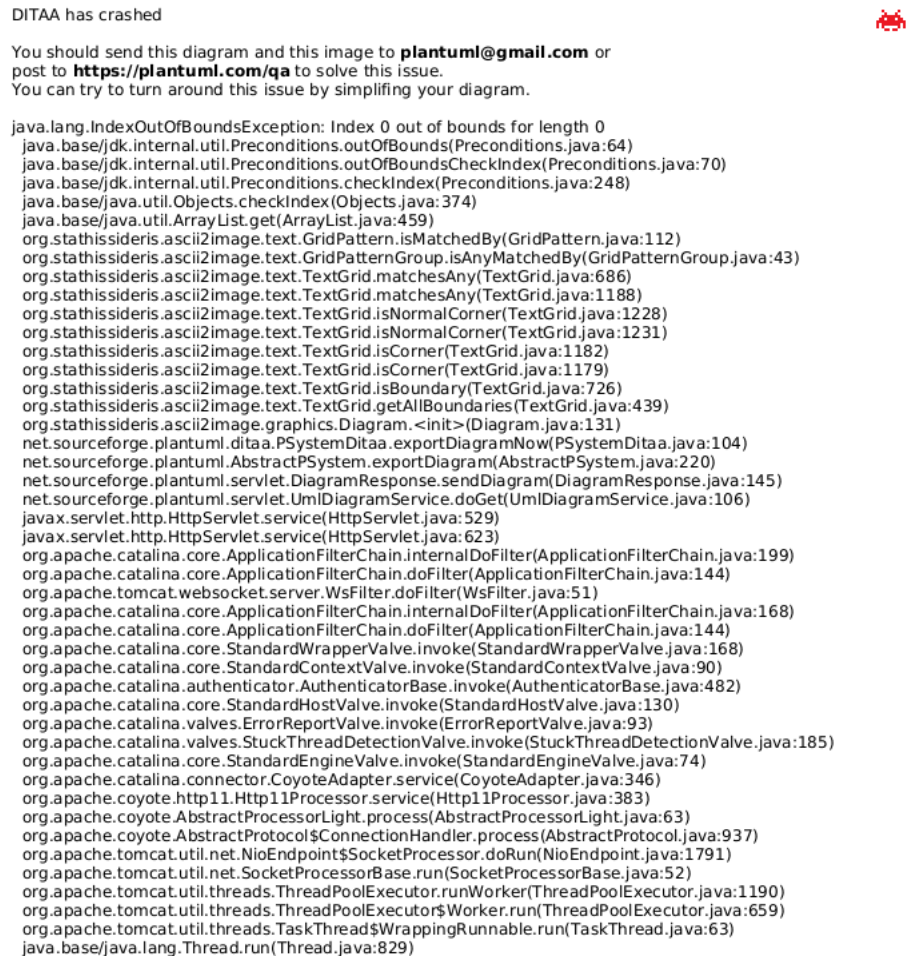


Fragen an die Veranstaltung

- Welche Funktionalität sollte eine ALU bereitstellen?
- Welcher Trade-Off ist mit der Entscheidung über die Wortbreite einer CPU und der Breite des möglichen OP-Codes verbunden?
- Welche Register gehören zum Programmiermodell eines Rechners?
- Welche Register sind für das Laden der Programmbefehle erforderlich und wie arbeiten sie zusammen?
- Erklären Sie die Vorgänge in der Fetch- und Execute-Phase der Befehlsabarbeitung.
- Wie funktioniert die Interrupt-Behandlung in einer CPU?
- Was versteht man unter dem Begriff "Instruktionsset-Architektur" (ISA)?
- Welche Adressierungsarten kennen Sie und wofür werden sie verwendet?



Ausgangspunkt



Der Prozessor ist die Einheit eines Rechners, die Daten nach einem vom Anwender spezifizierten Programm manipuliert. Das Programm wird als eine Folge von Anweisungen oder Befehlen für die CPU formuliert. Die CPU besteht daher:

1. aus einer Komponente, welche die Befehle liest, interpretiert und die korrekte Folge der Befehlsabarbeitung einhält - das Steuerwerk
2. aus einer Komponente, welche die Operationen auf den Daten, die durch die Befehle spezifiziert werden, ausführt

Das Steuerwerk besteht im wesentlichen aus einem Befehlsdecoder und einer Ablaufsteuerung. Befehle werden aus dem Befehlsspeicher gelesen und dekodiert.

Befehl	**Codierung**	$F_0 - F_2$	Z	$S_0 - S_1$
ADD_B	0110	011	0	11

Der Programmzähler berechnet welcher Befehl im nächsten Schritt bearbeitet werden soll. Die Ablaufsteuerung erzeugt, entsprechend dem auszuführenden Befehl, die Steuer- oder Kontrollsignale, um Daten zu lesen, im Datenpfad zu bearbeiten und die Resultate zu speichern.

Bei der Bearbeitung der Daten werden vom Datenpfad Statussignale generiert, die wiederum den Ablauf des Programms beeinflussen können, wie z.B. eine Programmverzweigung, die abhängig davon ist, ob das Resultat einer Berechnung größer oder kleiner einem bestimmten Wert ist.

Der Datenpfad besteht aus der ALU, die im einfachsten Falle arithmetische und logische Grundoperationen ausführen kann und einem Registersatz, in dem Daten bei der Verarbeitung zwischengespeichert werden.

Auf dieser Basis wollen wir nun einen Modellrechner entwerfen, an dem die zentralen Komponenten eingehender Untersucht werden können. Wo fangen wir an?

Wortformate

Um einen konkreten Maschinenbefehlssatz festlegen zu können, müssen die Randbedingungen unseres Prozessors betrachtet werden. Ein entscheidender Parameter ist die Wortbreite der Speicherschnittstelle. Die Wortbreite der Speicherschnittstelle bestimmt, wie viele Bits in einem Speicherzugriff zwischen Prozessor und Speicher transferiert werden können. Der intuitive Ansatz, diese Schnittstelle möglichst breit zu machen, hat weitreichende Auswirkungen auf die Rechnerarchitektur insgesamt.

Was nützt es, wenn wir zwar die Daten schnell aus dem Speicher lesen können, diese aber nur mit der halben Breite verarbeiten? Die Wortbreite des Speicher und der Breite der Verarbeitungseinheiten des Prozessors müssen abgestimmt sein.

Da Befehle und Daten **in unserem Modellrechner** in ein- und demselben Schreib/Lesespeicher stehen, muss die Wortbreite der Speicherschnittstelle auch für das Befehlswort berücksichtigt werden. Für unseren Modellrechner wählen wir eine Wortbreite von 16 Bit.

Welche Befehle soll unser Modellrechner nativ umsetzen können? Wir gehen von einem einfachen System aus, das arithmetisch/logische Operationen umsetzt, Sprünge realisiert und Eingaben von einem Schalterfeld lesen kann.

Opcode	Mnemonic	Opperand	Beschreibung
0000	HLT		HLT hält den Computer an. Kann auch manuell eingegeben werden. Nach HLT kann der Rechner nur manuell gestartet werden. Fortsetzung beim nächsten Befehl.
0001	JMA	addr	(Jump on Minus) Bedingter Sprung. Wenn das Ergebnis einer Berechnung negativ ist. Die Adresse "addr" wird in den Befehlszähler geladen. Der nächste Befehl wird von "addr" genommen.
0010	JMP	addr	Unbedingter Sprung. Die Adresse "addr" wird in den Befehlszähler geladen. Der nächste Befehl wird von "addr" genommen.
0011	JSR	addr	Unterprogrammsprung. Die Adresse, die im Befehlszähler enthalten ist, wird ins Register A geladen. Die Adresse "addr" wird in den Befehlszähler geladen. Der nächste Befehl wird von "addr" geladen.
0100	SWR		(Copy Switch Register to A) Der Zustand der Schalter wird in das Register A geladen.
0101	RAL		(Rotate A Left) Zyklischer Links-Shift. Der Inhalt von Register A wird um 1 Stelle nach links rotiert. Ringshift : Bit A0 ← Bit A15
0110	INP		INPUT
0111	OUT		OUTPUT
1000	NOT		Komplementbildung des Inhalts von Register A.
1001	LDA	addr	Laden des Registers A von Speicheradresse addr.
1010	STA	addr	Speichern des Registerinhalts auf Speicheradresse addr .
1011	ADD	addr	Inhalt des Speicherwortes mit Adresse adr wird auf den Inhalt des Registers A addiert. Der ursprünglich Inhalt von A wird mit dem Ergebnis überschrieben.
1100	XOR	addr	Inhalt des Speicherwortes mit Adresse adr wird mit dem Inhalt des Registers A durch XOR verknüpft. Der ursprüngliche Inhalt von A wird mit dem Ergebnis überschrieben.
1101	AND	addr	Inhalt des Speicherwortes mit Adresse adr wird mit dem Inhalt des Registers a durch log. UND verknüpft. Der ursprünglich Inhalt von A wird mit dem Ergebnis überschrieben.
1110	IOR	addr	Inhalt des Speicherwortes mit Adresse adr wird mit dem Inhalt des Registers A durch log. ODER verknüpft. Der ursprünglich Inhalt von A wird mit dem Ergebnis überschrieben.
1111	NOP		(No Operation) Der Befehlszähler wird um 1 erhöht.

Nun muss die Aufteilung ein Befehlswort in ein Feld für den Operationscode und ein Feld zur Adressierung des Operanden aufgeteilt werden. Mit 16 Bit für das Operandenfeld können insgesamt 64 k Worte adressiert werden. Jedes Bit, das wir für die Codierung der Befehle brauchen, halbiert diesen Wert. Als einen Kompromiss zwischen der Größe des Speicheradressraums und einem Befehlssatz, der die wesentlichen arithmetisch/logischen Operationen enthält, nehmen wir eine Aufteilung vor in ein:

Befehlsformat:

Datenformat:

Negative Zahlen werden im Zweierkomplement dargestellt.

Wie sieht ein Programm dann aus?

Aufgabe: Implementieren Sie einen variablen Rechtsshift - der Nutzer gibt bis zu 15 Shiftschritte an, die ausgeführt werden sollen.**

Herausforderung: Wir haben nur einen 1-Schritt zyklischen Linksschift auf dem Modellrechner vorgesehen. Die Lösung muss in Software realisiert werden.

Das eigentliche Programm belegt 16 aufeinander folgende Speicherplätze zwischen 0001000 und 00011111. Die nächste Spalte gibt die binäre Repräsentation des Programms an. Die mnemotechnische Darstellung und ein Kommentarfeld folgen.

Unser Rechner wird nur ein echtes Register A haben. Entsprechend müssen wir die Variablen im Speicher ablegen und von dort wieder laden. Das Programm besteht dann aus zwei Teilen

Adresse	Speicherinhalt	Programmzeilen	Kommentar
00010000	1001 10000001	`LDA 10000001`	Lade Anzahl der Rechts-Shifts aus dem Speicher 10000001 in Register a
00010001	1000 XXXXXXXXX	`NOT`	Komplementieren von a
00010010	1011 10000010	`ADD 10000010`	2-Komplement
00010011	1011 10000100	`ADD 10000100`	Berechne : 16 + (-AR)
00010100	1010 10000011	`STA 10000011`	Speichere Anzahl der Links-Shifts
00010101	1001 10000000	`NEXT: LDA 10000000`	Lade Datum
00010110	0101 XXXXXXXXX	`RAL`	Links-Shift
00010111	1010 10000000	`STA 10000000`	Speichere Datum
00011000	1001 10000010	`LDA 10000010`	Lade Konstante “1“
00011001	1000 XXXXXXXXX	`NOT`	Komplementieren von a
00011010	1011 10000010	`ADD 10000010`	2-Komplement
00011011	1011 10000011	`ADD 10000011`	Decrementiere Anzahl der L-Shifts
00011100	1010 10000011	`STA 10000011`	Speichere verbleibende Anzahl der L-Shifts
00011101	0001 00011111	`JMA DONE`	Bed. Sprung, wenn alle L-Shifts ausgeführt wurden
00011110	0010 00010101	`JMP NEXT`	Unbed. Sprung zum Anfang der Schleife
00011111	0000 XXXXXXXXX	`DONE: HLT`	

Der entsprechende Speicherauszug dazu:

Adresse	Daten	Bemerkung
10000000	0000000000001101	D: Zu shiftender Wert
10000001	0000000000000101	AR: Anzahl der Rechts-Shifts
10000010	0000000000000001	“1“: Konstante “1“
10000011	0000000000000000	TMP: Temp. Anzahl der Links-Shifts
10000100	0000000000010000	“16“: Konstante “16“

Da wir als Befehl nur den zyklischen Linksshift (`RAL`) um 1 Stelle zur Vefügung haben, müssen wir den zyklischen Rechtsshift um eine beliebige Anzahl von Stellen durch ein Programm simulieren. Im Modellrechner kann in einem Befehl kein Direktoperand angegeben werden. Deshalb werden alle Konstanten und Variablen im Speicher ablegen.

Und wie kommen wir dahin? In den seltensten Fällen wollen wir einen Rechner in der Assemblersprache programmieren. Deshalb nutzen wir Compiler, um ausgehend von einer Hochsprache (C, C++, ...) die für den Prozessor verständlichen Sprachkonstrukte zu generieren.

Aufgabe: Setzen Sie folgendes Fragment eines C-Programmes in einem Assembler-Code für unseren Modellrechner um.

```
#include<stdlib.h>

var = -10;

if (var < 0){
  var++;
  digitalWrite(); // Ausgabe des Wertes von Register A auf dem Ausgabedisplay SWR
    Befehl
}
else
{
  stop();         // Ausführen des HLT Befehls
}
```

Adresse	Speicherinhalt	Programmzeilen	Kommentar
00010000	1001 10000001	`LDA 10000001`	Lade den Wert von var aus dem Speicher 10000001 in Register A
00010001	0001 10000010	`Loop JMA 00010011`	Wenn ein negativer Wert vorliegt, springe zu AddrIf
00010010	0000 – – – – – – – –	`HLT`	Stoppe die Abarbeitung
00010011	1011 10000010	`AddrIf ADD 10000010`	Inkrementiere A
00010100	0100 – – – – – – – –	`SWR`	Zeige A auf dem LED Streifen an
00011100	1010 10000011	`STA 10000001`	Speichere A
00011100	0010 00010001	`JMP 00010001`	Springe zu Loop A

Adresse	Speicherinhalt	Bedeutung
00010000	100110000001	
00010001	000110000010	
00010010	000000000000	
00010011	101110000010	
00010100	010000000000	
00011100	101010000011	
00011100	001000010001	
...		
10000001	111111110110	int var = -10
10000010	000000000001	Konstante 1

Achtung! Unser Modellrechner hat bestenfalls didaktische Qualitäten. Vergleiche den Befehlssatz eines realen Systems (ATmega Handbuch oder Intel Instruction Set)

Entsprechend ändert sich auch das Ergebnis des Compiliervorganges in Abhängigkeit von der verwendeten Architektur! (vgl godbolt.org)

Elemente des Modelle-Rechners

Die Elemente des Rechners lassen sich, wie bereits in der vergangen Vorlesung dargestellt, in 4 Kategorien einteilen - Speicher, Rechenwerk, Steuerwerk und Ein-Ausgabe.

Um Ihr Verständnis zu fördern, hat ein Kommilitone - Fabian Bär - von Ihnen eine kleine Simulation vorbereitet, die das Zusammenspiel der Komponenten illustriert. Diese ist unter

Simulation

zu finden.

Speicherbezogene Komponente

In einem Speicherzyklus muss der Prozessor zunächst eine Adresse liefern, die während des gesamten Speicherzyklus anliegen muss. Für einen Schreibzyklus betrifft dies auch das entsprechend abzulegende Datum. Bei einem Lesezyklus steht das gewünschte Wort erst mit einer gewissen Verzögerung an der Schnittstelle zur Verfügung. Da der Speicher sowohl zum Schreiben als auch zum Auslesen eines Wortes länger braucht als die Zeit, in der der Prozessor eine elementare Operation ausführen kann, sind zwei Pufferregister vorgesehen:

1. Das Speicher-Adress-Register (MAR : Memory Address Register), in das die Adresse zu Beginn des Speicherzyklus geschrieben wird. In unserem Fall ist das MAR 12 Bit breit.
2. Das Speicher-Puffer-Register (MBR : Memory Buffer Register). Bei einer Schreiboperation legt der Prozessor ein Datenwort hier ab, so dass es durch den (langsamen) Schreibvorgang im Speicher unter der Adresse abgespeichert wird, die im MAR spezifiziert ist. Beim Lesen stößt der Prozessor den Lesevorgang an und kann später das adressierte Wort aus dem MBR auslesen. Die Adresse ist bei uns 12 Bit breit. Wir können also 2^12 Adressen ansprechen, die jeweils 16 Bit Daten repräsentieren.

Durch MBR und MAR sind Prozessor und Speicher bezüglich ihrer Zykluszeiten weitgehend entkoppelt.

Link auf den Simulator

Datenpfadbezogene Komponente

Der Datenpfad besteht aus der ALU, dem allgemeinen Register A (Akkumulator), einem Hilfsregister Z und dem Eingang-Ausgang-Schalterregister SWR. Die zweistelligen arithmetischen und logischen Befehle haben alle die Form:

< OPCODE, addr >

Sie setzen voraus, daß der eine Operand in A steht, der zweite Operand muss aus dem Speicher von Adresse `addr` gelesen wird. Dabei gehen wir davon aus, dass der Operand im MBR zur Verfügung steht. Da die ALU rein kombinatorisch aufgebaut werden soll, müssen beide Operanden während der Verarbeitungszeit an den Eingängen anliegen. Das Ergebnis der Operation wird in A verfügbar gemacht. Damit das Ergebnis der Operation nicht einen der Operanden in A überschreibt, ist das Hilfsregister Z vorgesehen. Während der Befehlsausführung wird der Operand aus A nach Z transferiert, damit das Ergebnis in A gespeichert werden kann.

Link auf den Simulator

Steuerwerk

Die Kontrolleinheit besteht aus:

1. dem Programmzähler,
2. dem Befehls- oder Instruktionsregister,
3. dem RUN/HLT Flip-Flop,
4. dem State-Flip-Flop,
5. dem Automaten, der die der sequentiellen Kontrolle realisiert.

Der Prozessor liest eine neue Instruktution aus dem MBR in das Instruction Register. Die Control Unit interpretiert den Operationscode und startet die Ausführung, währenddessen wird der Programmzähler bereits auf die Adresse der Instruktion gesetzt, die als nächste ausgeführt werden soll. Über den Programmzähler wird also die Sequenzierung der Instruktionen bei der Abarbeitung eines Programms gesteuert. Das State FF übernimmt die unterschiedliche Abarbeitung von Befehlen, die in einem Zyklus und zwei Zyklen umgesetzt werden können (HLT vs ADD).

> **Merke:** Das Steuerwerk implementiert einen Automaten für die Instruktionsabarbeitung des Prozessors.

Und jetzt alles zusammen

1. Der Akkumulator (`Register A`). In dem Modellrechner ist der Akkumulator das einzige allgemeine Register. Alle arithmetischen und logischen Befehle arbeiten auf diesem Register.
2. Das Schalterregister (`Switch Register: SWR`) besteht aus 16 Schaltern, die z.B. an der Frontplatte eines Rechners angebracht sind. Sie können manuell gesetzt und vom Rechner abgefragt werden.
3. Der Programmzähler (`Programm Counter`) wird beim Unterprogrammsprung(JSR, Jump Subroutine) in den Akkumulator geladen, durch den Befehl RTS (Return from Subroutine) wird der Inhalt des Akkumulators in den Programmzähler geladen.
4. Das `Halt-Flip-Flop` wird durch den Befehl HLT gesetzt. Es kann nur manuell zurückgesetzt werden.

Link auf den Simulator

Beschreibung der prozessorinternen Vorgänge

Schritt	Bedeutung
1. Befehl holen	Befehl entsprechend der Adressvorgabe aus dem MAR aus dem Speicher lesen und in MBR ablegen
2. Befehl dekodieren	aktuellen Befehl aus MBR nach IR verschieben und dekodieren
3. ggf Operanden bereitstellen	Daten entsprechend dem weiterbewegten PC (MAR) lesen und im MBR ablegen
4. Befehl ausführen	Kontrolleinheit definiert die entsprechenden Steuerleitungen
5. ggf Ergebnis speichern	Sichere den Inhalt von A ins MBR, Manipuliere den Inhalt des MBR

Register-Transfer-Sprache

Zur Bearbeitung einer Instruktion, z.b. einem "ADD addr", braucht der Prozessor mehrere Schritte. Wie lässt sich aber der Daten- und Kontrollfluss zwischen den einzelnen Komponenten abbilden?

Eine Möglichkeit der Darstellung ist die Beschreibung des der Struktur des Prozessors und die Beschreibung von dessen Verhalten auf der Register-Transfer Ebene:

1. Strukturbeschreibung - umfasst z.B. die vorhandenen Register, die arithmetisch/logischen Einheiten und die dazugehörige Verbindungsstruktur
2. Verhaltensbeschreibung - beschreibt das Zusammenspiel der Komponenten bei der Erfüllung einer bestimmten Aufgabe. Die Zustandsdiagramme, die wir zur Beschreibung von Automaten eingeführt haben, sind eine Form der Verhaltensbeschreibung.

Die hier verwendete Register-Transfer Sprache wurde von T.C. Bartee, I.L. Lebow, I.S. Reed: *Theory and Design of Digital Machines* beschrieben

Grundelemente von RTL

Element	Darstellung	Bedeutung
Register	R_n	bezeichnet die Bitstellen $n, n-1, ..., 0$ des Registers R, eine Slicing wird über die Indizes beschrieben R_{5-11}
Transfer	$\leftarrow$	$A \leftarrow B$ bezeichnet den Transfer des Inhalts von Register B nach A
Speicher	$M[addr]$	bezeichnet den Inhalt der Speicherzelle mit der Adresse "addr"
Bedingungen	$B:$	bezeichnet den Inhalt der Speicherzelle mit d z.B. $R = 0 : A \leftarrow B$, oder $CP7 \cdot RAL : A \leftarrow Z$

Beispiele

Kategorie	Beispiel	Bedeutung
Zwingende Operationen	$a_{3-0} \leftarrow b_{7-4}$	Inhalt der Bits 4-7 werden aus Register b auf die Stellen 0-3 des Registers a übertragen.
	$a \leftarrow SUM(a, b)$	
	$a \leftarrow SUM(a, 1)$	
	$a \leftarrow M[576]$	Das Speicherwort an der Adresse 576 wird in Register a transferiert.
Bedingte Operationen	$R = 0 : a \leftarrow b$	Wenn die Bedingung R gleich Null gilt, wird der Inhalt des Registers b nach a übertragen.
	$C_n \cdot CLR : a \leftarrow 0, b \leftarrow 0, c \leftarrow 0$	Wenn der Takt C_n anliegt, un dder Befehl "HLT" wird der Run-Flip-Flop auf 0 gesetzt.
	$C_n \cdot JMP : PC \leftarrow IR_{11-0}$	Wenn der Takt C_n anliegt, und der Befehl "JMP" ausgeführt wird, erfolgt der Transfer des Inhaltes des Instruktionsregisters an den Programmzähler.

Abläufe der Befehlsabarbeitung

Einige Befehle, insbesondere die, welche keinen zweiten Operanden oder ALU-Aktivitäten benötigen, können vollständig in der IF-Phase abgearbeitet werden. Dies gilt für HLT, NOP, CSA, und die Sprungbefehle JMP, JMA und SRJ. Bei diesen Befehlen wird im letzten Prozessorzyklus (CP8) eine neue Adresse in das MAR geladen, und dadurch der neue Speicherzyklus vorbereitet.

Während bei HLT, NOP und CSA der Programmzähler einfach inkrementiert wird, muss bei den Sprungbefehlen die Zieladresse des Sprungs, die im Operandenfeld der Instruktion (IR_{11-0}) steht, aus dem IR in das MAR geladen werden.

Bei den Befehlen, zu deren Ausführung die EX-Phase benötigt wird, wird in CP8 das SF in den Zustand E (Execute) gesetzt. Wird ein zweiter Operand benötigt, wird das Operandenfeld der Instruktion, das die Adresse enthält, in das MAR geladen, um den neuen Speicherzyklus zu initiieren

In der EX-Phase werden die arithmetisch/logischen Operationen, sowie Speicherbefehle LOAD/STORE und Ein/Ausgabebefehle ausgeführt.

OPCode	0000	0001	0010	0011	0100	0101	0110
	`HLT`	`JMA`	`JMP`	`JSR`	`SWR`	`RAL`	`INP`
CP1							
CP2							
CP3							
CP4							
CP5							
CP6							
CP7	$RF \leftarrow H$	$A_{15} = 1 : PC \leftarrow IR_{11-0}$	$PC \leftarrow IR_{11-0}$	$A_{11-0} \leftarrow PC$	$A \leftarrow SWR$	$Z \leftarrow A$	
CP8	$MAR \leftarrow PC$	$MAR \leftarrow PC$	$MAR \leftarrow PC$	$PC \leftarrow IR_{11-0}, MAR \leftarrow PC$	$MAR \leftarrow PC$	$SF \leftarrow E$	
$CP1$							
$CP2$						$A \leftarrow Z^*$	
$CP3$							
$CP4$							
$CP5$							
$CP6$							
$CP7$							
$CP8$						$MAR \leftarrow PC, SF \leftarrow F$	

Der folgende Automat bildet die Abarbeitung der Instruktionen `HLT`, `JMP`, `JMA` und `JSR` in einem Automaten ab.

Umsetzung als Schaltnetz / Schaltwerk

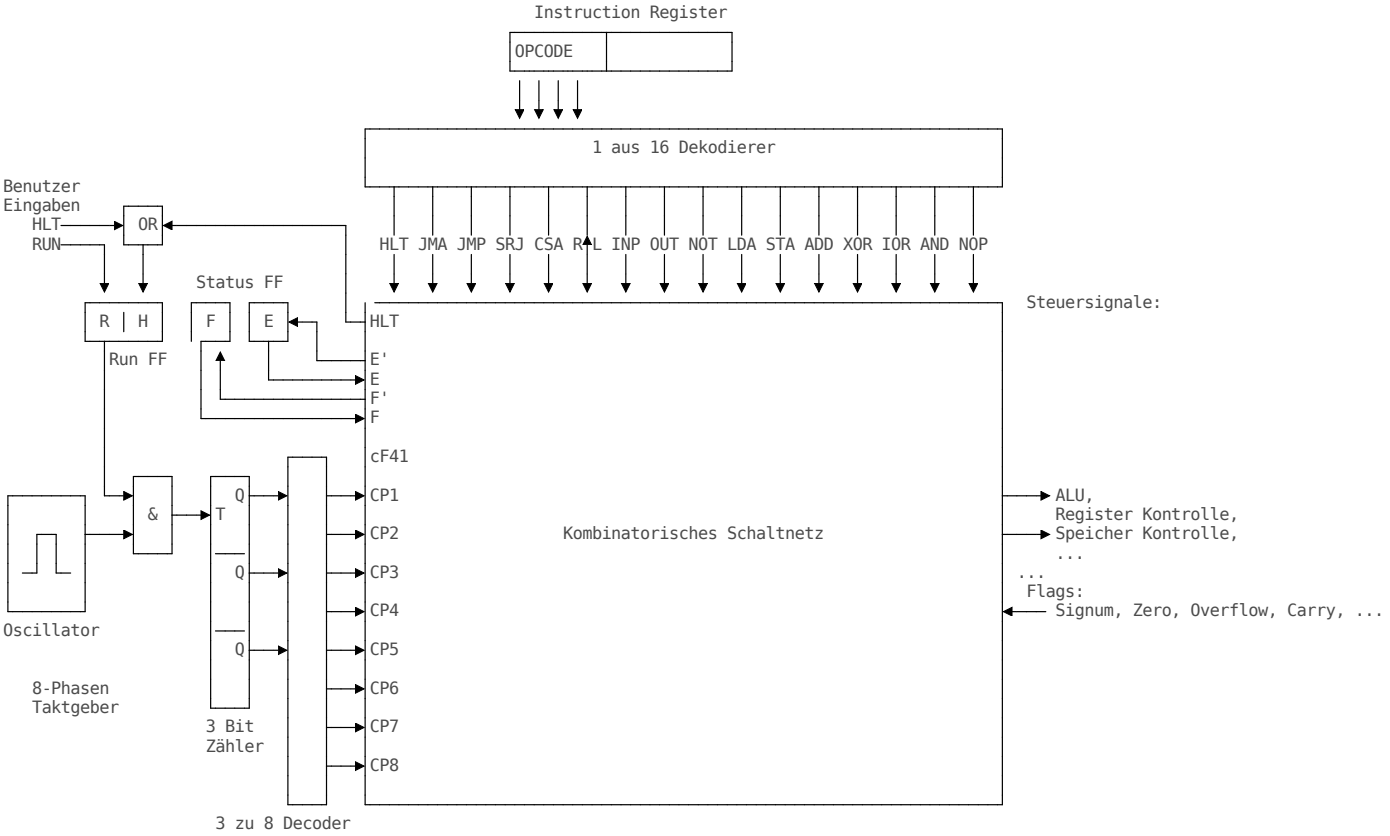
Taktvorgabe

Die Kontrolleinheit benötigt einen Trigger für die Abarbeitung der Instruktionen. Grundlage für den Zeitablauf einer Maschinenoperation ist der Speicherzyklus und der Prozessortakt. Im Modellrechner wird der Ablauf in Phasen zu 8 Taktintervalle unterteilt. In jedem dieser Taktintervalle kann eine Instruktion vollständig oder zur Hälfte ausgeführt werden.

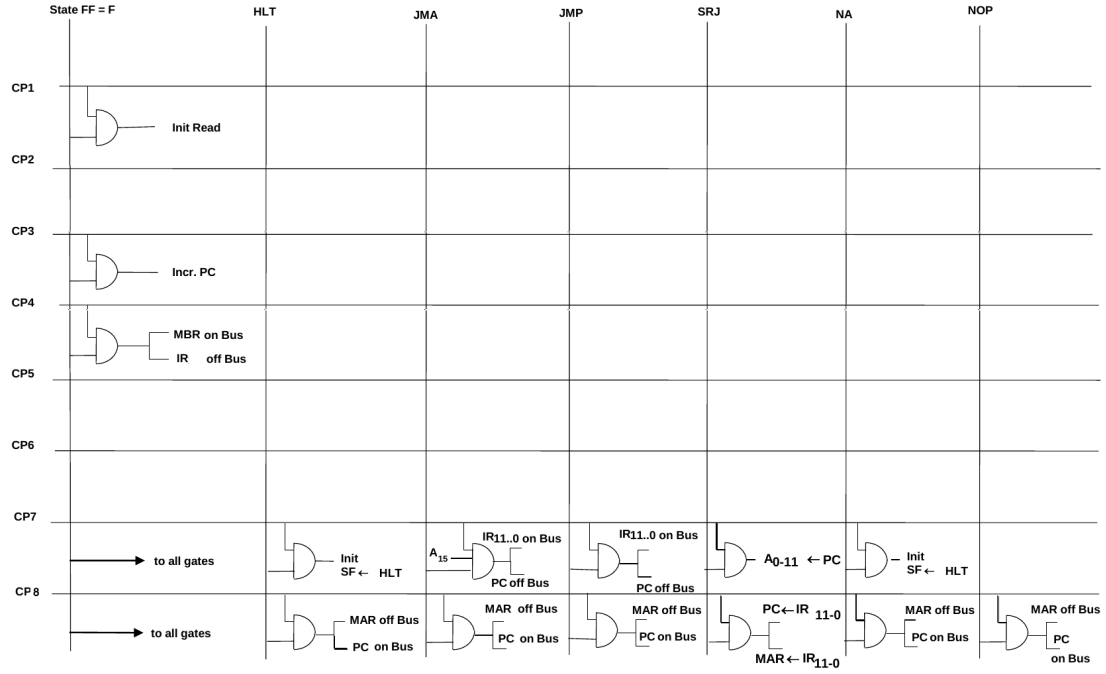
Integrierung der Taktvorgabe in Steuerwerk

Die folgende Abbildung gibt die Kontrolleinheit im schematische Aufbau wieder. Die wesentlichen Komponenten sind:

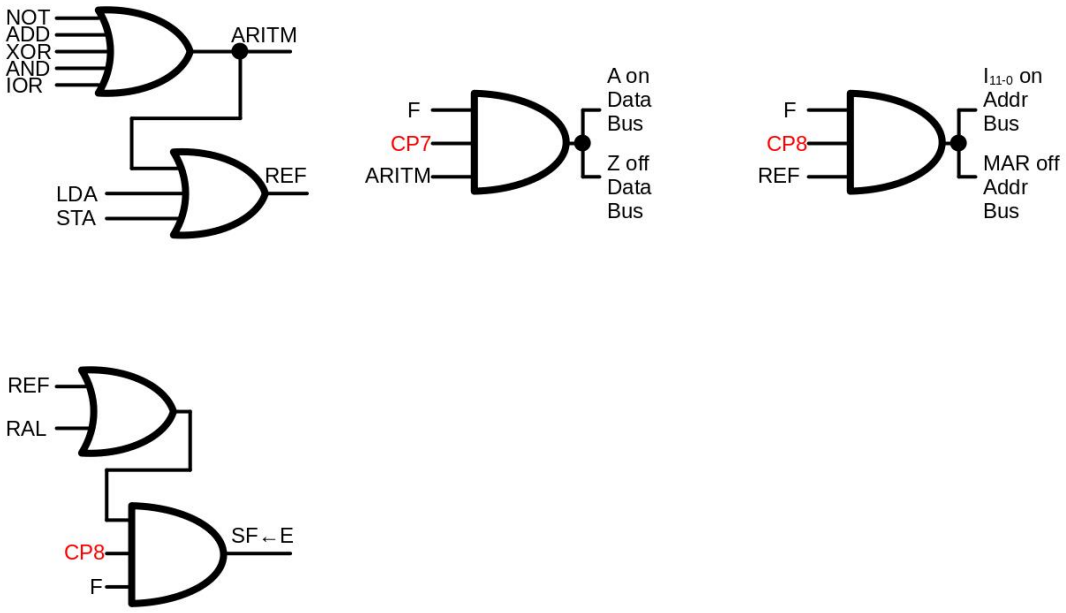
1. der 8-Phasen Taktgenerator
2. der Instruktionsdekoder
3. die Status-Flip-Flops
4. das kombinatorische Schaltnetz zu Erzeugung der Steuersignale.



Realsierung der kombinatorischen Logik in der Kontrolleinheit



`Bus off` == Lesender Zugriff auf den Bus / `Bus on` == schreibender Zugriff



Beschränkungen der aktuellen Lösung

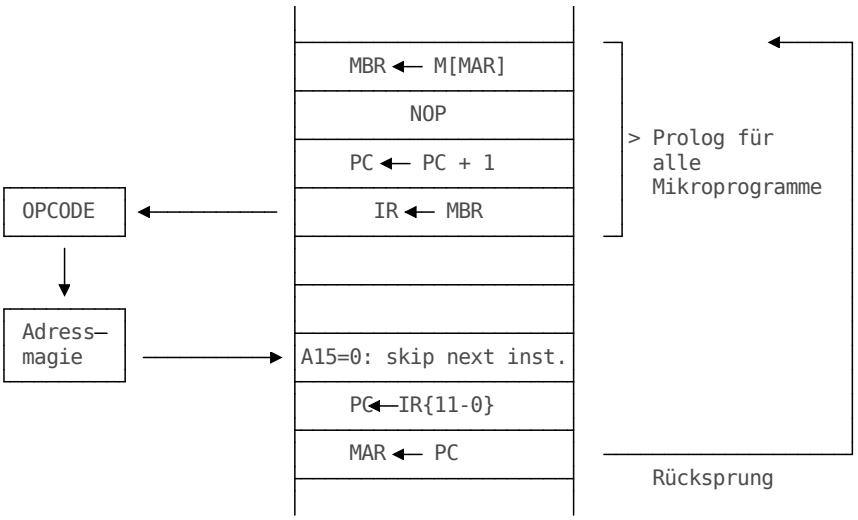
Aspekt	Kombinatorische Logik
Grundlegende Repräsentation	Endlicher Automat
Fortschaltung der Kontrolle	Expliziter Folgezustand
Logische Repräsentation	Boolesche Gleichungen
Implementierungstechnik	Gatter, Programmierbare Logikbausteine

Aspekt	Kombinatorische Logik	Mikroprogramm
Grundlegende Repräsentation	Endlicher Automat	Programm
Fortschaltung der Kontrolle	Expliziter Folgezustand	Programmzähler
Logische Repräsentation	Boolesche Gleichungen	Wahrheitstabelle
Implementierungstechnik	Gatter, Programmierbare Logikbausteine	R/W-Speicher, ROM

Mikroprogramme

Motivation

Realisierung am Beispiel der Implementierung eines `JMA`-Befehls als Mikroprogramm



- Fortlaufende Abarbeitung der Mikroprogramminstruktionen
- Die externe Quelle der Bedingungen auszuwählen.
- Einen Indexmechanismus einzuführen, der es erlaubt, eine Adresse aus dem Adressfeld der MIP und externen Adressinformationen zu bilden (bedingte Adressfortschaltung)

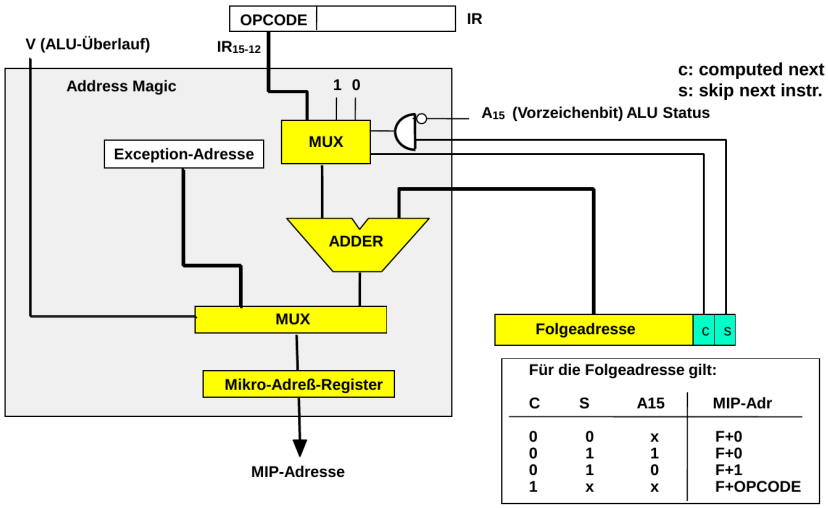
Umsetzung

Das Mikroprogrammwortes umfasst alle Steuersignale für Register, ALU, Speicher, usw., die jeweils durch ein Bit in der repräsentiert sind. Eine Steuerleitung wird aktiviert, wenn dieses Bit auf "1" gesetzt ist. Wird das Bit auf "0" gesetzt, wird die Steuerleitung deaktiviert. Der erste Teil des Mikroprogrammwortes wird entsprechend als Steuerwort bezeichnet.

Das Mikroprogrammwort enthält außer dem Steuerteil noch den Adressteil. Der Adressteil eines Mikroprogrammwortes enthält direkt die Adresse des nächsten Mikroprogrammwortes. Allerdings muss der Modellrechner auch auf Statusinformation der ALU und auf den Inhalt des OPCODE-Feldes des IR reagieren, so dass der Ablauf des Mikroprogramms nicht immer gleich ist.

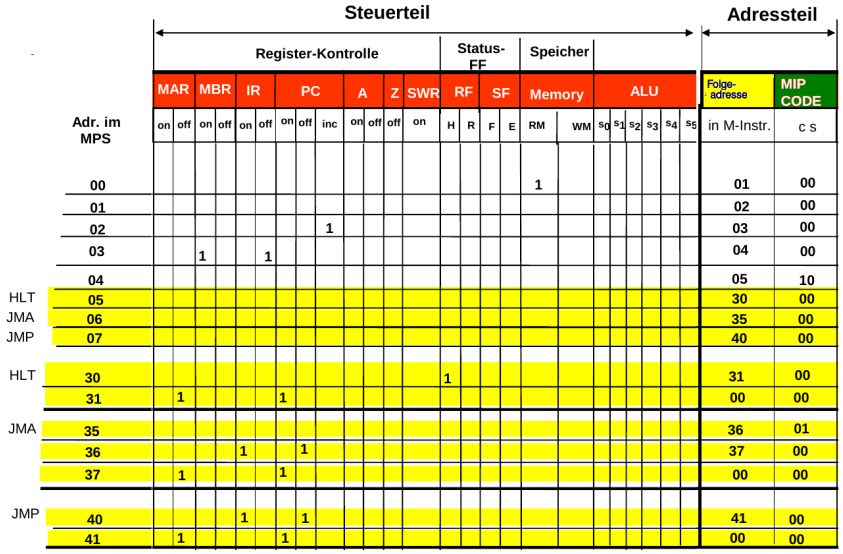
Um die Sprünge zu realisieren werden dem Mikroprogrammwort neben der Folgeadresse zwei Flags beigefügt:

- "computed next" c - Die Folgeadresse wird aus der Adresse im Adressfeld der MIP und dem OPCODE-Feld des IR_{15-12} gebildet. Dabei wird das OPCODE-Feld auf die Adresse im Adressfeld des MIP-Wortes addiert.
- skip next on condition s - Berücksichtigt wird das Vorzeichenbit des Akkumulators $A_{15} = 0$. Trifft die Bedingung zu, d.h. $A_{15} = 0$, wird die nächste Instruktion übersprungen (Folgeadresse +1). Trifft die Bedingung nicht zu, wird die nächste Instruktion von der spezifizierten Folgeadresse genommen.

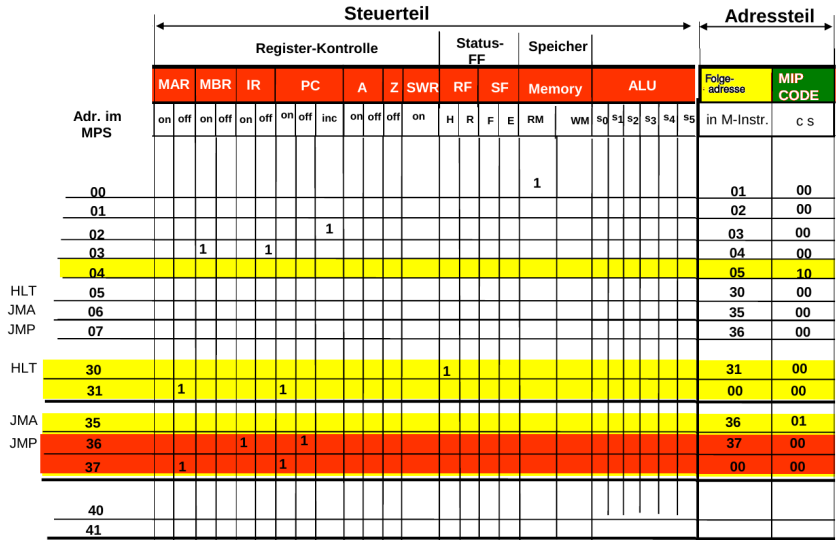


Layout eines Programmes im Mikroprogammspeicher

Und wie sieht das nun konkret aus?

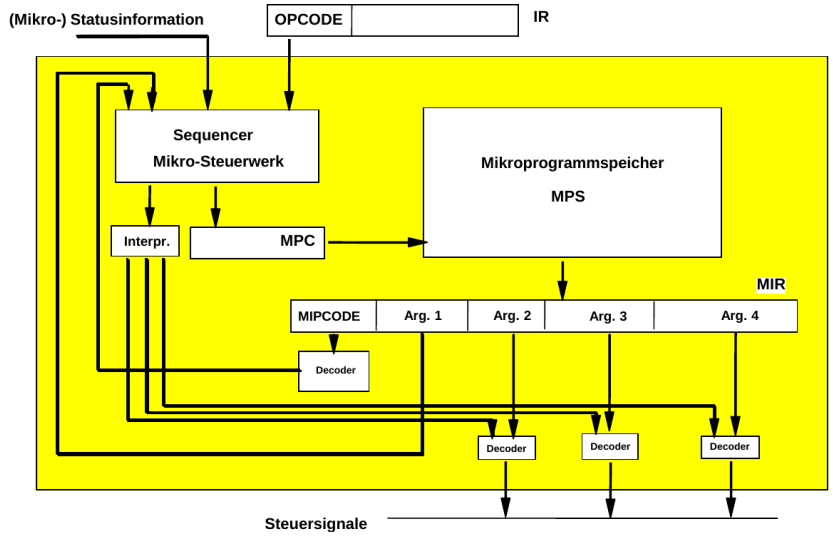


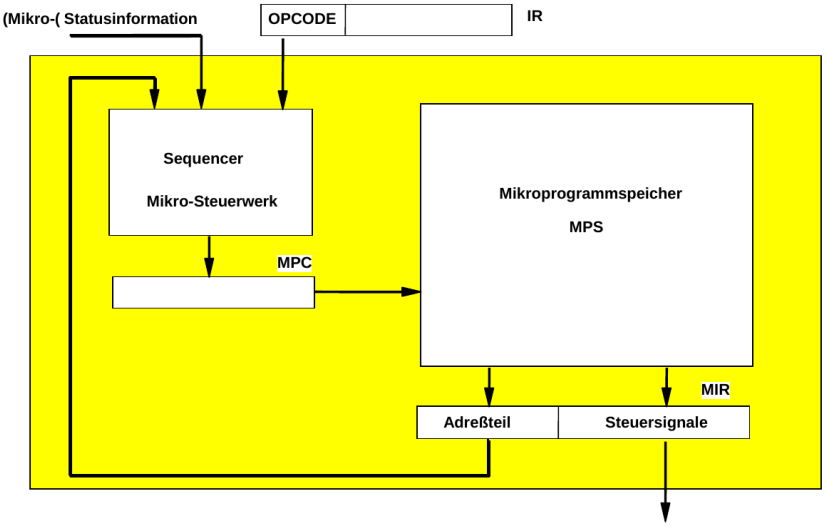
Frage: Welche Vereinfachungsmöglichkeiten sehen Sie?



Horizontale vs. vertikale Mikroprogrammierung

- *Wie kann der Micro-Code optimiert werden?* Zielkonflikt: Hardwarekosten gegen Geschwindigkeit und Flexibiltät
- *Wie kann des Micro-Wort verkürzt und der M-Speicher vekleinert werden?* Zielkonflikt: Wortlänge gegen Dekodierungsaufwand und Flexibilität
- *Wie erreicht man größtmöglichen Komfort bei der Programmierung?* Zielkonflikt: Komfort gegen Effizienz und Flexibilität





	Horizontale Mikroprogrammierung	Vertikale Mikroprogrammierung
Idee	Jedes Bit des Mikroinstruktionswortes steuert direkt eine Kontroll-Leitung	Mikroinstruktionswort ist in Felder aufgeteilt, die codierte Steuerinformation enthalten
Vorteil	maximale Flexibilität, Geschwindigkeit	Kurzes Mikroinstruktionswort, kleiner Mikroprogrammspeicher , übersichtlicher Mikroprogramme
Nachteil	sehr langes Mikroinstruktionswort, großer Mikroprogrammspeicher	Zusätzliche Dekodierungshardware, Flexibilität, Geschwindigkeit

Diskussion

Ausgangspunkt der Mikroprogrammierung:

1. Mächtige Maschinenbefehle unterstützen den Programmierer.
2. Leichte Änderbarkeit der Maschinenbefehle ist notwendig, um die Fortentwicklung des Befehlssatzes und Spezialanwendungen zu unterstützen.
3. Mikroprogramme können leichter entworfen werden als Logik.
4. Mikroprogramme können komplexe Operationen effizienter bereitstellen als eine Sequenz von Maschineninstruktionen, da sie die Parallelität der Hardware ausnutzen können.
5. Mikroprogramme können komplexe Operationen effizienter bereitstellen als eine Sequenz von Maschineninstruktionen, da Mikroprogrammspeicher schneller ist als Speicher für Maschinenprogramme.

Hausaufgaben

1. Entwerfen Sie die Implementierung einer Subtraktionsinstruktion für den Modellrechner.
2. Schreiben Sie ein Assemblerprogramm für den Modellrechner, dass im Speicher die Zahlen von 0-9 ablegt und darüber eine Summe bildet.