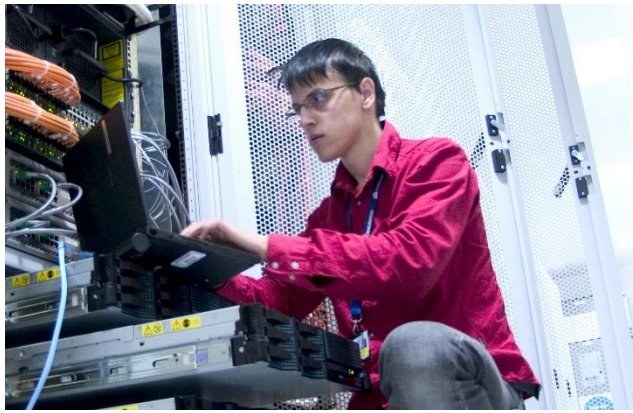
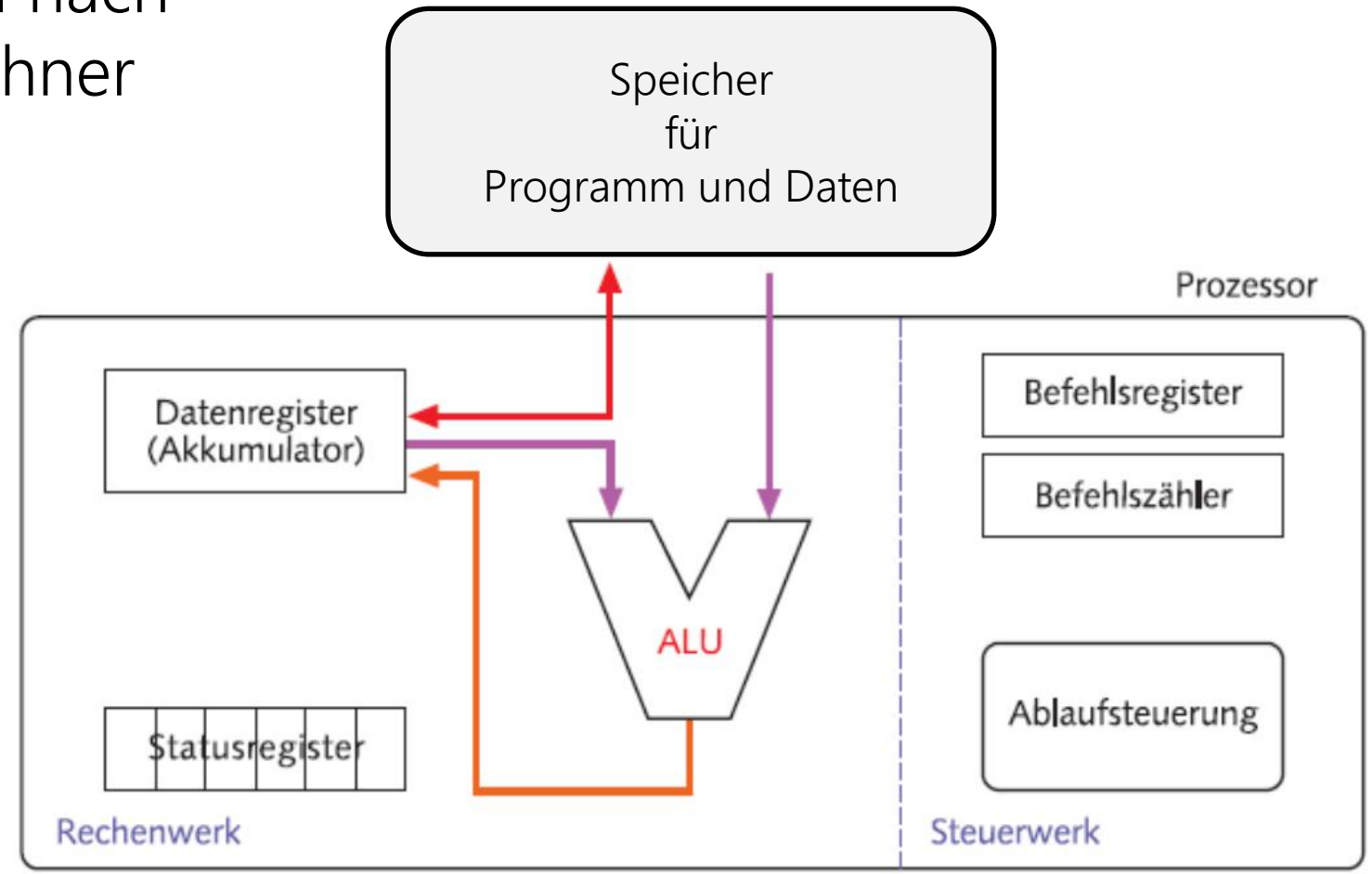
EMIL-VON-BEHRING GYMNASIUM

Maschinennahe Programmierung (Assembler)



Prinzip einer Registermaschine

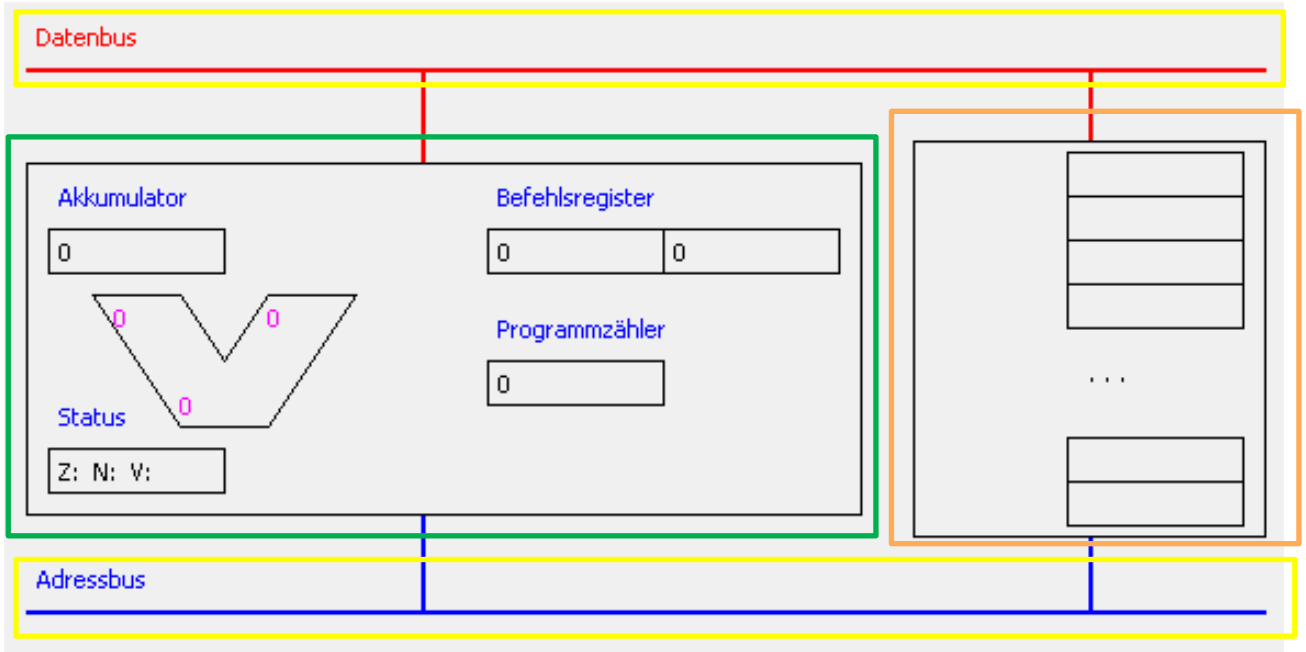
- Prinzipieller Aufbau nach Von-Neumann-Rechner



Aufbau der 16-Bit-Minimaschine (Simulator)

ALU mit Steuerwerk

Ein-Weg-Bus



Speicher

16-Bit-Adressraum

65536 Einträge/ Speicherzellen zu je 16bit (=128kB)

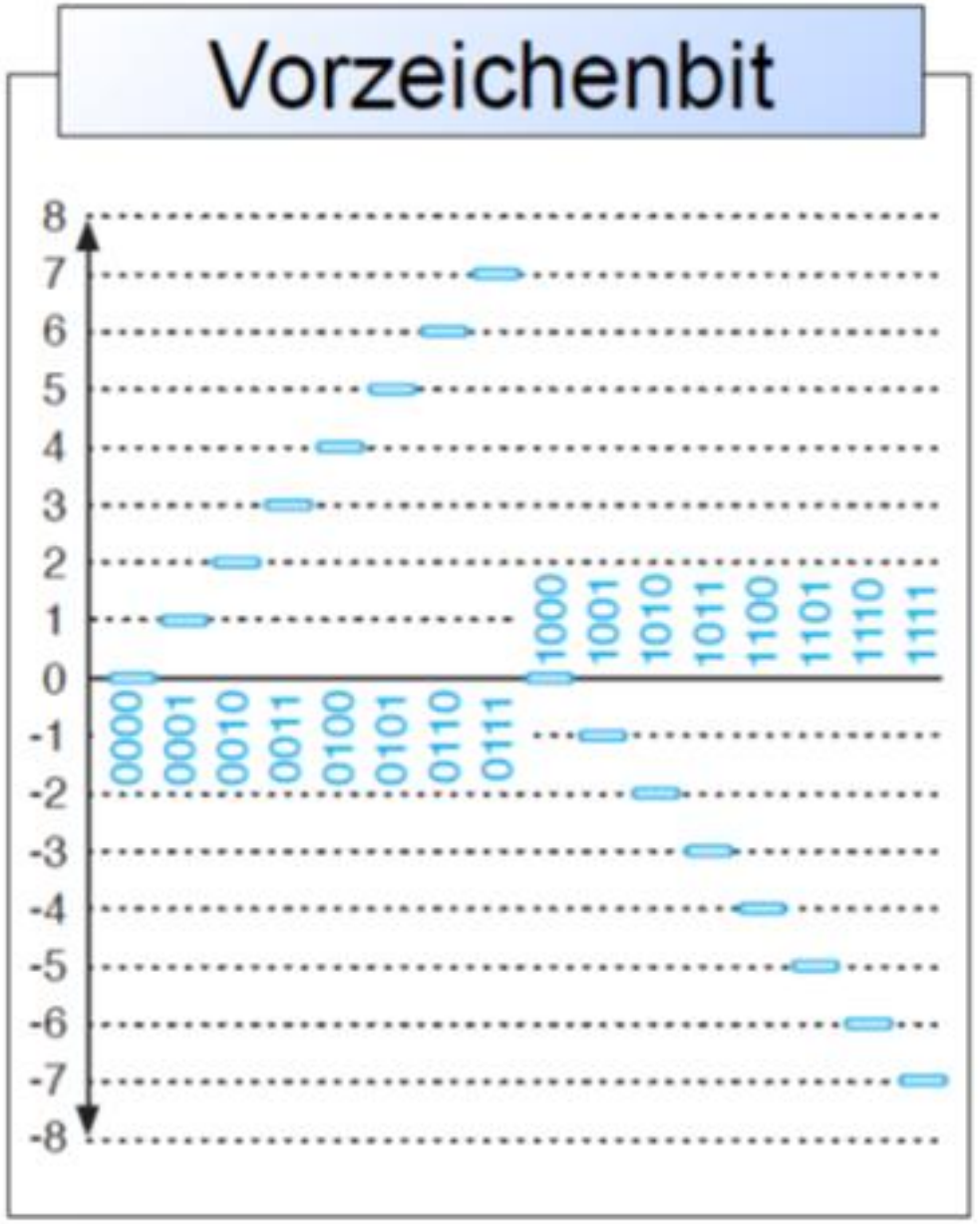
- 16 Bit → $\mathbf{2^{16}}$ = 65536
- → Codierung 11. Jgst: Dezimalzahl von 0 bis 65535 darstellbar
- Was ist mit negativen Zahlen?

Einschub: Codierung negativer Zahlen

- Man benötigt für die Information positiv oder negativ genau ein 1 Bit, weil es genau zwei „Zustände“ gibt, positiv oder negativ. Das erste Bit soll nun angeben, ob die Zahl positiv oder negativ ist. Ist das erste Bit 0, dann ist die Zahl positiv. Ist das erste Bit 1, dann ist die Zahl negativ.
- Hieraus ergeben sind drei/mehrere Varianten wie man die negativen Zahlen kodieren kann:
  - Vorzeichenbit
    Vorzeichen wird durch ein einzelnes Bit definiert
  - Einerkomplement
    Negative Zahlen werden durch das Invertieren aller Bits gebildet
  - Zweierkomplement
    Negative Zahlen: Invertieren und Addieren von 1

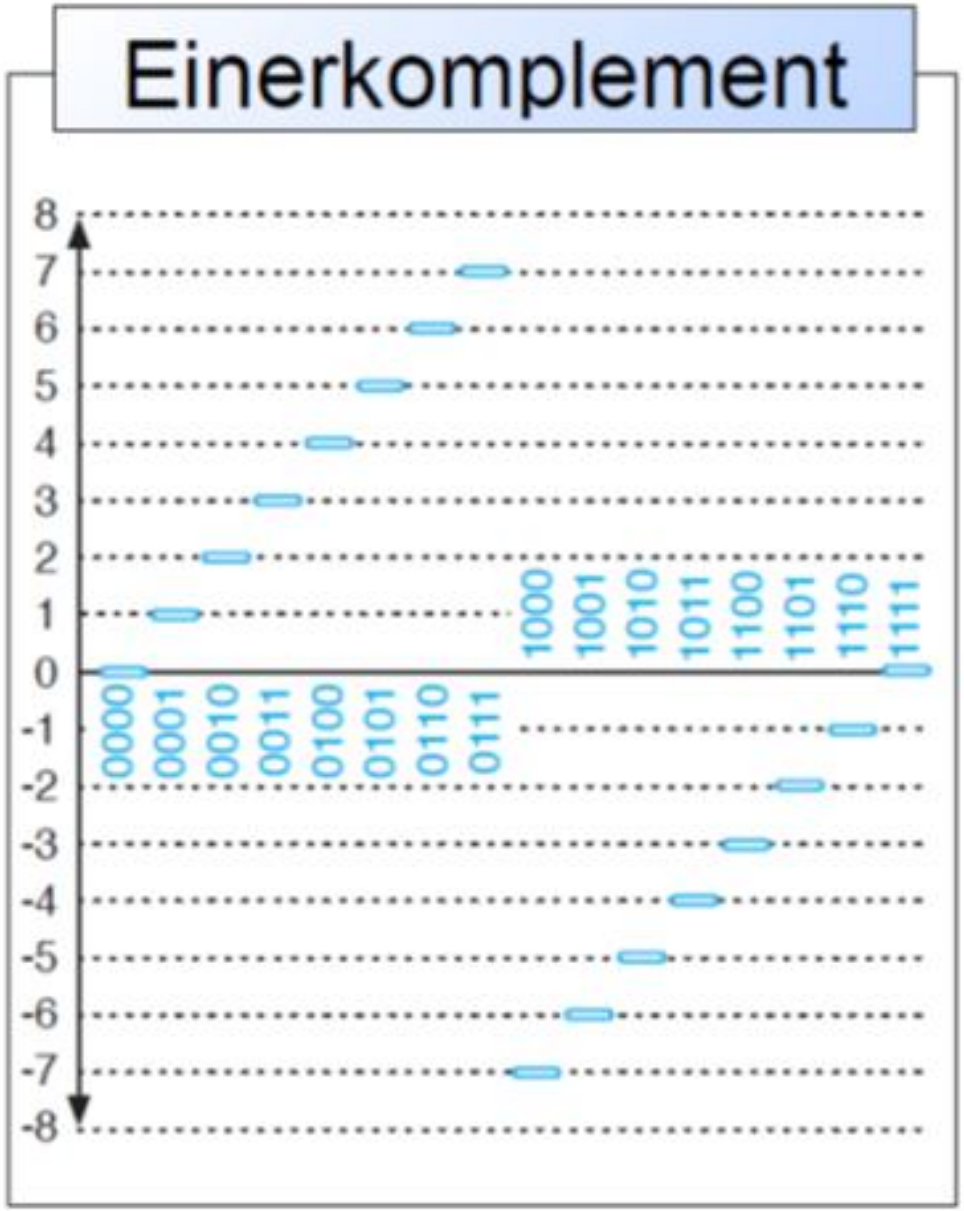
Vorzeichenbit

- Das erste Bit gibt das Vorzeichen an, die restlichen Bits geben die Zahl (Betrag der Zahl) an.
- Beispiele anhand eines Datentyps, der immer 4 Bit im Speicher reserviert:
  - 5=0101 -> -5=1101
  - 3=0011 -> -3=1011
  - 0=0000 -> -0=1000



Einerkomplement

- Das erste Bit gibt auch hier das Vorzeichen an, aber negative Zahlen werden durch das Invertieren aller Bits gebildet.
- Beispiele:
  - 5=0101 -> -5=1010
  - 3=0011 -> -3=1100
  - 0=0000 -> -0=1111
- Tatsächlich ist das sehr sinnvoll, denn es gilt -3 > -5. Dies gilt hier ebenfalls im Binären: 1100 > 1010

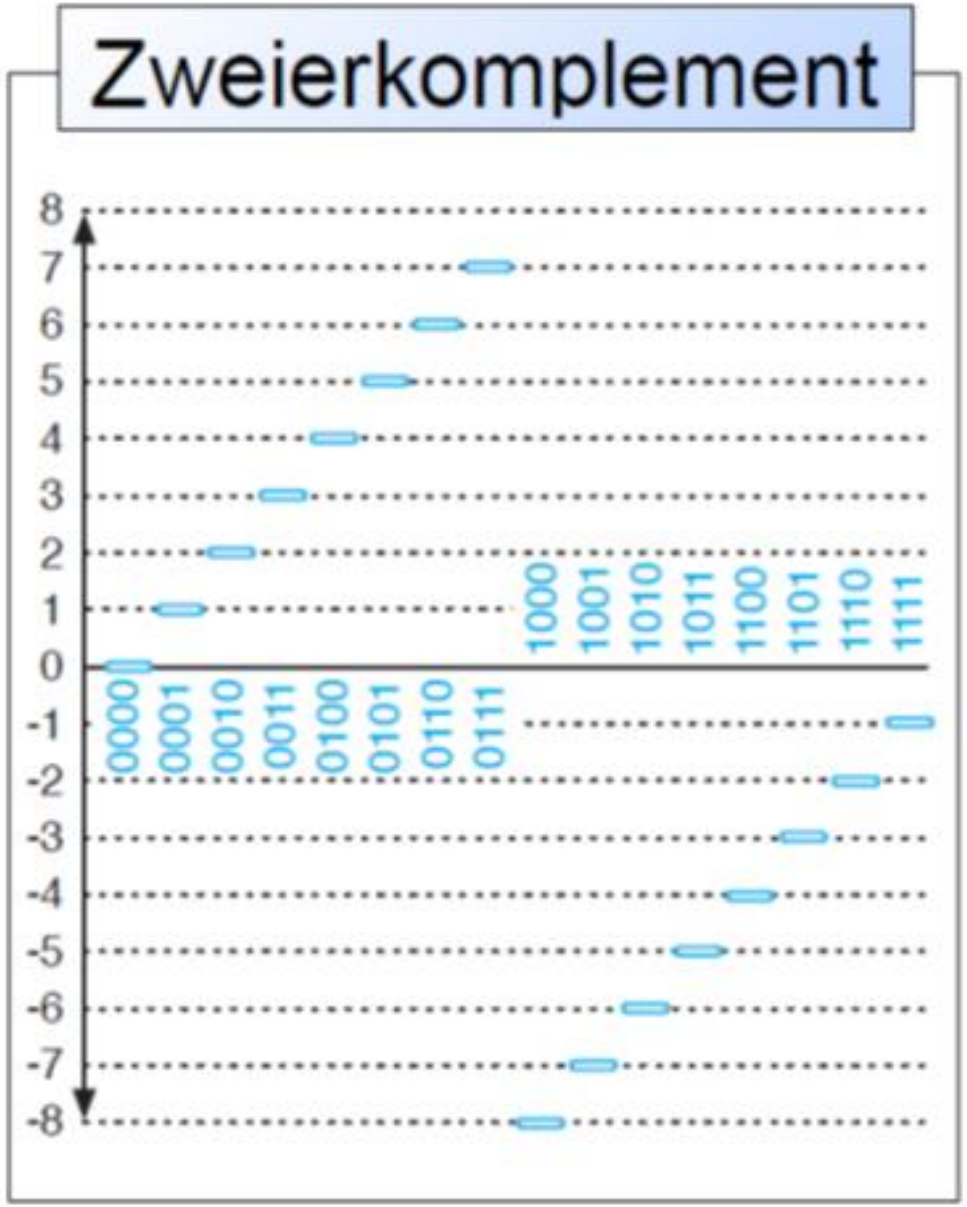


Zweierkomplement

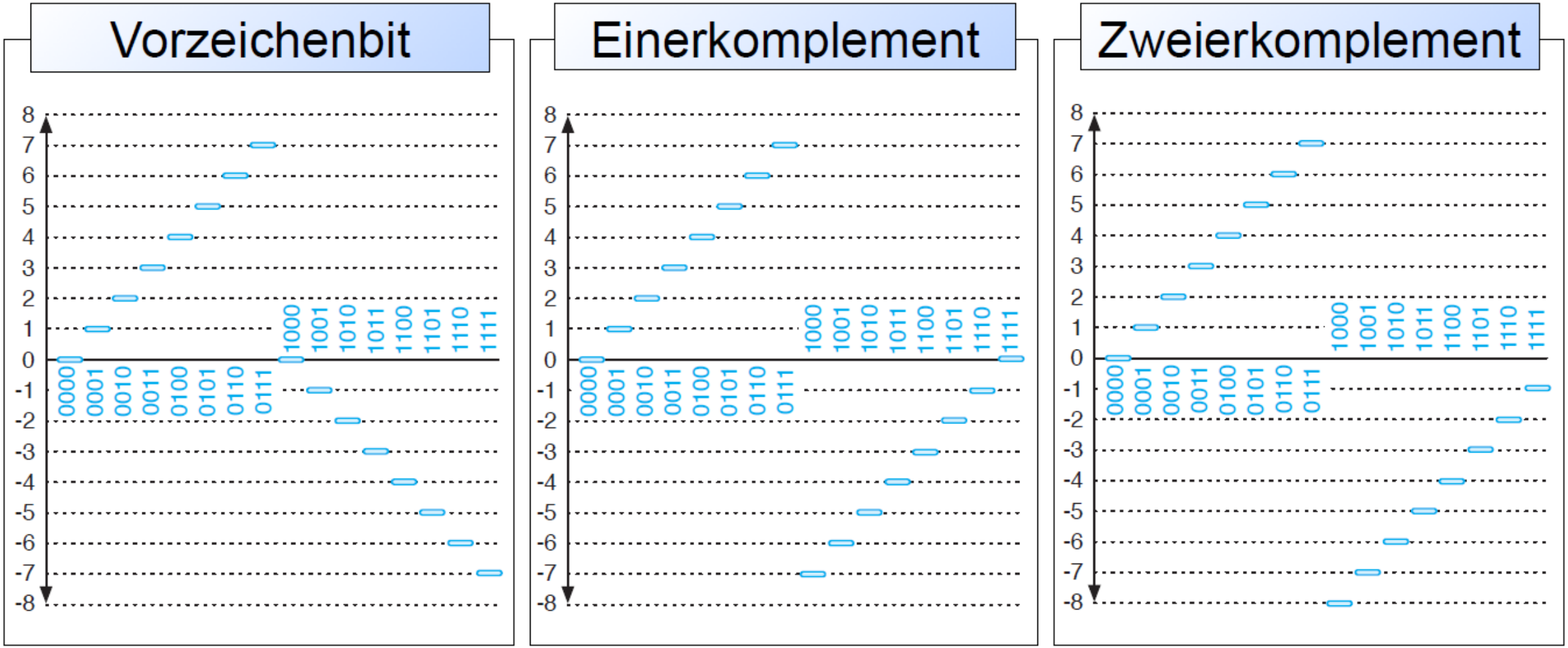
- Das erste Bit gibt auch hier das Vorzeichen an, aber negative Zahlen werden durch das Invertieren aller Bits + 1 gebildet.
- Beispiele:
  - -1: 1=0001 -> 1110 + 1 = 1111
  - -4: 4=0100 -> 1011 + 1 = 1100
  - -0 = 0, denn 0000 -> 1111 + 1 = 0000
  - Trick zum leichten Umrechnen: Die Negativ-1 kann hier als -8 gezählt werden.

    -8 +5 = -3

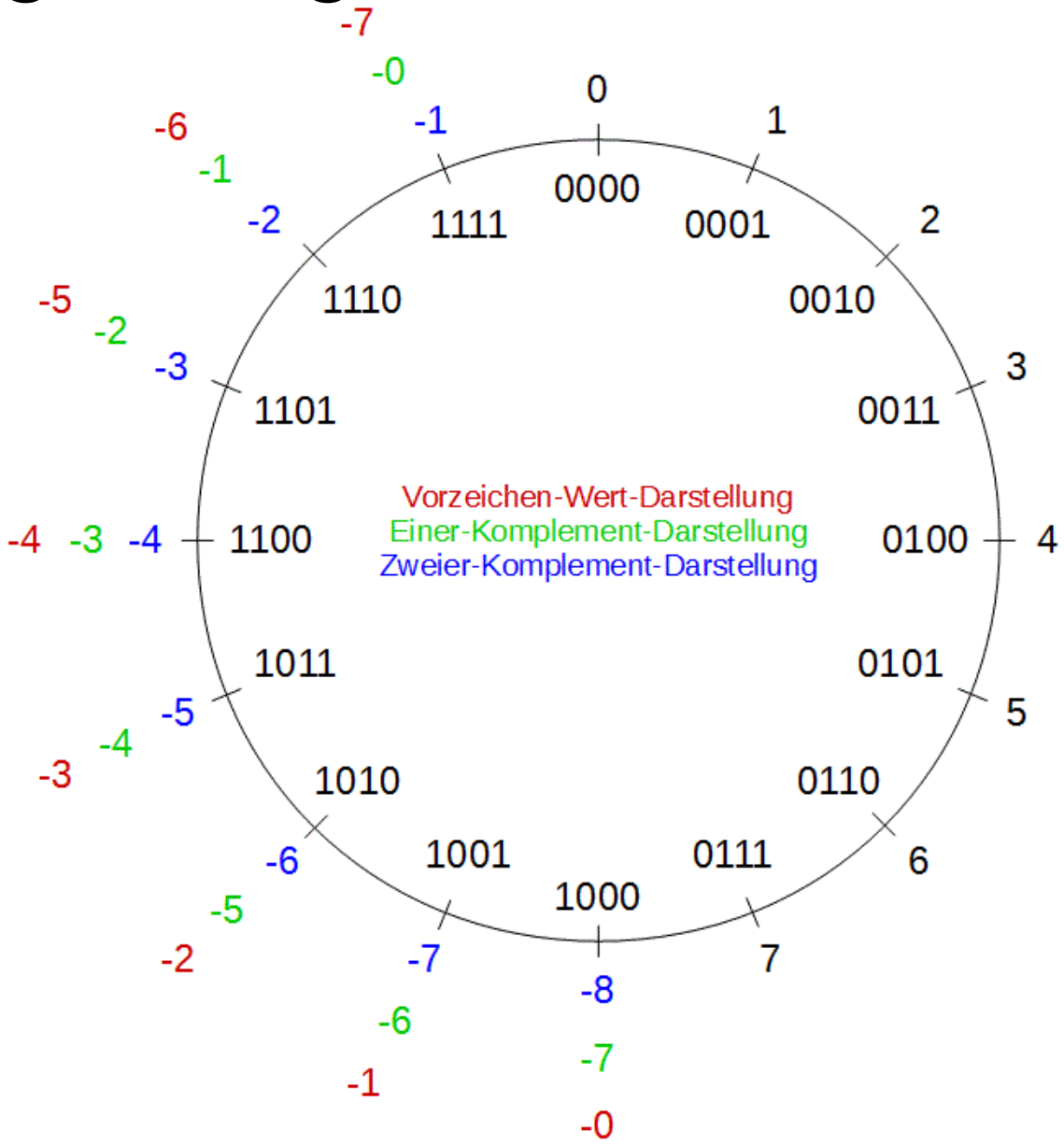
  - 1 1 0 1



Vorzeichenbit, Einerkomplement, Zweierkomplement



Kreisdarstellung der negativen Zahlen



Vorteil der Komplemente gegenüber dem Vorzeichenbit

- -2 + (+5) = +3
- Vorzeichenbitdarstellung

$$\begin{array}{r} 1010 \\ +0101 \\ =0011 \end{array}$$

Komplementdarstellung

$$\begin{array}{r} 1110 \\ +0101 \\ =0011 \end{array}$$

- Fazit: Die Komplementdarstellungen haben den großen Vorteil, dass man weiterhin normal im Stellenwertsystem rechnen kann, während bei der Vorzeichenbitdarstellung ständig geprüft werden müsste, ob ein Vorzeichenwechsel vorliegt, weil sonst die Rechnung nicht mehr stimmt.

Zusammenfassung

- Die Komplementdarstellungen haben den entscheidenden Vorteil, dass man weiterhin normal rechnen kann. Im Rechenwerk wird das Addieren wie das schriftliche Addieren auf Papier umgesetzt.
- Offensichtlich ist das Zweierkomplement das bessere System, da z.B. beim binären Hochzählen beim Einerkomplement darauf geachtet werden, dass die 0 nicht doppelt gezählt wird, während beim Zweierkomplement keine Probleme auftreten.
- Einerkomplement: 0=0000 und 1111, da 1111 der -0 entspricht
- Zweierkomplement: 0=0000, da -0: 0=0000 -> 1111 + 1 = 0000
- => Das Zweierkomplement ist die beste Variante!

Zweierkomplement in Java

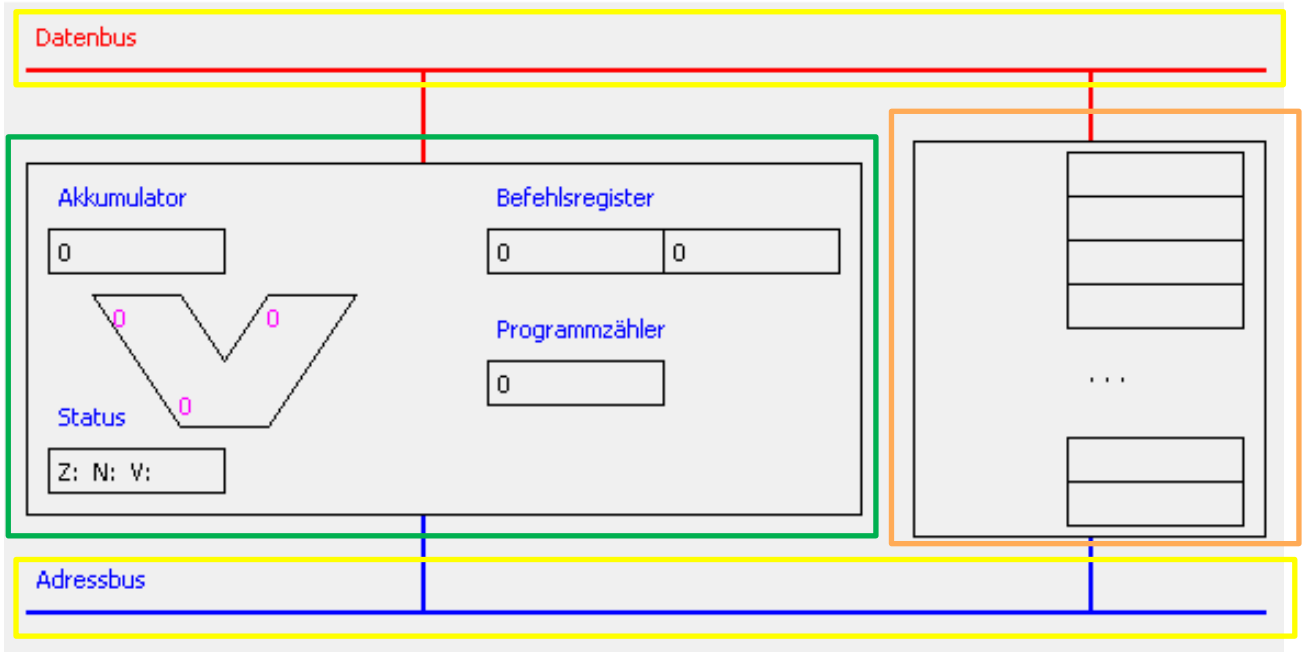
- Das Zweierkomplement ist die übliche Darstellungsweise für positive und negative ganze Zahlen.
- Auch Java speichert Zahlen im Zweierkomplement ab:

Typname	Größe[1]	Wrapper-Klasse	Wertebereich	Beschreibung
boolean	undefiniert[2]	java.lang.Boolean	true / false	Boolescher Wahrheitswert, Boolescher Typ[3]
char	16 bit	java.lang.Character	0 ... 65.535 (z. B. 'A')	Unicode-Zeichen (UTF-16)
byte	8 bit	java.lang.Byte	-128 ... 127	Zweierkomplement-Wert
short	16 bit	java.lang.Short	-32.768 ... 32.767	Zweierkomplement-Wert
int	32 bit	java.lang.Integer	-2.147.483.648 ... 2.147.483.647	Zweierkomplement-Wert
long	64 bit	java.lang.Long	-2^{63} bis $2^{63}-1$, ab Java 8 auch 0 bis $2^{64}-1$[4]	Zweierkomplement-Wert

Aufbau der 16-Bit-Minimaschine (Simulator)

ALU mit Steuerwerk

Ein-Weg-Bus



Speicher

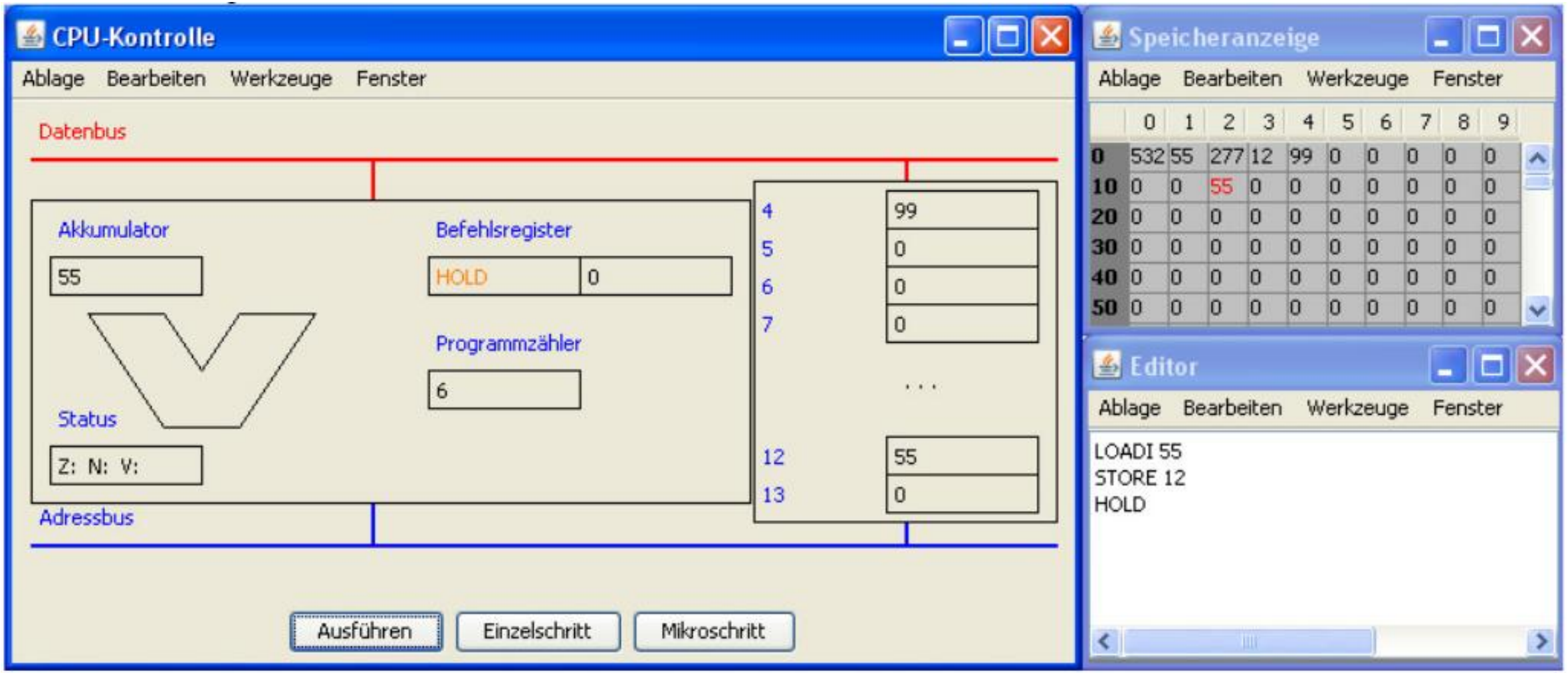
16-Bit-Adressraum

65536 Einträge/ Speicherzellen zu je 16bit (=128kB)

- 16 Bit → $\mathbf{2^{16}}$ = 65536
- → Codierung 11. Jgst: Dezimalzahl von 0 bis 65535 darstellbar
- Was ist mit negativen Zahlen?
- 16 Bit im Zweierkomplement -> Zahlen von -32768 bis +32767

Die Minimaschine

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



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 A1.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).

- Kopiere den Assemblertext in ein neues Fenster (Ablage → Neu) und speichere es unter dem Namen A1.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.

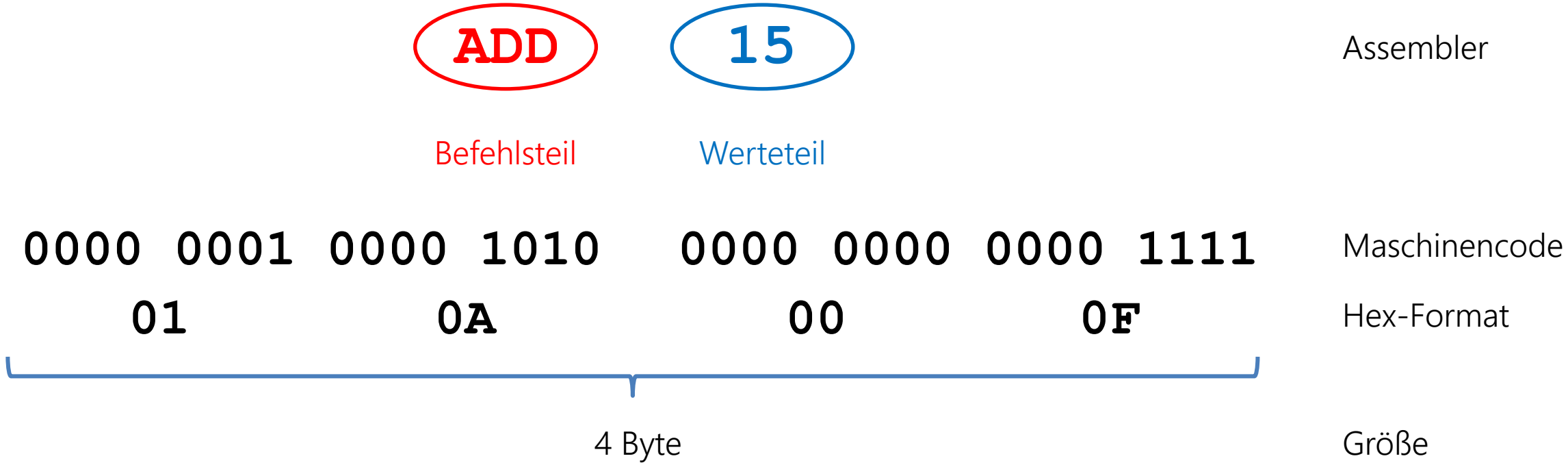
- Kannst du weitere Erkenntnisse gewinnen.

Arbeitsweise der Minimaschine nach von-Neumann-Zyklus

- FETCH-Phase, 1. Teil: operator
  Befehl holen: Einlesen des Maschinenbefehls (Operationskennung) aus der Speicherzelle, deren Adresse im Befehlszähler steht; die Operationskennung wird im Befehlsregister abgelegt; Weiterschalten des Befehlszählers um 1
- FETCH-Phase, 2. Teil: operand
  Befehl vervollständigen: Operand laden und im Befehlsregister ergänzen; Weiterschalten des Befehlszählers um 1
- FETCH-Phase, 3. Teil: indirect (lernen wir erst später)
  bei indirekter Adressierung den Wert der durch den bisherigen Operanden bezeichneten Speicherzelle laden und den Operanden im Befehlsregister durch diesen Wert ersetzen
- DECODE-Phase
  Analysieren des Maschinenbefehls durch Entschlüsselung des Operationscodes und Vorbereiten von Steuerwerk und Rechenwerk für die Ausführung des Befehls
- EXECUTE-Phase
  Ausführen des Befehls in aufeinanderfolgenden Schritten; die Schritte werden je nach Bedarf des Befehls ausgeführt. Hier ist der Ablauf für einen Arithmetikbefehl dargestellt:
  - Operanden holen: Holen des Operanden aus dem Speicher; Bereitstellung des Operanden und des Akkumulatorinhalts an den Eingängen der ALU
  - Befehl ausführen: Der Prozessor führt in der ALU die entsprechende Berechnung durch.
  - Rückschreiben: Das Ergebnis der Berechnung wird in den Akkumulator übertragen.

Struktur der Befehle in der Minimaschine

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



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

Was ist der Unterschied zwischen LOADI und LOAD?

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

Assemblerbefehle der Minimaschine

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

Direkter Wert zu Beginn

Speicherorganisation	
WORD zahl	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	
HOLD	Hält den Prozessor an. Dieser Befehl hat keine Adresse.
RESET	Setzt den Prozessor auf den Startzustand zurück. Dieser Befehl hat keine Adresse.
NOOP	Tut einfach nichts (NO OPeration). Dieser Befehl hat keine Adresse.

Grundlegender Aufbau eines Programms

1) Häufig müssen zu Beginn eines Programm Werte mittels der **I-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;

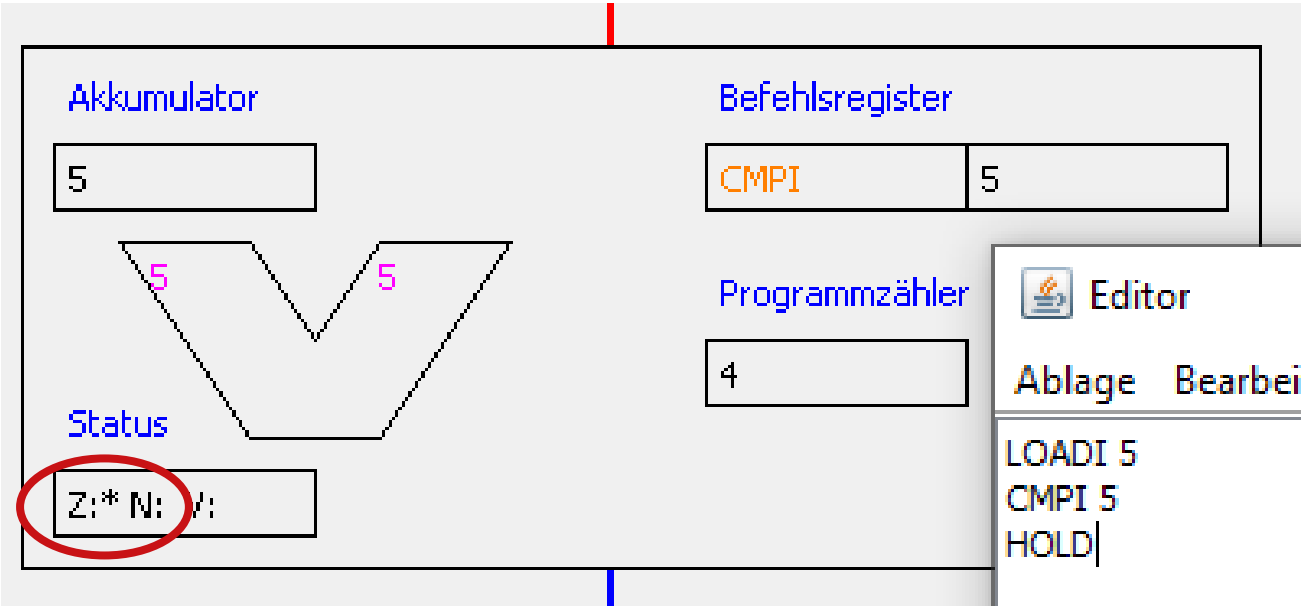
Status-Register

- Status-Register
  - *negative-flag N*
    Die letzte Rechenoperation lieferte ein negatives Ergebnis.
  - *zero-flag Z*
    Das letzte Rechenergebnis lieferte Null als Ergebnis.
  - *overflow-flag V*
    Der Akkumulator ist übergelaufen (bei vorzeichenbehafteter Interpretation).
  - *overflow-flag C*
    Der Akkumulator ist übergelaufen (bei vorzeichenloser Interpretation).

  - 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!

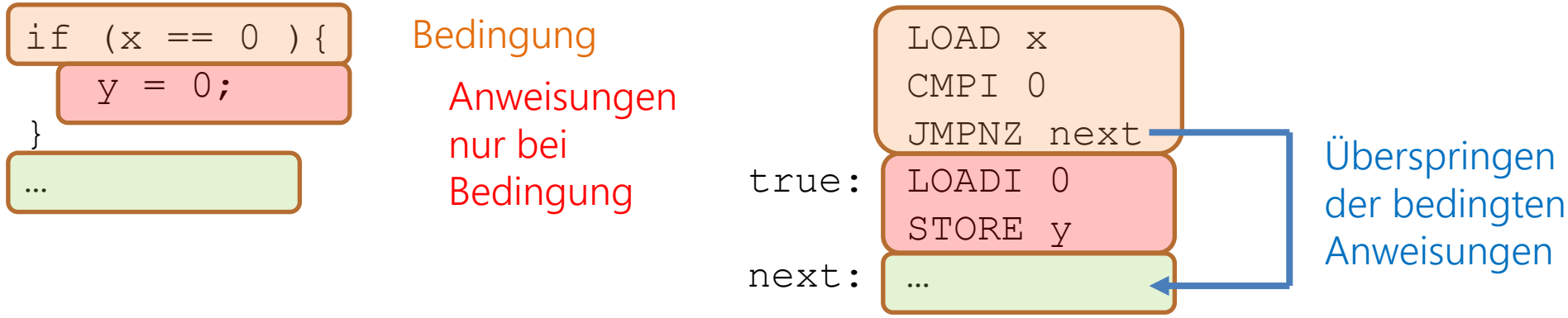
Sprungbefehle und Sprungmarken

- 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.

Sprungbefehle	
JMPP adresse	Springt zur angegebenen Adresse, wenn das Ergebnis der letzen Operation positiv (> 0) war, d. h. weder N noch Z-Flag sind gesetzt.
JMPNN adresse	Springt zur angegebenen Adresse, wenn das Ergebnis der letzen Operation nicht negativ (≥ 0) war, d. h. das N-Flag ist nicht gesetzt.
JMPN adresse	Springt zur angegebenen Adresse, wenn das Ergebnis der letzen Operation negativ (< 0) war, d. h. das N-Flag ist gesetzt.
JMPNP adresse	Springt zur angegebenen Adresse, wenn das Ergebnis der letzen Operation nicht positiv (≤ 0) war, d. h. das N-Flag oder as Z-Flag ist gesetzt.
JMPZ adresse	Springt zur angegebenen Adresse, wenn das Ergebnis der letzen Operation null (= 0) war, d. h. das Z-Flag ist gesetzt.
JMPNZ adresse	Springt zur angegebenen Adresse, wenn das Ergebnis der letzen Operation nicht null (≠ 0) war, d. h. das Z-Flag ist nicht gesetzt.
JMPV adresse	Springt zur angegebenen Adresse, wenn die letzte Operation einen Überlauf verursacht hat, d. h. das V-Flag ist gesetzt.
JMP adresse	Springt zur angegebenen Adresse.

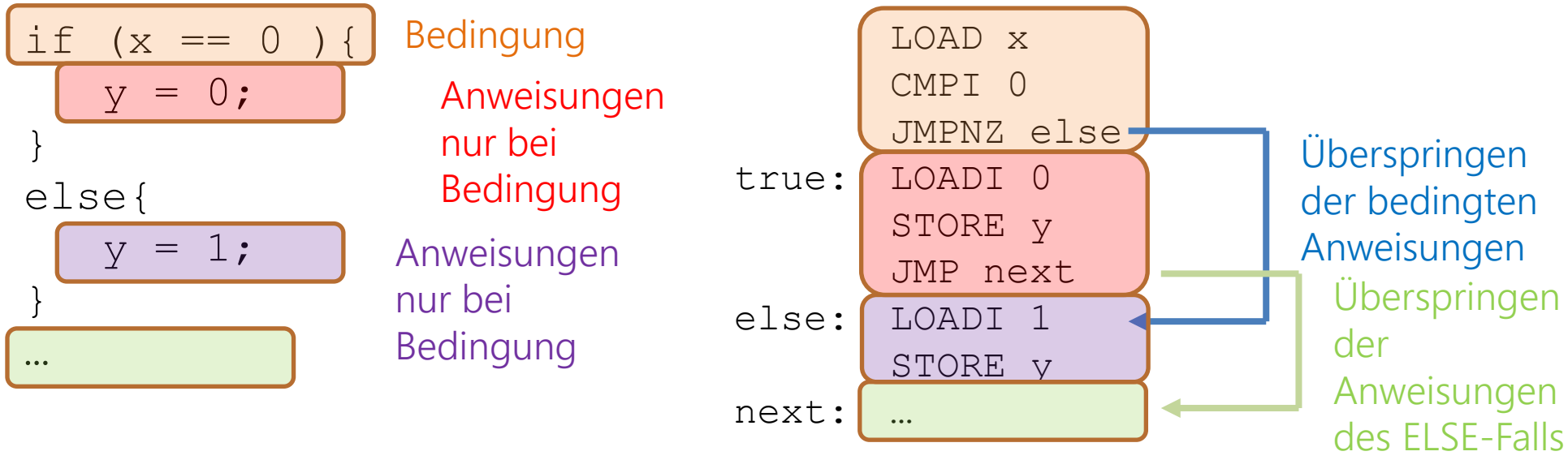
Bedingte Anweisung in Assembler

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



Fallunterscheidung in Assembler

- Grundstruktur einer Fallunterscheidung in Assembler



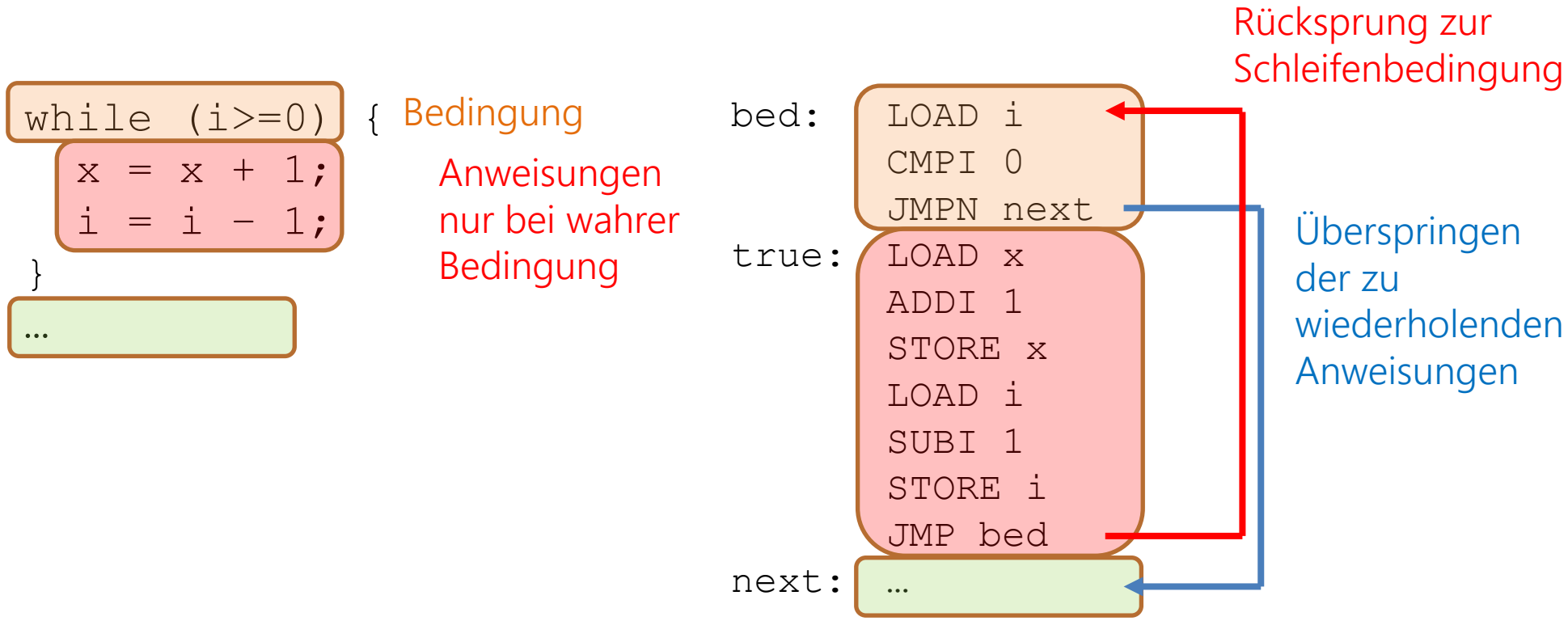
Aufgabe

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

```
for (int i=0; i<n; i++){
    x = x + 1;
}
…
```

Anweisungen zum Wiederholen nach Zähler

```
for:  LOADI 0
      STORE i
test: LOAD i
      CMP n
      JMPZ next
wied: LOAD x
      ADDI 1
      STORE x
      LOAD i
      ADDI 1
      STORE i
      JMP test
next: …
```

Rücksprung zum Schleifenzählertest

Überspringen der zu wiederholenden Anweisungen

Inkrementieren des Schleifenzählers

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!

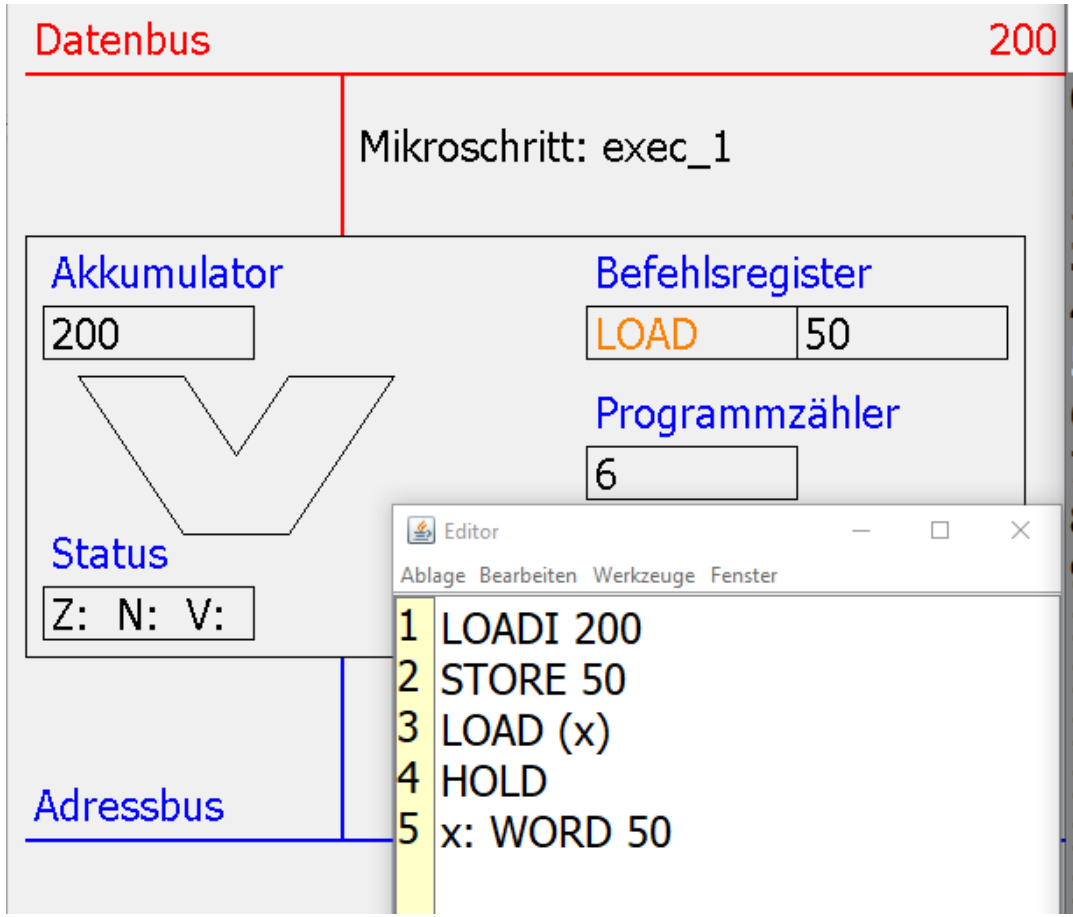
- Löse die Aufgaben des letzten Informatikabiturs in den Teilen III. und IV. zum Thema Assembler/Assembler-Programmierung!

- Die Informatikabiture sind zu finden unter mebis -> Prüfungsarchiv

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 die Variablen, also 50, in den Akkumulator.
- LOAD (x) lädt 7 in den Akkumulator.

Indirekte Adressierung



	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

Arrays in Assembler

- Ein Array ist eine Datenstruktur, bei der Elemente mit demselben Datentyp nebeneinander in einem zusammenhängenden Speicherbereich abgelegt werden.
- Die Zählwiederholung kann in Maschinensprache dazu verwendet werden, eine Reihe von Speicherzellen nacheinander zu durchlaufen und deren Werte ggf. zu modifizieren bzw. für Berechnungen zu verwenden. Dazu muss die indirekte Adressierung eingesetzt werden.

Folge von Quadratzahlen

- Es soll ein Array an der Speicheradresse 200 bis 220 mit den ersten 20 Quadratzahlen erstellt werden. Nutze eine Zählwiederholung (for) mit einer Laufvariable i, welche von 200 bis 220 läuft, um die indirekte Speicheradressierung zu realisieren.
- Ergebnis:

200	1	4	9	16	25	36	49	64	81	100
210	121	144	169	196	225	256	289	324	361	400

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:

	0	1	2	3	4	5	6	7
200	0	0	0	1	0	1	0	1

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