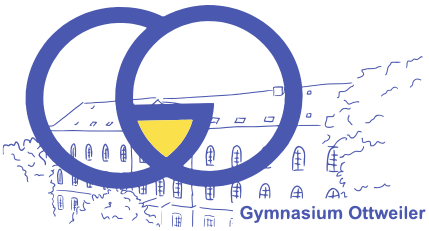
Kapitel 6

Funktionsweise von Computersystemen



Skript Informatik · Grundkurs

Bis hierher war der Rechner eine Blackbox: Sie schreiben `x := a + b`, und irgendwie kommt die Summe heraus. Dieses Kapitel öffnet die Box. Am Ende wissen Sie, in wie viele Einzelschritte eine einzige Zuweisung zerfällt, warum ein Programm im selben Speicher steht wie seine Daten und was genau passiert, wenn ein Prozessor „einen Befehl ausführt".

Der Weg dorthin führt über eine Maschine, die klein genug ist, um sie vollständig zu verstehen: die Registermaschine mit acht Befehlsarten, zwei Speichern und drei Registern. Wer darauf ein Programm zum Laufen bringt, hat den Kern jedes Prozessors verstanden — moderne CPUs machen dasselbe, nur milliardenfach pro Sekunde und mit erheblich mehr Beiwerk.

Lernziele

Am Ende dieses Kapitels können Sie

- die Funktionseinheiten eines Rechners benennen und ihr Zusammenspiel beschreiben,
- die fünf Prinzipien der von-Neumann-Architektur erläutern,
- Aufgabe und Inhalt von Akkumulator, Befehlsregister und Befehlszähler angeben,
- den fünfschrittigen Befehlszyklus an einem konkreten Befehl durchspielen,
- Programme in der Sprache der Registermaschine schreiben, von Hand ablaufen lassen und testen,
- einen Simulator für die Registermaschine programmieren,
- erklären, warum moderne Prozessoren vom reinen von-Neumann-Modell abweichen.

Gymnasium Ottweiler

6.1 Die von-Neumann-Architektur

ZUSATZ

> ZUSATZ
>
> **Dieses Kapitel steht in keinem der beiden Lehrpläne von 2019** — weder im Grundkurs noch im Leistungskurs. Es ist damit nicht verbindlich und kein Gegenstand einer Kursarbeit, solange es nicht ausdrücklich zum Unterrichtsgegenstand erklärt wird. Aufgenommen ist es, weil die Registermaschine zeigt, was hinter jeder Zuweisung und jeder Schleife steckt — und weil der Befehlszyklus die Brücke zwischen Kapitel 1 und der wirklichen Maschine schlägt.

Als der **ENIAC** 1946 an der Universität von Pennsylvania der Öffentlichkeit vorgestellt wurde — 17 468 Elektronenröhren und rund 1500 Relais auf 27 Tonnen Masse — konnte er wie seine Vorgänger Z3 und Mark I nur rechnen. Für eine freie Programmierung war er nicht geeignet: Ein neues Problem bedeutete, Kabel umzustecken und Schalter neu zu setzen, was Tage dauern konnte.

Schon während der ENIAC gebaut wurde, beschrieb der Mathematiker **John von Neumann** (1903–1957) in seinem *First Draft of a Report on the EDVAC* (1945) die Grundstruktur eines frei programmierbaren Rechners. Der entscheidende Gedanke: Das Programm wird nicht verdrahtet, sondern wie die Daten im Speicher abgelegt. Wer die Maschine umprogrammieren will, tauscht nur noch Speicherinhalt aus. Diese Struktur heißt bis heute **von-Neumann-Architektur**.

6.1.1 Funktionseinheiten

- **Zentraleinheit** mit **Steuerwerk**, **Rechenwerk** und **Arbeitsspeicher**
- **Ein- und Ausgabegeräte**

Steuerwerk und Rechenwerk sitzen gemeinsam auf einem Chip, dem **Prozessor** (CPU). Verbunden werden alle Einheiten über den **Systembus**, der aus drei Teilen besteht:

Bus	Aufgabe
Datenbus	transportiert die eigentlichen Daten
Adressbus	legt fest, welche Speicherzelle gemeint ist
Steuerbus	überträgt Signale wie „lesen” oder „schreiben”

Speicher. Der Speicher ist eine lineare Folge gleich großer Zellen. Jede Zelle hat eine Nummer, ihre **Adresse**, und kann einen Wert aufnehmen — einen Datenwert oder einen Befehl, die Maschine sieht da keinen Unterschied. Für die Modellmaschine in diesem Kapitel teilen wir den Speicher trotzdem in einen **Programmspeicher** (die Befehle) und einen **Datenspeicher** (die Werte) auf. Das ist eine didaktische Vereinfachung: Sie erspart uns das Mitzählen von Adressen und macht die Abläufe lesbar.

Steuerwerk. Es organisiert die Programmausführung: Befehle in der richtigen Reihenfolge holen, decodieren, interpretieren und die beteiligten Einheiten mit Steuersignalen versorgen.

Rechenwerk. Hier werden arithmetische und logische Verknüpfungen ausgeführt, daher der Name **ALU** (Arithmetic Logic Unit). Die Operanden bekommt sie vom Steuerwerk zugeteilt.

Register. Register sind besonders schnelle Speicherplätze im Prozessor selbst. Drei sind für uns wichtig:

- Der **Akkumulator** nimmt Werte aus dem Datenspeicher auf und sammelt Rechenergebnisse (von lat. *accumulare*, anhäufen). Vor einer Addition steht ein Operand im Akkumulator, der andere im Speicher; danach steht das Ergebnis im Akkumulator.
- Das **Befehlsregister** enthält den gerade auszuführenden Befehl.

- Der **Befehlszähler** (Befehlsadressregister) enthält die Adresse des Befehls, der als nächstes geholt wird. Solange ein Befehl noch bearbeitet wird, steht dort also dessen *eigene* Adresse; erst der letzte Schritt des Befehlszyklus rückt ihn weiter.

6.1.2 Die fünf Prinzipien

1. **Ein gemeinsamer, adressierbarer Speicher.** Der Speicher besteht aus gleich großen Zellen mit fortlaufenden Adressen, und **Befehle und Daten liegen im selben Speicher**. Genau das ist der Kerngedanke: Ein Programm ist für die Maschine nichts anderes als Daten und kann deshalb geladen, verändert und ausgetauscht werden.
2. **Programmgesteuertes Arbeiten.** Der Aufbau des Rechners ist unabhängig vom Problem; was er tut, bestimmt allein das Programm.
3. **Sequentielle Befehlsabarbeitung.** Die Befehle werden der Reihe nach ausgeführt. Sie enthalten nicht die Werte, sondern die *Adressen* der Werte.
4. **Sprungbefehle.** Von der Reihenfolge darf abgewichen werden; der Sprung kann von gespeicherten Werten abhängen. Erst dadurch entstehen Verzweigung und Schleife.
5. **Binärcodierung.** Sämtliche Daten und Befehle werden binär dargestellt.

Merke

Punkt 3 ist der Grund für Variablen: Ein Programm rechnet nicht mit Zahlen, sondern mit Adressen, hinter denen Zahlen stehen. Punkt 4 ist der Grund dafür, dass überhaupt etwas anderes als eine starre Rechenvorschrift möglich ist.

Stolperstein

Unsere Modellmaschine hat zwei getrennte Speicher — das widerspricht Prinzip 1. Trennt eine *echte* Maschine Programm- und Datenspeicher samt ihrer Busse, heißt das nicht mehr von-Neumann-, sondern **Harvard-Architektur**; viele Mikrocontroller sind so gebaut. Der Unterschied ist keine Spitzfindigkeit: Nur im gemeinsamen Speicher kann ein Programm ein anderes Programm als Daten behandeln — und genau darauf beruhen Compiler, Betriebssysteme und Virenscanner ebenso wie ein Teil der Sicherheitslücken, die Sie in Kapitel 8 wiedersehen.

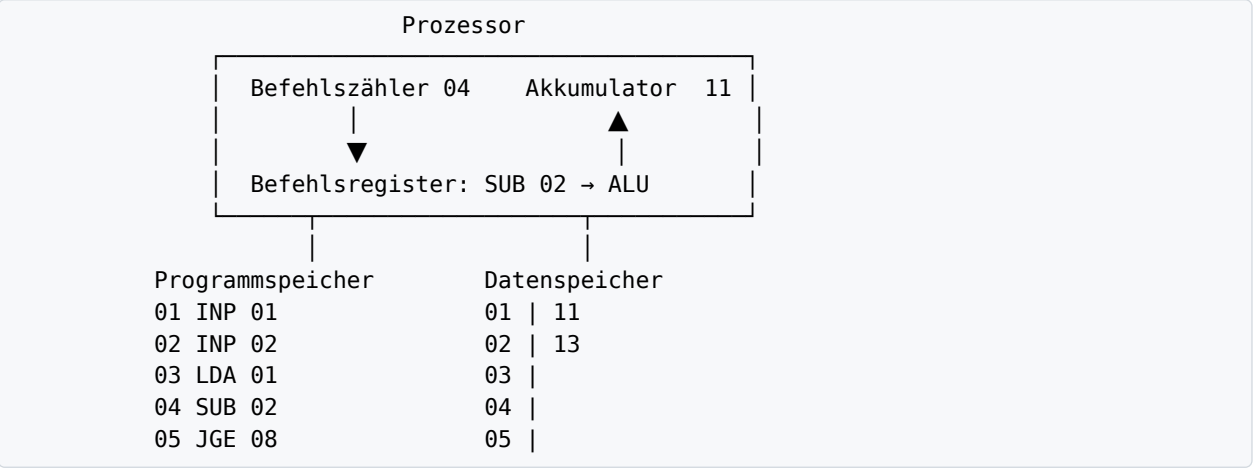
Arbeitsauftrag 1

1. Ordnen Sie zu, welche Funktionseinheit zuständig ist: eine Zahl addieren; den nächsten Befehl holen; einen Wert dauerhaft ablegen; entscheiden, ob gesprungen wird.
2. Erklären Sie, warum ein Rechner drei getrennte Busse braucht und nicht mit einem auskommt.
3. Was wäre nicht möglich, wenn es Prinzip 4 nicht gäbe? Nennen Sie zwei Beispiele aus Ihren bisherigen Programmen.
4. Recherchieren Sie den Unterschied zwischen von-Neumann- und Harvard-Architektur und nennen Sie je einen Einsatzbereich.

6.2 Die Registermaschine

ZUSATZ

Wir bauen uns eine Maschine mit dem Nötigsten: ein Programmspeicher, ein Datenspeicher, ein Akkumulator, ein Befehlsregister, ein Befehlszähler.



Die Zeichnung ist eine **Momentaufnahme mitten im Befehl** `04 SUB 02`: Der Befehl steht im Befehlsregister, der Operand wird gerade aus Datenzelle 02 geholt, im Akkumulator steht noch der alte Wert 11. Erst der vierte Schritt des Befehlszyklus schreibt $11 - 13 = -2$ hinein, der fünfte setzt den Befehlszähler auf 05.

Ein Befehl besteht immer aus drei Angaben: seiner **Befehlsadresse** im Programmspeicher, der **Operation** und einem **Adressteil**. Was der Adressteil bedeutet, hängt von der Operation ab: bei `LDA`, `STA`, `ADD` oder `SUB` eine Zelle des Datenspeichers, bei `LDK` die Zahl selbst, bei den Sprungbefehlen eine Adresse im *Programm*speicher, bei `OUT 00` den Akkumulator und bei `HLT 99` gar nichts.

6.2.1 Der Befehlssatz

Transportbefehle

Befehl	Wirkung
`LDA <Adresse>`	Lade den Inhalt der Adresse in den Akkumulator
`LDK <Zahl>`	Lade die Konstante in den Akkumulator
`STA <Adresse>`	Speichere den Akkumulatorinhalt in die Adresse

Verarbeitungsbefehle

Befehl	Wirkung
`ADD <Adresse>`	Addiere den Inhalt der Adresse zum Akkumulator
`SUB <Adresse>`	Subtrahiere den Inhalt der Adresse vom Akkumulator
`MUL <Adresse>`	Multipliziere den Akkumulator mit dem Inhalt der Adresse
`DIV <Adresse>`	Dividiere den Akkumulator durch den Inhalt der Adresse; der Rest fällt weg

Sprungbefehle

Befehl	Wirkung
`JMP <Adresse>`	Springe zur Adresse
`JEZ <Adresse>`	Springe, wenn Akkumulator $= 0$
`JNE <Adresse>`	Springe, wenn Akkumulator $\neq 0$
`JLZ <Adresse>`	Springe, wenn Akkumulator < 0
`JGZ <Adresse>`	Springe, wenn Akkumulator > 0
`JLE <Adresse>`	Springe, wenn Akkumulator ≤ 0
`JGE <Adresse>`	Springe, wenn Akkumulator ≥ 0

Sonstige Befehle

Befehl	Wirkung
`INP <Adresse>`	Lies eine Zahl ein und speichere sie in der Adresse
`OUT <Adresse>`	Gib den Inhalt der Adresse aus (Adresse 00 ist der Akkumulator)
`HLT 99`	Programmende

Stolperstein

`DIV` ist eine **Ganzzahldivision**: $7 : 2$ ergibt 3, der Rest 1 verfällt. Für unsere Programme rechnen wir nur mit nichtnegativen Zahlen — und das aus gutem Grund: Python rundet mit `//` immer nach unten (`-7 // 2` ergibt -4), Pascal schneidet mit `div` zur Null hin ab (`-7 div 2` ergibt -3). Für negative Operanden liefern die beiden Simulatoren weiter unten also verschiedene Ergebnisse. Das ist keine Schlamperei, sondern eine echte Festlegung, die jede Maschine treffen muss.

Eine solche maschinennahe Sprache heißt **Assemblersprache** (von *to assemble*, montieren): Die Befehle sind Bausteine, die zu einem Programm zusammengesetzt werden.

6.2.2 Beispiel: das Minimum zweier Zahlen

Gesucht ist $\min(a, b)$. Die beiden Zahlen sollen in den Datenzellen 01 und 02 liegen.

```
01 INP 01      ; a in Speicherzelle 01
02 INP 02      ; b in Speicherzelle 02
03 LDA 01      ; a in den Akkumulator
04 SUB 02      ; Akku = a - b
05 JGE 08      ; Sprung, wenn a - b >= 0, also b <= a
06 LDA 01      ; sonst: a ist kleiner
07 JMP 09      ; Sprung zur Ausgabe
08 LDA 02      ; b ist kleiner oder gleich
09 OUT 00      ; Ausgabe des Akkumulatorinhalts
10 HLT 99      ; Programmende
```

Verfolgen Sie den Ablauf für $a = 11$ und $b = 13$: In Zeile 04 steht -2 im Akkumulator, der Sprung in Zeile 05 findet nicht statt, Zeile 06 lädt $a = 11$, Zeile 07 überspringt Zeile 08, Zeile 09 gibt 11 aus. Richtig.

Merke

Die Maschine kennt kein `if`. Eine Verzweigung entsteht aus genau zwei Zutaten: einer Rechnung, die das Ergebnis der Bedingung ins Vorzeichen des Akkumulators bringt, und einem bedingten Sprung darauf. Genau das erzeugt Ihr Compiler aus jedem `if`.

6.3 Der Befehlszyklus

ZUSATZ

Wie führt die Maschine einen Befehl aus? In fünf Schritten. Wir spielen sie für den Befehl `04 SUB 02` durch.

1. **Befehl holen.** Aus dem Programmspeicher wird das Befehlswort geholt, dessen Adresse im Befehlszähler steht — hier `SUB 02` — und ins Befehlsregister gelegt.
2. **Befehl decodieren.** Der Operationsteil `SUB` wird entschlüsselt: Die ALU wird auf Subtraktion geschaltet. Jetzt erst ist bekannt, dass überhaupt ein Operand aus dem Datenspeicher gebraucht wird und wie der Adressteil zu lesen ist.
3. **Operand holen.** Der Adressteil `02` bezeichnet die Datenzelle, deren Inhalt gebraucht wird; die 13 wird bereitgestellt.
4. **Operation ausführen.** Die ALU rechnet $11 - 13 = -2$; das Ergebnis geht in den Akkumulator.
5. **Befehlszähler ändern.** Der Befehlszähler wird um 1 erhöht — bei einem Sprungbefehl stattdessen auf die Sprungadresse gesetzt.

Die Schritte 1 und 2 heißen zusammen **Holphase**, die Schritte 3 bis 5 **Ausführungsphase**.

Im Bedienfeld lässt sich beides durchspielen: ein Befehl je Schritt oder, mit *Befehlszyklus zeigen*, jede der fünf Phasen einzeln.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Registermaschine**.

Dass das Decodieren *vor* dem Operandenholen steht, ist kein Zufall: Ein `LDK` braucht gar keinen Speicherzugriff, ein `JMP` liest seinen Adressteil als Programmadresse. Ohne Decodieren wüsste die Maschine nicht, was sie holen soll.

Zu Beginn steht der Befehlszähler auf der ersten Programmadresse. Die Ausführung endet mit `HLT` — oder wenn die Maschine auf einen nicht ausführbaren Befehl trifft, etwa `JMP 25` in einem Programm mit 20 Zeilen.

Arbeitsauftrag 2

5. Spielen Sie den Befehlszyklus für `03 LDA 01` und für `05 JGE 08` einzeln durch und notieren Sie nach jedem Schritt Befehlszähler und Akkumulator.
6. Verfolgen Sie das Minimum-Programm vollständig für $a = 20$, $b = 7$ in einer Tabelle mit den Spalten Befehlszähler, Befehl, Akkumulator, Zelle 01, Zelle 02.
7. Was passiert, wenn in Zeile 07 `JMP 09` fehlt? Begründen Sie mit dem Ablauf.
8. Warum wird der Befehlszähler erst im *letzten* Schritt geändert und nicht im ersten?

6.4 Programmieren auf der Registermaschine

ZUSATZ

Arbeitsauftrag 3

9. Schreiben Sie ein Programm, das das arithmetische Mittel zweier Zahlen a und b berechnet und ausgibt.
10. Schreiben Sie ein Programm, das die größere zweier Zahlen ausgibt. Nutzen Sie das Minimum-Programm als Vorlage und ändern Sie möglichst wenig.
11. Schreiben Sie ein Programm, das die Zahlen von 1 bis n aufsummiert. Sie brauchen eine Zählvariable im Datenspeicher, eine Summe und einen Rücksprung.
12. Schreiben Sie ein Programm für ggT(a, b) nach dem Verfahren des Euklid: Solange $a \neq b$, ziehe die kleinere Zahl von der größeren ab.
13. Schreiben Sie ein Programm, das eine Zahl mit 8 multipliziert, ohne `MUL` zu benutzen.
14. Erweitern Sie Aufgabe 11 so, dass das Programm die Summe *ungerader* Zahlen bildet.
15. **Knobelaufgabe:** Schreiben Sie ein Programm, das prüft, ob eine Zahl gerade ist, und 1 oder 0 ausgibt. `DIV` lässt den Rest weg — daraus lässt sich der Rest zurückgewinnen.

Merke

Für Schleifen gibt es nur ein Muster: Zählvariable im Datenspeicher verändern, mit dem Zielwert vergleichen (durch Subtraktion), mit einem bedingten Sprung an den Schleifenanfang zurück. Alles, was Sie in Hochsprachen über `while` und `for` schreiben, wird genau dazu.

Gymnasium Ottweiler

6.5 Einen Simulator programmieren

ZUSATZ

Die Registermaschine lässt sich in wenigen Zeilen nachbauen. Das Programm ist eine Liste von Befehlen, der Datenspeicher eine Liste von Zahlen, der Rest sind drei Variablen.

Python

```python
def simuliere(programm, eingaben):
    """programm: Liste von (Operation, Adresse). Gibt die Ausgaben zurueck."""
    daten = [0] * 100          # Datenspeicher
    akku = 0                   # Akkumulator
    zaehler = 0                # Befehlszaehler (Index im Programm)
    eingabe_nr = 0
    ausgaben = []

    while 0 <= zaehler < len(programm):
        operation, adresse = programm[zaehler]
        zaehler += 1                       # Schritt 5, ausser bei Spruengen

        if operation == "INP":
            daten[adresse] = eingaben[eingabe_nr]
            eingabe_nr += 1
        elif operation == "LDA":
            akku = daten[adresse]
        elif operation == "LDK":
            akku = adresse
        elif operation == "STA":
            daten[adresse] = akku
        elif operation == "ADD":
            akku += daten[adresse]
        elif operation == "SUB":
            akku -= daten[adresse]
        elif operation == "MUL":
            akku *= daten[adresse]
        elif operation == "DIV":
            akku //= daten[adresse]
        elif operation == "JMP":
            zaehler = adresse - 1
        elif operation == "JEZ" and akku == 0:
            zaehler = adresse - 1
        elif operation == "JNE" and akku != 0:
            zaehler = adresse - 1
        elif operation == "JLZ" and akku < 0:
            zaehler = adresse - 1
        elif operation == "JGZ" and akku > 0:
            zaehler = adresse - 1
        elif operation == "JLE" and akku <= 0:
            zaehler = adresse - 1
        elif operation == "JGE" and akku >= 0:
            zaehler = adresse - 1
        elif operation == "OUT":
            ausgaben.append(akku if adresse == 0 else daten[adresse])
        elif operation == "HLT":
            break

    return ausgaben
```

```
50
51
52 MINIMUM = [("INP", 1), ("INP", 2), ("LDA", 1), ("SUB", 2), ("JGE", 8),
53            ("LDA", 1), ("JMP", 9), ("LDA", 2), ("OUT", 0), ("HLT", 99)]
54
55 print(simuliere(MINIMUM, [11, 13]))      # [11]
56 print(simuliere(MINIMUM, [20, 7]))       # [7]
```

Beachten Sie die Zeile `zaehler = adresse - 1` bei den Sprüngen: Die Registermaschine zählt ihre Programmzeilen ab 1, eine Python-Liste ihre Plätze ab 0. Ein Sprung nach Zeile 8 muss deshalb auf den Listenindex 7 führen. Das vorangegangene `zaehler += 1` spielt dabei keine Rolle — die Zuweisung überschreibt es ja. Genau solche Verschiebungen um eins übersieht man beim ersten Versuch, und der Simulator landet dann in der falschen Zeile oder in einer Endlosschleife. Die Bedingung `0 <= zaehler` in der Schleife fängt wenigstens den Fall ab, dass ein fehlerhafter Sprung ins Negative führt — ohne sie würde Python stillschweigend von hinten zählen.

Object Pascal

```
1  program Registermaschine;
2
3  type
4    TBefehl = record
5      Operation: string[3];
6      Adresse: Integer;
7    end;
8
9  const
10   PROGRAMM: array[1..10] of TBefehl = (
11     (Operation: 'INP'; Adresse: 1), (Operation: 'INP'; Adresse: 2),
12     (Operation: 'LDA'; Adresse: 1), (Operation: 'SUB'; Adresse: 2),
13     (Operation: 'JGE'; Adresse: 8), (Operation: 'LDA'; Adresse: 1),
14     (Operation: 'JMP'; Adresse: 9), (Operation: 'LDA'; Adresse: 2),
15     (Operation: 'OUT'; Adresse: 0), (Operation: 'HLT'; Adresse: 99));
16
17 var
18   daten: array[0..99] of Integer;
19   akku, zaehler, i: Integer;
20   b: TBefehl;
21
22 begin
23   for i := 0 to 99 do daten[i] := 0;
24   akku := 0;
25   zaehler := 1;
26
27   while (zaehler >= 1) and (zaehler <= Length(PROGRAMM)) do
28   begin
29     b := PROGRAMM[zaehler];
30     Inc(zaehler);                                  { Schritt 5 }
31
32     if b.Operation = 'INP' then ReadLn(daten[b.Adresse])
33     else if b.Operation = 'LDA' then akku := daten[b.Adresse]
34     else if b.Operation = 'LDK' then akku := b.Adresse
35     else if b.Operation = 'STA' then daten[b.Adresse] := akku
36     else if b.Operation = 'ADD' then akku := akku + daten[b.Adresse]
37     else if b.Operation = 'SUB' then akku := akku - daten[b.Adresse]
38     else if b.Operation = 'MUL' then akku := akku * daten[b.Adresse]
```

```
39      else if b.Operation = 'DIV' then akku := akku div daten[b.Adresse]
40      else if b.Operation = 'JMP' then zaehler := b.Adresse
41      else if (b.Operation = 'JEZ') and (akku =  0) then zaehler := b.Adresse
42      else if (b.Operation = 'JNE') and (akku <> 0) then zaehler := b.Adresse
43      else if (b.Operation = 'JLZ') and (akku <  0) then zaehler := b.Adresse
44      else if (b.Operation = 'JGZ') and (akku >  0) then zaehler := b.Adresse
45      else if (b.Operation = 'JLE') and (akku <= 0) then zaehler := b.Adresse
46      else if (b.Operation = 'JGE') and (akku >= 0) then zaehler := b.Adresse
47      else if b.Operation = 'OUT' then
48        if b.Adresse = 0 then WriteLn(akku) else WriteLn(daten[b.Adresse])
49      else if b.Operation = 'HLT' then Break;
50    end;
51  end.
```

Die Pascal-Fassung zählt ab 1 wie im Skript, die Python-Fassung ab 0 wie die Liste — deshalb steht dort `adresse - 1` und hier nicht. Ein guter Anlass, beide Programme nebeneinander zu legen.

Arbeitsauftrag 4

16. Ergänzen Sie den Simulator um eine Ausgabe nach jedem Schritt: Befehlszähler, Befehl, Akkumulator. Damit haben Sie einen Einzelschrittmodus.
17. Bauen Sie eine Sicherung ein, die nach 10 000 Schritten abbricht, und erklären Sie, warum das nötig ist — und warum sie das Problem nicht löst, sondern nur umgeht. (Vergleichen Sie Abschnitt 4.4.)
18. Lassen Sie Ihre Programme aus 6.4 im Simulator laufen und vergleichen Sie mit Ihrer Handrechnung.
19. Erweitern Sie den Simulator um einen Befehlszähler für die Statistik: Wie viele Schritte braucht Ihr ggT-Programm für $a = 1071$, $b = 462$?
20. **Projekt:** Schreiben Sie einen kleinen Assembler, der ein Programm aus einer Textdatei im Format `LDA 01` einliest und in die interne Darstellung übersetzt.

Gymnasium Ottweiler

6.6 Von der Registermaschine zum heutigen Prozessor

ZUSATZ

Unsere Maschine führt einen Befehl nach dem anderen aus und braucht dafür fünf Schritte. Ein heutiger Prozessor arbeitet nach demselben Grundprinzip, hat aber gelernt, es zu beschleunigen:

- **Pipelining.** Während ein Befehl ausgeführt wird, wird der nächste schon geholt und der übernächste decodiert — wie am Fließband. Ein Sprungbefehl kann die Pipeline leeren, deshalb versuchen Prozessoren, Sprünge vorherzusagen.
- **Cache.** Der Hauptspeicher ist im Vergleich zum Prozessor quälend langsam. Zwischen beiden liegen deshalb mehrere Stufen sehr schnellen Zwischenspeichers.
- **Viele Register.** Statt eines Akkumulators gibt es Dutzende Register, sodass Zwischenergebnisse nicht ständig in den Speicher zurückgeschrieben werden müssen.
- **Mehrere Kerne.** Ein Chip enthält mehrere vollständige Prozessoren, die echt gleichzeitig arbeiten.

Am von-Neumann-Prinzip ändert das nichts. Es bleibt aber ein grundsätzlicher Engpass: Befehle und Daten teilen sich den Weg zum Speicher. Dieser **von-Neumann-Flaschenhals** ist der Grund, warum mehr Rechenleistung ohne schnelleren Speicherzugriff wenig bringt — und warum Mikrocontroller oft der **Harvard-Architektur** mit getrennten Bussen für Programm und Daten folgen.

Arbeitsauftrag 5

21. Erklären Sie, warum ein bedingter Sprung für eine Pipeline ein Problem darstellt.
22. Der Zugriff auf den Hauptspeicher dauert etwa 100-mal so lange wie ein Rechenschritt. Was bedeutet das für die Aussage „der Prozessor schafft drei Milliarden Befehle pro Sekunde”?
23. Recherchieren Sie die Taktfrequenz und die Zahl der Kerne Ihres eigenen Rechners.
24. Diskutieren Sie: Warum ist es sicherheitstechnisch heikel, dass Programme und Daten im selben Speicher liegen? (Stichwort: ausführbarer Speicherbereich)

Gymnasium Ottweiler

6.7 Zusammenfassung

ZUSATZ

- Die von-Neumann-Architektur speichert Programm und Daten im selben Speicher; erst das macht freie Programmierbarkeit möglich.
- Ihre fünf Prinzipien sind: ein gemeinsamer Speicher aus adressierbaren Zellen, programmgesteuertes Arbeiten, sequentielle Abarbeitung, Sprungbefehle, Binärcodierung.
- Die Zentraleinheit besteht aus Steuerwerk, Rechenwerk (ALU) und Speicher; Busse verbinden die Einheiten.
- Akkumulator, Befehlsregister und Befehlszähler sind die drei Register, die man für das Verständnis braucht.
- Der Befehlszyklus umfasst Befehl holen, decodieren, Operand holen, ausführen und Befehlszähler ändern.
- Verzweigungen und Schleifen entstehen ausschließlich aus Rechnung plus bedingtem Sprung.
- Moderne Prozessoren beschleunigen dieses Modell durch Pipelining, Cache, viele Register und mehrere Kerne, folgen ihm aber weiterhin.

Wiederholungsaufgaben

25. Beschreiben Sie in eigenen Worten, was beim Ausführen von `ADD 03` genau geschieht.
26. Übersetzen Sie von Hand in Registermaschinencode: `if a > b then c := a - b else c := b - a;`
27. Erklären Sie den Unterschied zwischen `LDA 05` und `LDK 05` an einem Beispiel.
28. Warum ist `HLT 99` nötig, obwohl das Programm doch einfach zu Ende sein könnte?
29. Nennen Sie drei Gründe, warum man heute nicht mehr in Assembler programmiert — und einen, warum es trotzdem noch gemacht wird.

Gymnasium Ottweiler