Modellrechner mit Pseudo-Assembler (MOPS)

Der *MOPS* ist ein Modellrechner, der dem schematischen Aufbau eines Von-Neumann-Rechners (VNR) entspricht. Simuliert werden die Vorgänge, die sich beim Ablauf eines **Programms** im Herzen eines VNR abspielen („Von-Neumann-Zyklen“). Damit man den VNR in Aktion sehen kann, muss der MOPS mit Befehlen in **Maschinencode** gefüttert werden.

Weil echter Maschinencode aber für Menschen nicht gut lesbar ist, enthält der MOPS einen **Assembler**, der **mnemonischen Assemblercode** in Maschinencode umwandelt und diesen dem VNR zuführt. Im Rahmen des Befehlsvorrats dieses Assemblers ist der MOPS vom Anwender frei programmierbar.

Da jedoch kein echter Maschinencode erzeugt und ausgeführt wird, sondern nur ein **Pseudocode** für den simulierten VNR verwendung findet, ist der MOPS eben „nur“ ein **MO**dellrechner mit **PS**eudo-Assembler.

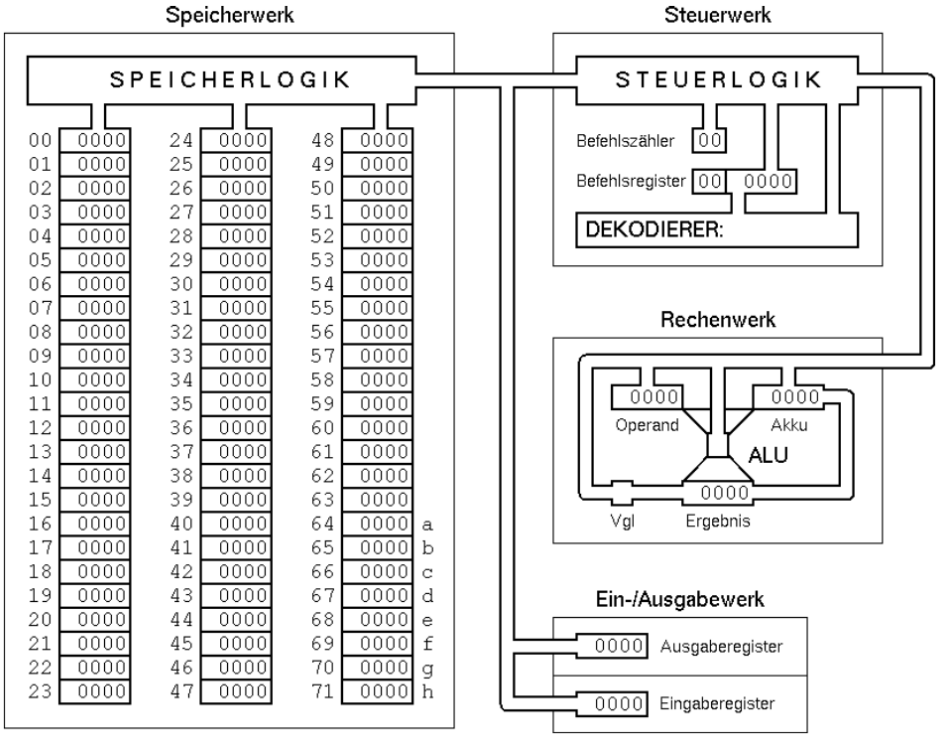

Abb. 1: Schematische Übersicht des MOPS

Übungen

1. Aufgabe ○ Erklären Sie die fett gedruckten Begriffe mit maximal zwei prägnanten Sätzen.

2. Aufgabe ⊖ Schreiben Sie ein Programm, das eine Zahl n aus dem *Eingaberegister* einliest und prüft, ob es sich um eine gerade Zahl handelt. Ist dies der Fall, soll eine 1 in das *Ausgaberegister* geschrieben werden, andernfalls eine 0.

3. Aufgabe ○ Schreiben Sie ein Programm, dass die Summe der ersten n Zahlen berechnet. **Beispiel**: Die Summe der ersten $n = 3$ Zahlen ist $1 + 2 + \mathbf{3} = 6$. Dabei soll n vom *Eingaberegister* eingelesenen und das Ergebnis in das *Ausgaberegister* geschrieben werden.

MOPS-Befehlssatz

Der Befehlssatz des MOPS-Assemblers umfasst insgesamt 15 Befehle. Die im folgenden aufgeführten Befehle beschreiben den eingebauten Befehlssatz. Dieser kann vom Benutzer bei Bedarf an die eigenen Vorstellungen angepasst werden: Es können für die einzelnen Befehle eigene mnemonische Codes festgelegt werden (max. 10 Zeichen, nur Buchstaben). Dazu verwendet man die optionale, aber mitgelieferte Datei mops.cfg. Wie man den MOPS mit einem eigenen Befehlssatz ausrüstet, wird in dieser Datei ebenfalls beschrieben. Der eingebaute Befehlssatz ist der folgende:

Tabelle 1: Befehlssatz des Pseudo-Assemblers

Befehl	**Code**	**Funktion**
ld *adr*	10	**load**: Lade den Wert an der Adresse *adr* in den Akku
ld *val*	11	load: Lade den Wert *val* in den Akku
st *adr*	12	**store**: Speichere den Wert des Akku an der Adresse *adr*
in *adr*	20	**input**: Schreibe den Wert des Eingaberegisters an die Adresse *adr*
out *adr*	22	**output**: Schreibe den Wert an der Adresse *adr* ins Ausgaberegister
out *val*	23	output: Schreibe den Wert *val* ins Ausgaberegister
add *adr*	30	**add**: Addiere den Wert an der Adresse *adr* zum Akku
add *val*	31	add: Addiere den Wert *val* zum Akku
sub *adr*	32	**subtract**: Subtrahiere den Wert an der Adresse *adr* vom Akku
sub *val*	33	subtract: Subtrahiere den Wert *val* vom Akku
mul *adr*	34	**multiply**: Multipliziere den Wert an der Adresse *adr* mit dem Akku
mul *val*	35	multiply: Multipliziere den Wert *val* mit dem Akku
div *adr*	36	**divide**: Dividiere den Akku durch den Wert an der Adresse *adr*
div *val*	37	divide: Dividiere den Akku durch den Wert *val* (nur ganzzahliger Teil)
mod *adr*	38	**modulo**: Rest bei Division des Akku durch den Wert an der Adresse *adr*
mod *val*	39	modulo: Rest bei Division des Akku durch den Wert *val*
cmp *adr*	40	**compare**: Vergleiche den Akkuinhalt mit dem Wert an der Adresse *adr*
cmp *val*	41	compare: Vergleiche den Akkuinhalt mit dem Wert *val*
jmp *tar*	50	**jump**: Springe zum Zielpunkt *tar* (Zeilennummer oder Marke)
jlt *tar*	52	**jump if less than**: Springe ..., wenn bei cmp der Akkuinhalt kleiner war
jeq *tar*	54	**jump if equal**: Springe ..., wenn bei cmp der Akkuinhalt gleich war
jgt *tar*	56	**jump if greater than**: Springe ..., wenn bei cmp der Akkuinhalt größer war
end	60	**end**: Beendet ein Programm