PROFESSOR ERKLÄRBÄR

Serie Informatik · Folge 17 · Advanced

INFORMATIK

LEHRERINFO ZUM VIDEO

Wir bauen eine eigene Mini-CPU

FACH	STUFE	DAUER	VIDEO
Informatik	Gymnasium · Sek II	16 min	youtube.com/@Erklaerbaer

01 Lernziele

Nach dem Video können die Schülerinnen und Schüler …

- den Aufbau einer einfachen 8-Bit-CPU aus ALU, Registern, Speicher, Befehlszähler, Befehlsregister und Steuerwerk erklären,
- einen Befehlssatz lesen und Befehle codieren (4 Bit Opcode + 4 Bit Adresse, Hex),
- das Steuerwerk als Automat mit Mikroschritten und Steuersignalen beschreiben,
- ein Assemblerprogramm schreiben, von Hand in Maschinencode übersetzen und Takt für Takt verfolgen,
- bedingte Sprünge mit dem Flag Z für Schleifen einsetzen,
- die Grenzen der Mini-CPU benennen und sie mit echten Prozessoren vergleichen.

02 Voraussetzungen

Hex und Zweierkomplement (Folge 2 · Advanced), Befehle als Codes (Folge 3 · Advanced), ALU, Flipflop, Register (Folge 4 · Advanced), Befehlszyklus und Von-Neumann-Architektur (Folge 5 · Advanced), Assembler (Folge 7 · Advanced), Automaten (Folge 8 · Advanced); der 4-Bit-Addierer aus Folge 17 · Grundlagen. Für den Simulator: Python 3.

03 Ablauf des Videos

ZEIT	KAPITEL	INHALT UND BEISPIEL
0:00	Einstieg	elf Bytes rechnen 3 · 4 = 12 · Ident · Fahrplan
0:34	Was du brauchst	Bausteine aus 02A, 03A, 04A, 05A, 07A, 08A und 17G
1:15	Der Bauplan	SAP-1 · Bus, Register, ALU, Speicher, Steuerwerk · Steckbrett
2:16	Die ALU	8 Volladdierer + XOR-Reihe · 4 − 1 · Flag Z als NOR
3:13	Der Befehlssatz	LDA 14 = 1E, JZ 8 = 78 PAUSE-FRAGE ADD 12, JMP 3 · von Neumann
4:55	Das Steuerwerk	T0/T1 holen · LDA, ADD Schritt für Schritt · ROM · Automat

6:23	Das Programm	3 · 4 als Schleife · Assembler von Hand · Frontplatte
7:48	Takt für Takt	erste Runde: 8 Befehle, 32 Takte, Bus und Register live
8:52	Zeitraffer	Runden 2–4 PAUSE-FRAGE Runden · OUT 12 · 5 · 7 = 0x23
9:59	Grenzen	16 Byte, Überlauf · 64 Bit, Caches, Pipeline · ARM, RISC-V, FPGA
11:09	Rückblick	Bausteine der Serie auf dem Bauplan · alle drei Projekte fertig
11:56	Prüfungsfallen	fünf Fallen (siehe 04)
12:55	Zusammenfassung	Kernaussagen · zurück zum Einstieg
13:34	Quiz	4 Fragen · Endkarte zur ganzen Serie

04 Typische Fehlvorstellungen

- „Der Befehlszähler wird erst nach dem Befehl erhöht“ – schon in T1 beim Holen; beim Abarbeiten zeigt er auf den nächsten Befehl.
- „JZ prüft den Akku“ – JZ prüft das Flag Z, das der letzte LDA, ADD oder SUB gesetzt hat (im Programm ist Z nach LDA 15 in Runde 1 gesetzt, stört aber nicht).
- „Befehle und Daten sind verschiedene Bits“ – von Neumann: 0x1E ist LDA 14 oder die Zahl 30, je nach Lesart.
- „4 Bit Adresse = 4 Zellen“ – 2^4 = 16 Zellen (0 bis 15).
- „Ohne Multiplizierbefehl kein Multiplizieren“ – Multiplikation als Schleife aus Addieren und bedingtem Springen.

05 Einsatz im Unterricht

Papiercomputer: Die Klasse spielt die CPU: je eine Person für Befehlszähler, Speicher, Akku, ALU und Steuerwerk; das Steuerwerk ruft die Signale aus der Tabelle (Anhang A) auf.

Simulator: `python mini_cpu.py --takt` zeigt jeden Takt der ersten Runde; eigene Programme in `uebungen.py` ergänzen.

Schaltung: Fortgeschrittene bauen die CPU in Logisim Evolution nach: Register aus D-Flipflops, ALU aus dem 8-Bit-Addierer mit XOR-Reihe, Steuerwerk als ROM.

Erweitern: einen Befehl JC (Sprung bei Übertrag) ergänzen: Welches Flag und welche Zeile im ROM braucht es zusätzlich?

Hinweis: Alle Werte im Video sind mit `mini_cpu.py` (Anhang B, Python 3.12.10) nachgerechnet: Ausgabe 12 nach 37 Befehlen und 145 Takten (erste Runde 8 Befehle, 32 Takte), 5 · 7 = 35 = 0x23. Ein Befehl dauert 3 bis 5 Takte. Anders als in manchen SAP-1-Vorlagen setzt bei uns auch LDA das Flag Z.

06 Quellen

- A. P. Malvino, J. A. Brown: Digital Computer Electronics, McGraw-Hill – Lehrbuchmodell SAP-1 («Simple As Possible»)
- J. von Neumann (1945): First Draft of a Report on the EDVAC
- RISC-V International: The RISC-V Instruction Set Manual – riscv.org/specifications
- Logisim Evolution – github.com/logisim-evolution/logisim-evolution

NAME _____ KLASSE _____ DATUM _____

Wir bauen eine eigene Mini-CPU

1 Befehle codieren und lesen ●○○

Übersetze mit der Befehlstabelle (Anhang A) in Binär und Hex: a) `LDA 13` · b) `STA 15` · c) `JZ 10` · d) `OUT`. **Was** bedeuten umgekehrt die Bytes `2E`, `3F`, `69` und `5A`?

2 Mikroschritte ●●○

Notiere für `SUB 12` die Mikroschritte T0 bis T4 mit Steuersignalen. **Gib** jeweils an, welcher Wert auf dem Bus liegt und wer ihn übernimmt.

3 Programm verfolgen ●●○

Im Programm 3 · 4 stehen jetzt x = 2 in Zelle 13 und y = 3 in Zelle 14. **Wie** viele Schleifenrunden laufen? **Welche** Ausgabe erscheint? **Wie** viele Befehle werden insgesamt abgearbeitet?

4 Countdown programmieren ●●●

Schreibe ein Programm, das nacheinander 3, 2, 1, 0 ausgibt und dann anhält. Tipp: `LDI`, `OUT`, `SUB`, `JZ`, `JMP` und eine Konstante 1 in Zelle 15. **Übersetze** es in Hex.

5 Weiterdenken ●●●

a) Warum kann `LDI` nur die Zahlen 0 bis 15 laden? · b) Was gibt das Programm 3 · 4 aus, wenn y = 0 ist – und warum? · c) Wie viele Takte braucht das Programm für y Runden?

Lösungen

1 Befehle codieren und lesen

a) 0001 1101 = `1D` · b) 0100 1111 = `4F` · c) 0111 1010 = `7A` · d) 1110 0000 = `E0` · `2E` = ADD 14 · `3F` = SUB 15 · `69` = JMP 9 · `5A` = LDI 10

2 Mikroschritte

T0 `CO MI`: Befehlszähler → Adressregister · T1 `RO II CE`: Speicher (der Befehl 3C) → Befehlsregister, Zähler + 1 · T2 `IO MI`: Adresse 12 aus dem Befehlsregister → Adressregister · T3 `RO BI`: Inhalt von Zelle 12 → Register B · T4 `EO AI SU FI`: ALU rechnet A + NOT B + 1 = A − B → Akku, Flag Z wird gesetzt

3 Programm verfolgen

3 Runden (Produkt 2, 4, 6) · Ausgabe 6 · 3 · 8 + 2 + 3 = 29 Befehle (8 pro Runde, dann LDA 14 und JZ 8, zum Schluss LDA 15, OUT, HLT)

4 Countdown programmieren

0 `LDI 3` 53 · 1 `OUT` E0 · 2 `SUB 15` 3F · 3 `JZ 5` 75 · 4 `JMP 1` 61 · 5 `OUT` E0 · 6 `HLT` F0 · 15: 01 → Ausgabe 3, 2, 1, 0 nach 14 Befehlen (geprüft mit `uebungen.py`)

5 Weiterdenken

a) Im Befehlsbyte bleiben nur 4 Bit für die Zahl: 0000 bis 1111 · b) 0: LDA 14 lädt die 0, Z = 1, JZ springt sofort zu Adresse 8, ausgegeben wird das Produkt 0 (3 · 0 = 0) · c) 32 Takte pro Runde + 17 Takte (LDA 14, JZ, LDA 15, OUT, HLT) → 32 · y + 17, bei y = 4 also 145

Q Quiz im Video

1 C (16 Zellen) · 2 B (springt, wenn Flag Z gesetzt) · 3 A (der aktuelle Befehl) · 4 C (wiederholtes Addieren in einer Schleife)

Anhang A: Befehlssatz und Steuersignale

Ein Befehl = 1 Byte: obere 4 Bit Opcode, untere 4 Bit Adresse a (bei LDI die Zahl n). Das Flag Z setzen LDA, ADD und SUB (Z = 1, wenn das Ergebnis 0 ist).

OPCODE	BEFEHL	WIRKUNG	T2	T3	T4
0001 (1)	`LDA a`	A ← RAM[a]	`IO MI`	`RO AI FI`	–
0010 (2)	`ADD a`	A ← A + RAM[a]	`IO MI`	`RO BI`	`EO AI FI`
0011 (3)	`SUB a`	A ← A − RAM[a]	`IO MI`	`RO BI`	`EO AI SU FI`
0100 (4)	`STA a`	RAM[a] ← A	`IO MI`	`AO RI`	–
0101 (5)	`LDI n`	A ← n (0 bis 15)	`IO AI`	–	–
0110 (6)	`JMP a`	Befehlszähler ← a	`IO J`	–	–
0111 (7)	`JZ a`	Befehlszähler ← a, wenn Z = 1	`IO J` (nur bei Z = 1)	–	–
1110 (E)	`OUT`	Ausgabe ← A	`AO OI`	–	–
1111 (F)	`HLT`	Takt anhalten	`HLT`	–	–

Holen bei jedem Befehl: T0 `CO MI` (Befehlszähler → Adressregister) · T1 `RO II CE` (Speicher → Befehlsregister, Befehlszähler + 1).
Signale: CO Zähler aus · CE Zähler + 1 · J Sprung (Zähler übernimmt) · MI Adressregister ein · RO/RI Speicher aus/ein · II/IO Befehlsregister ein/aus (untere 4 Bit) · AI/AO Akku ein/aus · BI Register B ein · EO ALU aus · SU subtrahieren · FI Flag Z übernehmen · OI Ausgabe ein · HLT Takt anhalten

A Programm 3 · 4

ADRESSE	ASSEMBLER	HEX	BEDEUTUNG	ADRESSE	INHALT	HEX	BEDEUTUNG
0	`LDA 14`	1E	Zähler y laden	8	`LDA 15`	1F	Produkt laden
1	`JZ 8`	78	fertig, wenn y = 0	9	`OUT`	E0	ausgeben
2	`SUB 12`	3C	y − 1	10	`HLT`	F0	anhalten
3	`STA 14`	4E	y speichern	11	–	00	frei
4	`LDA 15`	1F	Produkt laden	12	1	01	Konstante 1
5	`ADD 13`	2D	+ x	13	3	03	x
6	`STA 15`	4F	Produkt speichern	14	4	04	y (Zähler)
7	`JMP 0`	60	nächste Runde	15	0	00	Produkt

Anhang B: mini_cpu.py

Simulator der Mini-CPU, Takt für Takt mit denselben Mikroschritten wie im Video. `python mini_cpu.py --takt` zeigt die erste Schleifenrunde.

```
"""Mini-CPU aus Folge 17A: 8-Bit-Datenbus, Register A und B, ALU mit Flag Z, 16 Byte RAM, 4-Bit-Befehlszähler,
Befehlsregister, Ausgaberegister. Jeder Befehl ist ein Byte: 4 Bit Opcode + 4 Bit Adresse (oder Zahl).
Das Steuerwerk arbeitet jeden Befehl in Mikroschritten T0 bis T4 ab (Steuersignale wie im Lehrbuchmodell SAP-1).

Aufruf:  python mini_cpu.py            → Programm 3 · 4, Ausgabe 12
         python mini_cpu.py --takt     → zusätzlich jeder Takt der ersten Schleifenrunde
"""
import sys

OPS = {"LDA": 0x1, "ADD": 0x2, "SUB": 0x3, "STA": 0x4, "LDI": 0x5, "JMP": 0x6, "JZ": 0x7, "OUT": 0xE, "HLT": 0xF}
NAME = {v: k for k, v in OPS.items()}

# Mikroschritte: T0 und T1 holen den Befehl (Fetch), ab T2 führt das Steuerwerk ihn aus.
FETCH = [["CO", "MI"], ["RO", "II", "CE"]]
MIKRO = {
    "LDA": [["IO", "MI"], ["RO", "AI", "FI"]],
    "ADD": [["IO", "MI"], ["RO", "BI"], ["EO", "AI", "FI"]],
    "SUB": [["IO", "MI"], ["RO", "BI"], ["EO", "AI", "SU", "FI"]],
    "STA": [["IO", "MI"], ["AO", "RI"]],
    "LDI": [["IO", "AI"]],
    "JMP": [["IO", "J"]],
    "JZ": [["IO", "JZ"]],
    "OUT": [["AO", "OI"]],
    "HLT": [["HLT"]],
}


def assemble(lines):
    """Assembler von Hand: 'LDA 14' → 0x1E. Reine Zahlen sind Daten."""
    ram = [0] * 16
    for adr, z in enumerate(lines):
        if z is None:
            continue
        if isinstance(z, int):
            ram[adr] = z & 0xFF
            continue
        teile = z.split()
        arg = int(teile[1]) if len(teile) > 1 else 0
        assert 0 <= arg <= 15, z
        ram[adr] = (OPS[teile[0]] << 4) | arg
    return ram


class CPU:
    def __init__(self, ram):
        self.ram = list(ram)
        self.pc = self.mar = self.ir = self.a = self.b = 0
        self.z = 0
        self.ausgabe = []
        self.halt = False
        self.takte = 0
        self.befehle = 0

    def schritt(self, signale):
        """Ein Takt: genau ein Wert liegt auf dem Bus, die Register mit 'in'-Signal übernehmen ihn."""
        s = set(signale)
        bus = None
        su = "SU" in s
        alu = (self.a + ((~self.b) & 0xFF) + 1) & 0xFF if su else (self.a + self.b) & 0xFF
        if "CO" in s: bus = self.pc
        if "RO" in s: bus = self.ram[self.mar]
        if "IO" in s: bus = self.ir & 0x0F
        if "AO" in s: bus = self.a
        if "EO" in s: bus = alu
        if "MI" in s: self.mar = bus & 0x0F
        if "II" in s: self.ir = bus
        if "AI" in s: self.a = bus
```

```
        if "BI" in s: self.b = bus
        if "RI" in s: self.ram[self.mar] = bus
        if "OI" in s: self.ausgabe.append(bus)
        if "FI" in s: self.z = 1 if bus == 0 else 0
        if "CE" in s: self.pc = (self.pc + 1) & 0x0F
        if "J" in s: self.pc = bus
        if "JZ" in s and self.z: self.pc = bus
        if "HLT" in s: self.halt = True
        self.takte += 1

    def befehl(self, protokoll=None):
        for i, sig in enumerate(FETCH):
            self.schritt(sig)
            if protokoll is not None:
                protokoll.append((f"T{i}", sig, self.zustand()))
        op = NAME[self.ir >> 4]
        for i, sig in enumerate(MIKRO[op]):
            self.schritt(sig)
            if protokoll is not None:
                protokoll.append((f"T{i + 2}", sig, self.zustand()))
        self.befehle += 1
        return op

    def zustand(self):
        return dict(PC=self.pc, IR=f"{self.ir:02X}", A=self.a, B=self.b, Z=self.z, R14=self.ram[14], R15=self.ram[15])

    def lauf(self, limit=1000):
        while not self.halt and self.befehle < limit:
            self.befehl()
        return self.ausgabe


PROG_3MAL4 = ["LDA 14", "JZ 8", "SUB 12", "STA 14", "LDA 15", "ADD 13", "STA 15", "JMP 0",
              "LDA 15", "OUT", "HLT", 0, 1, 3, 4, 0]

if __name__ == "__main__":
    ram = assemble(PROG_3MAL4)
    print("Maschinencode:", " ".join(f"{b:02X}" for b in ram))
    cpu = CPU(ram)
    if "--takt" in sys.argv:
        for _ in range(8):
            prot = []
            op = cpu.befehl(prot)
            for t, sig, z in prot:
                print(f"  {t} {' '.join(sig):<12} {z}")
            print("  →", op)
    aus = cpu.lauf()
    print("Ausgabe:", aus, "· Befehle:", cpu.befehle, "· Takte:", cpu.takte)
```

Anhang C: uebungen.py

Drei Übungsprogramme mit Lösung: 5 · 7, Countdown und Summe 1 bis 4. Jede Lösung prüft sich selbst (`assert`).

```
"""Drei Übungsprogramme für die Mini-CPU aus Folge 17A (Lösungen).
Aufruf im Ordner Code:  python uebungen.py
"""
from mini_cpu import CPU, assemble, PROG_3MAL4


def lauf(name, programm):
    ram = assemble(programm)
    cpu = CPU(ram)
    aus = cpu.lauf()
    print(f"{name}: Maschinencode {' '.join(f'{b:02X}' for b in ram)}")
    print(f"   Ausgabe {aus} · {cpu.befehle} Befehle · {cpu.takte} Takte")
    return aus


# Übung 1: 5 · 7 – nur die Daten ändern (Zelle 13 = x, Zelle 14 = y)
p1 = list(PROG_3MAL4)
p1[13], p1[14] = 5, 7
assert lauf("Übung 1 · 5 · 7", p1) == [35]

# Übung 2: Countdown 3, 2, 1, 0 – OUT in der Schleife
p2 = ["LDI 3",   # 0  A = 3
      "OUT",     # 1  ausgeben
      "SUB 15",  # 2  A - 1 (Zelle 15 = 1)
      "JZ 5",    # 3  bei 0 zum Ende
      "JMP 1",   # 4  nächste Runde
      "OUT",     # 5  die 0 ausgeben
      "HLT",     # 6
      None, None, None, None, None, None, None, None,
      1]         # 15 Konstante 1
assert lauf("Übung 2 · Countdown", p2) == [3, 2, 1, 0]

# Übung 3: Summe 1 + 2 + 3 + 4 – Zähler n zählt herunter, jede Runde Summe + n
p3 = ["LDA 14",  # 0  A = n
      "JZ 8",    # 1  fertig, wenn n = 0
      "ADD 15",  # 2  A = n + Summe
      "STA 15",  # 3  Summe speichern
      "LDA 14",  # 4  A = n
      "SUB 13",  # 5  n - 1
      "STA 14",  # 6  n speichern
      "JMP 0",   # 7  nächste Runde
      "LDA 15",  # 8  Summe laden
      "OUT",     # 9  ausgeben
      "HLT",     # 10
      0, 0,      # 11, 12 frei
      1,         # 13 Konstante 1
      4,         # 14 n
      0]         # 15 Summe
assert lauf("Übung 3 · Summe 1 bis 4", p3) == [10]
print("alle drei Lösungen geprüft")
```