

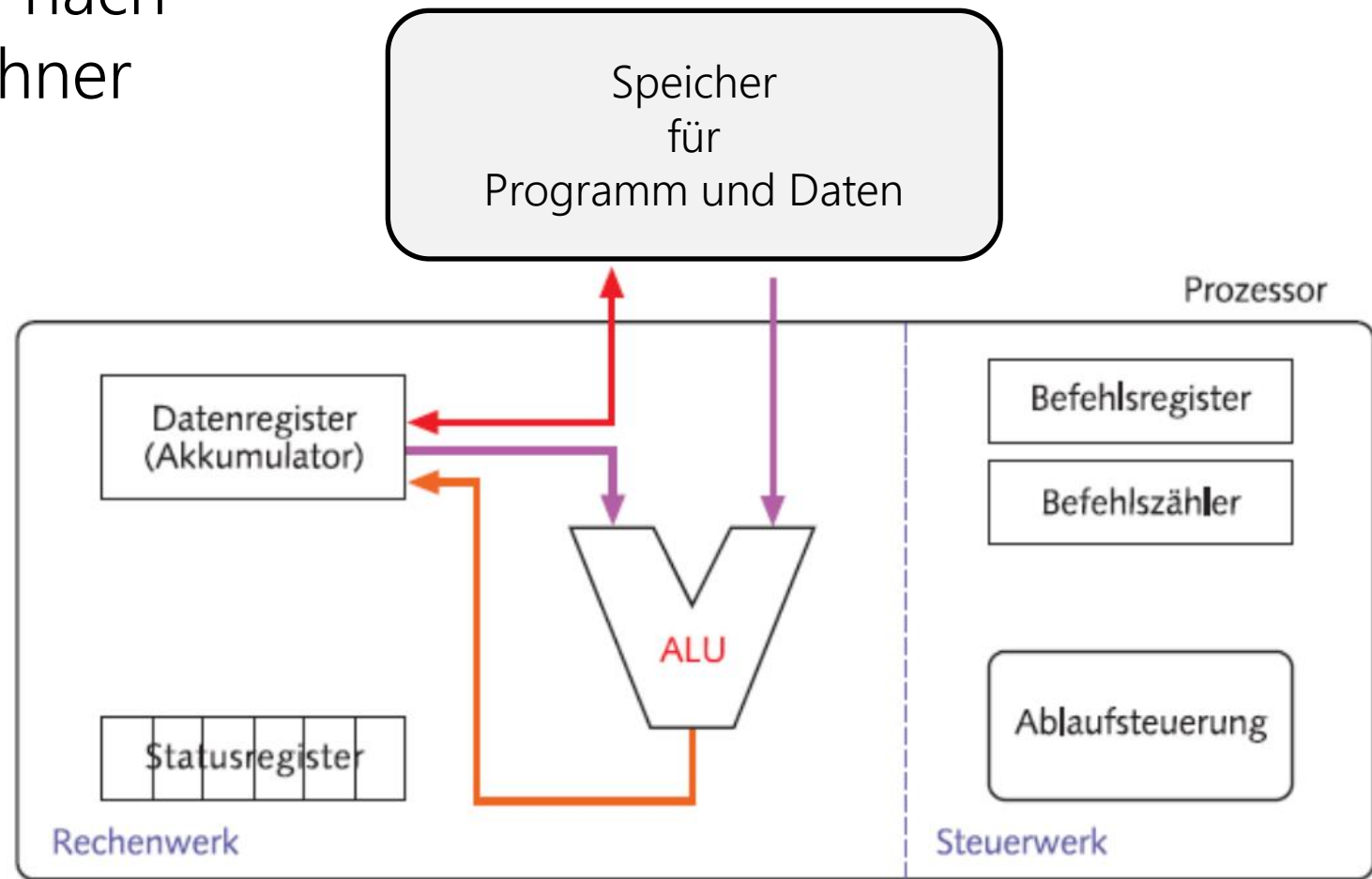


Maschinennahe Programmierung (Assembler)



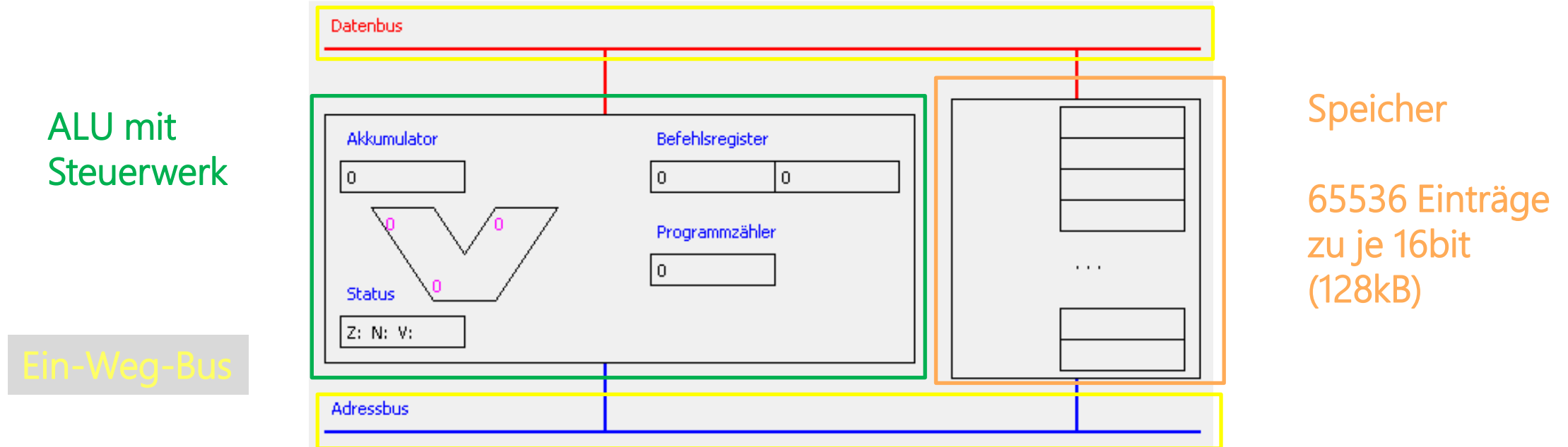
Prinzip einer Registermaschine

- Prinzipieller Aufbau nach Von-Neumann-Rechner



Aufbau der 16-Bit-Minimaschine (Simulator)

- 16 Bit im Zweierkomplement -> Zahlen von -32768 bis +32767



Die Minimaschine

Download für alle Betriebssysteme: <https://schule.awiedemann.de/minimaschine.html>

The screenshot displays the Minimaschine software interface, which is divided into several windows and panels.

CPU-Kontrolle Window:

- Datenbus:** A red horizontal line representing the data bus.
- Adressbus:** A blue horizontal line representing the address bus.
- Akkumulator:** A box containing the value 55.
- Status:** A box containing the flags Z, N, and V.
- Befehlsregister:** A box containing the instruction HOLD and the value 0.
- Programmzähler:** A box containing the value 6.
- Memory Stack:** A vertical list of memory addresses and their contents:
 - 4: 99
 - 5: 0
 - 6: 0
 - 7: 0
 - ...
 - 12: 55
 - 13: 0
- Buttons:** Ausführen, Einzelschritt, and Mikroschritt.

Speicheranzeige Window:

- Table:** A table showing memory addresses (0 to 50) and their contents (0 to 99). The value 55 is highlighted in red at address 10.

Editor Window:

- Code:** LOADI 55, STORE 12, HOLD.

Aufgabe

- a) Öffne die Minimaschine. Es erschienen zunächst zwei Fenster: CPU-Kontrolle und Speicheranzeige. Öffne im Fenster CPU-Kontrolle über das Hauptmenü: Ablage → Neu ein Editor-Fenster.
- b) Schreibe darin in der Mini-Sprache folgendes Programm:
 - ```
PROGRAM Addition;
VAR x, y, z;
BEGIN
 x := 3;
 y := 2;
 z := x + y;
END Addition.
```
  - Speichere dieses Programm unter dem Namen Aufgabe\_01.mis.

# Aufgabe

- c) Übersetze dieses Programm (Editor-Fenster, Hauptmenü: Werkzeuge → übersetzen). Lasse dir nun das entsprechende Maschinen-Programm anzeigen (Editor-Fenster, Hauptmenü: Werkzeuge → Assemblertext zeigen).
  - Speichere es unter dem Namen Aufgabe\_01.mia.
  - Versuche den Maschinen-Code zu verstehen. Was bedeuten die einzelnen Befehle?
  
- d) Lade das Maschinen-Programm nun in die Minimaschine (Editor-Fenster, Hauptmenü: Werkzeuge → Assemblieren) und betrachte das Fenster Speicheranzeige.
  - Was steht in den einzelnen Speicherzellen? Erkennst du irgendeine Logik?

# Aufgabe

- e) Führe nun das Assembler-Programm in Einzelschritten aus und betrachte bei jedem Schritt genau, was sich in den Fenstern CPU-Kontrolle und Speicheranzeige ändert.
  - Kannst du eine Logik erkennen?
- f) Assembliere das Programm erneut (Editor-Fenster, Hauptmenü: Werkzeuge → Assemblieren) und führe diesmal Mikroschritte aus und betrachte wieder die Veränderungen in den Fenstern CPU-Kontrolle und Speicheranzeige.
  - Kannst du weitere Erkenntnisse gewinnen.

# Arbeitsweise der Minimaschine

- Die Minimaschine arbeitet ähnlich wie der von-Neumann-Zyklus:
  - FETCH – Befehlsabruf: Aus dem Speicher wird der nächste zu bearbeitende Befehl entsprechend der Adresse im Befehlszähler in das Befehlsregister geladen und der Befehlszähler wird um die Länge des Befehls erhöht.
  - FETCH OPERANDS – Operandenabruf: Aus dem Speicher werden nun die Operanden geholt. Das sind die Werte, die durch den Befehl verändert werden sollen oder die als Parameter verwendet werden.
  - DECODE – Dekodierung: Der Befehl wird durch das Steuerwerk in Schaltinstruktionen für das Rechenwerk aufgelöst.
  - EXECUTE – Befehlsausführung: Eine arithmetische oder logische Operation wird vom Rechenwerk ausgeführt. Bei Sprungbefehlen und erfüllter Sprungbedingung wird an dieser Stelle der Befehlszähler verändert.
  - WRITE BACK – Rückschreiben des Resultats: Sofern notwendig, wird das Ergebnis der Berechnung in den Speicher zurückgeschrieben.



# Struktur der Befehle in der Minimaschine

- Ein Befehl besitzt ebenfalls eine Größe von 16 Bit, obwohl es nicht annähernd 65536 viele Befehle existieren.

**ADD**

**15**

Assembler

Befehlsteil

Werteteil

0000 0001 0000 1010

0000 0000 0000 1111

Maschinencode

01

0A

00

0F

Hex-Format

4 Byte

Größe

- 0000 0001 0000 1010 kann je nach Interpretation der Befehl ADD oder die Zahl 266 oder die Speicheradresse 266 sein.

# Assemblerbefehle der Minimaschine

- Bei vielen Assemblerbefehlen der Minimaschine gibt es zwei Versionen desselben Befehls: einer mit I und einer ohne I.
- Beim Befehl **mit I** wird die konkrete Zahl im Operanden angegeben
- Beim Befehl **ohne I** ist die Speicheradresse gemeint und es wird die dort hinterlegte Zahl benutzt

## Speicherbefehle

|                             |                                                                                                          |
|-----------------------------|----------------------------------------------------------------------------------------------------------|
| <b>LOAD</b> <i>adresse</i>  | Lädt den Wert von der angegebenen Adresse in den Akkumulator.                                            |
| <b>LOADI</b> <i>zahl</i>    | Lädt die angegebenen Zahl in den Akkumulator, negative Werte sind möglich, Adressen sind nicht zulässig. |
| <b>STORE</b> <i>adresse</i> | Speichert den Wert im Akkumulator an der angegebenen Adresse.                                            |

# Assemblerbefehle der Minimaschine

## *Arithmetikbefehle*

|                    |                                                                                                                 |
|--------------------|-----------------------------------------------------------------------------------------------------------------|
| <b>ADD</b> adresse | Addiert den Wert von der angegebenen Adresse zum Akkumulator.                                                   |
| <b>SUB</b> adresse | Subtrahiert den Wert der angegebenen Adresse vom Akkumulator.                                                   |
| <b>MUL</b> adresse | Multipliziert den Wert von der angegebenen Adresse zum Akkumulator.                                             |
| <b>DIV</b> adresse | Dividiert den Wert im Akkumulator durch den Wert der angegebenen Adresse.                                       |
| <b>MOD</b> adresse | Dividiert den Wert im Akkumulator durch den Wert der angegebenen Adresse und speichert den Rest im Akkumulator. |
| <b>CMP</b> adresse | Vergleicht den Wert der angegebenen Adresse mit dem Akkumulator und setzt Null- und Negativflag entsprechend.   |
| <b>ADDI</b> zahl   | Addiert den angegebenen Wert zum Akkumulator.                                                                   |
| <b>SUBI</b> zahl   | Subtrahiert den angegebenen Wert vom Akkumulator.                                                               |
| <b>MULI</b> zahl   | Multipliziert den angegebenen Wert zum Akkumulator.                                                             |
| <b>DIVI</b> zahl   | Dividiert den Wert im Akkumulator durch den angegebenen Wert.                                                   |
| <b>MODI</b> zahl   | Dividiert den Wert im Akkumulator durch den angegebenen Wert und speichert den Rest im Akkumulator.             |
| <b>CMPI</b> zahl   | Vergleicht den angegebenen Wert mit dem Akkumulator und setzt Null- und Negativflag entsprechend.               |

# Aufbau des Speichers der Minimaschine

- Speicher
  - Adressen von 0 bis 65535
  - Jede Zelle nimmt zwei Byte (=16 Bit) auf
  - Variablen können definiert werden und zeigen auf Speicheradressen
  - Zuteilung der Zelle an Variablen durch Assemblierung (=Programmierung)
- Die Minimaschine arbeitet rein dual. Zur Vereinfachung werden in der Minimaschine alle Zahldarstellungen dezimal vorgenommen.
- Auch die Befehle der Assemblersprache werden durch Zahlen dargestellt, die nach dem Einlesen erst interpretiert werden müssen.
  - Um die Arbeit mit komplexeren Programmen zu erleichtern, stehen symbolische Adressen (=Marken) zur Verfügung.  
Beispiel: Variable/Symbol **PLZ** entspricht der Speicherzelle 21

# Marken (Variablen und Sprungadressen)

- Marken (Variablen und Sprungadressen)
  - Festlegung von Marken für Adressen mit **Markenname** :
  - Assembler ersetzt Markennamen durch Adresse
  - Der Befehl **WORD** besetzt eine Speicherzelle mit der angegebenen Zahl, negative Werte sind möglich.

**X** :      **WORD**      17

|            |            |               |
|------------|------------|---------------|
| Markenname | Datentyp   | Direkter Wert |
|            | hier nur   | zu Beginn     |
|            | Doublebyte |               |

## ***Speicherorganisation***

|                  |                                                                                   |
|------------------|-----------------------------------------------------------------------------------|
| <b>WORD zahl</b> | Besetzt eine Speicherzelle mit der angegebenen Zahl, negative Werte sind möglich. |
|------------------|-----------------------------------------------------------------------------------|

# Speichernutzung eines Programms

- Da nach von-Neumann-Architektur das Programm und die Daten im selben Speicher liegen und Befehle nicht von Zahlen unterschieden werden können muss man mittels des Befehls **HOLD** angeben, wann das Programm „zu Ende“ ist. Den Variablen (=Marken die mit WORD einen Wert im Speicher erhalten) wird Speicher „nach“ dem Programm zugeteilt.

|    | 0   | 1  | 2   | 3  | 4   | 5 | 6   | 7  | 8   | 9  |
|----|-----|----|-----|----|-----|---|-----|----|-----|----|
| 0  | 532 | 3  | 277 | 16 | 532 | 2 | 277 | 17 | 276 | 16 |
| 10 | 266 | 17 | 277 | 18 | 99  | 0 | 3   | 2  | 5   | 0  |
| 20 | 0   | 0  | 0   | 0  | 0   | 0 | 0   | 0  | 0   | 0  |

↑  
HOLD

↑ x      ↑ y      ↑ z

## Sonstige Befehle

|              |                                                                                   |
|--------------|-----------------------------------------------------------------------------------|
| <b>HOLD</b>  | Hält den Prozessor an. Dieser Befehl hat keine Adresse.                           |
| <b>RESET</b> | Setzt den Prozessor auf den Startzustand zurück. Dieser Befehl hat keine Adresse. |
| <b>NOOP</b>  | Tut einfach nichts (NO OPERATION). Dieser Befehl hat keine Adresse.               |



# Grundlegender Aufbau eines Programms

- 1) Häufig müssen zu Beginn eines Programm Werte mittels der **L-Befehle** und **STORE** gespeichert werden.
- 2) Anschließend muss der **Algorithmus** bzw. das Programm umgesetzt werden.
- 3) Befehl **HOLD**
- 4) Variablen, die im Algorithmus verwendet werden, werden zum Schluss mittels Marken angegeben **Variable : WORD zahl**

# Aufgabe

Erstelle jeweils ein Assemblerprogramm für:

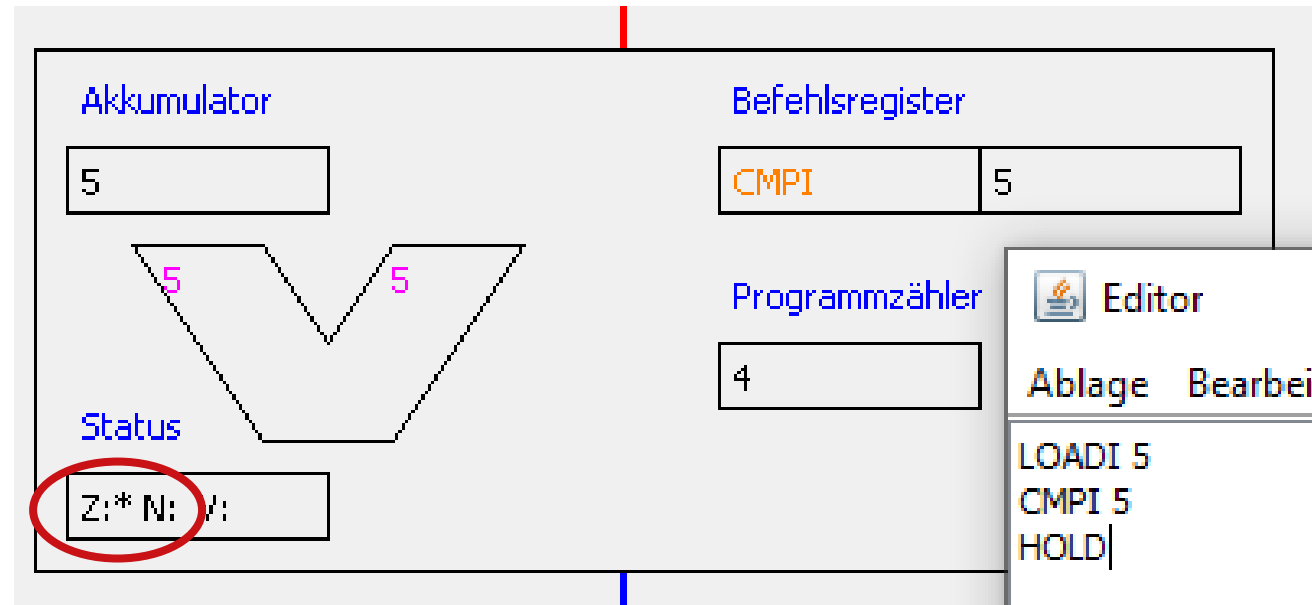
- Speichere die Zahl 2 in einer Variablen n, die zu Beginn 0 ist.
- Addiere die beiden Zahlen 2 und 3 und speichere das Ergebnis in einer Variablen erg.
- Überführe folgenden Java-Code in Assembler:
  - `int x, y;`
  - `x = 5;`
  - `y = 2;`
  - `x = x - 2;`
  - `x = x * y;`

# Status-Register

- Status-Register
  - *negative-flag*  
Die letzte Rechenoperation lieferte ein negatives Ergebnis.
  - *zero-flag*  
Das letzte Rechenergebnis lieferte Null als Ergebnis.
  - *overflow-flag*  
Das Ergebnis der letzten Rechenoperation überschritt den Zahlbereich von 16bit.  
Das letzte Ergebnis gibt somit (ohne entsprechende Sonderinterpretation) keinen Sinn.
  - Mit den Werten dieser Flags können wir im Folgenden Fallunterscheidungen oder Wiederholungen realisieren. Die Abfrage eines einzelnen Bits (boolean-Wert) kann nämlich sehr viel schneller erfolgen als das nachträgliche Interpretieren eines gespeicherten 16-Bit-Werts (gewöhnliche Speicherzelle).

# Flags und Vergleichsbefehle

- Die Befehle CMP und CMPI ziehen **nur intern** den Wert des Operanden vom Wert des Akkumulators ab. Falls beide Zahlen gleich groß sind kommt hier 0 heraus -> zero-flag wird gesetzt!



- Die flags können durch **Sprungbefehle** ausgelesen und genutzt werden, um die reine sequentielle Abarbeitung zu umgehen!

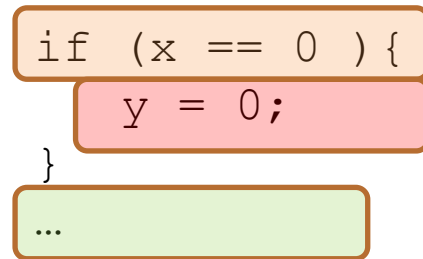
# Sprungbefehle und Sprungmarken

- Anstatt einer Speicheradresse kann auch (wie bei Variablen) eine **Marke** angegeben werden, um an eine andere Stelle im Algorithmus zu springen. Es können an beliebigen Stellen Marke gesetzt werden.

| <b>Sprungbefehle</b> |                                                                                                                                                        |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>JMPP adresse</b>  | Springt zur angegebenen Adresse, wenn das Ergebnis der letzten Operation positiv ( $> 0$ ) war, d. h. weder N noch Z-Flag sind gesetzt.                |
| <b>JMPNN adresse</b> | Springt zur angegebenen Adresse, wenn das Ergebnis der letzten Operation nicht negativ ( $\geq 0$ ) war, d. h. das N-Flag ist nicht gesetzt.           |
| <b>JMPN adresse</b>  | Springt zur angegebenen Adresse, wenn das Ergebnis der letzten Operation negativ ( $< 0$ ) war, d. h. das N-Flag ist gesetzt.                          |
| <b>JMPNP adresse</b> | Springt zur angegebenen Adresse, wenn das Ergebnis der letzten Operation nicht positiv ( $\leq 0$ ) war, d. h. das N-Flag oder das Z-Flag ist gesetzt. |
| <b>JMPZ adresse</b>  | Springt zur angegebenen Adresse, wenn das Ergebnis der letzten Operation null ( $= 0$ ) war, d. h. das Z-Flag ist gesetzt.                             |
| <b>JMPNZ adresse</b> | Springt zur angegebenen Adresse, wenn das Ergebnis der letzten Operation nicht null ( $\neq 0$ ) war, d. h. das Z-Flag ist nicht gesetzt.              |
| <b>JMPV adresse</b>  | Springt zur angegebenen Adresse, wenn die letzte Operation einen Überlauf verursacht hat, d. h. das V-Flag ist gesetzt.                                |
| <b>JMP adresse</b>   | Springt zur angegebenen Adresse.                                                                                                                       |

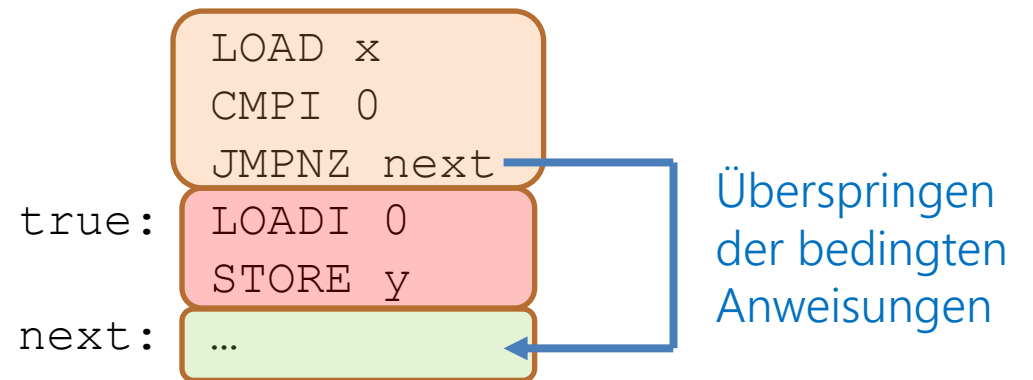
# Bedingte Anweisung in Assembler

- Bedingte Anweisungen in Assembler benötigen einen Sprungbefehl. Um den Sprungpunkt zu dem gesprungen werden soll kann hier an dem entsprechenden Stelle im Code eine Marke gesetzt werden.



Bedingung

Anweisungen  
nur bei  
Bedingung

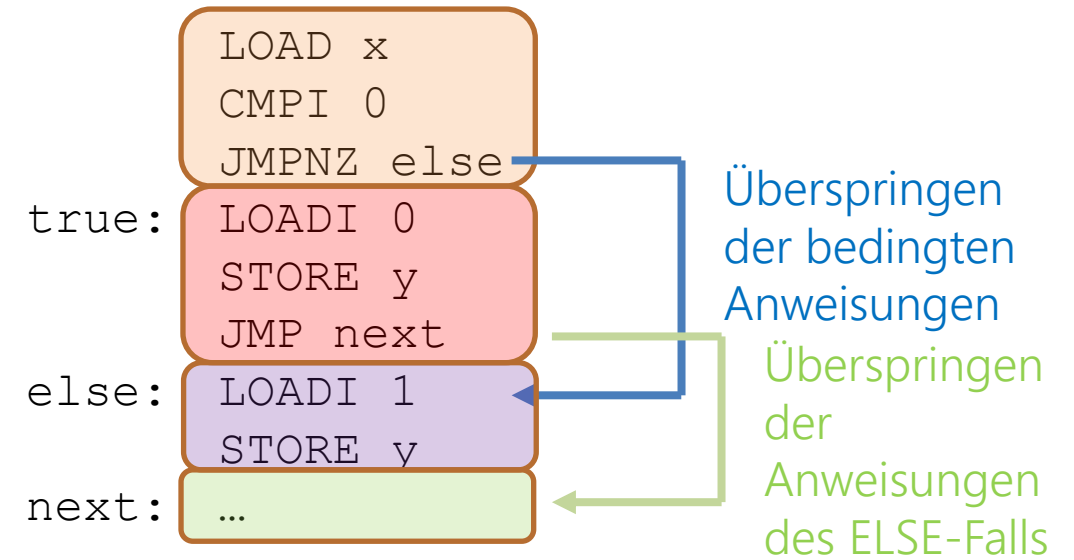
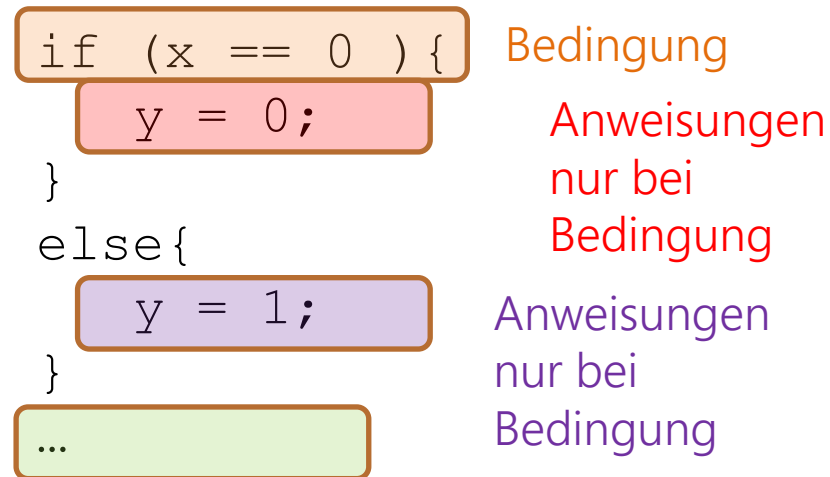


Überspringen  
der bedingten  
Anweisungen



# Fallunterscheidung in Assembler

- Grundstruktur einer Fallunterscheidung in Assembler



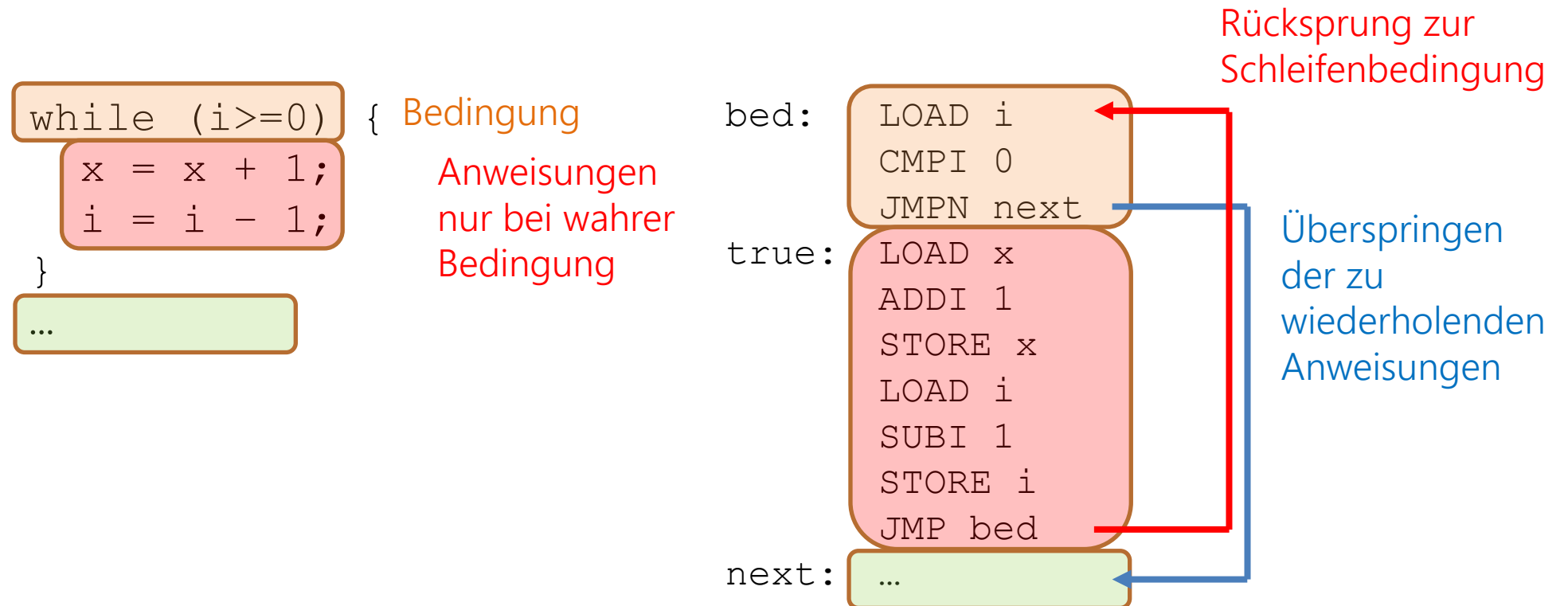
# Aufgabe

Erstelle jeweils ein Assemblerprogramm zur:

- Bei zwei gegebenen Variablen `n`, `m` (Werte dürfen selbst gewählt werden) soll das Programm überprüfen, ob `n` größer als `m` ist. Falls ja schreibe in eine Variable `erg` die Zahl 10, ansonsten die Zahl 20
- Bei einer gegebenen Variable `n` (Wert darf selbst gewählt werden) soll das Programm überprüfen, ob `n` gerade ist oder nicht. Falls ja schreibe in eine Variable `erg` die Zahl 10, ansonsten die Zahl 20

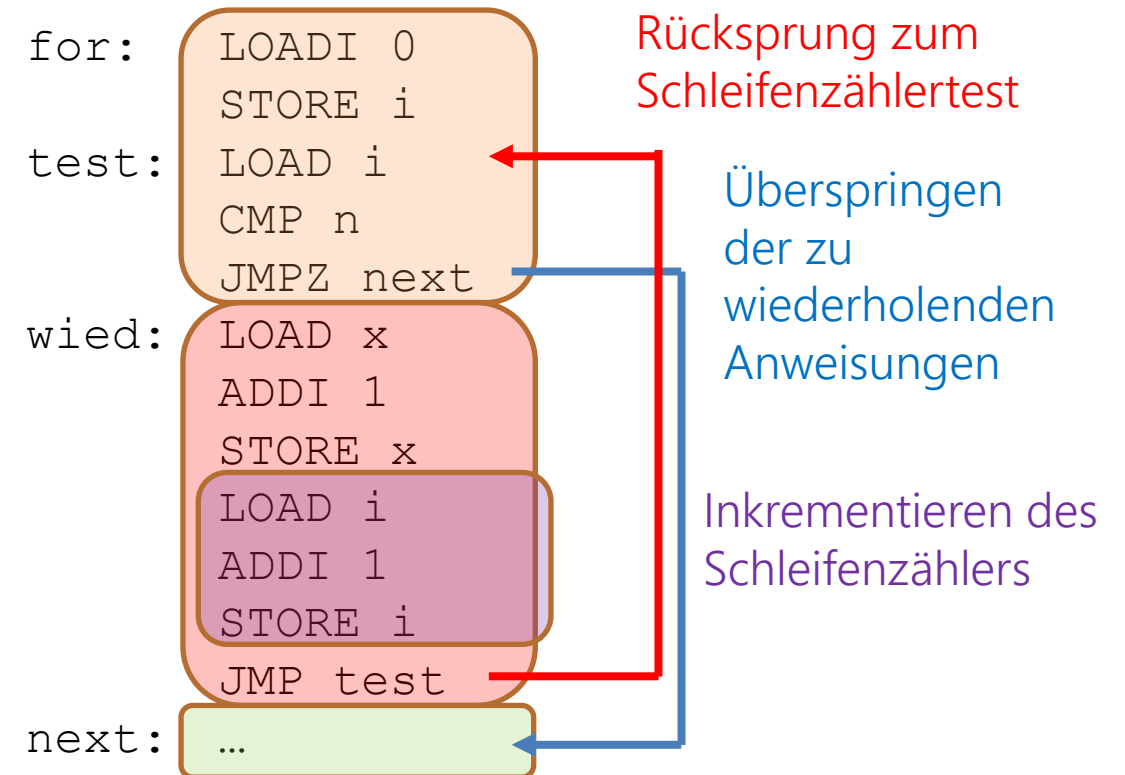
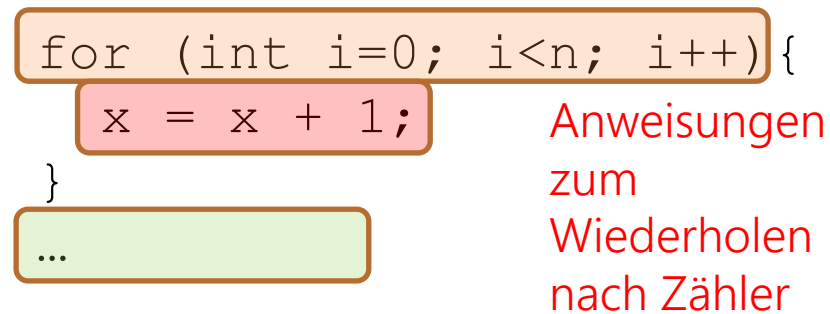
# Wiederholungen mit Bedingung

- Ähnlich zum Syntaxdiagramm muss bei der Wiederholungen mit Bedingung zu einer Marke zurückgesprungen werden



# Wiederholungen mit fester Anzahl

- Wiederholungen mit fester Anzahl funktionieren ganz ähnlich zu Wiederholung mit Bedingung



# Aufgabe

Erstelle jeweils ein Assemblerprogramm für:

- $n$  ist eine Variable mit selbstgewähltem Wert. Berechne die Summe der Zahlen von 1 bis  $n$
- $a, n$  sind Variablen mit selbstgewählten (positiven) Werten. Berechne die Potenz  $a^n$ .
- $a, b$  sind Variablen mit selbstgewählten (positiven) Werten. Berechne den größten gemeinsamen Teiler von  $a$  und  $b$  (euklidischer Algorithmus -> Falls unbekannt, siehe Wikipedia)

# Aufgabe

- Assembleraufgaben kommen nahezu ausnahmslos immer beiden Teilen III. und IV. des Informatikabiturs vor! Daher die Vorbereitung in Assembler-Programmierung sehr wichtig!
- Löse die Aufgaben des **letzten** Informatikabiturs in den Teilen III. und IV. zum Thema Assembler/Assembler-Programmierung!
- Die Informatikabiture sind zu finden unter mebis -> Prüfungsarchiv

## Zusatz: Indirekte Adressierung

- Der Befehl **LOAD (x)** ermöglicht die indirekte Adressierung, kopiert den Wert aus derjenigen Speicherzelle in den Akkumulator, deren Adresse in der Speicherzelle x steht.
- => In der Speicherzelle 50 ist die Zahl 7 gespeichert. Die Variable x hat die Speicheradresse 20 und in dieser die Zahl 50.
- LOAD x lädt – wie bekannt – den Wert der Variablen, also 50, in den Akkumulator.
- LOAD (x) lädt 7 in den Akkumulator.



# Zusatz: Indirekte Adressierung

Datenbus

200

Mikroschritt: exec\_1

Akkumulator

200

Befehlsregister

LOAD 50

Programmzähler

6

Status

Z: N: V:

Adressbus

Editor

Ablage Bearbeiten Werkzeuge Fenster

1 LOADI 200

2 STORE 50

3 LOAD (x)

4 HOLD

5 x: WORD 50

|     | 0   | 1   | 2   | 3  | 4   | 5 | 6  | 7 | 8  | 9 |
|-----|-----|-----|-----|----|-----|---|----|---|----|---|
| 0   | 532 | 200 | 277 | 50 | 788 | 8 | 99 | 0 | 50 | 0 |
| 10  | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 20  | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 30  | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 40  | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 50  | 200 | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 60  | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 70  | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 80  | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 90  | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 100 | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 110 | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 120 | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 130 | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 140 | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |
| 150 | 0   | 0   | 0   | 0  | 0   | 0 | 0  | 0 | 0  | 0 |

# Zusatzaufgabe

- Schreibe ein Assemblerprogramm, welches eine gegebene **Binärzahl mit 8 Bit in eine Dezimalzahl** umwandelt und zurückgibt.
- Hinweise: Zu Beginn des Programms ist die Binärzahl mit je einer Ziffer in den Zellen 200 bis 207 gespeichert. Beispiel für die Dezimalzahl 21:

|            | <b>0</b> | <b>1</b> | <b>2</b> | <b>3</b> | <b>4</b> | <b>5</b> | <b>6</b> | <b>7</b> |
|------------|----------|----------|----------|----------|----------|----------|----------|----------|
| <b>200</b> | 0        | 0        | 0        | 1        | 0        | 1        | 0        | 1        |

- Die Rückgabe der Dezimalzahl wird simuliert durch das Speichern des Werts in der Variable ergebnis.