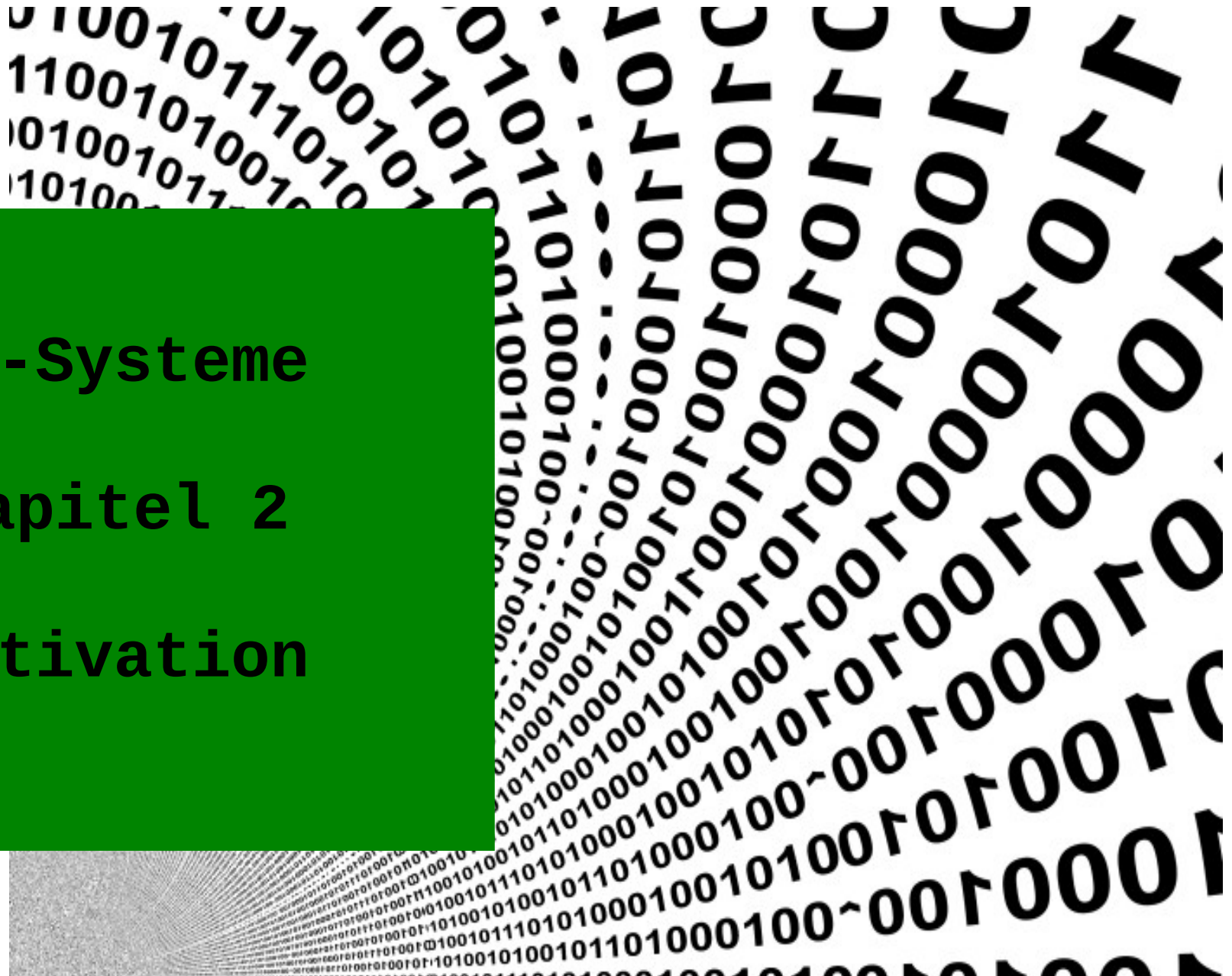


DV-Systeme

Kapitel 2

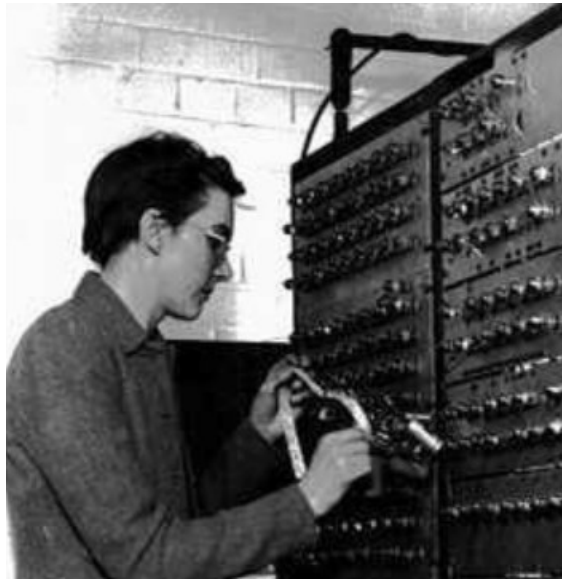
Motivation



Inhaltsverzeichnis

Thema	Seite
Was ist Assembler?	3
Aufbau von Assembler-Code am Beispiel eines simplen Befehlssatzes	10
Der Maschinenbefehlssatz	12
Wofür braucht man Assembler	16
Optimierung der Ausgabe des Compilers	18
Weg vom C-Code zum Maschinencode	19
Einschub: Hardware-Pipelining	26

Was ist Assembler?

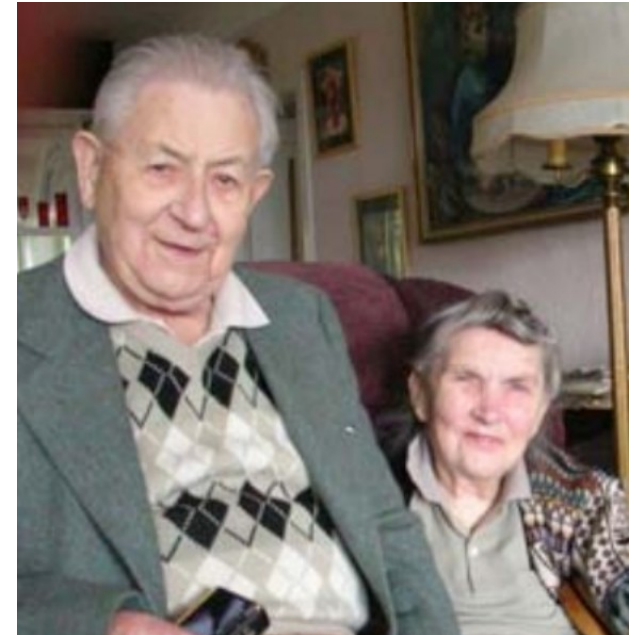


Kathleen Hylda Valerie Booth

geb. 9.7.1922 in England, gest. 29.9.2022

Sie war eine der ersten Computerprogrammiererinnen, die 1947 eine Assemblersprache entwickelte, die der Computerprogrammierung half, von der Eingabe von Nullen und Einsen zur Interpretation durch die Maschine wegzukommen. Booth half beim Design und war die Hauptprogrammiererin von drei frühen Computern: ARC, SEC und APE(X)C, die von Andrew Booth (ihrem Ehemann) am Birkbeck College konstruiert wurden.

$M \times R \rightarrow cA$. Clear accumulator, multiply M by R and place L.H. 39 digits of answer in A and R.H. 39 digits in R.
 $A \div M \rightarrow cR$. Clear register, divide A by M, leave quotient in R and remainder in A.
 $C \rightarrow M_1$.
 $C \rightarrow M_r$.
 $Cc \rightarrow M_1$.
 $Cc \rightarrow M_r$.
 $A \rightarrow M$.
 If number in A ≥ 0 shift control to M_1 .



2008 mit Ehemann Andrew, der mit **von Neumann** in Princeton die ersten Computer baute.

Was ist Assembler?

Das Wort Assembler hat zwei Bedeutungen:

1. Eine maschinenorientierte Programmiersprache. D.h. die Sprache orientiert sich an der Maschine (dem Prozessor), auf der sie läuft. Deshalb gibt es einige bekannte und eine ganze Reihe weniger bekannter Assemblersprachen.

- bekannte:

- für x86-Prozessoren unter DOS / Windows (Intel-Syntax):

TASM - Borland Turbo Assembler (16-/32-Bit-Output)

MASM - Microsoft Macro Assembler (64-Bit-Output)

FASM - Flat Assembler (64-Bit-Output) (auch unter Linux)

- für x86-Prozessoren unter Linux (AT&T-Syntax)

GAS - GNU-Assembler (64-Bit-Output) (**GNU** = **GNU's not Unix**)

- für verschiedene Plattformen

NASM - Netwide Assembler (Variante der Intel-Syntax)

- unbekannte:

Sind in erster Linie solche, die auf Embedded Systems für spezielle Zwecke laufen, und dafür oft einen sonst unbekannten Ass-Code nutzen.

Beispiele: Unterschiedliche Assembler-Dialekte für Hello World.

PowerPC-CPU

```
.section .rodata
    .align 2
.s:
    .string "Hello World!\n"

.section ".text"
    .align 2
    .globl _start
_start:
    li 0,4          # SYS_write
    li 3,1          # fd = 1 (stdout)
    lis 4,.s@ha     # buf = .s
    la 4,.s@l(4)
    li 5,13         # len = 13
    sc              # syscall

    li 0,1          # SYS_exit
    li 3,0          # returncode = 0
    sc              # syscall
```

Kompilierbefehl:

gcc -nostdlib -s hello.s

SPARC-CPU

```
.section .data

hello_str:
    .asciz "Hello World!\n"
    .align 4
hello_str_len:
    .word . - hello_str

.section .text

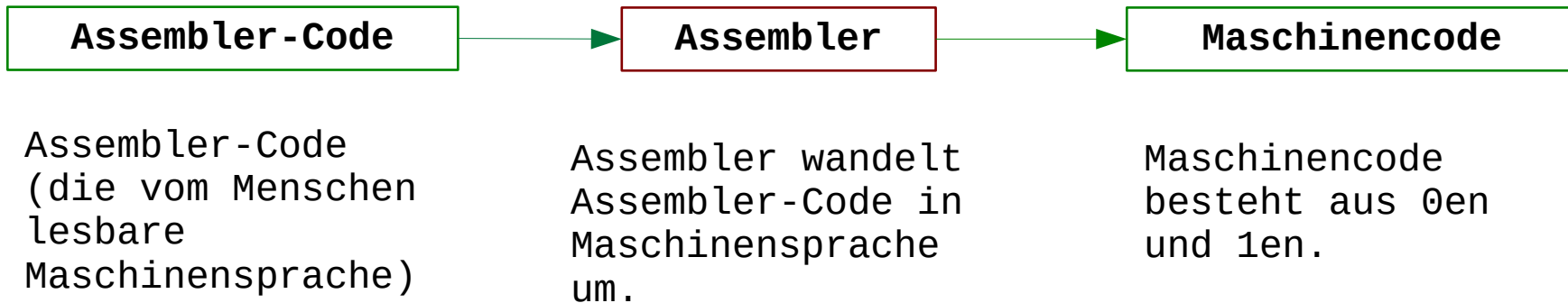
.global _start
_start:
    set 4, %g1
    set 1, %o0
    set hello_str, %o1
    set hello_str_len, %l0
    ld [%l0], %o2
    t 0x110
    set 1, %g1
    set 0, %o0
    t 0x110
```

Kompilierbefehl: **as -o hello.o hello.s;
ld -o hello hello.o**

Was ist Assembler?

In diesem Sinn ist von Assemblersprachen die Rede, die vom Menschen lesbar bzw. programmierbar sind.

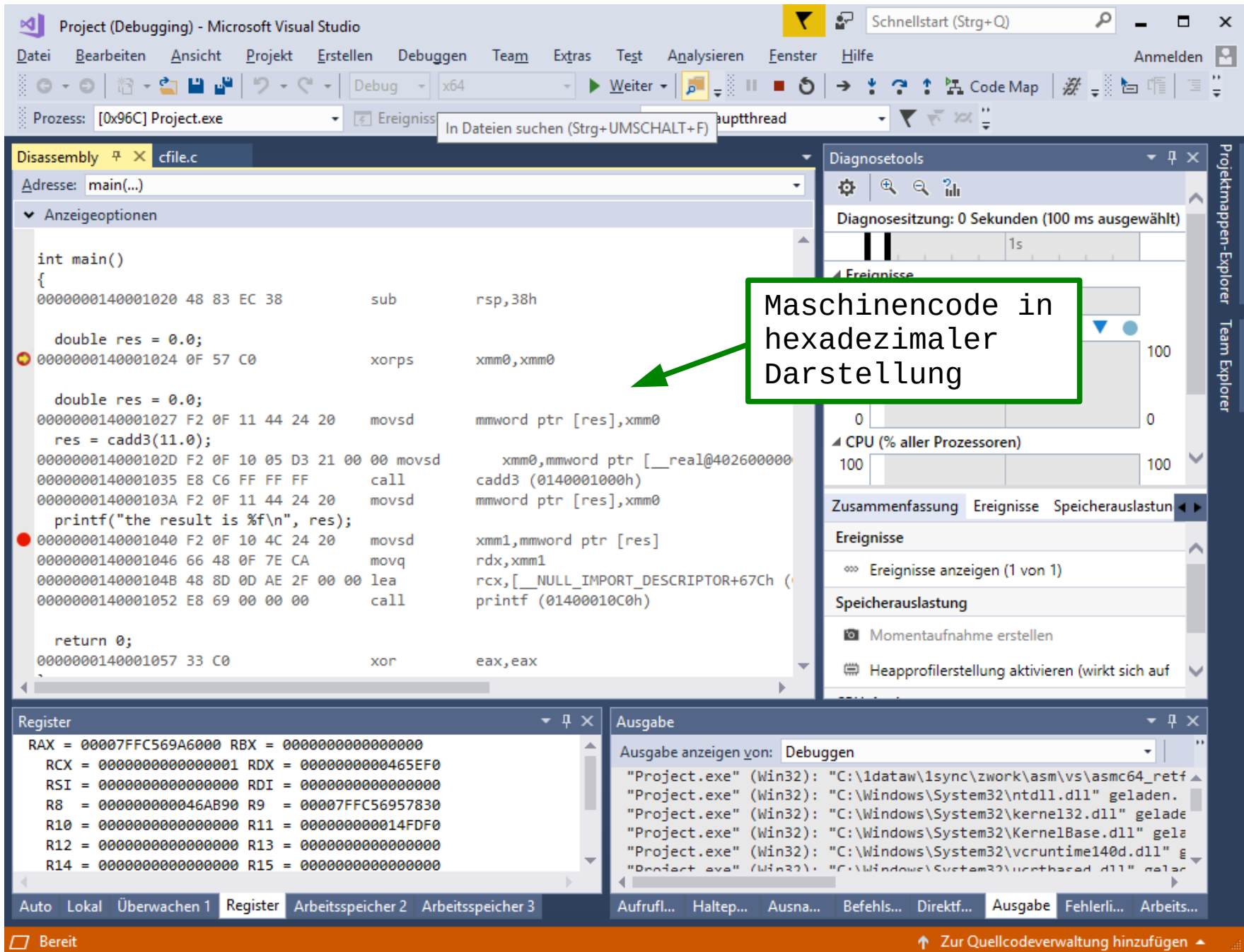
2. Das Programm, das den Assemblercode in den eigentlichen Maschinencode überführt, der nur noch aus 0en und 1en besteht – meist in hexadezimaler Form dargestellt.



Maschinencode entsteht z.B. wenn das HelloWorld-Programm mit

bzw. `gcc helloWorld.c -o helloWorld` - für C-Programme
`g++ helloWorld.cpp -o helloWorld` - für C++-Programme

in eine ausführbare Datei umgewandelt wird.



Was ist Assembler?

Die einzelnen Schritte bis zum ausführbaren Programm:

Präprozessor ausführen:

```
gcc -E hello.c -o hello.i
```

Assembler-Code erzeugen:

```
gcc -S hello.i -o hello.s
```

Objektcode erzeugen:

```
gcc -c hello.s -o hello.o
```

Ausführbare Datei erzeugen:

```
gcc hello.o -o hello
```

alle Schritte auf einmal:

```
gcc -save-temps hello.c -o hello
```

C-Programm

```
#include <stdio.h>
int main(){
    printf("Hello World!\n");
    return 0;
}
```

C-Code: **hello.c**

Ergebnis:

Dateien zusätzlich zu **hello.c**:
hello.i, hello.s, hello.o, hello

Alternative:

alle Schritte auf einmal, ohne
Zwischendateien zu erzeugen:

```
gcc hello.c -o hello
```

Ergebnis:

Dateien
hello.c, hello

Was ist Assembler?**C++-Programm**

Die einzelnen Schritte bis zum ausführbaren Programm:

Präprozessor ausführen:

```
g++ -E hello.cpp -o hello.i
```

Assembler-Code erzeugen:

```
g++ -S hello.i -o hello.s
```

Objektcode erzeugen:

```
g++ -c hello.s -o hello.o
```

Ausführbare Datei erzeugen:

```
g++ hello.o -o hello
```

alle Schritte auf einmal:

```
g++ -save-temps hello.cpp -o hello
```

```
#include <iostream>
int main(){
    std::cout << "Hello World!\n";
    return 0;
}
```

C++-Code: **hello.cpp**

Ergebnis:

Dateien zusätzlich zu **hello.cpp**:
hello.i, hello.s, hello.o, hello

Alternative:

alle Schritte auf einmal, ohne Zwischendateien zu erzeugen:

```
g++ hello.cpp -o hello
```

Ergebnis:

Dateien
hello.cpp, hello

Aufbau von Assembler-Code am Beispiel eines simplen Befehlssatzes

Das Aussehen des Assembler-Codes wird durch die **Instruction Set Architecture (ISA)** bestimmt. ISA enthält eine Liste von Assemblerbefehlen mit dazu passendem Maschinencode.

Beispiel eines sehr einfachen 16-Bit-Codes (von Atmel AVR):

Legende:

R = Register, **r** = Quelle, **d** = Ziel
MOV = kopieren (engl. move / bewegen)

ISA-Syntax:

MOV Rd, Rr

Maschinencode:

0010 11rd dddd rrrr

konkrete Umsetzung:

MOV R16, R1

0010 1101 0000 0001

Aufbau von Assembler-Code am Beispiel eines simplen Befehlssatzes

Aufgabe:

1. Welchen Teil des Binär-Codes stellt der Befehl **MOV** dar?
 2. Stellen Sie den Befehl **MOV R10, R11** entsprechend der Angaben binär dar. Verwenden Sie dabei für den Kopierbefehl, Quelle und Ziel jeweils unterschiedliche Farben.
-

Legende:

R = Register, r = Quelle, d = Ziel
MOV = kopieren (engl. move / bewegen)

ISA-Syntax:

MOV Rd, Rr

zgh. Maschinencode :

0010 11rd dddd rrrr

Der Maschinenbefehlssatz (Instruction Set)

Der Maschinenbefehlssatz (**Instruction Set**) ist ein direktes Abbild aller Operationen, die der Prozessor durchführen kann.

Mit Maschine ist hier der Prozessor gemeint.

Die Maschinenbefehle teilen sich in folgende Gruppen auf:

- Transportbefehle
- Arithmetische Befehle
- Bitweise logische Befehle
- Schiebe- und Rotations-Befehle
- Einzelbit-Befehle
- Sprung-Befehle
- Prozessorsteuerungs-Befehle

Der Maschinenbefehlssatz (Instruction Set)

Transportbefehle

Befehle mit denen Daten zwischen Komponenten des Rechnersystems transportiert werden. Z.B.:

- Register <--> Speicher
- Register <--> Register
- Ein-/Ausgabebaustein <--> Register
- Ein-/Ausgabebaustein <--> Speicher

Sehr oft wird dabei eine Kopie des Quell-Operanden angelegt.
Spezielle Transportbefehle sind die Stackbefehle PUSH und POP.

Arithmetische Befehle

Hierbei werden die zu verarbeitenden Bitmuster als Zahlen interpretiert.
Manche Befehle unterscheiden zwischen vorzeichenlosen und vorzeichenbehafteten Zweierkomplement-Zahlen. Typische Befehle:

- Addition, Subtraktion, Multiplikation, Division, Dekrement, Inkrement und diverse Vergleichsbefehle

Der Maschinenbefehlssatz (Instruction Set)

Bitweise logische Befehle

Hier werden Operanden bitweise durch die logischen Operatoren UND, ODER, exklusives ODER verknüpft; dazu kommt die bitweise Invertierung eines Operanden.

Schiebe- und Rotationsbefehle

Veränderung von Bitmustern durch Schiebe- oder Rotationsbefehle, wie z.B. Schieben nach links, Schieben nach rechts, Rotieren nach links und Rotieren nach rechts. Häufig wird dabei das Carry-Flag einbezogen.

Einzelbitbefehle

Diese Befehle umfassen das Setzen, Verändern oder Abfragen einzelner Bits in Dateneinheiten.

Sprungbefehle

Sie verändern den Inhalt des Programmzählers (PC) und veranlassen dadurch die Fortsetzung des Programms an einer anderen Stelle. Unbedingte Sprungbefehle werden immer ausgeführt, bedingte Sprungbefehle abhängig vom Inhalt des Maschinenstatusregisters (Flags). Spezielle Sprungbefehle sind der Unterprogramm-Aufruf und der Rücksprung aus einem Unterprogramm.

Der Maschinenbefehlssatz (Instruction Set)

Prozessorsteuerungsbefehle

Jeder Prozessor verfügt über Spezialbefehle, mit denen man die Betriebsart einstellen kann. Dazu zählt z.B. das Freischalten von Interrupts, die Umschaltung auf Einzelschrittbetrieb, die Konfiguration der Speicherverwaltung etc. Oft werden dazu Flags in Steuerregistern gesetzt.

Dies sind nur die wichtigsten Befehlsgruppen, moderne Prozessoren besitzen weitere Befehle, z.B. für die Gleitkommaeinheit. Andererseits sind in einfachen Prozessoren nicht alle aufgeführten Befehlsgruppen vorhanden, z.B. gibt es einfache Prozessoren ohne Einzelbitbefehle.

Will man eine Familie von Prozessoren mit ähnlichem Befehlssatz zusammenfassen, so spricht man von einer Befehlssatzarchitektur (englisch: **Instruction Set Architecture**, kurz: **ISA**). Verbreitete Befehlssatzarchitekturen sind beispielsweise:

- **IA-32** 32-Bit-ISA der ursprünglichen 16-bittigen x86-Architektur; (auch die 64-Bit-Erweiterung **x64** ist IA-32 zuzuordnen)
- **IA-64** die Itanium-Architektur; nicht zu verwechseln mit x64
- **ARM** basierend auf RISC, z.B. für Handys (RISC = Reduced Instruction Set Computer)

Wofür braucht man Assembler?

- Assembler-Code ansehen, wenn sich ein Programm merkwürdig verhält.
- Code analysieren, den andere geschrieben und schon kompiliert haben (z.B. exe-Dateien) – das nennt man Reverse Engineering.
- Code für spezielle Hardware schreiben (die ersten Gameboys wurden in Assembler geschrieben)
- Treiber für spezielle Maschinen schreiben.
- Malware analysieren – feststellen, ob es sich um bösartigen Code handelt.
- Für Entwickler von Embedded Systems, die mit Prozessoren oder Entwicklungs-Tools arbeiten, die man braucht, aber nicht ausreichend getestet wurden.
- Wenn man zufällig auf einen Compilerfehler trifft oder die Compiler-Dokumentation nicht ausreichend ist.
- Einfach aus Interesse, um zu sehen, was im Inneren des Programms abläuft.
- Wenn man Code schneller laufen lassen möchte (insbesondere bei zeitfressenden Schleifen), könnte man direkt in den Assemblercode eingreifen und ihn verbessern.
- Besseres Verständnis für Abläufe in der CPU entwickeln.

Wofür braucht man Assembler?

Zusammenfassend

- Um zu verstehen, wie Computer arbeiten.
- Schärft die Wahrnehmung bzgl. gutem oder schlechtem Code.
- Man versteht, weshalb manche Dinge schneller oder langsamer laufen als andere.
- Als Software-Entwickler trifft man bessere Entscheidungen bei der Programmierung in Hochsprachen.

Die ausgewählte Assemblersprache hängt immer von dem Prozessor ab, den man nutzt – jeder Prozessor hat seinen eigenen Befehlssatz.

Im Allgemeinen ist es wichtig, zu verstehen was der Assembler aus dem Code der Hochsprache macht.

Beim Kompilieren sollte man zur Untersuchung des Assemblercodes Optimierungen ausschalten, weil der ursprüngliche Assembler-Code zu stark verändert würde.

Optimierung der Ausgabe des Compilers

Neben dem Aktivieren einzelner spezifischer Optimierungen bieten moderne Compiler die Möglichkeit, Optimierungen in Gruppen zu aktivieren.

Die wichtigsten im Fall von **gcc**: **-O0**, **-O1**, **-O2**, **-O3**, **-Os** und **-Og**

Gruppe -O0:

Deaktivieren aller Optimierungen (Default)

Gruppen -O1 bis -O3:

Optimieren Geschwindigkeit in verschiedener Stärke, wobei -O3 die stärkste Optimierung darstellt. Für auf Geschwindigkeit optimierte Builds sollte im Allgemeinen -O2 verwendet werden, da -O3 unter Umständen zu erheblich größerem Kompilat führen kann und Code dann in manchen Formen zu undefiniertem Verhalten führen kann.

Gruppe -Os:

Optimierung wenig Speicherverbrauchs des Kompilats. Entspricht weitgehend der Gruppe -O2, lässt jedoch Optimierungen, die den Code vergrößern würden, aus und fügt einige die ihn verlangsamen, aber verkleinern, hinzu.

Gruppe -Og:

Erzeugt besonders debuggerfreundliche Binaries. Dafür werden die Optimierungen der Gruppe -O1 angewandt (-O0 allein ist kaum lesbar), und es wird auf jene verzichtet, die das Debuggen erschweren würden.

<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-O3-719>

Weg vom C-Code zum MaschinencodeBeispiel eines einfachen Summenprogramms:

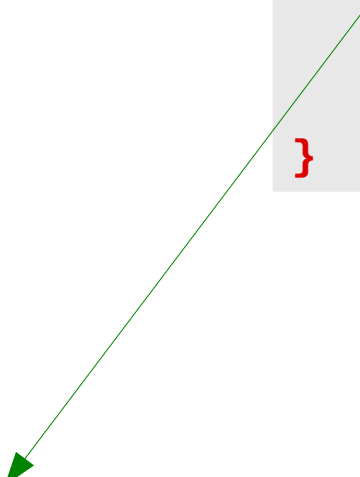
- Definition von drei Variablen
- Zuweisung Wert 2 an Variable x
- Zuweisung Wert 5 an Variable y
- Addition von x und y und Zuweisung des Ergebnisses an die Variable z

```

int main()
{
    int x = 2;
    int y = 5;
    int z = x + y;
    return 0;
}

```

sum.c


Programm wird kompiliert:

- **gcc sum.c -o sum**
Es entsteht der Maschinencode.

Resultat bei einer 16-Bit-CPU

```

000000001100001000000000100001101000000001100010100000001000011100000000
001000110100000000110001110000000010000111100000001110000000

```

Nur von Maschinen, aber nicht von Menschen lesbarer Code.

Weg vom C-Code zum Maschinencode

```

int main()
{
    int x = 2;
    int y = 5;
    int z = x + y;

    return 0;
}

```

sum.c

Resultat bei einer 16-Bit-CPU

```

000000001100001000000000100001101000000001100010100000001000011100000000
00100011010000000011000111000000001000011110000001110000000

```

Insgesamt 128 Bits, also 8 Befehle, die aus jeweils 16 Bits bestehen:

```

0000000011000010 | 00000000100001101 | 0000000011000101 | 00000000100001110
0000000010001101 | 00000000110001110 | 00000000100001111 | 000000001110000000

```


Weg vom C-Code zum Maschinencode

Der entsprechende Assemblercode könnte folgendermaßen aussehen:

```
int main()
{
    int x = 2;
    int y = 5;
    int z = x + y;

    return 0;
}
```

sum.c

	Reserve	Befehl	Nummer	Operand		
1)	000000	001	1	000010	LOAD	#2
2)	000000	010	0	001101	STORE	13
3)	000000	001	1	000101	LOAD	#5
4)	000000	010	0	001110	STORE	14
5)	000000	001	0	001101	LOAD	13
6)	000000	011	0	001110	ADD	14
7)	000000	010	0	001111	STORE	15
8)	000000	111	0	000000	HALT	

- 1) Lade Wert 2 in den Akkumulator
- 2) Speichere Inhalt vom Akkumulator an Speicherzelle 13
- 3) Lade Wert 5 in den Akkumulator
- 4) Speichere Inhalt vom Akkumulator an Speicherzelle 14
- 5) Lade Inhalt von Speicherzelle 13 in den Akkumulator
- 6) Addiere den Inhalt von Zelle 14 zum Inhalt des Akkumulators
- 7) Speichere den Inhalt vom Akkumulator in Speicherzelle 15
- 8) Beende das Programm

Weg vom C-Code zum Maschinencode

```
int main()
{
    int x = 2;
    int y = 5;
    int z = x + y;

    return 0;
}
```

sum.c

Wie sind die einzelnen Befehle codiert?

000	-->	NOOP	no operation (PC wird um 1 erhöht, sonst passiert nichts)
001	-->	LOAD	laden eines Wertes oder einer Adresse
010	-->	STORE	speichern
011	-->	ADD	addieren
100	-->	SUB	subtrahieren
101	-->	EQUAL	auf Gleichheit prüfen
110	-->	JUMP	Sprung
111	-->	HALT	Programm beenden

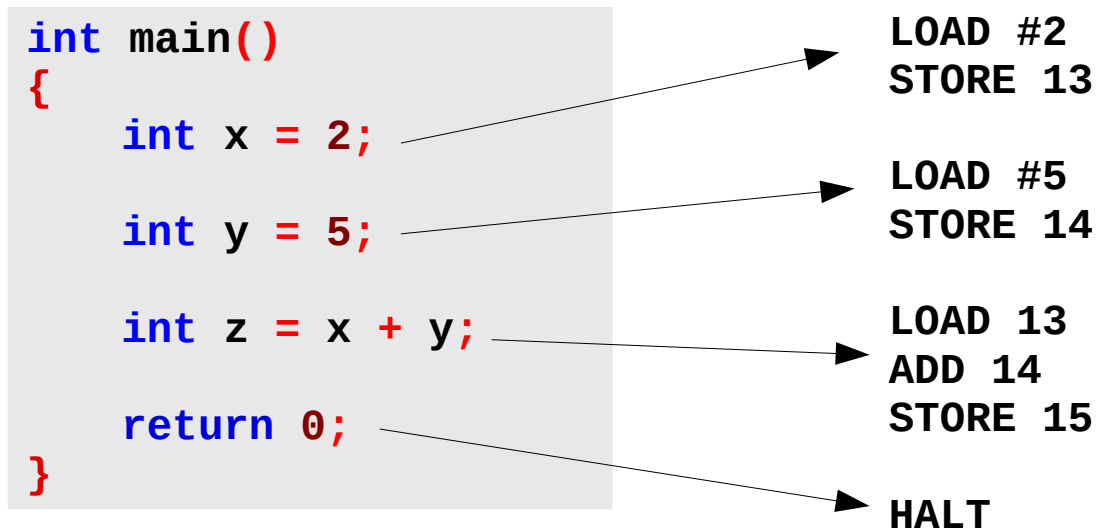
Die Befehle werden üblicherweise auf dem Akkumulator ausgeführt.

Transport-Befehle (z.B. LOAD, STORE)
Arithmetisch-logische Befehle (z.B. ADD, SUB)
Programmsteuerbefehle (z.B. JUMP, EQUAL)
Systemsteuerbefehle (z.B. HALT)

Weg vom C-Code zum MaschinencodeWas bedeuten die Nummern?

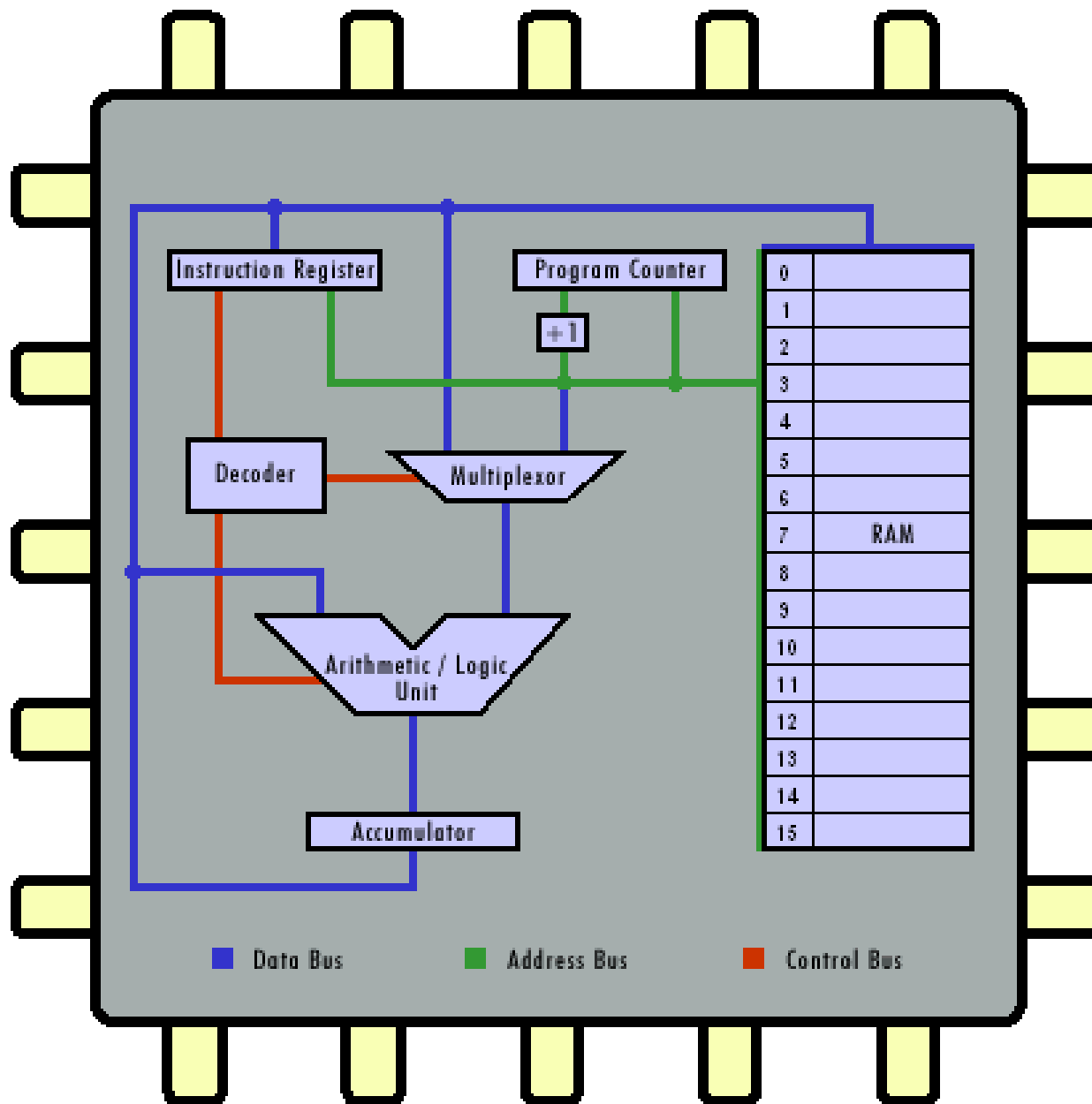
- 0 --> Operand ist Speicheradresse
1 --> Operand ist Zahl (beginnt mit Doppelkreuz #)

Die "Reserve" kann noch für andere Zwecke genutzt werden.
Die Operanden enthalten die binäre Darstellung der Zahlen oder Adressen.

Zusammenfassung:

Frage:

Wieviele Adressen
könnte man mit diesem
Befehlssatz ansteuern?

Weg vom C-Code zum Maschinencode

Hier finden Sie die fünf Grundkomponenten einer einfachen CPU mit 16 Byte RAM.

RAM, Register, Busse, ALU und Steuereinheit

Weg vom C-Code zum Maschinencode

Referenzen

Brookshear, J. G. (1997), Computer Science: An Overview, Fifth Edition, Addison-Wesley, Reading, MA.

Intel (2000), "Virtual press kit for 0.18 micron processor launch"

Eine Animation zu den Abläufen in der CPU zu diesem Programm finden Sie auf den Seiten der Virginia Polytechnic Institute and State University (Flashplayer notwendig):

[Animation Assembler-Code und Maschinencode](#)

Falls Sie keinen Flashplayer installiert haben, gibt es noch zwei entsprechende Youtube-Videos:

Die Links finden Sie unter dem Titel "Sum program (Maschinensprache)" und "Sum program (Assembler)"

Einschub: Hardware - Pipelining

Befehlsararbeitung - Geschirrspüler zur Zeit von John von Neumann



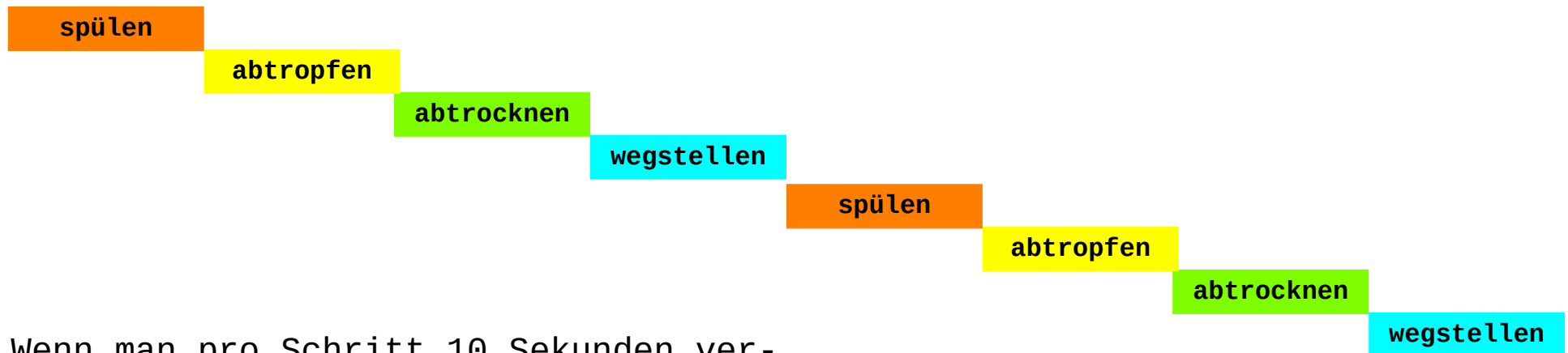
1. Teller abspülen
2. Teller abtropfen lassen
3. Teller abtrocknen
4. Teller in den Schrank stellen

- | | |
|----|------------|
| 1. | spülen |
| 2. | abtropfen |
| 3. | abtrocknen |
| 4. | wegstellen |

Einschub: Hardware - Pipelining

Befehlsararbeitung - Geschirrspüler zur Zeit von John von Neumann

1. Teller abspülen
2. Teller abtropfen lassen
3. Teller abtrocknen
4. Teller in den Schrank stellen



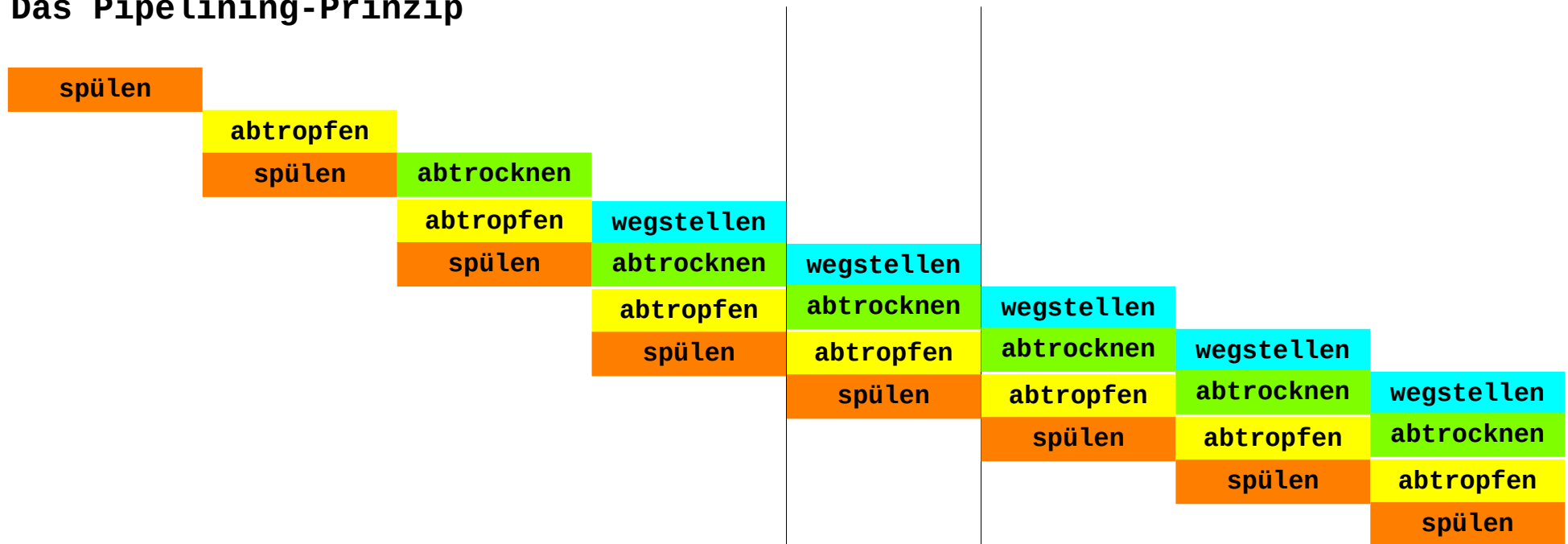
Wenn man pro Schritt 10 Sekunden veranschlagt, benötigt ein Teller bis er trocken im Schrank liegt 40 Sekunden.

u.s.w.

2 Teller = 80 Sekunden

Einschub: Hardware - Pipelining

Das Pipelining-Prinzip



Ergebnis (insgesamt):

1 Teller verbraucht $4 * 10s = 40s$

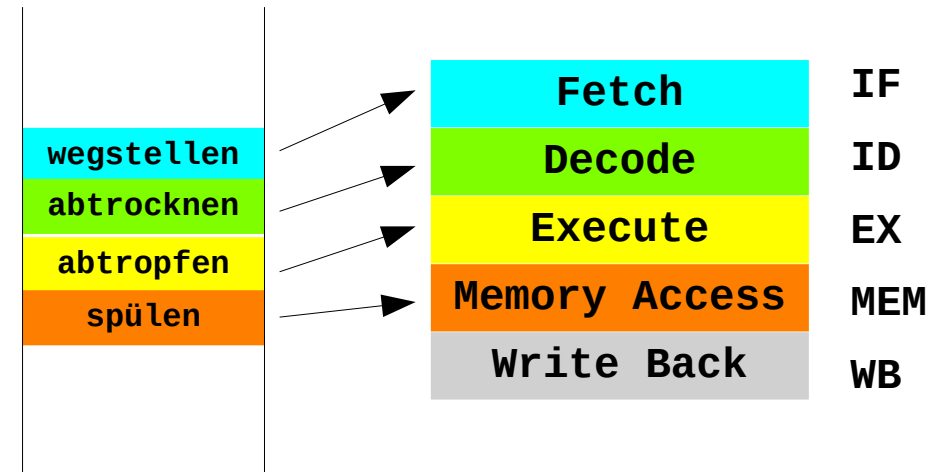
in 80s schafft man jetzt 5 - 8 Teller

u.s.w.

Einschub: Hardware - Pipelining

Das Pipelining-Prinzip:

1. Pipeline als Abfolge von Schritten
2. Parallele Abarbeitung
3. Performancegewinn
4. Durchsatzverbesserung



Write Back bedeutet Zwischenspeichern von Endergebnissen in Registern oder dem Arbeitsspeicher, um sie evtl. für weitere Aufgaben zur Verfügung zu stellen.

Einschub: Hardware - Pipelining

Das Pipelining-Prinzip

1. Pipeline als Abfolge von Schritten:

Eine Pipeline ist eine Abfolge von unabhängigen Schritten oder Phasen, die zur Verarbeitung einer Aufgabe oder eines Prozesses nacheinander durchlaufen werden. Jeder Schritt erledigt eine spezifische Teilaufgabe.

2. Parallele Abarbeitung:

In einer Pipeline können die verschiedenen Schritte parallel oder gleichzeitig abgearbeitet werden. Dies bedeutet, dass, während ein Schritt eine Aufgabe erledigt, ein anderer Schritt bereits mit einer neuen Aufgabe beginnen kann. Das ermöglicht eine gleichzeitige Verarbeitung mehrerer Aufgaben.

3. Performancegewinn:

Im besten Fall, wenn die Pipeline nicht gestört wird und die Schritte gleichmäßig lang sind, kann dies zu einer deutlichen Verbesserung der Leistung führen. Der Gewinn an Verarbeitungsgeschwindigkeit entspricht in gewisser Weise der "Tiefe" der Pipeline, was bedeutet, wie viele Schritte auf einmal bearbeitet werden können.

Einschub: Hardware - Pipelining

Das Pipelining-Prinzip

4. Durchsatzverbesserung, keine *Latenzverbesserung:

Die Pipeline verbessert den Durchsatz, was bedeutet, dass insgesamt mehr Aufgaben pro Zeiteinheit verarbeitet werden können. Dies führt zu einer effizienteren Nutzung der Ressourcen. Allerdings hat die Pipeline keinen Einfluss auf die Latenz, was die Zeit von der Aufgabe bis zur Fertigstellung einer einzelnen Aufgabe bedeutet. Die Latenz kann durch die Pipeline sogar minimal erhöht werden, da es Zeit braucht, um die Pipeline zu füllen und zu leeren.

*Latenz ist im Allgemeinen der Zeitraum zwischen einem Ereignis und dem Eintreten einer sichtbaren Reaktion darauf.

Einschub: Hardware - Pipelining

Ideal pipelined timing

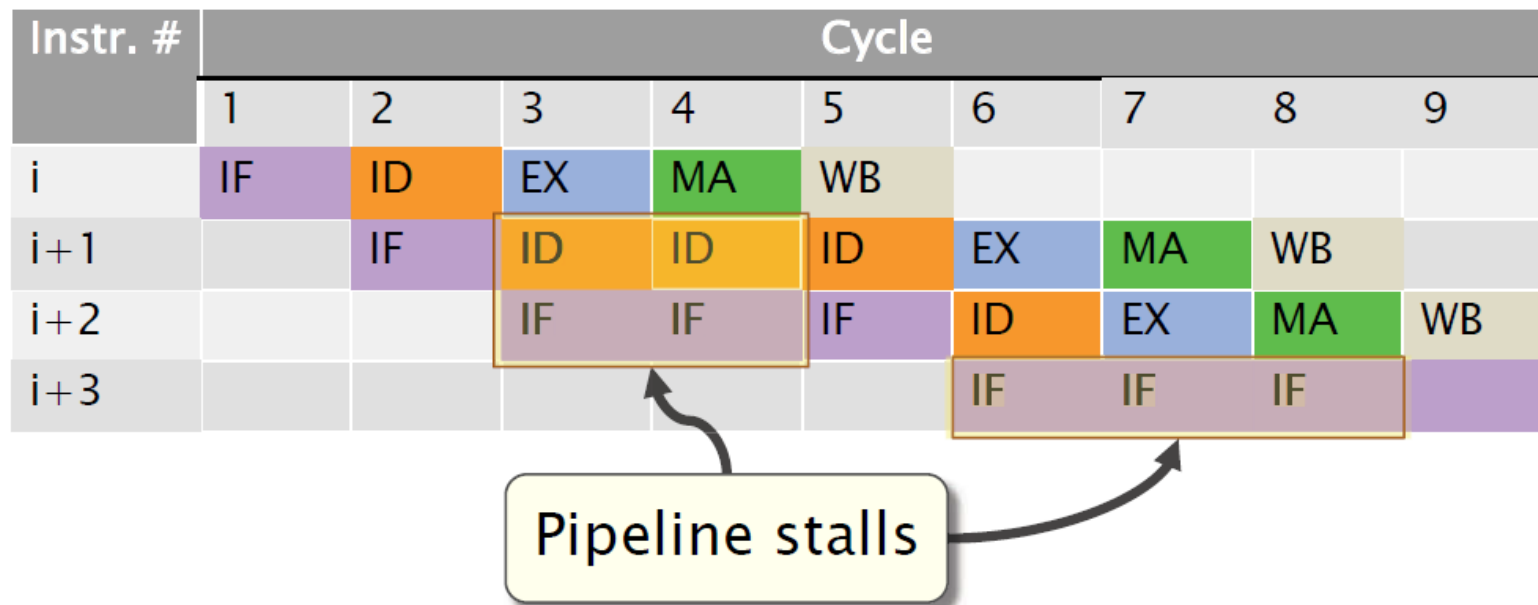
Instr. #	Cycle									
	1	2	3	4	5					
i	IF	ID	EX	MA	WB					
i+1		IF	ID	EX	MA	WB				
i+2			IF	ID	EX	MA	WB			
i+3				IF	ID	EX	MA	WB		
i+4					IF	ID	EX	MA	WB	

Each pipeline stage is executing a different instruction.

Basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). The vertical axis is successive instructions; the horizontal axis is time. So in the green column, the earliest instruction is in WB stage, and the latest instruction is undergoing instruction fetch.

Einschub: Hardware - Pipelining

In der Praxis können verschiedene Probleme während einem Zyklus eine Anweisung an ihrer Ausführung hindern, was dazu führen kann, dass die Pipeline ins Stocken gerät.



dt.: Pipeline-Verzögerung
Pipeline-Stillstand

Einschub: **Hardware - Pipelining**

Drei Arten von Risiken, auch als Hazards bezeichnet, können die Ausführung eines Befehls während seines geplanten Taktzyklus verhindern.

1.Strukturgefährdung (structure hazard)

Eine Strukturgefährdung tritt auf, wenn zwei Anweisungen gleichzeitig versuchen, dieselbe funktionale Einheit in der Pipeline zu nutzen.

2.Datenrisiko (data hazard)

Ein Datenrisiko liegt vor, wenn eine Anweisung von den Ergebnissen einer vorherigen Anweisung in der Pipeline abhängt.

3.Steuerungsrisiko (control hazard)

Ein Steuerungsrisiko tritt auf, wenn das Abrufen und Dekodieren des nächsten auszuführenden Befehls aufgrund einer Entscheidung im Steuerungsfluss, beispielsweise einem bedingten Sprung, verzögert wird.