Rechnerarchitektur

Marián Vajteršic und Helmut A. Mayer

Fachbereich Computerwissenschaften
Universität Salzburg
marian@cosy.sbg.ac.at und helmut@cosy.sbg.ac.at
Tel.: 8044-6344 und 8044-6315

28. September 2017

Lehrveranstaltungszeiten

Dienstag 12:00–13:30, HS T01/T02
Sprechstunden: jeweils nach der Lehrveranstaltung

Lehrveranstaltungsinhalte

Rechnerarchitektur
- Die Von-Neumann-Architektur
- Maschinenarchitektur
- Speicheradressierung
- Operationen des Befehlssatzes
- Performance eines Rechnersystems

Die DLX-Maschine

Pipelining
- Die DLX-Pipeline
- Pipeline Hazards

Literatur

- J. L. Hennessy, D. A. Patterson: Rechnerarchitektur. Vieweg Verlag, 1996
- A. S. Tanenbaum, T. Austin: Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner. Pearson, 2014

Folien und Übungszettel

- **Folien zur Lehrveranstaltung:** elektronisch abrufbar unter (PLUSonline)
  https://online.uni-salzburg.at
  UV Rechnerarchitektur → LV-Unterlagen
- **Übungszettel:** elektronisch abrufbar unter (PLUSonline)
  https://online.uni-salzburg.at
  UV Rechnerarchitektur → LV-Unterlagen

Rechnerarchitektur

Rechnerarchitektur

Im Prinzip: Ein heutiger **Rechner** (Computer) ist ein einziges großes Schaltwerk, gesteuert von einer zentralen Taktfrequenz. Die Komplexität eines solchen Schaltwerks ist sehr groß.

$\rightarrow$ Aufteilung in wesentliche Teilschaltwerke, diese betrachtet man als **Teile des Aufbaus von Rechnern**.

Rechnerarchitektur

Von-Neumann-Architektur: Die Grundstruktur sequenzieller Rechner
(John von Neumann war der erste, der diese Struktur formuliert hat.)

Diese prinzipielle Architektur blieb unverändert; der Fortschritt an Rechenleistung beruht auf technologischen Innovationen:

Moore's Law. The density of transistors on a chip will double every 18 months thus increasing the price performance of computing power by a factor of two every $1\frac{1}{2}$ years.

Die Von-Neumann-Architektur

Die Von-Neumann-Architektur

Was macht ein Schaltwerk zu einem Computer?

- Rechenwerk
- Speicher
- Programm

Bemerkung: Rechenwerk „rechnet“ (d. h. führt arithmetische/logische Operationen aus)
$\rightarrow$ ein Teil des Computers
(also **Computer ist komplexer als Rechenwerk**).

Die Von-Neumann-Architektur

Beispiel [Taschenrechner (als Vorstufe zu einem Computer)]

- Über eine **Eingabe (Input)**: Zahlen und Operationen eingeben (Tastatur)
- **Ausgabe (Output)**: Das Ergebnis (Display)
- **Rechenwerk**: Ausführung von verschiedenen arithmetischen und logischen Operationen

Die Von-Neumann-Architektur

- Operationen **ohne Speicher**
  Addieren wir $A \oplus B$ und danach $C \oplus D$, so ist $A \oplus B$ vergessen. (Um das Ergebnis $A \oplus B$ neuerlich zu erhalten, muss man die Eingabe und die Addition wiederholen.)
- Operationen **mit dem Speicher**
  Wir können die Ergebnisse sowie auch die einzelnen Daten speichern, um diese aus dem Speicher zu holen. Man muss wissen, wo diese im Speicher „beheimatet“ sind (also an welcher **Adresse**).

$\rightarrow$ Wir benötigen eine **Kennzeichnung des Speichers** (genannt **Adressierung**).

Die Von-Neumann-Architektur

Mit den gespeicherten Daten können wir die Operationen ausführen.

- Alternative 1:
  Daten holen, Operation 1 **eingeben/ausführen** → Ergebnis 1
  (Neue) Daten holen, Operation 2 **eingeben/ausführen** → Ergebnis 2
  ...
- Alternative 2
  (Nicht **nur** die Daten, sondern) auch **die Abfolge der Operationen** (Operation 1, Operation 2, ...), d. h. ein Programm (P) im Speicher ablegen.

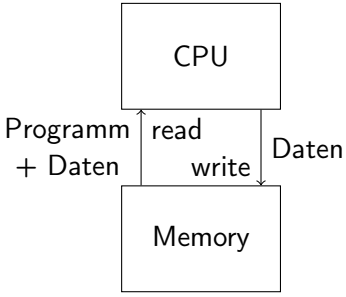
  Wenn sich die Daten ändern, so brauchen wir dem Rechner nur mitzuteilen, das Programm P auszuführen. Das Programm würde dann die Daten von den entsprechenden Speicherstellen **holen**, sie **verarbeiten** und die Ergebnisse **abspeichern** (um die Ergebnisse wieder verwenden zu können).

Die Von-Neumann-Architektur

Auch das Programm selbst muss aus dem Speicher geholt werden!

→ Die wesentliche Erkenntnis der **Von-Neumann-Architektur**:
Programm und Daten in einem gemeinsamen Speicher.
(Diese Architektur wird deswegen auch Stored-Program-Architektur genannt.)

Schema der Von-Neumann-Architektur



CPU: Central Processing Unit (Kernstück davon: Prozessor)

Holt Programm und Daten aus dem gemeinsamen Speicher (Memory)

Führt die einzelnen Anweisungen (Instructions) aus und schreibt Ergebnisdaten wieder in den Speicher zurück.

Schema der Von-Neumann-Architektur

Die CPU führt folgende Schritte aus:

1. Hole Anweisung aus dem Speicher
2. Hole die Daten für diese Anweisung
3. Führe die Anweisung aus
4. Speichere das Ergebnis der Anweisung
5. Gehe zu 1. oder Programmende

Implizit angenommen: Programm liegt linear in Speicherzellen.

Die Leitungen zwischen CPU und Speicher heißen **Bus**. Man unterscheidet zwischen Adressbus und Datenbus.

Schema der Von-Neumann-Architektur

Um ein Datum oder die Anweisung aus dem Speicher zu holen, muss die CPU zuerst die Adresse des jeweiligen Speicherplatzes angeben.
$\rightarrow$ Die Binärdarstellung der **Adresse** wird an den **Adressbus** angelegt (was voraussetzt, dass die **Busbreite** (Anzahl der parallelen Leitungen) diese Adresse aufnehmen kann).

Hat die CPU eine Adresse ausgewählt, so werden die Daten/Instruktionen über den **Datenbus** zurück an die CPU gesendet.

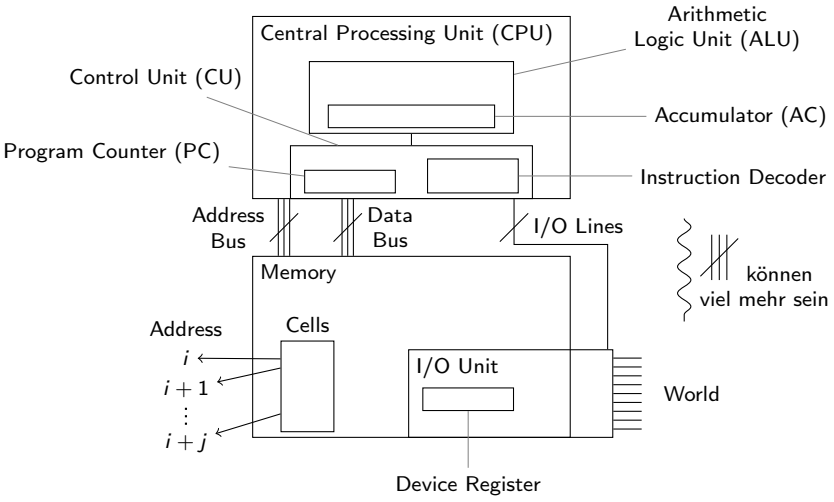
(Für die Übertragung von einem Byte sollte der Datenbus aus 8 parallelen Leitungen bestehen; ein 16-Bit-Wort müsste dann in zwei Schritten (von zwei verschiedenen Adressen) geholt werden $\rightarrow$ das dauert also doppelt so lange.)

Der Universalrechenautomat (nach Händler)

Ziel: Einzelne Komponenten (CPU, Speicher, Bus) mit mehr Details

Dazu werden wir eine Verfeinerung der prinzipiellen Von-Neumann-Architektur behandeln: Universalrechenautomat (Händler).

Der Universalrechenautomat (nach Händler)



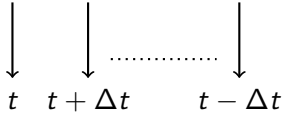
Der Universalrechenautomat (nach Händler)

Die Unterteilung von **CPU** und **Memory** (Speicher)

- CPU
  - Control Unit (CU)
    (Steuerwerk für Programmsteuerung und Befehlscodierung)
  - Arithmetic Logical Unit (ALU)
    (Rechenwerk für arithmetische und logische Operationen)
- Memory
  - Adressierbare Speicherzellen
  - I/O Unit
    (Ein-/Ausgabe von Daten und Programmen (Tastatur(I), Monitor(O), Maus(I), Drucker(O), Scanner(I), Modem(I/O), Festspeicher(O)))

Der Universalrechenautomat (nach Händler)

- Verbindung CPU-Speicher (Bus)
  Adressbus, Datenbus
  - Bus: Menge von Leitungen, die CPU mit Speicher verbindet
    Adressbus: Von welcher Datenstelle Daten geholt werden
    Datenbus: Darüber werden Daten vom/zum Speicher transferiert
  - Busbreite: Anzahl von parallelen 1-Bit-Leitungen
    (8-Bit-Bus: 8 Leitungen (auf 1 Leitung kommt 1 Bit))
    Parallel: Alle zur selben Zeit behandelt
    Sonst: Mischung aus Zukunft und Vergangenheit



Der Universalrechenautomat (nach Händler)

Die Zusammenfassung der Funktion des URA:

1. Der URA wird logisch und räumlich in die Teile CPU (CU, ALU), Memory und I/O-Unit zerlegt.
2. - Das Programm des URA wird über die I/O-Unit eingegeben und im Speicher abgelegt.
   - Das Programm ermöglicht die Bearbeitung eines Problems, das von der Struktur des Rechners unabhängig ist.
3. Programm und Daten sind im selben Speicher.
4. Der Speicher ist eine nummerierte Folge von Speicherzellen. Die Nummer einer Speicherzelle ist deren Adresse, **über die der Inhalt** der Speicherzelle **gelesen** oder **geschrieben** werden kann.
5. Ein Programm ist eine Folge von Befehlen, die in aufeinanderfolgenden Speicherzellen abgelegt sind. Wird ein Befehl von der Adresse a geholt und abgearbeitet, dann wird der nächste Befehl (im Normalfall) von der Adresse $a + 1$ geholt.

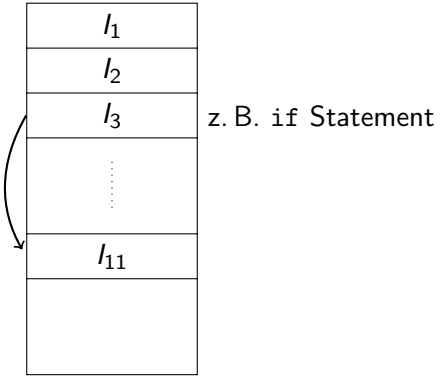
Der Universalrechenautomat (nach Händler)

6. **Sprungbefehle** bewirken eine Änderung der Reihenfolge der Befehlsbearbeitung, die bewirkt, dass der nächste Befehl nicht von der Adresse $a + 1$ sondern von $a \pm n$ geholt wird.
7. **Bedingte Sprünge** sind Sprungbefehle, die nur ausgeführt werden, wenn eine (programmierte) Bedingung erfüllt wird. Ist die Bedingung nicht erfüllt, wird der nächste Befehl (wie üblich) von der Adresse $a + 1$ geholt.
8. Programm und Daten sind binär codiert.

Punkt 7. ermöglicht, dass sich das Programm in Abhängigkeit von den (zunächst unbekannten) Eingabedaten verhält (siehe Abbildung auf der nächsten Folie).
$\rightarrow$ Die Flexibilität von URA (ohne dieser Eigenschaft wären z. B. Benutzeroberflächen (Interaktion mit dem Anwender) nicht möglich (und damit der Rechner nicht universell verwendbar)).

Der Universalrechenautomat (nach Händler)

Bedingter Sprung (Branch)



Maschinenarchitektur

Die Architektur des Befehlssatzes

Der **Befehlssatz** der verwendeten CPU ist die primäre Schnittstelle **zwischen** einem Programmierer und einem Computer.

Die Übersetzung der Anweisungen einer Programmierhochsprache in Maschinenbefehle übernimmt ein Compiler.

$\rightarrow$ Ein **Compiler (Übersetzer)** ist ein Hilfsmittel, das es dem Programmierer erlaubt, eine **CPU** zu programmieren, **ohne deren** intrinsischen **Befehlssatz zu kennen**.

(Die direkte Programmierung in Maschinensprache ist doch immer nötig in zeit- und speicherkritischen Anwendungen).

Die Architektur des Befehlssatzes

Wichtigste Frage beim Entwurf von CPU: Wie soll **die Art und Anzahl der verschiedenen Befehle** einer **Maschinensprache** sein?

- Einerseits sollten die Befehle die Programmierung von allen möglichen Anwendungen ermöglichen.
- Andererseits erhöhen zu viele verschiedene Befehle die Komplexität einer CPU und damit wird auch die Geschwindigkeit negativ beeinflusst.

Maschinenarchitekturen

Die Unterteilung hängt vom Aufbau des **internen Speichers der CPU** ab, denn die Architektur des Befehlssatzes ist wesentlich durch den Aufbau, die Manipulierbarkeit und Kapazität **des internen Speichers** bestimmt (weil Daten und Befehle, die vom Speicher geholt werden müssen, zwischengespeichert werden, um die Befehle abarbeiten zu können).

Maschinenarchitekturen

Um die von uns im weiteren präsentierten **Maschinenarchitekturen** zu illustrieren, werden wir ein Maschinenprogramm zeigen:

Die Zahlen A, B **aus dem Speicher addieren** und das **Ergebnis** C **in die Speicherstelle** C ablegen.

(Hier werden symbolische Namen und keine Zahlen für die Speicherstellen benutzt.)

$\rightarrow$ Abstraktionsschritt von der **Maschinenprogrammierung** zur **Assemblerprogrammierung**

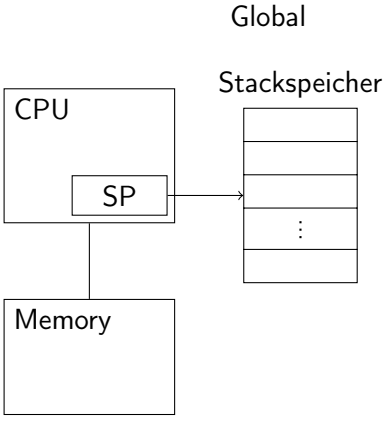
Maschinenarchitekturen

Ein **Assembler übersetzt** die **einzelnen Befehle** und die **symbolischen Adressen in Maschinensprache**

Im Assembler wird also **mit Namen und nicht mit binären Zahlen** gearbeitet.

(Was für den menschlichen Programmierer viel leichter ist (weil die Arbeit mit Namen wesentlich leichter ist als mit binären Zahlen).)

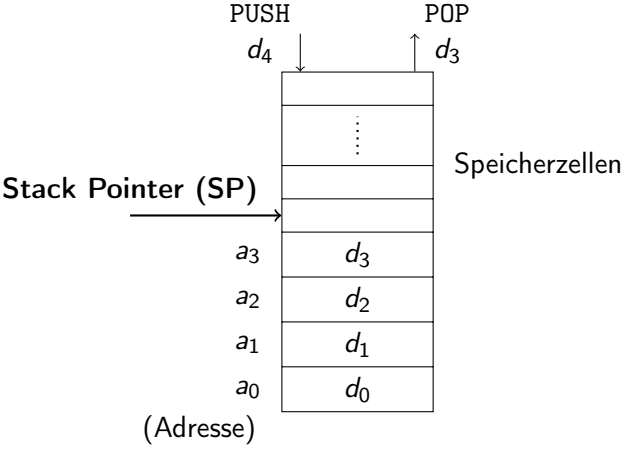
Die Stackmaschine



Bei Stackmaschinen befindet sich der **Stack**(speicher) **in** der **CPU** und es gibt dafür explizite Hardware.

Die Stackmaschine

Der **interne** Speicher: Stapelspeicher (Stack)



Die Stackmaschine

Stack Pointer (ein internes Register) **zeigt** immer auf **die nächste freie** Speicherstelle.

`PUSH`: Mit diesem Befehl wird vom Speicher ein neues Datum (d_4) auf den Stack (d. h. in die Speicherstelle auf die der Stack Pointer zeigt) gelegt.

`POP`: Holt das oberste Datum (d_3) vom Stack.

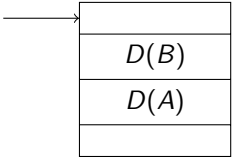
Die Stackmaschine

Beispiel

$C \leftarrow D(A) + D(B)$

`PUSH A`: Hole Datum $D(A)$ von Speicherstelle A (A: **Stelle im Hauptspeicher**) und lege es auf den Stack (Datum geholt).
(Es passiert noch etwas: Stack Pointer wird um 1 erhöht. Es gibt Architekturen, wo diese Erhöhung explizit gemacht werden muss; Manchmal wird es bei Stack automatisch gemacht.)

`PUSH B`: Hole Datum $D(B)$ aus der Speicherstelle B auf den Stack



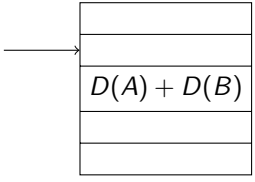
Die Stackmaschine

Beispiel (Fortsetzung)

`ADD`: (Heißt implizit) addiere die beiden Daten, die ganz oben liegen und speichere das Ergebnis am Stack.

- Man muss die Adressen von Operanden nicht mehr angeben, sobald diese auf dem Stack sind.
- Die einzelnen Befehle produzieren einen kompakten Code (**da nur die Operation, nicht aber die Adresse der Operanden codiert werden müssen**).
- Stack-Maschine: Null-Adress-Maschine

Danach schaut der Stack so aus:



(Werte $D(A)$ und $D(B)$ werden schon durch `ADD` Befehl gepoppt, sind also nicht mehr in der CPU)

Die Stackmaschine

Beispiel (Fortsetzung)

`POP C`: Hole Datum vom Stack und lege es in Speicherstelle C.

Warum wird bei heutigen CPUs nicht mehr die Stack-Architektur verwendet?

Es kann immer nur das oberste Datum am Stack explizit angesprochen werden.

$\rightarrow$ Die Daten müssen oft mehrmals aus dem Speicher geholt werden (z. B. $D(A)$ und $D(B)$ sind nach `ADD`-Operation nicht mehr in CPU vorhanden).

Die Stackmaschine

Der Stack-Speicher ist eine LIFO-Speicher (**L**ast **I**n **F**irst **O**ut).

Bemerkung: Dieser Speicher unterstützt die **U**mgekehrte **P**olnische **N**otation (UPN), bei der arithmetische Operationen in einer speziellen Reihenfolge ausgeführt werden.

Beispiel: $A + B$ wird geschrieben als $AB+$

$((A + B) \cdot (C + D))$ wird geschrieben als $AB + CD + \cdot$

(Wegfall von Klammern)

Die Stackmaschine

$AB + CD + \cdot$:

- AB: Hole Daten von den Speicherstellen A und B in den Stack
- $+$: Verknüpfe durch $+$ $D(A)$ und $D(B)$ und lege das Ergebnis auf den Stack ($D(A)$ und $D(B)$ werden gepoppt).
- CD: Hole $D(C)$, $D(D)$ auf den Stack.
- $+$: Verknüpfe $D(C)$ und $D(D)$ mit $+$ ($D(C)$ und $D(D)$ gepoppt).
- $\cdot$: Verknüpfe zwei oberste Werte (d. h. $D(C) + D(D)$ und $D(A) + D(B)$) mit mal.

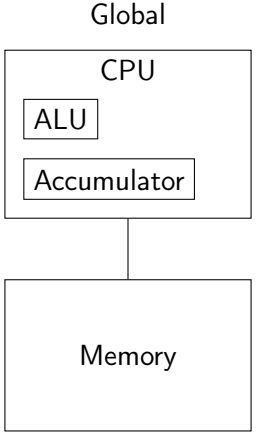
Virtual Maschine (VM)

Eine Virtual Maschine (VM) ist eine CPU, die nur aus Software besteht, d. h. es ist eine **Nachbildung einer physischen CPU**.

$\rightarrow$ Die VM **lebt im Speicher**
Hier entfällt der Nachteil des wiederholten Ladens von Daten in die CPU, aber die effiziente Befehlsabarbeitung bleibt erhalten.

Die Akkumulatormaschine

Hier enthält der interne CPU-Speicher ein **einziges internes Register**, das vom Programmierer angesprochen werden kann: **Akkumulator**.



Die Akkumulatormaschine

Das Beispiel $C \leftarrow D(A) + D(B)$:

- `LOAD A`: Hole Datum ($D(A)$) aus Speicherstelle A in den Akkumulator
- Wenn jetzt `LOAD B` gemacht werden würde, würde $D(A)$ im Akkumulator überschrieben werden.
- `ADD B`: Addiere $D(B)$ zum Akkumulatorinhalt.
- `STORE C`: Schreibe Akkumulatorinhalt nach C.
  (Also wird das Ergebnis gespeichert, um den Akkumulator für ein anderes Datum verwenden zu können.)

Die Akkumulatormaschine

Zu `ADD B`: Wie `ADD` codiert wird, hängt davon ab, **wie viele Befehle es gibt**.

B muss auch codiert werden.

`ADD`: 0100 und z. B. 12 Bits für die Speicheradresse

Somit verbraucht `ADD B` $4 + 12$ Bits.

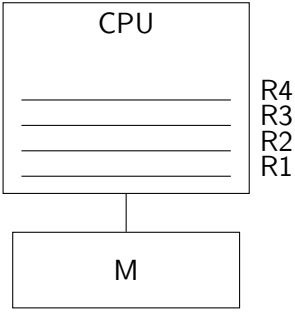
Die Befehle für die Akkumulatormaschine weisen nur **eine Adresse** auf $\rightarrow$ **Ein-Address-Maschine** (`ADD A`: nur die Adresse von *A* nötig).

Implizit läuft jeder Datentransfer und jede Operation über den Akkumulator, der nicht adressiert werden muss.

$\rightarrow$ kompakter Code, viele Speicherzugriffe (wie bei der Stackmaschine).

Die Registermaschinen

Die Registermaschinen enthalten einen Satz von allgemein verwendbaren Registern:



(Wenn nur 4 Register:
2 Bit für Adresse)

Die Registermaschinen

→ Der Vorteil: Oft verwendete Daten können in der CPU zwischengespeichert werden.

→ Die Befehle für das Nachladen von Daten entfallen.

Unterteilung von Registermaschinen:

- Register-Memory
- Register-Register

Die Registermaschinen

Das Beispiel mit Register-Memory-Maschine:

- `LOAD R1, A`: Hole Datum von Speicherstelle A nach Register $R1$.
- `ADD R1, B`: Addiere $D(B)$ zu $R1$.
- `STORE C, R1`: Speichere $R1$ nach C.

(Auch hier müsste man $D(A)$ und $D(B)$ bei mehrmaligen Verwendungen nachladen, aber durch einen zusätzlichen Befehl `ADD R1, R2` (Addition von Registern) könnte man die Teilergebnisse in der CPU zwischenspeichern.)

Da bei allen diesen Operationen immer zwei Operanden adressiert werden müssen $\rightarrow$ **Zwei-Address-Maschine**

Die Registermaschinen

Das Beispiel mit Register-Register-Maschine:

- `LOAD R1, A`: Hole Datum von Speicherstelle A nach Register $R1$.
- `LOAD R2, B`: Hole Datum von B nach $R2$.
- `ADD R3, R2, R1`: Addiere Inhalt von $R1$ und $R2$ nach $R3$.
- `STORE C, R3`: Speichere $R3$ nach C.

Nach diesen Anweisungen stehen alle Variablen und das Ergebnis (auch nach dem Abspeichern) in der CPU zur Verfügung.

Die Registermaschinen

Andere Namen:

- **Drei-Address-Maschine** (da es Befehle für Registeroperationen gibt, bei denen **drei** Register adressiert werden).
- **Load-Store-Maschine** (weil alle Daten in Register geladen werden, dort dann verarbeitet und von Registern wieder gespeichert werden).

Die meisten heutigen Maschinen sind vom Typ Register-Register.

Die Befehlszeiten sind in Taktzyklen (Clock Cycles) angegeben.

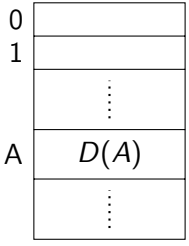
Wenn die Befehle ähnliche Ausführungszeiten haben
$\rightarrow$ **Pipelining** von Befehlen möglich.

Speicheradressierung

Speicheradressierung

Wie holt man die Daten aus dem Speicher?

Das Datum $D(A)$ liegt an der Adresse A
$\rightarrow$ **die Adresse ist eine Nummer**



Speicheradressierung

Wie viele Bits stehen an einer Speicheradresse?

Die meisten heutigen Prozessoren sind Byte-Adressiert (d. h. an jeder Adresse wird **ein Byte gespeichert**).

Zusätzlich können heute auch Halbwörter (16 Bits), Wörter (32 Bits) und Doppelwörter (64 Bits) im Speicher angesprochen werden, die sich über entsprechend viele Adressen erstrecken.

Speicheradressierung

Beispiel

In einem Speicher stehen ab Speicheradresse 1000 16 Bytes in hintereinanderliegenden Speicherzellen.

- Von Adresse 1007 kann man z. B. ein Byte lesen.
- Von Adresse 1004 kann man z. B. ein Wort lesen.
  (Wobei das nächste Wort an Adresse 1008 stehen würde.)
  (32 Bits (ein Wort) $= 4 \cdot 8$ Bits $= 4$ Bytes)
- Von Adresse 1008 können wir ein Doppelwort (8 Bytes) lesen.

Ausgerichtete Daten (Data Alignment)

Für **ausgerichtete Daten** gilt

$$A \mod s = 0$$

wobei A die Adresse eines Datums ist und s die Größe des Datums in Bytes.

- Ein Lesen/Schreiben eines Halbworts (2 Bytes) an einer ungeraden Adresse würde zu einem **Address Error** führen.
- Der Programmierer muss dafür sorgen, dass solche inkorrekten Zugriffe nicht auftreten.

Ein **Bus Error** tritt auf, wenn auf eine Speicherzelle zugegriffen wird, die es nicht gibt.

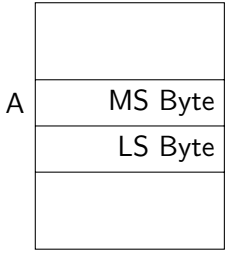
Byte Order

Ein weiteres Problem: **Byte Order** (**Anordnung der Bytes**)

Ist z. B. ein Halbwort an einer Adresse gespeichert, so gibt es genau **zwei Möglichkeiten**, die beiden Bytes abzuspeichern:

entweder steht das **MS Byte** (Most Significant Byte) an der Adresse oder das **LS Byte**.

- Wenn an der **ersten Speicherstelle** (d. h. Adresse) das **MS Byte** (Bits 15-8) steht:



→ Daten werden im **Big Endian**-Format (großes Ende) gespeichert:

MS Byte	LS Byte

Byte Order

- Wenn an der **oberen Speicherstelle** das **LS Byte** (Bits 7-0) gespeichert wird

  $\rightarrow$ | LS Byte | MS Byte |

  $\rightarrow$ **Little Endian**-Format (kleines Ende)

Beim Datenaustausch zwischen zwei verschiedenen Maschinen muss die Byteanordnung unbedingt berücksichtigt werden.

Adressierung

Wieviel Bits müssen für die Adressierung verwendet werden?

Da jede Adresse über eine eigene Nummer angesprochen werden kann, ist der gesamte Speicher für ein Programm zugänglich.

- Diese Nummer wird größer, je größer der Speicher wird.
- Auch der Befehl, der diese Speicherstelle anspricht, wird größer (da das Adressfeld mehr Bits bereitstellen muss).
- Deshalb erhöht sich der Speicherbedarf von Programmen.

Da die Daten oft miteinander in Beziehung stehen (Arrays), kann man effizientere Adressierungsmodi zur Adressierung verwenden.

Adressierungsmodi

Wir werden jeweils einen Move-Befehl betrachten, der ein Datum ins Register R1 transferiert.

- Direkte Adressierung: Auch **absolute** Adressierung genannt, da hier die absolute Speicheradresse (Nummer) angegeben wird.

  `MOVE R1,100`: Hole **Inhalt** der **absoluten** Speicheradresse 100 (nach R1).

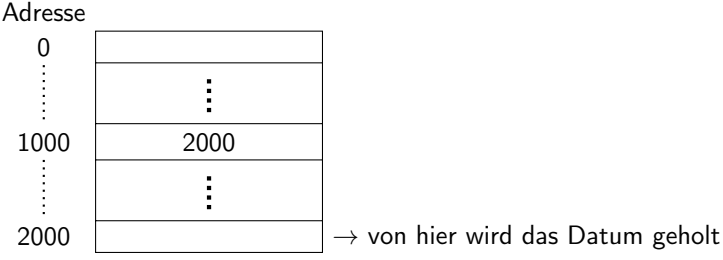
  (Die Größe des Datums hängt von der Spezifikation des Move-Befehls ab.)

  Nachteil dieses Modus: **Platzbedarf des Befehls**.
  $\rightarrow$ Viel günstiger ist es, wenn das Datum schon in einem Register steht:
  `MOVE R1,R2`: Hole Inhalt des Registers R2
  $\rightarrow$ R1 und R2 haben das selbe Datum.

Adressierungsmodi

- Indirekte Adressierung: Dieser Modus ist ähnlich wie das Pointer-Konzept in Hochsprachen:
  - `MOVE R1,(1000)`: Hole **Inhalt der Adresse, die in 1000 steht** (nach R1).

    `( )`: Indirekt. Interpretiert **Inhalt** von 1000 als Adresse.
    → Indirekte Adressierung auch Pointer-Konzept genannt



    → **Memory Deferred** Modus (da **der Pointer im Speicher** abgelegt ist)

Adressierungsmodi

- (Fortsetzung indirekte Adressierung)
  - `MOVE R1,(R2)`: Hole Inhalt der Adresse, die im R2 steht.

    R2 | 2000 | → R1 bekommt das Datum aus der Speicherstelle 2000

    → **Register Deferred** Modus (da der Pointer im **Register** abgelegt ist).

- Immediate Adressierung:

  Es gibt auch Daten, die weder im Speicher noch in einem Register stehen müssen, sondern direkt (d. h. immediate) im Programmcode angegeben sind:

  `MOVE R1,#100`: Lade R1 mit der Zahl 100

Adressierungsmodi

Erweiterungen der indirekten Adressierung über Register:

Displacement und Indexed

Displacement:
`MOVE R1,100(R2)`: Hole Inhalt der Adresse R2+100 nach R1.

z. B. R2 = 1000 $\rightarrow$ Hole Datum aus der Speicherstelle $1000 + 100 = 1100$.

100: Offset (der zu R2 addiert wird) (Offset muss auch codiert werden).

Indexed:
`MOVE R1,(R2+R3)`: Hole Inhalt der Adresse R2+R3 nach R1.

Adressierungsmodi

- **Benchmarks**:
  Für DLX-Maschine:

  Displacement und **Immediate** machen 75–99 % aller verwendeten Adressierungsmodi aus.

  Die Größe des **Displacements** wird auf 16 Bit gesetzt (dies reicht in 99 % der gemessenen Fälle aus).

  Die Größe der **Immediate**-Adressierung wird auch auf 16 Bit gesetzt (reicht für 80 % aller Fälle).

Operationen des Befehlssatzes

Operationen des Befehlssatzes

Einteilung in Gruppen:

- **Datentransfer** (um Daten aus und in den Speicher zu bewegen) – bereits präsentiert.
- **Arithmetik und Logik**
  - Integeroperationen (Addition (`ADD`), Multiplikation)
  - Logische Verknüpfungen der Operanden
  - Floating-Point-Operationen (ausgeführt in einem speziellen Rechenwerk der CPU)

    Operanden für Addition, Subtraktion, Multiplikation und Division sind (auf den meisten heutigen Maschinen) im IEEE-Format codiert.

    Manchmal gibt es Hardwareimplementierungen für das Berechnen von Quadratwurzeln oder trigonometrischen Funktionen.

Operationen des Befehlssatzes

- **Control-Operationen** (Operationen für die Steuerung des Programm-Ablaufs):
  - Jump (unbedingter Sprung)
  - Branch (bedingter Sprung)
  - Call (Prozeduraufruf)
  - Return (Rücksprung aus Prozedur)

  Die Auswirkungen dieser Befehle:

  Das Programm wird nicht mit dem nächsten, sondern mit einem an anderer Stelle angegebenem Befehl, fortgeführt. Diese Befehle verändern den **PC** (Program Counter – Register in der CU, welches die Adresse des nächsten auszuführenden Befehls beinhaltet).

Operationen des Befehlssatzes

Branches: Sprünge nicht groß (überspringen nur einige Instruktionen (Lokalität)).
$\rightarrow$ Zur Adressierung der Sprünge wird meist die Distanz relativ zum PC angegeben (in den meisten Fällen genügen 8 Bits dafür, d. h. die Sprünge liegen im Bereich PC $\pm$ 127).

Jumps: Auch hier sind die Sprungadressen vor der Programmausführung bekannt.

Dagegen sind die Sprungadressen von **Call** und **Return** nicht vor dem Programmstart bekannt $\rightarrow$ diese müssen **dynamisch** bestimmt werden.

Operationen des Befehlssatzes

Beispiel

`A`, `B`, `C`: Prozeduren eines Programms.
`C` kann sowohl **von** `A`, sowie auch **von** `B` **aufgerufen werden**. **Return: Steht am Prozedurende von** `C`.
Ein Compiler kann nicht entscheiden, an welche Adresse gesprungen werden soll, da dies davon abhängt, ob `A` oder `B` die Prozedur `C` aufgerufen hat.

$\rightarrow$ Die Rücksprungadresse kann nur zur Laufzeit des Programms gebildet werden.

Operationen des Befehlssatzes

Die Rücksprungadresse ist jene Adresse, an der die Befehlsabarbeitung nach Prozedurende fortgesetzt wird.

Branches: Sprünge, die von einer Bedingung abhängig sind.

Die Bedingung wird im Programm definiert und muss zur Laufzeit interpretiert werden.

Branch Condition

```
      BEQZ R1,LOOP
                ⋮
LOOP: ADD  R1,R2,R3
```

(Springe zum Label `LOOP`, wenn R1 gleich 0)
„**B**ranch **eq**ual **Z**ero“

Operationen des Befehlssatzes

Einfaches Problem

Unterprogramm schreibt auch in R1

→ Irgendjemand (Programmierer) muss dafür sorgen, dass der Wert nicht überschrieben wird

HP (Hauptprogramm)

```
R1
CALL UP
```

```
UP SIN
R1
   R2,R3
```

Operationen des Befehlssatzes

Caller Save (HP verantwortlich)
Aufrufer (HP) ist verantwortlich, dass `R1` temporär gespeichert wird.

Callee Save (meistens)
UP verantwortlich
$\rightarrow$ Es muss zuerst `R1` im Speicher ablegen und vor dem Zurückspringen wiederherstellen.
UP weiß, welche Register verwendet werden.
`R1`, `R2`, `R3` im Speicher abgelegt und vor return wiederhergestellt.

Meistens macht das nicht die Hardware, man muss sich als Programmierer darum kümmern.

Befehlshäufigkeit

Aufgrund von Messungen von Benchmark-Programmen gibt es folgende Häufigkeit der Befehle (für Intel 80x86-Prozessoren):

Befehl	Häufigkeit	
LOAD	0.22	
BRANCH	0.20	Warum ist COMPARE ungefähr gleich oft wie BRANCH?
COMPARE	0.16	⟶ (COMPARE wird sehr oft mit BRANCH verwendet: Zuerst
STORE	0.12	vergleichen, dann branchen.)
ADD	0.08	
AND	0.06	
SUB	0.05	
MOVE	0.04	⟶ Innerhalb der Register (nicht Speicher/Register)
CALL	0.01	⟶
RETURN	0.01	⟶ gleich häufig
Gesamt	0.95	

Befehlshäufigkeit

Multiplikation fehlt.
(Da sie sehr lange dauert: Durch Addition (in Software) ersetzt.)

95 % des Programms wird von 10 Befehlen abgedeckt.
Leitsatz für die Architektur des Befehlssatzes:
Make the common case fast.
(Die Befehle, die am häufigsten auftreten, sollen am schnellsten ausgeführt werden → Hardware-Realisierung).

Also relativ kleiner Befehlssatz genügt für DLX: wenige elementare Befehle zur Verfügung.

Operandentypen

- Welches ist die geeignete Größe von Operanden (Zahlen)?
  Hängt von der Rechnerarchitektur (Datenbusbreite) ab:
  Daten, deren Größe gleich der Breite des **Datenbuses** ist, können in einem Taktzyklus übertragen werden.

  Wenn 32-Bit-Wort transferiert wird:
  Auf einem Rechner mit
  32 Bit Busbreite: in einem Zyklus
  16 Bit Busbreite: 2 Takte.

Operandentypen

- Typ des Operanden
  Wird durch die Instruktion festgelegt:
  In **Opcode** (Teil eines Befehls, der den Befehl selbst codiert)

  `101010` $\rightarrow$ `ADD`

  Jeder Befehl 32 Bits lang, 6 Bit davon sind Opcode.
  Codiert: `ADD`-Befehl und Operandentyp (z. B. dass es sich um Floating-Points handelt).

  Aus Opcode $\rightarrow$ welchen Typ die jeweiligen Operanden haben (müssen).

Operandentypen

Integers: Heute meist als Wort und Doppelwort repräsentiert.

Für negative Zahlen: Es wird ausschließlich das 2-Komplement benutzt.

Für Floating-Point-Repräsentation: IEEE-32-Bit, IEEE-64-Bit.

Also für DLX folgt: Einfache 32-Bit-Architektur, mit Unterstützung beider IEEE-Formate.

Befehlssatzcodierung

Instruktion ist **Code fester Länge**.

Ein Codierter Befehl hat **drei wesentliche** Teile:

- **Befehlsfeld** (**Opcode**) (gibt den Befehl an (`ADD`))
- **Adressfeld** (**Address Field**) (enthält eine Adresse, die in Abhängigkeit eines eventuellen Address Specifier interpretiert wird).

  Der Address Specifier kann entfallen, wenn die Spezifizierung der Adresse im Opcode vorhanden ist.

  z. B. `LW` (für Load Word)
  (Hier ist die Information vorhanden, dass ein 32-Bit-Wort adressiert wird.)

Befehlssatzcodierung

- Code variabler Länge:
  Für jeden Befehl muss die entsprechende Anzahl von Bytes aus dem Speicher geholt werden.
  Dies muss aus dem Opcode bestimmt werden
  $\rightarrow$ höhere Ansprüche an die CU der CPU.
- Code fester Länge:
  Jedes Befehlswort hat die gleiche Länge
  $\rightarrow$ Programmsteuerung einfacher aber nicht optimal (was die Programmgröße anbelangt).
  D. h. Programmcode wird vergrößert weil es immer Befehle geben wird, die nicht die vorgesehene Länge brauchen.

DLX: Es werden Befehlscodes fester Länge verwendet und dabei werden möglichst alle Bits des Codeworts ausgenutzt.

Befehlssatz und Compiler

Heutige Programmierung wird meist in einer Hochsprache durchgeführt.
$\rightarrow$ Der Befehlssatz muss auch auf die für die Maschine vorhandenen Compiler abgestimmt werden.

Die modernen Compiler versuchen die Programmcodes zu optimieren. Eine der wichtigsten Optimierungen betrifft die Verwendung von Registern für Variablen einer Hochsprache.

Welche Variablen kommen in Register?
Ein komplexes Problem (Graph Coloring). Je mehr Variablen in Register, desto schneller wird die Ausführung des Codes.

Aber: Die Registeranzahl ist beschränkt. (Graph Coloring robust, wenn mindestens 16 Register vorhanden sind.)

Befehlssatz und Compiler

Make the frequent cases fast and the rare cases correct.

Die häufig auftretenden Codesequenzen sollten auf Geschwindigkeit optimiert werden, während die seltenen nur korrekt sein sollten.

$\rightarrow$ Einige prinzipielle Anforderungen an die Architektur des Befehlssatzes:

Befehlssatz und Compiler

1. Orthogonalität des Befehlscodes
   (Operationen, Datentypen und Adressierungsmodi sollten beliebig kombinierbar sein.)
2. Einfache Grundoperationen
   (Manche Architekturen haben sehr spezielle Befehle, die aber selten von Compilern verwendet werden.)
3. Einfache Abhängigkeiten
   (Es genügt heute nicht, nur die Codegröße zu optimieren: Pipelines, Branch Prediction, etc. beeinflussen die Codeeffizienz.)
   Z. B. bei Pipelines: Bei exakt der selben Taktrate ergeben sich Verbesserungen bis zu Faktor 5.
4. Statische Instruktionen
   (Zur Compilezeit soll man möglichst viel wissen.)
   Nicht statisch: Return

Befehlssatz und Compiler

1.–4. $\rightarrow$ Es ergibt sich die Forderung nach einem einfachen, klar strukturierten Befehlssatz (Entwicklung der letzten 20 Jahre).

(Vorher versuchte man mit sehr komplexe Befehlssätze (z. B. VAX (DEC)) die Maschinensprache an Hochsprachen anzunähern.)

Heute: RISC-Architektur (Reduced Instruction Set Computer)
Geschwindigkeitsvorteile (Die Codegröße ist höher, aber dieser Nachteil wird durch größere Speicher eliminiert.)

$\rightarrow$ DLX-Maschine hat RISC-Architektur.

Performance eines Rechnersystems

Performance eines Rechnersystems

Quantifizierung von Verbesserungen:

Ausführungszeit eines Programms: t_P
Ausführungszeit auf der ursprünglichen Maschine: $t_{P,old}$
Ausführungszeit auf der verbesserten Maschine: $t_{P,new}$
Verbesserungen (z. B. schnellerer Speicher, schnellere Befehle, höhere Taktfrequenz, etc.):

Performance eines Rechnersystems

- **Speedup** (Beschleunigung):

$$s = \frac{t_{P,old}}{t_{P,new}}, \quad s > 1$$

$s = 1$ (keine Verbesserung)
$s < 1$ (Slowdown)

Wenn f_e: Anteil der verbesserten Befehle (Anteil der Befehle, auf die sich die Verbesserung auswirkt (im Bezug auf Gesamtprogramm)
s_e: Speedup der speziellen Verbesserung

$$\rightarrow t_{P,new} = (1 - f_e)t_{P,old} + \frac{f_e}{s_e}t_{P,old}$$

Performance eines Rechnersystems

- **Amdahl's Law**:

$$s = \frac{1}{\frac{1-f_e}{1} + \frac{f_e}{s_e}} = \frac{t_{P,old}}{t_{P,new}}$$

Der Anteil der nicht verbesserten wird auf Speedup 1 bezogen

Anteil der verbesserten Befehle auf s_e bezogen

Performance eines Rechnersystems

Beispiel

Eine Verbesserung einer Maschine betrifft 40 % aller Befehle. Diese Verbesserung erhöht die Geschwindigkeit dieser Befehle um Faktor 10. Wie groß ist der Speedup der Maschine?

$f_e = 0.4$
$s_e = 10$

$\rightarrow s = \frac{1}{(1-0.4)+\frac{0.4}{10}} \approx 1.56$

Performance eines Rechnersystems

- Clocks Per Instruction (CPI)

$$CPI = \frac{n_C}{n_I}$$

n_C: Anzahl der **Taktzyklen**, die Programm zur Ausführung braucht (abhängig von Befehlssatzarchitektur und Compilertechnologie)

n_I: Anzahl der **Instruktionen** des Programms

CPI: Mittlere Anzahl der Taktzyklen

$\rightarrow t_P = CPI \cdot n_I \cdot t_C = n_C \cdot t_C$

t_P: Ausführungszeit des Programms P
t_C: Periodendauer eines Taktzyklus (abhängig von der Technologie der Hardware)

Performance eines Rechnersystems

Wenn CPI_i die CPI-Rate einer Teilmenge des Befehlssatzes bezeichnet und f_i der Anteil dieser Teilmenge an der Anzahl der Gesamtinstruktionen ist (es gilt $\sum_{i=1}^{n} f_i = 1$, wobei n die Anzahl der Teilmengen ist) gilt:

$$CPI = \frac{1}{n} \sum_{i=1}^{n} CPI_i f_i$$

$$CPI_i = \frac{n_C}{n_{I\,i}} = \frac{n_C}{f_i n_I}$$

$$\frac{1}{n} \sum_{i=1}^{n} CPI_i f_i = \frac{1}{n} \sum_{i=1}^{n} \frac{n_C}{f_i n_I} f_i = \frac{1}{n} \sum_{i=1}^{n} \frac{n_C}{n_I} = \frac{1}{n} \cdot n \cdot CPI = CPI$$

Performance eines Rechnersystems

- Eine andere Maßzahl: **MIPS** (Million Instructions Per Second)

$$MIPS = \frac{n_I}{t_P \cdot 10^6} = \frac{1}{CPI \cdot t_C \cdot 10^6} = \frac{\cancel{n_I}}{\underbrace{CPI \cdot \cancel{n_I} \cdot t_C}_{t_P} \cdot 10^6}$$

(Ungenügende Maßzahl, weil diese sogar bei derselben Rechnerarchitektur und demselben Programm unterschiedliche (compilerabhängige) Ergebnisse bringen kann.)

- Ähnliche (ungenügende) Maßzahl: **MFLOPS** (Million Floating Point Operations Per Second)

$$MFLOPS = \frac{n_{FP}}{t_P \cdot 10^6}$$

n_{FP}: Anzahl der Floating-Point-Operationen im Programmcode.

Die DLX-Maschine

Pipelining

Die DLX-Pipeline

Pipeline Hazards