

Modul Informatik-II

Kurs Informatik-3: Teil-3

www.engineering.zhaw.ch/de/engineering/studium/bachelor/informatik/studium-zurich.html

Prof. Dr. Olaf Stern
Leiter Studiengang Informatik
+41 58 934 82 51
olaf.stern@zhaw.ch

Lernziele: (Allgemein)

- Die Studierenden kennen die *grundlegende Architektur von Rechnern* und die wichtigsten *Architekturelemente*.
- Sie sind vertraut mit der *elementaren Arbeitsweise eines Computers* und der *hardwarenahen Programmierung*. Sie können diese an einfachen Beispiel erläutern.
- Die Studierenden kennen die grundsätzlichen Aufgaben eines *Betriebssystems*. Sie können die *typischen* Verfahren und *Algorithmen*, die bei der *Entwicklung* von *Betriebssystemen* zur Anwendung gelangen, beschreiben.

Lernziele: (Allgemein)

- Die Kurse der Module Informatik I und Informatik II (der Modulgruppen "Grundlagen der Informatik I+II") vermitteln den Studierenden die *Grundlagen der Informatik, die jede / jeder Studierende unabhängig von der Wahl der Wahlpflichtmodule im Fachstudium erlangen sollte.*
- Die vermittelten Grundlagen *werden in den Modulen im Fachstudium vorausgesetzt.*

Lernziele: Spezifisch Teil-3

- Die Studierenden kennen die grundlegenden Arten und den Aufbau von *Befehlen* für einen Computer.
- Sie können den Unterschied zwischen *direkter* und *indirekter Adressierung* erläutern.
- Sie sind vertraut mit dem Ansatz der Programmierung mit Hilfe von *mnemonischen Symbolen*, die auch in der Programmiersprache *Assembler* genutzt werden, wie auch mit *Maschinen-Code*.
- Sie können an Beispielen einfache *Programme* schreiben und kennen die *elementaren Programmkonstrukte* wie *bedingte* und *unbedingte Verzweigungen*, *Schleifen* und *Unterprogramme*.

Themenüberblick Teil-3

Technische Informatik / Rechnerarchitektur

- Einführung / Übersicht
- Grundlegende Rechnerarchitektur
- Prozessoren
- **Befehle – die „Wörter“ des Rechners**
 - Aufbau und Arten
 - Direkte und indirekte Adressierung
 - Assembler-Sprache / mnemonische Symbole
 - Programm-Konstrukte: Einfache Operationen, Schleifen, Unterprogramme
- „Mini-Power-PC“ - Prozessormodell
- Speicher
- „Mini-Power-PC“ (Fortsetzung)

Lerninhalte Teil-3

- **Befehle – die „*Wörter*“ eines Rechners**
 - **Einführung / Motivation – Was ist ein Befehl?**
 - **Aufbau / Arten**
 - **Direkte und indirekte Adressierung**
 - **Verzweigungen und Unterprogramme**
 - **Assembler-Sprache / *mnemonische Symbole (Mnemonics)***
 - **Einfaches Prozessormodel mit Befehlssatz**
 - **Beispielprogramme**
 - Einfache Addition
 - Schleifen - Summenbildung

Lerninhalte Teil-3

- **Befehle – die „*Wörter*“ eines Rechners**
 - **Maschinen-Code**
 - Beispiel – vereinfachter Befehlssatz
 - Programmbeispiele

Befehle – die „Wörter“ eines Rechners

Einführung / Motivation

- Was ist ein **Befehl**?
- Was ist ein **Befehlssatz** (*instruction set*)?



- Mit Hilfe von **Befehlen** wird ein **Rechner gesteuert!**

Befehle – die „Wörter“ eines Rechners

Einführung / Motivation

- Was ist ein *Befehl*?

- Rechner soll *arithmetisch-logische Funktionen* berechnen:

- => *Arithmetische Funktionen* sind i. d. R. 1- oder 2-stellige mathematische Grundfunktionen

- Beispiele: *Addition* und *Subtraktion*

- Beispiel: $s = a + b$

- Zahl a** mit **Zahl b** addieren und **Ergebnis** (Summe) in **s** schreiben.

Befehle – die „*Wörter*“ eines Rechners

Einführung / Motivation

- Was ist ein *Befehl*?

- Rechner soll *arithmetisch-logische Funktionen* berechnen:

- => *Logische Funktionen* sind i. d. R. 1- oder 2-stellige boolesche Funktionen

- Beispiele: *ODER* und *OR*

- Beispiel: $w = u \text{ OR } v$

- Zeichenreihe u mit Zeichenreihe v (bitweise) *ODER*-
verknüpfen und Ergebnis in w schreiben.

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten

▪ Schematischer Aufbau

- Ein *Befehl* setzt sich aus dem *Operationscode* (*OP-Code*), ggf. *Optionen* und den *Operanden* zusammen:

=> „*Was wird (wie) mit wem gemacht*“.

- Dabei ist für die *Befehle* eines Rechners (Prozessors) festgelegt, an welcher Stelle der *OP-Code* steht und an welcher Stelle in welcher Reihenfolge die *Operanden*.

Typischer Aufbau eines *Befehls*:

OP-Code, (Option), Operand-1, Operand-2, (Operand-3)

Befehle – die „Wörter“ eines Rechners

Aufbau / Arten

▪ Schematischer Aufbau

- Ein **Befehl** setzt sich aus dem **Operationscode (OP-Code)**, ggf. **Optionen** und den **Operanden** zusammen:

=> „Was wird (wie) mit wem gemacht“.

- Dabei ist für die **Befehle** eines Rechners (Prozessors) festgelegt, an welcher Stelle der **OP-Code** steht und an welcher Stelle in welcher Reihenfolge die **Operanden**.

Typischer Aufbau eines **Befehls**: (Alternative – Bsp.)

OP-Code, Operand-1, Operand-2, Operand-3, (Option)

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten

▪ Schematischer Aufbau

- Beispiel: $s = a + b$

Zahl a mit **Zahl b** addieren und **Ergebnis** (Summe) in **s** schreiben:

=> **ADD, a , b , s , -**

Für einen Rechner bedeutet dieses:

- **Addiere** den **Inhalt** des **Speichers** mit der **Adresse a** mit dem **Inhalt** des **Speichers** mit der **Adresse b**

und
- **Schreibe** das **Ergebnis** in den **Speicher** mit der **Adresse s** .

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten

▪ Schematischer Aufbau

- Beispiel: $w = u \text{ OR } v$

Zeichenreihe u mit Zeichenreihe v (bitweise) über die **ODER-**Funktion verknüpfen und Ergebnis in w schreiben:

=> **OR, u , v , w , -**

Für einen Rechner bedeutet dieses:

- **Verknüpfe** (bitweise) den **Inhalt** des **Speichers** mit der **Adresse u** mit dem **Inhalt** des **Speichers** mit der **Adresse v**
und
- **Schreibe** das **Ergebnis** in den **Speicher** mit der **Adresse w** .

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten

▪ Schematischer Aufbau

- Wie lange würde die Ausführung eines solchen *Befehls* dauern?
Bzw. was ist daran sehr ungünstig und wie kann dieses verbessert werden?

(nachdem, was Sie bisher gelernt haben)

=> Vor der **eigentlichen Berechnung** (Operation) **muss zweimal** auf den **Speicher zugegriffen** werden und **anschliessend** noch **einmal** (in der Summe **dreimal**).

Speicherzugriffe sind **sehr langsam** (gegenüber Operationen im Prozessor – Faktor 100 und noch schlechter).

Befehle – die „Wörter“ eines Rechners

Aufbau / Arten

▪ Schematischer Aufbau

- Daher werden für *arithmetisch-logische* Funktionen häufig für die *Operanden* Register verwendet (statt auf den Speicher zuzugreifen).

Aus: **ADD, a, b, s,-** => **ADD, R-1, R-2, R-3**

Für einen Rechner bedeutet dieses nun:

- *Addiere* den Inhalt des Registers **R-1** mit dem Inhalt des Registers **R-2** und
- *Schreibe* das **Ergebnis in** Register **R-3**.



Für die Abarbeitung des Befehls ist nun kein einziger Speicherzugriff mehr erforderlich!

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten

▪ Schematischer Aufbau

- Daher werden für *arithmetisch-logische* Funktionen häufig für die *Operanden* Register verwendet (statt auf den Speicher zuzugreifen).

=> Als Folge davon sind *Lese-* und *Schreibbefehle* erforderlich, mit denen die **Werte** eines **Speichers** in ein **Register** geschrieben oder aus einem Register übernommen werden.

=> Die Anzahl der in einem **Prozessor** verfügbaren (**Arbeits-**) **Register** ist auf **wenige beschränkt**.
Z. B. **32**, da ansonsten die **HW** sehr **teuer** und **langsamer** wird (hat **direkten Einfluss** auf die **Zykluszeit**).

Befehle – die „Wörter“ eines Rechners

Aufbau / Arten

▪ Befehlstypen

- Anstelle eines Registers für einen Operanden kann auch eine Konstante stehen (nicht für das Ergebnis!).

Beispiel: `ADD, R-1, 100, R-2`

Für einen Rechner bedeutet dieses nun:

- Addiere zum Inhalt des Registers *R-1* den Wert *100* und schreibe das Ergebnis in das Register *R-2*.

Anmerkung: Operationen mit Konstanten werden häufig als «direkte» Befehle bezeichnet.

Befehle – die „Wörter“ eines Rechners

Aufbau / Arten

▪ Befehlstypen

- Bei den ersten Mikroprozessoren betrug die Wortbreite 4 bzw. 8 Bit (heute i. d. R. 32 oder 64 Bit).

Wie sollen mit 4 ... 16 Bit **Op-Code** und drei **Operanden** (Register, Konstante oder Speicheradressen) angegeben werden?

Lösung a: Ausgezeichnetes Register, der **Akkumulator**

- Es gibt nur zwei oder einen **Operanden** und dafür ein ausgezeichnetes Register, den **Akkumulator**.
- Der **Akkumulator** wird für viele Operationen implizit als ein **Operand** genutzt (und ggf. als dritter für das Ergebnis).

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten

▪ Befehlstypen

- Bei den ersten Mikroprozessoren betrug die Wortbreite 4 bzw. 8 Bit (heute i. d. R. 32 oder 64 Bit).

Wie sollen mit 4 ... 16 Bit *Op-Code* und drei *Operanden* (Register, Konstante oder Speicheradressen) angegeben werden?

Lösung a: => Beispiel: *ADD, R-1*

- *Addiere* zum *Inhalt* des *Akkumulators* den *Wert* des Registers *R-1* und *schreibe* das *Ergebnis* in den *Akkumulator*.

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten

▪ Befehlstypen

- Bei den ersten Mikroprozessoren betrug die Wortbreite 4 bzw. 8 Bit (heute i. d. R. 32 oder 64 Bit).

Wie sollen mit 4 ... 16 Bit *Op-Code* und drei *Operanden* (Register, Konstante oder Speicheradressen) angegeben werden?

Lösung b: Ein Befehl ist grösser als ein Wort

- Ein **Befehl** setzt sich aus **zwei, drei oder mehr Wörtern** zusammen.
- Dafür müssen **mehrere Wörter** in das **Befehlsregister** geladen werden (dies ist auch entsprechend breiter).

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – Adressierung

▪ Adressierung

- Bei den ersten Mikroprozessoren betrug die Wortbreite 4 bzw. 8 Bit (heute i. d. R. 32 oder 64 Bit).

Wie sollen mit 4 ... 32 Bit *Op-Code* und Speicher adressiert werden, der i. d. R. sehr gross ist?

(4 GiB erfordern 32 Bit für die Adressierung)

Lösung a: Ein Befehl ist grösser als ein Wort – bekannt

- Ein **Befehl** setzt sich aus **zwei, drei oder mehr Wörtern** zusammen.
- Dafür müssen **mehrere Wörter** in das **Befehlsregister** geladen werden (dies ist auch entsprechend breiter).

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – Adressierung

▪ Adressierung

- Bei den ersten Mikroprozessoren betrug die Wortbreite 4 bzw. 8 Bit (heute i. d. R. 32 oder 64 Bit).

Wie sollen mit 4 ... 32 Bit *Op-Code* und Speicher adressiert werden, der i. d. R. sehr gross ist?

(4 GiB erfordern 32 Bit für die Adressierung)

Lösung b: *Indirekte Adressierung*

next
Slide



Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – Adressierung

■ Adressierung

- ... **Wie sollen mit 4 ... 32 Bit *Op-Code* und Speicher adressiert werden, der i. d. R. sehr gross ist?**
(4 GiB erfordern 32 Bit für die Adressierung)

Lösung b: *indirekte Adressierung*

- Statt eine **Speicheradresse direkt anzugeben**, wird eine **indirekte Adresse genutzt**:
 - z. B. ein **Register**: der **Inhalt des Register** spezifiziert die **Speicheradresse** oder / und
 - ein **Basisregister**: ein **festgelegtes Register** definiert einen **Basiswert**.

Befehle – die „Wörter“ eines Rechners

Aufbau / Arten – Adressierung

▪ Adressierung

- ... Wie sollen mit 4 ... 32 Bit **Op-Code** und Speicher adressiert werden, der i. d. R. sehr gross ist?
(4 GiB erfordern 32 Bit für die Adressierung)

Lösung b: => Beispiel: **Load, R-1**

Für einen Rechner bedeutet dieses z. B. (Festlegung):

- **Lade** den **Inhalt** des durch den **Wert des Registers R-1** adressierten **Speichers** in den **Akkumulator**.

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – Adressierung

■ Adressierung

- Ein Programm (Programmdaten und das Programm selber) kann irgendwo im Speicher abgelegt sein.

Wie soll der Speicherzugriff programmiert werden, wenn die absolute Speicheradresse zur Zeit der Programmierung nicht bekannt ist?

Lösung: Wiederum *indirekte Adressierung*

- Statt eine **Speicheradresse** direkt anzugeben, wird ein **Register** angegeben, dessen Inhalt zur **Adressierung** mit verwendet wird; der **Wert** dieses Registers wird zum **Programmstart festgelegt** (und ggf. gespeichert).

Befehle – die „Wörter“ eines Rechners

Aufbau / Arten – Adressierung

▪ „Offset“

- Häufig wird bei Lese- und Schreibbefehlen zum Wert eines Registers noch ein **Offset** addiert.

(=> z. B. sinnvoll für komplexe Sprünge)

Beispiel: **Ldoff, R-1, 4** („Load offset“)

Für einen Rechner bedeutet dieses z. B. (Festlegung):

- **Lade** den **Inhalt** des durch den **Wert des Registers R-1** zuzüglich der **Konstanten 4 adressierten Speichers** in den **Akkumulator**.

Befehle – die „Wörter“ eines Rechners

Aufbau / Arten – Sprungbefehle & Verzweigungen

▪ Sprungbefehle / (bedingte) Verzweigungen

- Häufig wird bei **bedingten Sprungbefehlen** ein ausgezeichnetes **Register auf Null geprüft**.

(=> Vergleich auf Null sehr einfach und schnell realisierbar.)

Beispiel: **Bnull, R-1** („Branch if null“) *Branch - go s'where else*

Für einen Rechner bedeutet dieses z. B. (Festlegung):

- Wenn das **Register R-1**, z. B. der **Akkumulator**, **Null** ist,
 - **Verzweige** zum **Inhalt** der durch den **Wert** des **Registers R-1** angegebenen **Adresse**.
 - **Sonst** fahre mit dem folgenden Befehl fort.

Befehle – die „Wörter“ eines Rechners

Aufbau / Arten – Sprungbefehle & Verzweigungen

▪ Sprungbefehle / (bedingte) Verzweigungen

In der Regel beinhaltet ein Befehlssatz viele bedingte Sprungbefehle (*Branch instructions*):

- Test auf **Null**, auf **grösser Null** oder auf **kleiner Null** eines (ausgezeichneten) Registers.
- Test auf **Gleichheit** / **Ungleichheit** von zwei (ausgezeichneten) Registern.
- Test auf **Gleichheit** / **Ungleichheit** eines (ausgezeichneten) Registers mit einer Konstanten.

Anmerkung: Neben dem **Akkumulator** werden häufig **spezielle Register** wie das **Statusregister**, **Interrupt-Register** oder der **Stack-Pointer** angesprochen.

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – Sprungbefehle & Verzweigungen

- **Sprungbefehle / (bedingte) Verzweigungen**

In der Regel beinhaltet ein Befehlssatz viele bedingte Sprungbefehle (*Branch instructions*):

Frage: Warum gibt es so viele verschiedene Sprungbefehle, wenn doch schon einer aus Sichtweise „*Mächtigkeit*“ genügen würde?

=> Ziel: Vereinfachung der Programmierung (Effizienz)

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – Sprungbefehle & Verzweigungen

▪ Sprungbefehle / (bedingte) Verzweigungen

Beispiel: Umsetzung einfaches „*if*-Konstrukt“

Wie kann eine einfache *if*-Anweisung mit einem *bedingten Sprung* mit Test auf *ungleich Null* realisiert werden?

1. Bedingungen berechnen und **Resultat** in ein **Register** schreiben
2. Test des **Registers** auf **Null**
 - a) Test ist **positiv** => **kein Sprung** (Programmfortsetzung)
=> **Ausführung der Befehle der *if*-Anweisung**
 - b) Test **negativ** => **Sprung zum 1. Befehl nach der *if*-Anweisung**

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – Sprungbefehle & Verzweigungen

- **Sprungbefehle / (bedingte) Verzweigungen**

Beispiel: Umsetzung einfaches „*if-else*-Konstrukt“

Wie kann eine einfache *if-else*-Anweisung mit einem *bedingten Sprung* mit Test auf *ungleich Null* realisiert werden?

Wie *if*-Konstrukt mit zusätzlichem **unbedingten Sprung** am **Ende** der **Befehle** des *if*-Teils an das **Ende** der *if-else*-Anweisung und zusätzlichem **unbedingten Sprung** zum 1. **Befehl** des *else*-Teils falls **Test negativ** ausgeht.

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – Sprungbefehle & Verzweigungen

▪ Sprungbefehle / (bedingte) Verzweigungen

Beispiel: Umsetzung einfaches „*while*-Konstrukt“

Wie kann eine einfache *while*-Schleife mit einem *bedingten Sprung* mit Test auf *ungleich Null* realisiert werden?

1. Bedingungen berechnen und **Resultat** in ein **Register** schreiben

2. Test des **Registers** auf **Null**

a) Test *positiv* => kein Sprung (Programmfortsetzung)
=> Ausführung der Befehle des *Schleifenkörpers*
=> Rücksprung zum Beginn (Schritt 1)

b) Test *negativ* => Sprung zum 1. Befehl nach dem *Schleifenkörper*

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – Sprungbefehle & Verzweigungen

- **Sprungbefehle / (bedingte) Verzweigungen**

Beispiel: Umsetzung «*for*-Konstrukt»

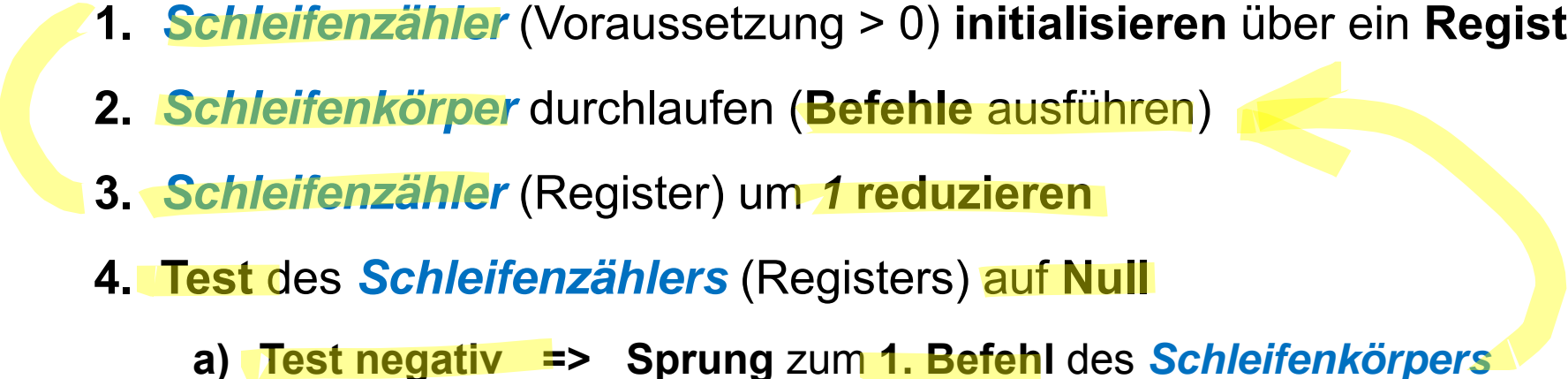
Realisierung einer einfachen *for*-Schleife mit einem *bedingten Sprung* mit Test auf ungleich Null?

Befehle – die „Wörter“ eines Rechners

Aufbau / Arten – Sprungbefehle & Verzweigungen

▪ Sprungbefehle / (bedingte) Verzweigungen

Beispiel: Umsetzung «**for-Konstrukt**»

1. **Schleifenzähler** (Voraussetzung > 0) **initialisieren** über ein **Register**
 2. **Schleifenkörper** durchlaufen (**Befehle** ausführen)
 3. **Schleifenzähler** (Register) um **1 reduzieren**
 4. **Test** des **Schleifenzählers** (Registers) **auf Null**
 - a) **Test negativ** \Rightarrow **Sprung** zum **1. Befehl** des **Schleifenkörpers**
(Schritt 2)
 - b) **Test positiv** \Rightarrow **kein Sprung** = **exit**
- 

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – Sprungbefehle & Verzweigungen

- Sprungbefehle / (bedingte) Verzweigungen

Beispiel: Umsetzung «*case/switch*-Konstrukt»

Realisierung einer einfachen *case/switch*-Anweisung mit einem *bedingten Sprung* mit Test auf ungleich Null?

Am einfachsten über eine *Folge* von *if*-Anweisungen

[Alternativ kann eine **Tabelle** mit **Sprungadressen** genutzt werden, die **zuvor indiziert** wurde. Manche **Befehlssätze** sehen dafür ein bereits **ausgezeichnetes Register** vor, das „*Jump-Register*“].

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – „*Unterprogramme / Prozeduren*“

- **Strukturierte Programme sind ohne Prozedur-Aufrufe / Unterprogramme nicht denkbar.**

Realisierung eines einfachen *Prozedur*-Aufrufs?

Beim Aufruf einer Prozedur / eines Unterprogramms:



Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – „*Unterprogramme / Prozeduren*“

- **Strukturierte Programme sind ohne Prozedur-Aufrufe / Unterprogramme nicht denkbar.**

Realisierung eines einfachen *Prozedur*-Aufrufs?

Beim **Aufruf** einer **Prozedur** / eines **Unterprogramms**:

- Wird wie bei einer Verzweigung das Programm an einer **anderen Stelle fortgesetzt**

=> **Über einen unbedingten Sprungbefehl.**

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – „*Unterprogramme / Prozeduren*“

- **Strukturierte Programme sind ohne Prozedur-Aufrufe / Unterprogramme nicht denkbar.**

Realisierung eines einfachen *Prozedur*-Aufrufs?

Beim Aufruf einer Prozedur / eines Unterprogramms:

- Wird wie bei einer Verzweigung das Programm an einer anderen Stelle fortgesetzt
- **Müssen Parameter (Argumente) übergeben werden**

=> Über Abspeichern der Werte an vereinbarten Stellen (Speicher bzw. spezielle Register).

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – „*Unterprogramme / Prozeduren*“

- **Strukturierte Programme sind ohne Prozedur-Aufrufe / Unterprogramme nicht denkbar.**

Realisierung eines einfachen *Prozedur*-Aufrufs?

Beim Aufruf einer Prozedur / eines Unterprogramms:

- ...
- **Wird das *Ergebnis der Prozedur* / des Unterprogramms zurückgegeben**

=> Ebenfalls über *Abspeichern der Werte an vereinbarten Stellen (Speicher bzw. spezielle Register).*

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – „*Unterprogramme / Prozeduren*“

- Strukturierte Programme sind ohne Prozedur-Aufrufe / Unterprogramme nicht denkbar.

Realisierung eines einfachen *Prozedur*-Aufrufs?

Beim Aufruf einer Prozedur / eines Unterprogramms:

- ...
- Wird zum **ursprünglichen** Programm **zurückgekehrt**
 - => Über einen **unbedingten Sprungbefehl** an die **zuvor abgespeicherte Adresse** des Befehlszählers des aufrufenden Programms; dieses wird dann normal fortgesetzt.

Befehle – die „*Wörter*“ eines Rechners

Aufbau / Arten – „*Unterprogramme / Prozeduren*“

- **Strukturierte Programme sind ohne Prozedur-Aufrufe / Unterprogramme nicht denkbar.**
 - Beim verschachtelten Aufruf von *Prozeduren / Unterprogrammen* müssen die erforderlichen *Registerinhalte* jeweils abgespeichert und wiederhergestellt werden.
 - Die dafür geeignete *Datenstruktur* wird über einen *Stack* realisiert, i. d. R. als *LIFO-Warteschlange*.
[vgl. Kurs Algorithmen & Datenstrukturen und Informatik 2]
 - Zur Vereinfachung der Verwaltung wird häufig ein „*Stack Pointer*“ – realisiert über ein festgelegtes Register – zur Verfügung gestellt.

Befehle – die „Wörter“ eines Rechners

Aufbau / Arten – Hauptgruppen

▪ **Befehlsgruppen** (elementare):

- Einfache **arithmetische** und **logische** Befehle
- **Lade-** oder **Speicher-Befehle** (**Load** / **Store**)
- Kontrollfluss: **bedingte** / **unbedingte Sprünge** (**Branch**)
- **Sonderfunktionen** (z. B. **Stack-Verwaltung**, **Interrupts**, ...)
e.g. ESC-Interrupt

▪ **Adressierung / Operandenangabe**

- **Absolut**: Werte / Adressen absolut im Befehl angegeben
- **Indirekt**: Werte / Adressen aus **Registerinhalt**
- Verwendung **Offset**: Wert / Adresse aus **Registerinhalt** und einem im Befehl absolut angegebenen **Wert** zusammengesetzt.

Befehle – die „Wörter“ eines Rechners

Programmbeispiele:

- „Prozessormodell“ (vereinfacht):
 - Wortbreite: 2 Byte (16 Bit),
 - Zahlendarstellung: 2-er-Komplement, 16 Bit mit MSb und MSB („most significant bit/byte“) je ganz links
 - Arbeitsspeicher: 1 KiB (2^{10} Bytes)
 - Register: Befehlsregister, Befehlszähler, Akkumulator „Akku“, Arbeitsregister „R1“, „R2“ und „R3“, Carry-Flag
 - Cache: keiner
 - Zykluszeit: 200 ns
 - CPI: 1 (alle Befehle)

Befehle – die „Wörter“ eines Rechners

Programmbeispiele:

▪ „Prozessormodell“ (vereinfacht):

– Befehlssatz (Ausschnitt - vereinfacht):

- **ADD Rnr** \Rightarrow $Akku := Akku + \langle Rnr^+ \rangle$
- **ADDD #Zahl** \Rightarrow $Akku := Akku + \#Zahl$
- **LWDD Rnr, #Adr** \Rightarrow $\langle Rnr^+ \rangle := \text{Inhalt Speicher}(\text{Adr})$
- **SWDD Rnr, #Adr** \Rightarrow $\text{Inhalt Speicher}(\text{Adr}) := \langle Rnr^+ \rangle$
- **SRA** \Rightarrow $Akku: \text{Verschieben (arithmetisch) nach rechts}$
- **SLA** \Rightarrow $Akku: \text{Verschieben (arithmetisch) nach links}$

Bei der Notation handelt es sich um **mnemonische Symbole**
(kurz **Mnemonics**)

^+nr ist 0, 1, 2 oder 3 (und steht für Register *Akku*, *R1*, *R2* bzw. *R3*)

Befehle – die „*Wörter*“ eines Rechners

Programmbeispiele:

▪ Code mit *mnemonischen Symbolen*:

- Ist erheblich „*lesbarer*“ (für Programmierer) als *Maschinen-Code*

- Ein Compiler, der Programme mit *mnemonischen Symbolen* in *Maschinen-Code* übersetzt, ist ein *Assembler-Compiler*.

- *Code* auf Basis von *mnemonischen Symbolen* wird entsprechend auch als *Assembler-Code* (oder *Assembler-Programm*) bezeichnet, der *Umfang der Symbole* als *Assembler (-Sprache)*.

Befehle – die „*Wörter*“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Einfache Addition

- Gegeben sind folgende (abstrakte) Anweisungen:

$s := b + a$

$r := 4 + b$

$s := s + r$ (a und b seien *Integer-Zahlen* mit der Länge **16 Bit**)

- Das *Programm* liegt ab Speicher(offset) **100** im Speicher
- Die *Daten* liegen ab Speicher(offset) **200** im Speicher,
d. h. die Variable a liegt im Speicher mit den Adressen **200 und 201**, die Variable b im Speicher mit den Adressen **202 und 203**.
- Nach Ende des Programms soll das *Ergebnis* (die Variable s) im Speicher mit den Adressen **204 und 205** abgelegt sein.

Befehle – die „Wörter“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Einfache Addition

- Wie könnte ein äquivalentes Programm mit *mnemonischen Symbolen* (Assemblerprogramm) lauten?

Programm-Code:

$s := b + a$

$r := 4 + b$

$s := s + r$

```
100 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b
102 LWDD R1, #200 ; R1 := Inhalt Speicher(200 + 201) = a
104 ADD R1        ; Akku := Akku + R1 = s
106 ...
```


Befehle – die „Wörter“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Einfache Addition

- Wie könnte ein äquivalentes Programm mit *mnemonischen Symbolen* (Assemblerprogramm) lauten?

Programm-Code:

$s := b + a$

$r := 4 + b$

$s := s + r$

$\Rightarrow s := b + a$

$r := b + 4$

$s := s + r$

```

100 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b
102 LWDD R1, #200 ; R1 := Inhalt Speicher(200 + 201) = a
104 ADD R1 ; Akku := Akku + R1 = s
106 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku = s
108 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b
110 ADDD #4 ; Akku := Akku + 4 = r
112 SWDD R0, #206 ; Inhalt Speicher(206 + 207) := Akku = r
114 LWDD R0, #204 ; Akku := Inhalt Speicher(204 + 205) = s
116 LWDD R1, #206 ; R1 := Inhalt Speicher(206 + 207) = r
118 ADD R1 ; Akku := Akku + R1 = s
120 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku = s
    
```

Befehle – die „*Wörter*“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Einfache Addition

- Wie lang ist die Ausführungszeit näherungsweise?

Programm-Code:

```
s := b + a  
r := 4 + b  
s := s + r
```

5 Ladebefehle: **5 * 200 ns**

3 Speicherbefehle: **3 * 200 ns**

3 Additionsbefehle: **3 * 200 ns**

=> Ausführungszeit: 2200 ns

Anmerkung: Ein allfälliger **Überlauf** bei der Addition wurde vernachlässigt!

Befehle – die „*Wörter*“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Einfache Addition

- Wie lang ist die Ausführungszeit näherungsweise?

Programm-Code:

```
s := b + a  
r := 4 + b  
s := s + r
```

5 Ladebefehle: **5 * 200 ns**

3 Speicherbefehle: **3 * 200 ns**

3 Additionsbefehle: **3 * 200 ns**

=> Ausführungszeit: 2200 ns

Geht das mit den gegebenen Befehlen nicht besser?

Befehle – die „Wörter“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Einfache Addition (2)

- Wie könnte ein äquivalentes Programm mit *mnemonischen Symbolen* (Assemblerprogramm) lauten?

Programm-Code:

$s := b + a$

$r := 4 + b$

$s := s + r$

=> $s := b + a$

$r := b + 4$

$s := r + s$

```
100 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b
102 LWDD R1, #200 ; R1 := Inhalt Speicher(200 + 201) = a
104 ADD R1        ; Akku := Akku + R1          = s
106 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku = s
108 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b
110 ADDD #4       ; Akku := Akku + 4           = r
112 LWDD R1, #204 ; R1:= Inhalt Speicher(204 + 205) = s
114 ADD R1        ; Akku := Akku + R1          = s
116 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku = s
```

Befehle – die „*Wörter*“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Einfache Addition (2)

– Wie lang ist die Ausführungszeit näherungsweise?

Programm-Code:

$s := b + a$

$r := 4 + b$

$s := s + r$

=> $s := b + a$

$r := b + 4$

$s := r + s$

4 Ladebefehle: **$4 * 200\text{ ns}$**

2 Speicherbefehle: **$2 * 200\text{ ns}$**

3 Additionsbefehle: **$3 * 200\text{ ns}$**

=> Ausführungszeit: 1800 ns

Anmerkung: Ein allfälliger **Überlauf** bei der Addition wurde vernachlässigt!

=> Geht das noch besser?

Befehle – die „Wörter“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Einfache Addition (3)

- Wie könnte ein äquivalentes Programm mit *mnemonischen Symbolen* (Assemblerprogramm) lauten?

Programm-Code:

```
s := b + a  
r := 4 + b  
s := s + r  
  
=> s := b * 2  
    s := s + 4  
    s := s + a
```

```
100 LWDD R0, #202 ; Akku := Inhalt Speicher(202 + 203) = b  
102 SLA          ; Akku := Akku * 2 (Shift arithmetisch) = 2 * b  
104 ADDD #4      ; Akku := Akku + 4 = 2 * b + 4  
106 LWDD R1, #200 ; R1 := Inhalt Speicher(200 + 201) = a  
108 ADD R1       ; Akku := Akku + R1 = s + a  
110 SWDD R0, #204 ; Inhalt Speicher(204 + 205) := Akku = s
```

Befehle – die „*Wörter*“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Einfache Addition (3)

– Wie lang ist die Ausführungszeit näherungsweise?

Programm-Code:

```
s := b + a  
r := 4 + b  
s := s + r  
  
=> s := b * 2  
    s := s + 4  
    s := s + a
```

2 Ladebefehle: **2 * 200 ns**

1 Speicherbefehle: **1 * 200 ns**

2 Additionsbefehle: **2 * 200 ns**

1 Shift-Operation: **1 * 200 ns**

=> Ausführungszeit: 1200 ns

Anmerkung: Ein allfälliger **Überlauf** bei der Addition wurde vernachlässigt!

Befehle – die „Wörter“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ „Prozessormodell“ (vereinfacht): Befehlssatz

- | | | | |
|--------------------------|----|--|------------------------------|
| ▪ CLR <i>Rnr</i> | => | « <i>Rnr</i> » := 0 | |
| ▪ ADD <i>Rnr</i> | => | Akku := Akku + « <i>Rnr</i> » | MSB / LSB |
| ▪ ADDD #Zahl | => | Akku := Akku + #Zahl | = Most/Least significant Bit |
| ▪ INC | => | Akku := Akku + 1 | |
| ▪ DEC | => | Akku := Akku - 1 | e.g. 10000101 |
| | | | MSB LSB |
| ▪ LWDD <i>Rnr</i> , #Adr | => | « <i>Rnr</i> » := Inhalt Speicher(Adr) | |
| ▪ SWDD <i>Rnr</i> , #Adr | => | Inhalt Speicher(Adr) := « <i>Rnr</i> » | |
| ▪ SRA | => | Akku: Verschieben arithmetisch nach rechts | |
| ▪ SLA | => | Akku: Verschieben arithmetisch nach links | |
| ▪ SRL | => | Akku: Verschieben logisch nach rechts | |
| ▪ SLL | => | Akku: : Verschieben logisch nach links | |

Anmerkung: Bei arithmetischen Operationen (ADD, ADDD, SRA, SLA) bleibt das **MSb** des **MSB** erhalten. Zudem wird das **Carry-Flag** (auch bei allen Schiebe-Operationen) entsprechend gesetzt.

Befehle – die „Wörter“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ „Prozessormodell“ (vereinfacht): Befehlssatz (2)

- AND *Rnr* => *Akku* := *Akku* AND «*Rnr*» (bitweise)
- OR *Rnr* => *Akku* := *Akku* OR «*Rnr*» (bitweise)
- NOT => *Akku* := -*Akku* (bitweise negiert, alle Bit)

- BZ *Rnr* => Wenn *Akku* = 0, verzweige (Adresse «*Rnr*»)
- BNZ *Rnr* => Wenn *Akku* ≠ 0, verzweige (Adresse «*Rnr*»)
- BC *Rnr* => Wenn das Carry-Flag gesetzt ist, verzweige
 (zu Adresse «*Rnr*»)
- B *Rnr* => Verzweige (Adresse «*Rnr*»)

- BZD #*Adr* => Wenn *Akku* = 0, verzweige (Adresse #*Adr*)
- BNZD #*Adr* => Wenn *Akku* ≠ 0, verzweige (Adresse #*Adr*)
- BCD #*Adr* => Wenn das Carry-Flag gesetzt ist, verzweige
 (zu Adresse #*Adr*)
- DBD #*Adr* => Verzweige (zu Adresse #*Adr*)

Befehle – die „*Wörter*“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Summenbildung (über einfache *for*-Schleife)

– Aufgabe:

Realisieren Sie mit einer einfachen *for*-Schleife:

$$S = \sum_{k=i}^j k \quad \text{mit } j > i \text{ und } i, j \in \mathbb{N}$$

Beispiel: Eingabe *i* und *j*

s = 0

For *k* = *i* to *j*

s := *s* + *k*

Next

Ausgabe *s*

Anmerkung: Das *Programm* liegt wiederum ab Speicher(offset) 100 im Speicher, die *Daten* (*i*, *j*, *s*) ab Speicher(offset) 200.

Befehle – die „Wörter“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Summenbildung (über einfache *for-Schleife*)

– Aufgabe: $S = \sum_{k=i}^j k$ (und $j > i$)

Programm-Code:

Eingabe i und j

$s = 0$

For $k = i$ to j

$s := s + k$

Next

Ausgabe s

```

100 LWDD R0, #200 ; Akku = k
102 SWDD R0, #204 ; Summe s initialisieren und speichern
104 LWDD R0, #200 ; Akku = k
106 INC ; k = k + 1
108 SWDD R0, #200 ; Inkrementierten Wert speichern
110 LWDD R1, #204 ; R1 = s
112 ADD R1 ; s = s + k
114 SWDD R0, #204 ; Summe speichern
116 LWDD R0, #200 ; k (inkrementierter Wert) laden
118 NOT ; Akku = Akku invertieren
120 INC ; Akku = Akku + 1 (= 2er-Komplement von k)
122 LWDD R1, #202 ; R1 = j
124 ADD R1 ; Akku := Akku + R1 (= j - k)
126 BNZD #104 ; Wenn j - k ≠ 0 springe zu 104 zurück
  
```

Befehle – die „Wörter“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

- **Summenbildung (über einfache *for*-Schleife)**
 - Aufgabe: $S = \sum_{k=i}^j k$ mit $i = 4$ und $j = 8$
 - Analyse Programmschritte: (proportional Ausführungszeit)

Programm-Code:

Eingabe i und j
 $s = 0$
 For $k = i$ to j
 $s := s + k$
 Next
 Ausgabe s

100	LWDD R0, #200	; Akku = k
102	SWDD R0, #204	; Summe s initialisieren und speichern
104	LWDD R0, #200	; Akku = k
106	INC	; k = k + 1
108	SWDD R0, #200	; Inkrementierten Wert speichern
110	LWDD R1, #204	; R1 = s
112	ADD R1	; s = s + k
114	SWDD R0, #204	; Summe speichern
116	LWDD R0, #200	; k (inkrementierter Wert) laden
118	NOT	; Akku = Akku invertieren
120	INC	; Akku = Akku + 1 (=> 2er-Komplement von k)
122	LWDD R1, #202	; R1 = j
124	ADD R1	; Akku := Akku + R1 (= j - k)
126	BNZD #104	; Wenn j - k ≠ 0 springe zu 104 zurück

Initialisierung: 2 Befehle

Befehle – die „Wörter“ eines Rechners

Programmbeispiele mit *mnemonischen Symbolen*:

▪ Summenbildung (über einfache *for*-Schleife)

- Aufgabe: $S = \sum_{k=i}^j k$ mit $i = 4$ und $j = 8$
- Analyse Programmschritte: $2 + 4 \times 12 = 50$

Programm-Code:

Eingabe i und j
 $s = 0$
 For $k = i$ to j
 $s := s + k$
 Next
 Ausgabe s

100	LWDD R0, #200	; Akku = k	Initialisierung: 2 Befehle
102	SWDD R0, #204	; Summe s initialisieren und speichern	
104	LWDD R0, #200	; Akku = k	Schleife: 12 Befehle wird 4x durchlaufen
106	INC	; k = k + 1	
108	SWDD R0, #200	; Inkrementierten Wert speichern	
110	LWDD R1, #204	; R1 = s	
112	ADD R1	; s = s + k	
114	SWDD R0, #204	; Summe speichern	
116	LWDD R0, #200	; k (Inkrementierter Wert) laden	
118	NOT	; Akku = Akku invertieren	
120	INC	; Akku = Akku + 1 (=> 2er-Komplement von k)	
122	LWDD R1, #202	; R1 = j	
124	ADD R1	; Akku := Akku + R1 (= j - k)	
126	BNZD #104	; Wenn j - k ≠ 0 springe zu 104 zurück	

Befehle – die „Wörter“ eines Rechners

Programmbeispiele: (Variante)

▪ Summenbildung (über einfache *for*-Schleife)

– Aufgabe: $S = \sum_{k=i}^j k$ (und $j > i$)

Programm-Code:

Eingabe i und j

$s = 0$

For $k = i$ to j

$s := s + k$

Next

Ausgabe s

```

100 LWDD R0, #200    ; Akku = i
102 LWDD R2, #200    ; R2 = i
104 SWDD R0, #204    ; Summe initialisieren und speichern
106 NOT              ; Akku = Akku invertieren
108 INC              ; Akku = Akku + 1 (= 2er-Komplement von i)
110 LWDD R1, #200    ; R1 = j
112 ADD R1           ; Akku = j - i = k
114 SWDD R0, #206    ; Zähler k zwischenspeichern
116 LWDD R1, #204    ; R1 = s
118 ADD R1           ; Summe aufaddieren
120 ADD R2           ; Summe aufaddieren
122 SWDD R0, #204    ; Summe speichern
124 ...

```

Befehle – die „Wörter“ eines Rechners

Programmbeispiele: (Variante)

▪ Summenbildung (über einfache *for*-Schleife)

– Aufgabe: $S = \sum_{k=i}^j k$ (und $j > i$)

Programm-Code:

Eingabe i und j

$s = 0$

For $k = i$ to j

$s := s + k$

Next

Ausgabe s

122 ...

124 LWDD R0, #206 ; Zähler k laden (für Schleife)

126 DEC ; Zähler dekrementieren

128 SWDD R0, #206 ; Zähler k zwischenspeichern

130 BNZD #116 ; Wenn Zähler $k \neq 0$ springe zu 116 zurück

Befehle – die „Wörter“ eines Rechners

Programmbeispiele: (Variante)

▪ Summenbildung (über einfache *for*-Schleife)

– Aufgabe: $S = \sum_{k=i}^j k$ mit $i = 4$ und $j = 8 \Rightarrow$ Analyse Prog.-Schritte:

Programm-Code:

Eingabe i und j

$s = 0$

For $k = i$ to j

$s := s + k$

Next

Ausgabe s

100 LWDD R0, #200	; Akku = i
102 LWDD R2, #200	; R2 = i
104 SWDD R0, #204	; Summe initialisieren und speichern
106 NOT	; Akku = Akku invertieren
108 INC	; Akku = Akku + 1 (= 2er-Komplement von i)
110 LWDD R1, #200	; R1 = j
112 ADD R1	; Akku = $j - i = k$
114 SWDD R0, #206	; Zähler k zwischenspeichern
116 LWDD R1, #204	; R1 = s
118 ADD R1	; Summe aufaddieren
120 ADD R2	; Summe aufaddieren
122 SWDD R0, #204	; Summe speichern
124 LWDD R0, #206	; Zähler k laden (für Schleife)
126 DEC	; Zähler dekrementieren
128 SWDD R0, #206	; Zähler k zwischenspeichern
130 BNZD #116	; Wenn Zähler $k \neq 0$ springe zu 116 zurück

Befehle – die „Wörter“ eines Rechners

Programmbeispiele: (Variante)

▪ Summenbildung (über einfache *for*-Schleife)

– Aufgabe: $S = \sum_{k=i}^j k$ mit $i = 4$ und $j = 8 \Rightarrow$ Analyse Prog.-Schritte:

Programm-Code:

Eingabe i und j

$s = 0$

For $k = i$ to j

$s := s + k$

Next

Ausgabe s

Initialisierung: 8 Befehle

```

100 LWDD R0, #200      ; Akku = i
102 LWDD R2, #200      ; R2 = i
104 SWDD R0, #204      ; Summe initialisieren und speichern
106 NOT                ; Akku = Akku invertieren
108 INC                ; Akku = Akku + 1 (= 2er-Komplement von i)
110 LWDD R1, #200      ; R1 = j
112 ADD R1              ; Akku = j - i = k
114 SWDD R0, #206      ; Zähler k zwischenspeichern

116 LWDD R1, #204      ; R1 = s
118 ADD R1              ; Summe aufaddieren
120 ADD R2              ; Summe aufaddieren
122 SWDD R0, #204      ; Summe speichern
124 LWDD R0, #206      ; Zähler k laden (für Schleife)
126 DEC                ; Zähler dekrementieren
128 SWDD R0, #206      ; Zähler k zwischenspeichern
130 BNZD #116          ; Wenn Zähler k ≠ 0 springe zu 116 zurück
  
```

Befehle – die „Wörter“ eines Rechners

Programmbeispiele: (Variante)

▪ Summenbildung (über einfache *for*-Schleife)

– Aufgabe: $S = \sum_{k=i}^j k$ Analyse Prog.-Schritte: $8 + 4 \times 8 = 40$

Programm-Code:

Eingabe i und j

$s = 0$

For $k = i$ to j

$s := s + k$

Next

Ausgabe s

100	LWDD R0, #200	; Akku = i
102	LWDD R2, #200	; $R2 = i$
104	SWDD R0, #204	; Summe initialisieren und speichern
106	NOT	; $Akku = Akku$ invertieren
108	INC	; $Akku = Akku + 1$ (= 2er-Komplement von i)
110	LWDD R1, #200	; $R1 = j$
112	ADD R1	; $Akku = j - i = k$
114	SWDD R0, #206	; Zähler k zwischenspeichern
116	LWDD R1, #204	; $R1 = s$
118	ADD R1	; Summe aufaddieren
120	ADD R2	; Summe aufaddieren
122	SWDD R0, #204	; Summe speichern
124	LWDD R0, #206	; Zähler k laden (für Schleife)
126	DEC	; Zähler k dekrementieren
128	SWDD R0, #206	; Zähler k zwischenspeichern
130	BNZD #116	; Wenn Zähler $k \neq 0$ springe zu 116 zurück

Initialisierung: 8 Befehle

Schleife: 8 Befehle

werden 4x durchlaufen

Befehle – die „Wörter“ eines Rechners

Maschinensprache (*Operations-Code*):

- Ein Befehl in *Maschinensprache* besteht aus einer festgelegten Bitfolge (i. d. R. Wortlänge oder Vielfaches davon).
- In einem Befehl sind – wie bereits erwähnt – auch die **Operanden** und **Optionen** kodiert.
- Beispiel: *00100x<Adresse>* entspricht: **BD #Adr**
 => *00100000 01100100* entspricht: **BD #100**
 - **Bit 1 bis Bit 5** spezifizieren den **Befehl**:
 - => Im Beispiel: **BD** steht für „**Branch direct**“, „**direkter**“ oder auch „**unbedingter**“ Sprung
 - **Bit 6** ist **nicht relevant** (kann den Wert **0** oder **1** haben).
 - **Bit 7 bis Bit 16** geben eine absolute Adresse an.

Befehle – die „Wörter“ eines Rechners

Maschinensprache (*Operations-Code*):

- Weitere Beispiele (für das „Prozessormodell“)

Maschinen-Code (Op-Code)	Mnemonics ("Assembler")	Kurzbeschreibung	Beschreibung
0 0 0 0 x x 1 0 1 < n o t u >	CLR Rnr	«Rxx» := 0	Lösche das Register «Rxx» (alle Bit auf 0 setzten) und das Carry-Flag (00 bis 11 für: Akku, R-1, R-2 bzw. R-3).
0 0 0 0 x x 1 1 1 < n o t u >	ADD Rnr	Akku := Akku + «Rxx»	Addition zweier 16-Bit-Zahlen (Zahl im Akku und Zahl im Register «Rxx»; 00 bis 11 für Akku, R-1, R-2 bzw. R-3) im 2er-Komplement; bei Überlauf wird das Carry-Flag gesetzt (= 1), sonst auf den Wert 0.
1 < - - - - Z a h l - - - - >	ADDD #Zahl	Akku := Akku + #Zahl	Addition der 16-Bit-Zahl im Akku mit der 15-Bit-Zahl als direkten Operanden im 2er-Komplement; bei Überlauf wird das Carry-Flag gesetzt (= 1), sonst auf den Wert 0. Vor der Addition wird die 15-Bit-Zahl des Operanden auf 16 Bit erweitert (mit MSb des MSB auf 1 wenn negativ, sonst auf 0).
0 0 0 0 0 0 0 1 n o t u s e d	INC	Akku := Akku + 1	Der Akku (16-Bit-Zahl im 2er-Komplement) wird um den Wert 1 inkrementiert; bei Überlauf wird das Carry-Flag gesetzt (= 1), sonst auf den Wert 0.
0 0 0 0 0 1 0 0 n o t u s e d	DEC	Akku := Akku - 1	Der Akku (16-Bit-Zahl im 2er-Komplement) wird um den Wert 1 dekrementiert; bei Überlauf wird das Carry-Flag gesetzt (= 1), sonst auf den Wert 0.

- Vollständiger Befehlssatz für das „Prozessormodell“ siehe Moodle bzw. ausgeteiltes Zusatzblatt.

Befehle – die „*Wörter*“ eines Rechners

Maschinensprache (*Operations-Code*):

- Für das „*Prozessormodell*“

- Alle Befehle sind gleich lang (16 Bit – ein Wort).

(Vereinfacht die Realisierung – muss aber nicht so sein; Befehle können auch unterschiedlich lang innerhalb eines Befehlssatzes sein.)

- Es gibt 22 verschiedene Befehle:

- 5 arithmetische Befehle
 - 7 logische Befehle
 - 2 Lade- / Speicherbefehle
 - 8 Sprungbefehle
 - 1 END-Befehl

Vollständiger Befehlssatz für das „*Prozessormodell*“: siehe Moodle bzw. ausgeteiltes Zusatzblatt.

Befehle – die „Wörter“ eines Rechners

Maschinensprache (*Operations-Code*):

- Für die Ausführung eines Programms muss der mit *mnemonischen Symbolen* (*Assembler*) geschriebene *Programm-Code* in *Maschinensprache*, den *Operations-Code*, übersetzt werden.
- Dieses erfolgt in der Regel durch einen (für die konkrete *Maschinensprache* entwickelten) *Assembler-Compiler*.

Sowohl die *Sprache* (für eine konkrete Architektur) als auch der zugehörige *Compiler* werden *Assembler* genannt:

- *Assembler-Sprache*
- *Assembler-Compiler*

Befehle – die „Wörter“ eines Rechners

Maschinensprache (*Operations-Code*):

- Beispiel: Programm zur „*Addition*“, 3. Version

- a) *Assembler-Code*:

100	LWDD R0, #202	; Akku := Inhalt Speicher(202 + 203)
102	SLA	; Akku := Akku * 2 (<i>Shift arithmetisch</i>)
104	ADDD #4	; Akku := Akku + 4
106	LWDD R1, #200	; R1:= Inhalt Speicher(200 + 201)
108	ADD R1	; Akku := Akku + R1
110	SWDD R0, #204	; Inhalt Speicher(204 + 205) := Akku

- b) *Maschinen-Code* (vereinfacht):

100	01000000 11001010	; LWDD R0, #202
102	00001000 00000000	; SLA
104	10000000 00000100	; ADDD #4
106	01000100 11001000	; LWDD R1, #200
108	00000111 10000000	; ADD R1
110	01100000 11001100	; SWDD R0, #204

Befehle – die „Wörter“ eines Rechners

Maschinensprache (*Operations-Code*):

▪ Beispiel: Programm zur „*Summenbildung*“, 1. Version

a) *Assembler*-Code:

100	LWDD R0, #200	; Akku = k
102	SWDD R0, #204	; Summe s initialisieren und speichern
104	LWDD R0, #200	; Akku = k
106	INC	; k = k + 1
108	SWDD R0, #200	; Inkrementierten Wert speichern
110	LWDD R1, #204	; R1 = s
112	ADD R1	; s = s + k
114	SWDD R0, #204	; Summe speichern
116	LWDD R0, #200	; k (inkrementierter Wert) laden
118	NOT	; Akku = Akku invertieren
120	INC	; Akku = Akku + 1 (= 2er-Komplement von k)
122	LWDD R1, #202	; R1 = j
124	ADD R1	; Akku = Akku + R1 (= j - k)
126	BNZD #104	; Wenn j - k ≠ 0 springe zu 104 zurück

Befehle – die „Wörter“ eines Rechners

Maschinensprache (*Operations-Code*):

- Beispiel: Programm zur „*Summenbildung*“, 1. Version

a) *Assembler*-Code:

100	LWDD R0, #200	=>	01000000 11001000
102	SWDD R0, #204	=>	01100000 11001100
104	LWDD R0, #200	=>	01000000 11001000
106	INC	=>	00000001 00000000
108	SWDD R0, #200	=>	01100000 11001000
110	LWDD R1, #204	=>	01000100 11001100
112	ADD R1	=>	00000111 10000000
114	SWDD R0, #204	=>	01100000 11001100
116	LWDD R0, #200	=>	01000000 11001000
118	NOT	=>	00000000 10000000
120	INC	=>	00000001 00000000
122	LWDD R1, #202	=>	01000100 11001010
124	ADD R1	=>	00000111 10000000
126	BNZD #104	=>	00101000 01101000

Befehle – die „*Wörter*“ eines Rechners

