Projekt „Core Wars“

Ziel

- In Core Wars geht es darum, zwei – in einer Assembler ähnlichen Sprache geschriebene – Programme gegeneinander antreten zu lassen
- Ziel ist die Vernichtung eines der Programme

Ablauf

- Das Spiel besteht aus einem Server, welcher die „Assemblersprache“ der gegeneinander antretenden Programme Interpretiert.
- Abwechselnd werden jeweils eine Anweisung der beiden Programme ausgeführt.
- Nach jeder Anweisung wird der Programm-Counter des Cores inkrementiert.
- Verloren hat das Programm, das als erstes auf eine nicht ausführbare Anweisung trifft.

Server

- Der Server verwaltet einen gemeinsamen Speicherbereich für die interpretierten Programme.
  -> Überlegen Sie sich, welche Informationen der Speicher haben kann
- Der Speicherbereich ist zusammenhängend, d.h wird etwas über die Maximalgröße hinaus adressiert, fängt er vorne wieder an --> Tip: Modulo rechnen!
- Grundzustand des Speichers ist überall DAT #0
- Vor dem laden eines Programms (Core) ist eine Zufalls-Adresse zu ermitteln, an der das Programm im Speicher anfängt.

Aufbau des Assembler-Programms

- Das Assembler-Programm hat beispielsweise folgenden Aufbau:

```
mov 0,1
```

- Hier wird der gesamte Inhalt von Speicheradresse 1 auf Speicheradresse 0 kopiert.

```
add #4,5
```

- Mit add wird der Inhalt von Argument A zur Adresse von Argument B hinzuaddiert – also hier wird 4 zur Fünft-Nächsten Speicheradresse addiert.

Aufbau des Assembler-Programms

- Zu einer bestimmten Speicheradresse kann mit einem Jump-Befehl gesprungen werden:

  ```
  jmp -200
  ```

  Progamm-Sprünge sind grundsätzlich relativ, d.h hier wird von der aktuellen Speicheradresse aus um 200 zurück gesprungen.

Labels

- Labels legen einen Namen für eine Variable an

```
Ptr    DAT    #0
```

  über ptr ist nun die Variable ansprechbar (Wert ist 0)

- Mit Labels kann eine bestimmte Adresse angesprungen werden:

```
    DAT #0
P1  add #1,-1
    jmp P1
```

```
DAT #0
add #1,-1
jmp -1
```

  Die zwei Programme bewirken das gleiche. Hier wird endlos eine Variable um 1 erhöht

Adressierungsarten

- Zahl - direkt
  Bsp: `MOV Quelle Ziel`
  Kopiert den Inhalt der Adresse Quelle zur Adresse Ziel
- #Zahl - unmittelbar
  Bsp: `MOV #5 Ziel`
  füllt den Inhalt der Adresse Ziel mit der Zahl 5
- @Zahl - indirekt
  Bsp: `MOV #5 @Ziel`
  füllt den Inhalt der in der Adresse Ziel angegebene Adresse mit der Zahl 5
- <Zahl - dekrementierend und Indirekt
  Bsp: `MOV #5 <Ziel`
  Hier wird zunächst die Zieladresse um 1 erniedrigt, anschließend ein MOV #5 @Ziel.

Assembler-Operanden

- DAT
  Bsp: `Var DAT #-1`
  Erstellt eine Variable mit dem Label Var und dem Inhalt -1
- MOV
  Bsp: `MOV #5 Ziel`
  füllt den Inhalt der Adresse Ziel mit der Zahl 5
- ADD
  Bsp: `ADD #5 @Ziel`
  Addiert den Wert 5 zum Inhalt der angegebenen Adresse „Ziel“
- SUB
  Bsp: `SUB #5 @Ziel`
  Subtrahiert den Wert 5 vom Inhalt der angegebenen Adresse „Ziel“

Assembler-Operanden

- JMP
  Bsp: `JMP -2`
  Springt zwei Programmzeilen zurück
- JMZ
  Bsp: `JMZ -5 ptr`
  Spring 5 Programmzeilen zurück, wenn der Inhalt von ptr Null ist
- JMG
  Bsp: `JMG ptr @var`
  Springe nach „ptr“ falls der Inhalt von „var“ ungleich Null ist
- DJN
  Bsp: `DJN ptr @var`
  Dekrementiere den Inhalt von var um 1 und springe nach ptr falls der Inhalt von var noch nicht Null ist

Assembler-Operanden

- CMP
  Bsp: `CMP @var #42`
  Vergleiche den Inhalt von „var“ mit 42 und überspringe die nächste Instruktion, falls die Werte **nicht** übereinstimmen

Assembler - Beispiel:

```
ptr     DAT     #0          ; legt eine „Variable“ ptr mit dem Wert 0 an
start   ADD     #5, ptr     ; addiert 5 in den Speicherbereich „ptr“
        MOV     #0, @ptr    ; kopiert füllt den Inhalt in der Adresse „ptr“
                            ; mit 0
        JMP     start       ; springt an die Stelle mit dem Label „start“
```

Assembler

Objekt-Code

Was ist Objekt-Code?:

„Objekt-Code besteht aus einer folge von Bits“ (bei uns Integern)

Objekt-Code = { Instruktionen }

Instruktion = { DAT Value |
MOV Operands |
ADD Operands |
SUB Operands |
JMP Operand |
JMZ Operands |
JMG Operands |
DJN Operands |
CMP Operands }

Operands = Operand | Operand

Operand = Adressing Mode Value

Adressing Mode = Immediate | Direct | Indirect | Decrement

Was ist Objekt-Code?:

„Objekt-Code besteht aus einer folge von Bits“

Objekt-Code = { Instruktionen }

Instruktion = { DAT Value |
MOV Operands |
ADD Operands |
SUB Operands |
JMP Operands |
JMZ Operands |
JMG Operands |
DJN Operands |
CMP Operands }

DAT=0
MOV=1
ADD=2
SUB=3
....
CMP=8

Operands = Operand | Operand

Operand = Adressing Mode Value

Imediate = 0
Direct = 1
Indirect = 2
Decrement = 3

Adressing Mode = Immediate | Direct | Indirect | Decrement

Zurück zum Beispiel:

```
ptr     DAT     #0          ; legt eine „Variable“ ptr mit dem Wert 0 an
start   ADD     #5, ptr     ; addiert 5 in den Speicherbereich „ptr“
        MOV     #0, @ptr    ; kopiert füllt den Inhalt in der Adresse „ptr“
                            ; mit 0
        JMP     start       ; springt an die Stelle mit dem Label „start“
```

Assembler

```
0   0
2   0   5   1   -1
1   0   0   2   -2
4   1   -2
```

Objekt-Code

Simulator:

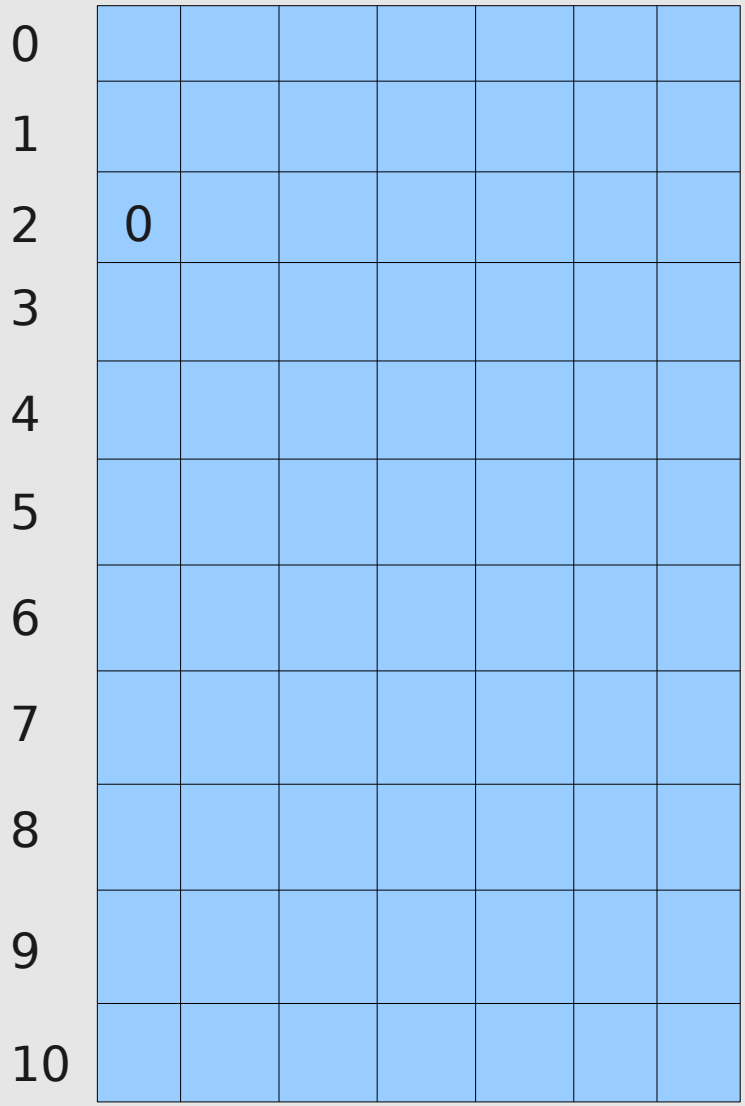
- Der Objekt-Code wird nun interpretiert:

```
ptr     DAT    #0
start   ADD    #5, ptr
        MOV    #0, @ptr

        JMP    start
```

```
0   0
2   0   5   1   -1
1   0   0   2   -2
4   1   -2
```

Cunter: 0



Simulator:

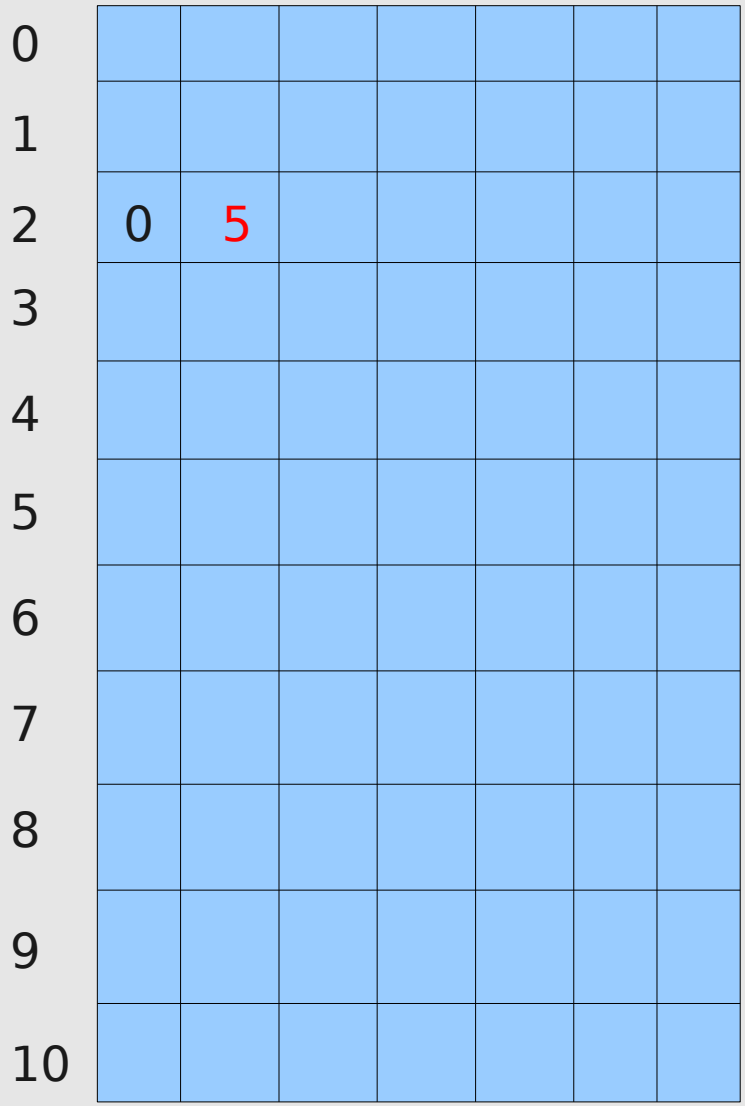
- Der Objekt-Code wird nun interpretiert:

```
ptr     DAT     #0
start   ADD     #5, ptr
        MOV     #0, @ptr

        JMP     start
```

```
0   0
2   0   5   1   -1
1   0   0   2   -2
4   1   -2
```

Cunter: 1



Simulator:

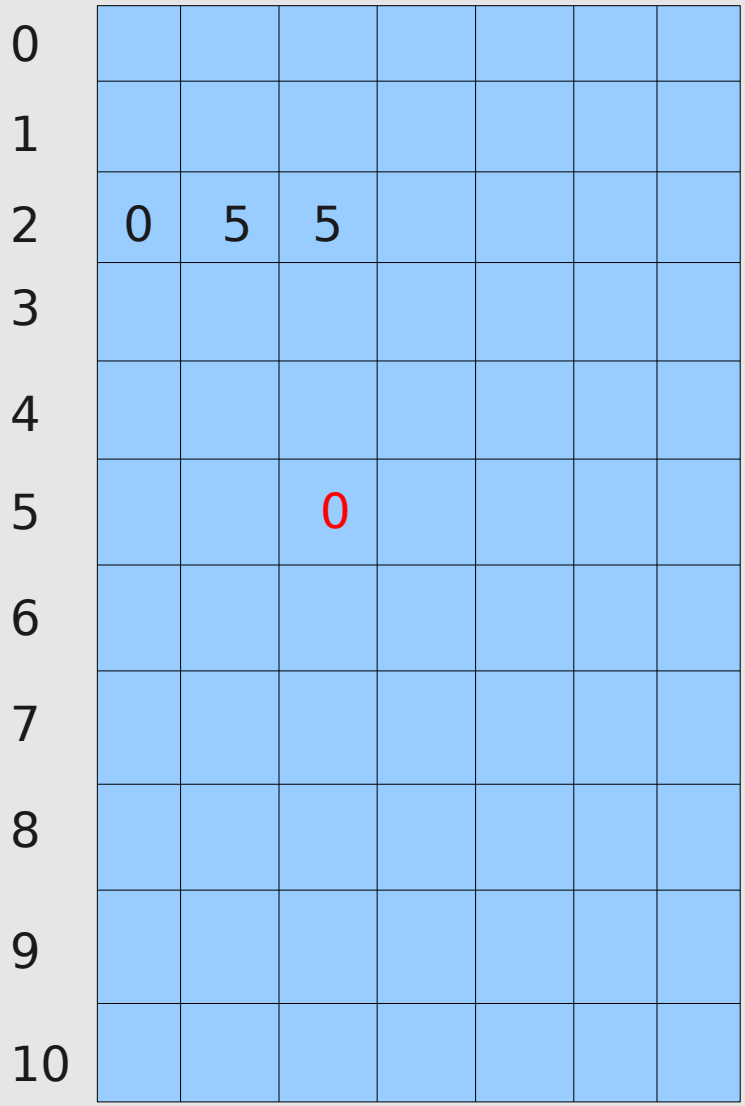
- Der Objekt-Code wird nun interpretiert:

```
ptr     DAT     #0
start   ADD     #5, ptr
        MOV     #0, @ptr

        JMP     start
```

```
0   0
2   0   5   1   -1
1   0   0   2   -2
4   1   -2
```

Cunter: 2



Simulator:

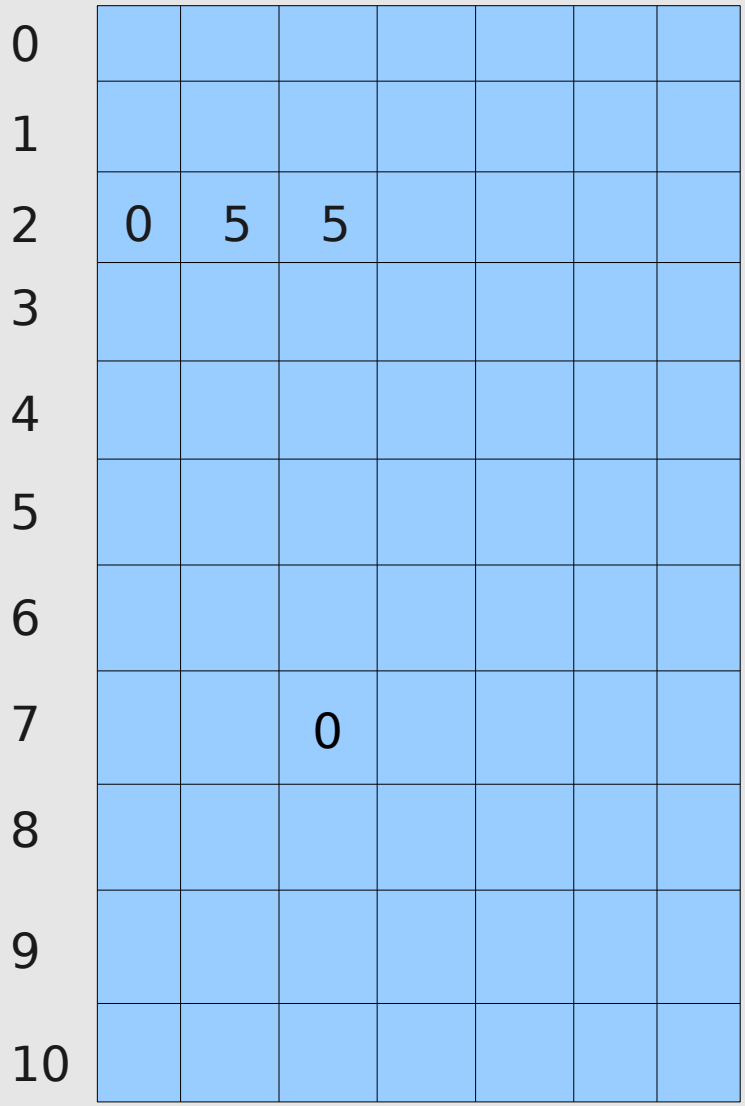
- Der Objekt-Code wird nun interpretiert:

```
ptr     DAT    #0
start   ADD    #5, ptr
        MOV    #0, @ptr

        JMP    start
```

```
0   0
2   0   5   1   -1
1   0   0   2   -2
4   1   -2
```

Cunter: 3



Simulator:

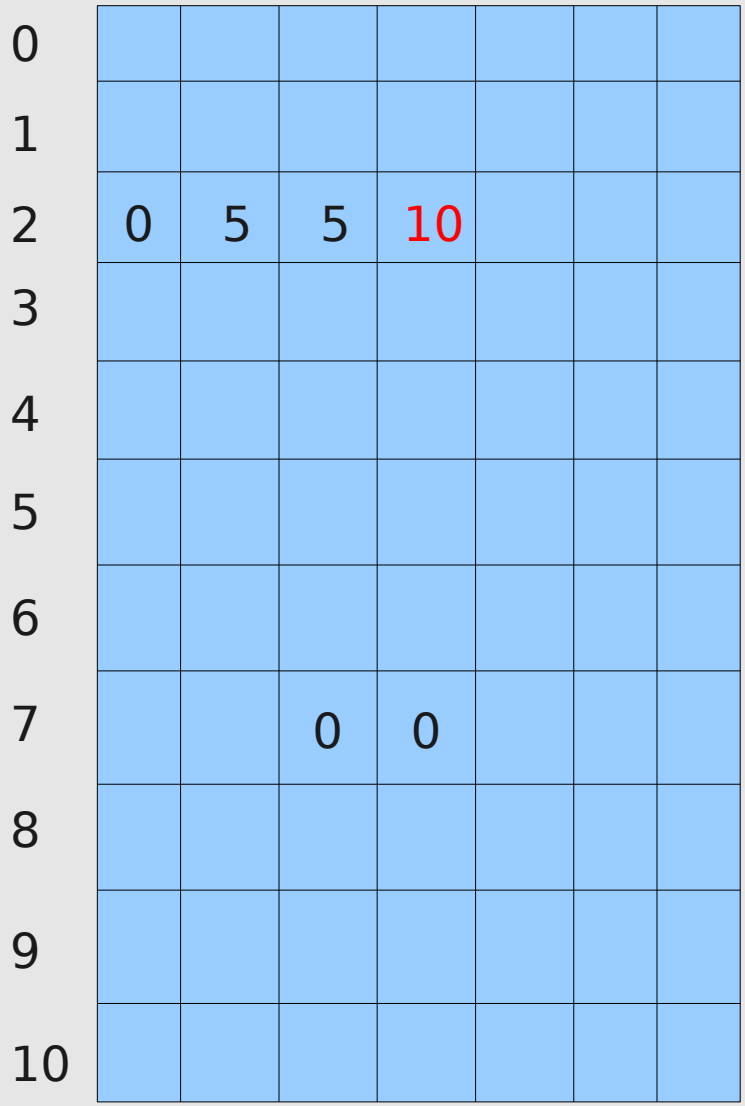
- Der Objekt-Code wird nun interpretiert:

```
ptr     DAT     #0
start   ADD     #5, ptr
        MOV     #0, @ptr

        JMP     start
```

```
0   0
2   0   5   1   -1
1   0   0   2   -2
4   1   -2
```

Cunter: 4



Simulator:

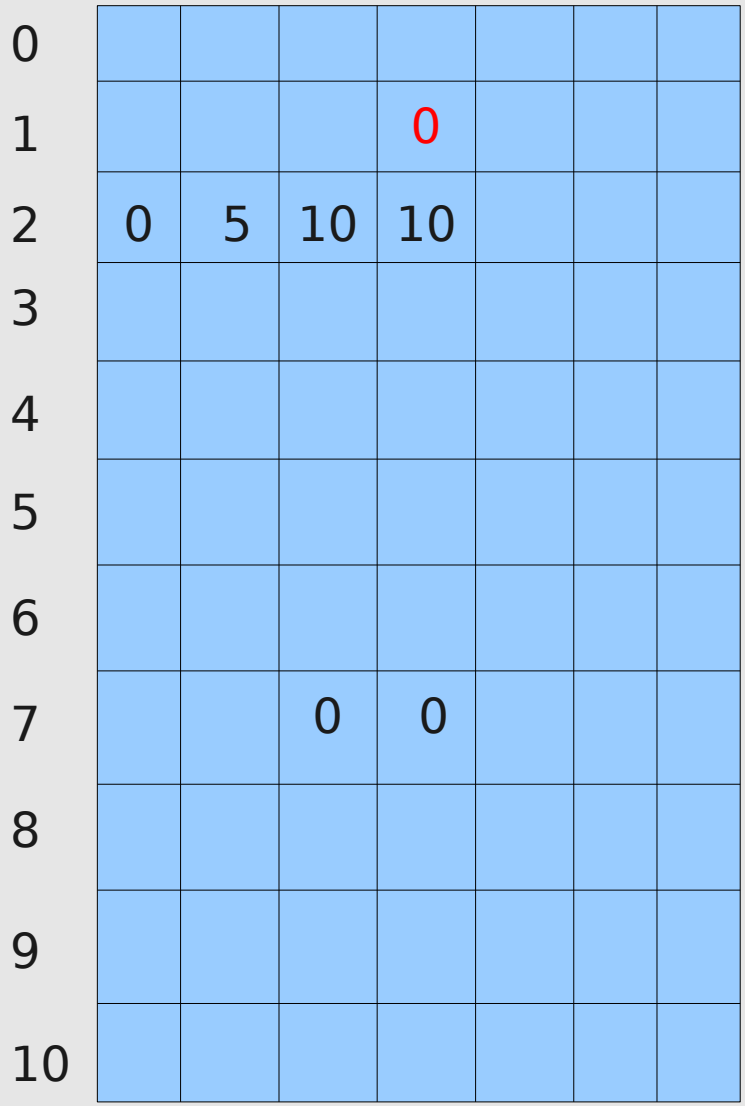
- Der Objekt-Code wird nun interpretiert:

```
ptr     DAT     #0
start   ADD     #5, ptr
        MOV     #0, @ptr

        JMP     start
```

```
0   0
2   0   5   1   -1
1   0   0   2   -2
4   1   -2
```

Cunter: 5



Simulator:

- Der Objekt-Code wird nun interpretiert:

```
ptr     DAT    #0
start   ADD    #5, ptr
        MOV    #0, @ptr

        JMP    start
```

```
0   0
2   0   5   1   -1
1   0   0   2   -2
4   1   -2
```

Cunter: 6

0							
1				0			
2	0	5	10	10			
3							
4							
5							
6							
7			0	0			
8							
9							
10							

Simulator:

- Der Objekt-Code wird nun interpretiert:

```
ptr     DAT    #0
start   ADD    #5, ptr
        MOV    #0, @ptr

        JMP    start
```

```
0   0
2   0   5   1   -1
1   0   0   2   -2
4   1   -2
```

Cunter: 7