



Systemnahe Programmierung I

M.Sc. Benedikt Klein - DHBW Ravensburg Campus Friedrichshafen

WELCOME

- Voraussetzungen
- Zeitlicher Aufwand
- Prüfungsleistung
- Kontakt
- Working Space

- Zahlensysteme & Codes & logische Verknüpfungen
- Grundlagen digitaler Systeme
- 22h Vorlesung
- 36h Selbststudium (Skript, Literatur, Aufgaben,...)
- unbenotetes Testat
- b.m.w.klein@arcor.de
- Raspberry Pi oder virtuelle Umgebung (ARM-Core)
- Raspbian (Linux)
- GCC, GDB

LITERATUR

- Larry D. Pyeatt: Modern Assembly Language Programming with the ARM Processor, Newnes, 1. Auflage 2016, ISBN 978-0-12-803698-3
- Tobias Häberlein: Technische Informatik - Ein Tutorium der Maschinenprogrammierung und Rechnertechnik, 1. Auflage 2011, ISBN 978-3-8348-1372-5
- Eben Upton: Learning Computer Architecture with Raspberry Pi, Wiley, 1. Auflage 2.9.2016, ISBN 978-1-119-18393-8
- ARM Infocenter, <http://infocenter.arm.com>, letztes Abrufdatum: 10.01.2018, Achtung: ggf. Registrierung notwendig
- ARM Developer-Center, <https://developer.arm.com/>, letztes Abrufdatum: 10.01.2018, Achtung: ggf. Registrierung notwendig
- Azeria Laboratories, <https://azeria-labs.com/>, letztes Abrufdatum: 10.01.2018
- Think In Geek, <https://thinkingeek.com/arm-assembler-raspberry-pi/>, letztes Abrufdatum: 10.1.2018

Inhalt

1. Einleitung & Motivation
2. Grundlagen der Rechnerarchitektur
3. Speicherorganisation & Adressierungsarten
4. Maschinencode & Befehlsausführung
5. Ausführungszeit & Optimierung
6. Polling & Interrupts
7. Einführung des Übungsrechners
8. Einrichtung der Entwicklungsumgebung
9. ARM Assembler & GNU Toolchain
10. Hands On: ARM Assembler auf dem Übungsrechner

1

Einleitung & Motivation

Worüber sprechen wir überhaupt?

1. Einleitung & Motivation

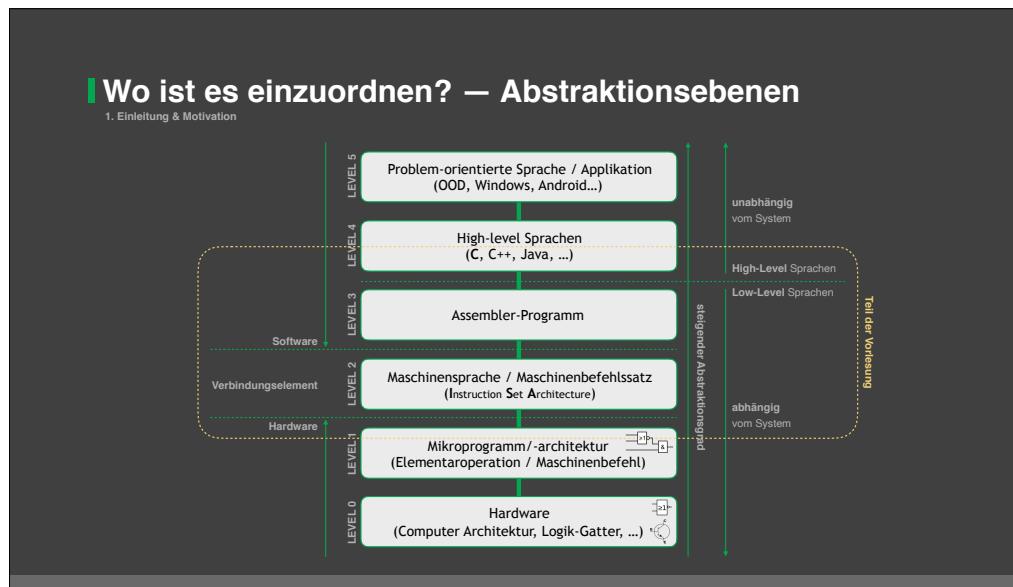
- Maschinenorientierte oder systemnahe Programmierung
 - Low-Level
 - direkte Interaktion mit Hardware
 - maschinenabhängig / Prozessor-spezifisch
- Programmiersprachen
 - C
 - Assembler (Maschinensprache)

In dieser Vorlesung soll der Bezug auf sehr tiefe Level (Low-Level) eines Rechnersystems genommen werden. Im Gegensatz zu Hochsprachen kommen dabei Programmietechniken zum Einsatz, die die direkte Kommunikation mit der Hardware und dem Betriebssystem ermöglichen. Aus diesem Grund muss der Programmierer die Architektur des Zielsystems, mir ihren Möglichkeiten und Grenzen, kennen. Es handelt sich also um eine maschinenabhängige / Prozessor-spezifische Programmierung. Genau hier liegt auch der Unterschied zu den Hochsprachen, dort ist die direkte Interaktion nicht erwünscht, d.h. hierbei muss der Programmierer keine tiefreichende Kenntnis über das Zielsystem haben.

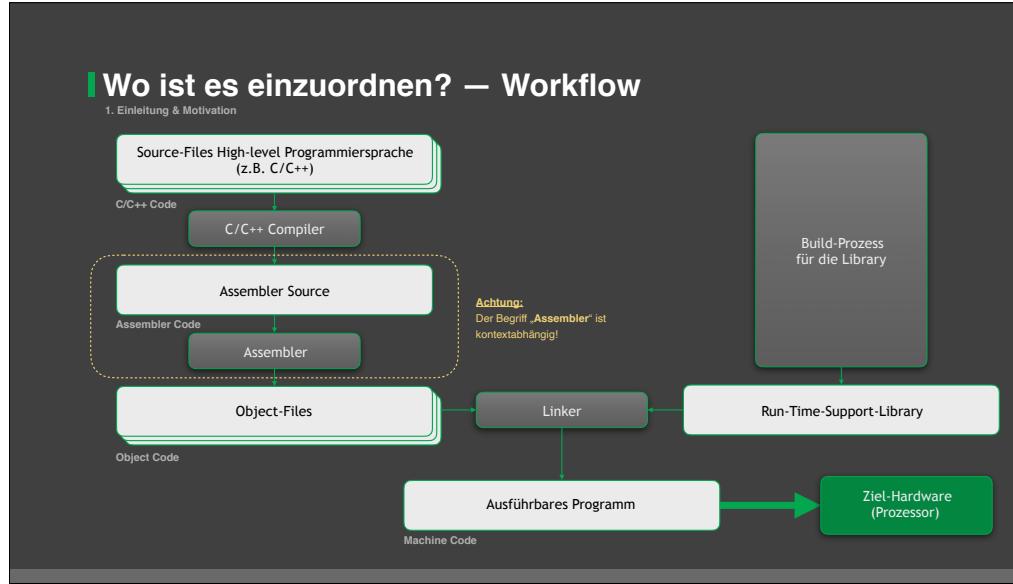
Von besonderem Interesse sind die Programmiersprachen C und die Maschinensprache (bezeichnet als Assembler). Das Sprachkonzept von C beinhaltet Möglichkeiten für den direkten Bezug zur Hardware (z.B. direkter Zugriff auf Speicherstrukturen eines Rechners), wodurch sie sich ebenfalls zur systemnahen Programmierung eignet. Sie ist jedoch bereits unter den Hochsprachen einzuordnen. Aufwendiger zu programmieren ist Assembler. Im Gegensatz zu höheren Programmiersprachen ist Assembler architektur- bzw. maschinenabhängig (prozessorspezifisch). Die dabei verwendeten Befehle werden direkt auf der Ziel-Hardware ausgeführt und haben dadurch direkten Einfluss auf die zugrundeliegende Architektur.

Wo ist es einzuordnen? — Abstraktionsebenen

1. Einleitung & Motivation



Jeder Maschinenbefehl ist durch ein Mikroprogramm implementiert oder fest verdrahtet.



Der hier dargestellte Workflow zeigt den Erzeugungsprozess, welcher notwendig ist um von einem Programm, geschrieben in einer Hochsprache, letztendlich zur Ausführung auf der Hardware zu kommen. Da Prozessoren ausschließlich mit Daten in binärer Form arbeiten können, ist ein Maschinenprogramm auf seiner tiefsten Ebene eine Abfolge von Maschinenbefehlen, die jeweils durch Bitmustern (binär) definiert sind. Diese für uns Menschen nur sehr schwer lesbare Form wird als Maschinencode bezeichnet.

Aus der Grafik geht hervor, dass der Begriff „Assembler“ kontextabhängig ist. Zum einen ist „Assembler“ definiert als Programmiersprache, in der der Maschinenbefehlssatz des Ziel-Prozessors / der Ziel-Hardware in seiner symbolischer Form verwendet wird. Der Begriff steht gleichzeitig für den Übersetzer, welcher die symbolische Form der Maschinenbefehle in binären Maschinencode umsetzt. Diese Folge von Bitmustern kann entsprechend auf einem Rechner verarbeitet werden. In dieser Vorlesung verwenden wir den Begriff vorwiegend als Programmiersprache.

Um dies hier abzurunden sei noch angemerkt, dass ebenfalls der umgekehrte Prozess möglich ist. Mit Hilfe von sogenannten Disassembly-Tools und der Kenntnis der zugrundeliegenden Rechnerarchitektur kann aus einem Maschinencode die symbolische Form, d.h. die Assemblerprogrammierung wiederhergestellt werden.

Zur Vervollständigung ein paar Ergänzung zu den Datenformaten, die während der Erstellung des ausführbaren Programms auftreten:

- ELF - Executable and Linkable Format
- DWARF - Debugging with Attributed Record Format

Diese werden aus den Object-Files erstellt. Die Object-Files beinhalten weiterführende Debugging Informationen, wie z.B. Symbole für den Linker und das Debugging selbst.

Die ELF / DWARF Dateien können dann zu einer Binary (.bin) Datei weiterverarbeitet werden, falls dies zur Übertragung auf das Zielsystem notwendig ist.

Wie sieht es aus?

1. Einleitung & Motivation

```
.data           /* the .data section is dynamically created*/
var1: .word 3   /* variable 1 in memory */
var2: .word 4   /* variable 2 in memory */

.text          /* start of the text (code) section */
.global _start

_start:
    ldr r0, adr_var1 @ load the memory address of var1 via label adr_var1 into R0
    ldr r1, adr_var2 @ load the memory address of var2 via label adr_var2 into R1
    ldr r2, [r0]       @ load the value (0x03) at memory address found in R0 to register R2
    str r2, [r1]       @ store the value found in R2 (0x03) to the memory address found in R1
    bkpt

adr_var1: .word var1 /* address to var1 stored here */
adr_var2: .word var2 /* address to var2 stored here */
```

Warum lernen wir es?

1. Einleitung & Motivation

- Wissensaufbau: Funktionsweise von Prozessoren
- Compiler-Implementierung für Hochsprachen
- Performance-Optimierung
 - Grenzen des Compilers überwinden
 - Nutzung neuer Prozessor-Features
 - In-Line Assembler
 - Interrupt-Handling
- Entwicklung von Betriebssystemen
- Speicher-arme Embedded Systeme
- Debugging / Programmanalyse

... und viele weitere Gründe.

Wissensaufbau: Funktionsweise von Prozessoren

- Durch die Assemblerprogrammierung und das dadurch gewonnene tiefergehende Verständnis darüber wie ein Prozessor funktioniert, wird oft Klarheit geschaffen. Dies ist z.B. bei dem Erlernen von Pointer-Mechanismen in C hilfreich. Es ist sogar fast unmöglich fortgeschrittene Konzepte wie z.B. microcode oder pipelineing ohne dieses Wissen zu verstehen.

Compiler-Implementierung für Hochsprachen

- Bei der Implementierung von Compilern für Hochsprachen ist Assembler unverzichtbar, da der resultierende Compiler basierend auf der Hochsprache Assembler-Code erzeugen muss.
- Die Qualität des erzeugten Assembler-Codes kann maßgeblich für die Performance des Programms auf der Ziel-Hardware sein.

Performance-Optimierung

- Auch hoch optimierte Compiler sind nicht perfekt und arbeiten ausserhalb ihres Anwendungsfalls, für den sie entwickelt wurden, nicht optimal. Gerade die Features neuerer CPUs, wo z.B. auf mehreren Dateninstanzen zur selben Zeit Arbeit verrichtet wird (Single Instruction Multi Data (SIMD)), werden von den Compilern selten genutzt.
- Manche Prozessoren bieten Features um effektiver mit Vektoren (Arrays) umgehen zu können. Compiler nutzen diese Vektor Instruktionen nicht wirklich effektiv aus, was aber gerade für Audio und Video Anwendungen wichtig ist.
- In-Line Assembler: Optimierung von Zeit-kritischen Codeabschnitten, d.h. innerhalb dieser Abschnitte wird Assembler verwendet, um die maximale Performance des Prozessors zu erzielen.
- Interrupt-Handling: Erfolgt z.B. bei einem Mikrocontroller ein externes Event an einem der Eingänge, so kann dies je nach Implementierung zu einer Unterbrechung des Programmablaufs führen. Diese Unterbrechung sollte so optimal und schnell wie möglich ablaufen um eine möglichst geringe Beeinflussung des Programms zu erzielen.

Entwicklung von Betriebssystemen

- Manche Funktionen können nur in Assembler geschrieben werden, obwohl der Rest des Systems in einer Hochsprache geschrieben ist. Hierzu zählen z.B. Treiber oder die Verwaltung von Speicher- und Memory-Protection-Hardware. Natürlich kann man mit einem guten Design die notwendigen Assembler-Implementierungen möglichst gering halten.

Speicher-arme Embedded Systeme

- Heute sind Mikrocontroller nahezu überall aufzufinden. Um Kosten zu sparen, werden auch sehr kleine Derivate mit geringem Speicher eingesetzt. Damit dennoch der Anwendungsfall abgedeckt werden kann, muss die Implementierung des Source-Codes so effizient und effektiv wie möglich erfolgen, oder anders gesprochen es kommt auf jedes Byte an. Auch hier ist Assembler eine gute Wahl.
- Es sei an dieser Stelle angemerkt, dass die Wahl eines Mikrocontrollers immer eine gewisse Kapazitätsreserve beinhalten sollte. Nur so haben Funktionalitäten, die nachträglich hinzukommen oder zu Beginn falsch abgeschätzt wurden, eine Chance in das Produkt einzufließen.

Debugging / Programmanalyse

- Mit Hilfe der erlernten Kenntnisse im Bereich der systemnahen Programmierung ist es möglich Implementierungen auf einem sehr tiefen Level zu verstehen. Das Verständnis darüber, was hinter den Szenen eines kompilierten Programms einer Hochsprache (z.B. C/C++) abläuft, kann bei der Fehlersuche maßgeblich über den Erfolg entscheiden. Dabei kann es notwendig sein durch die einzelnen Instruktionen des CPUs durchzusteppen und dessen Register zu betrachten.
- Die erlernten Fähigkeiten können so gewinnbringend bei der Programmanalyse, der Durchführung von Optimierungen und zur Fehlersuche eingesetzt werden.

Mögliche weitere Gründe:

- In den ersten Schritten um einen Computer zu starten.
- Low-Level Locking Code für Multi-Thread Programme
- Sourcecode für Prozessoren für die kein Compiler existiert.
- Sourcecode der Low-Level Zugriff auf die Prozessor-Features/-Architektur benötigen.

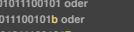
MISSION



„ Die Fähigkeit Mikroprozessoren **systemnah**
(Assembler) programmieren und das Wissen zur
Programmanalyse nutzen zu können. ”

2

**Grundlagen der
Rechnerarchitektur**

Zahlensysteme & Co. (Wiederholung)					
3. Maschinencode und Befehlausführung					
System	Darstellung				Schreibweise
Hex	F	2	E	5	0xF2E5 oder F2E5h oder F2E5H
Binary					0b1111001011100101 oder 111001011100101b oder 111001011100101B
Voltage					
Bus lines	Line 15	System bus			Line 0

Welchen Zahlenwert repräsentiert der Hex-Wert im Dezimalsystem?

Unser dezimales Zahlensystem hat die Basis 10. Das Hexadezimale System verwendet die Basis 16, wohingegen das binäre System die Basis 2 verwendet.

Logic Gates (Wiederholung)

3. Maschinencode und Befehlausführung

- NOT

$$\begin{array}{l} \text{NOT } 1 = 0 \\ \text{NOT } 0 = 1 \end{array}$$

- AND

$$\begin{array}{l} 1 \text{ AND } 1 = 1 \\ 1 \text{ AND } 0 = 0 \\ 0 \text{ AND } 1 = 0 \\ 0 \text{ AND } 0 = 0 \end{array}$$

- OR

$$\begin{array}{l} 1 \text{ OR } 1 = 1 \\ 1 \text{ OR } 0 = 1 \\ 0 \text{ OR } 1 = 1 \\ 0 \text{ OR } 0 = 0 \end{array}$$

- XOR

$$\begin{array}{l} 1 \text{ XOR } 1 = 0 \\ 1 \text{ XOR } 0 = 1 \\ 0 \text{ XOR } 1 = 1 \\ 0 \text{ XOR } 0 = 0 \end{array}$$

Quelle: Eben Upton: Learning Computer Architecture with Raspberry Pi, Wiley, 1. Auflage 2.9.2016, ISBN 978-1-119-18393-8

Externe vs. interne Architektur

2. Grundlagen der Rechnerarchitektur

- Rechnerarchitektur
 - Design und Organisation eines Rechners
 - bestimmt externen & internen Aufbau
 - definiert Schnittstelle zwischen SW & HW
- externe Architektur
 - definiert Programmiermodell
 - maßgeblich für Codegenerierung
 - Maschinenbefehlssatz (ISA)
 - enthält Elementarbefehle (Elementaroperationen)
- interne Architektur
 - Hardwareaufbau / interne Realisierung
 - Rechnerorganisation



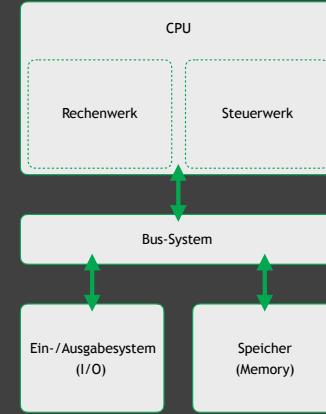
Das Programmiermodell, also die Art wie ein Rechner von der Seite der Software angesprochen werden kann, wird durch die externe Architektur, genauer durch deren Maschinenbefehlssatz, definiert. Dieser kann auch als die Sicht des Programmierers auf den Prozessor bezeichnet werden und ist die Schnittstelle, welche für die Codegenerierung maßgeblich ist. Der Maschinenbefehlssatz enthält alle Elementaroperationen (z.B. Laden, Speichern oder einfache Mathematik (z.B. Addition)) und ist auch als Befehlssatzarchitektur oder ISA (Instruction Set Architecture) bezeichnet.

Die interne Architektur definiert den inneren Hardwareaufbau eines Rechners, sie beschreibt die Hardware selbst. Sie beschreibt wie die einzelnen Komponenten zusammenarbeiten um die externe Architektur zu implementieren. Hinter der interne Architektur steht die Rechnerorganisation (Mikroarchitektur), d.h. High-Level Aspekte wie z.B. das Design der CPU, des Speichersystems oder des Systembusses.

Interne Architektur

2. Grundlagen der Rechnerarchitektur

- Prozessor (CPU)
 - Rechenwerk / Operationswerk
 - Steuerwerk
- Bus-System
- Speicher (Memory)
- Ein-/Ausgabesystem (Input / Output)



Um das Programmiermodell verstehen zu können muss zunächst die interne Architektur verstanden sein. Hierzu gibt es eine separate Vorlesung „Rechnertechnik“, dennoch soll kurz auf die Kernelemente und deren Zusammenhänge, welche für die systemnahe Programmierung notwendig sind, eingegangen werden.

Der Prozessor (die CPU) stellt die zentrale Einheit des Rechners dar. Er besteht aus dem Steuerwerk und dem Rechenwerk.

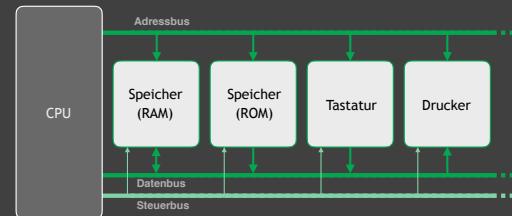
Das Rechenwerk oder Operationswerk, wendet arithmetische und logische Operationen auf Daten an. Es manipuliert / verarbeitet diese und deckt dadurch den datenverarbeitenden Teil des Prozessors ab. Die in dem Rechenwerk beheimatete Arithmetic Logic Unit (ALU) stellt hierzu die notwendigen Basisfunktionen zur Verfügung und besitzt zusätzlich Register zur Zwischenspeicherung der Ergebnisse. Durch Binärwerte (boolean), welche als „Condition Flags“ zusammengefasst werden, werden besondere Merkmale der Operanden oder des Ergebnisses bei der Datenverarbeitung signalisiert.

Im Gegensatz zum Rechenwerk ist das Steuerwerk oder Leitwerk der koordinierende Teil des Prozessors. Das Steuerwerk ist für die Abarbeitung und Ausführung von Befehlen aus dem Speicher zuständig. Es interpretiert die im Programmablauf definierten Maschinenbefehle schrittweise (Steuerung des Programmablaufs). Dabei übernimmt es unterschiedliche Aufgaben, wie z.B. das Laden und Decodieren von Befehlen, das Laden und Speichern von Operanden und Ergebnissen und die Steuerung der Befehlausführung.

Bus-System

2. Grundlagen der Rechnerarchitektur

- BUS (Binary Unit System)
- Steuerung, Adressierung und Übertragung von Daten
- Systembus verbindet:
 - CPU
 - Speicher
 - Ein-/Ausabesystem
- Aufteilung in:
 - Adressbus
 - Datenbus
 - Steuerbus
- Wichtig: Busbreite



Der Systembus (das Bus-System) dient der Steuerung, Adressierung und Übertragung von Daten. Durch den Begriff „Bus“ ist definiert, dass durch mehrere Leitungen einzelne Funktionseinheiten verbunden sind. Um hier die Anzahl der Leitungen auf dem „Die“ gering zu halten, wurde die parallele Verbindung der Funktionseinheiten gewählt. Hierbei kann weiter in unidirektionale und bidirektionale Busse unterschieden werden. Ein Drucker z.B. nur Daten annehmen oder eine Tastatur nur Daten als Eingabe liefern. Nicht so z.B. der RAM, hier können Daten sowohl gelesen als auch geschrieben werden.

Seriell = 1 Leitung

Parallel = mehrere Leitungen

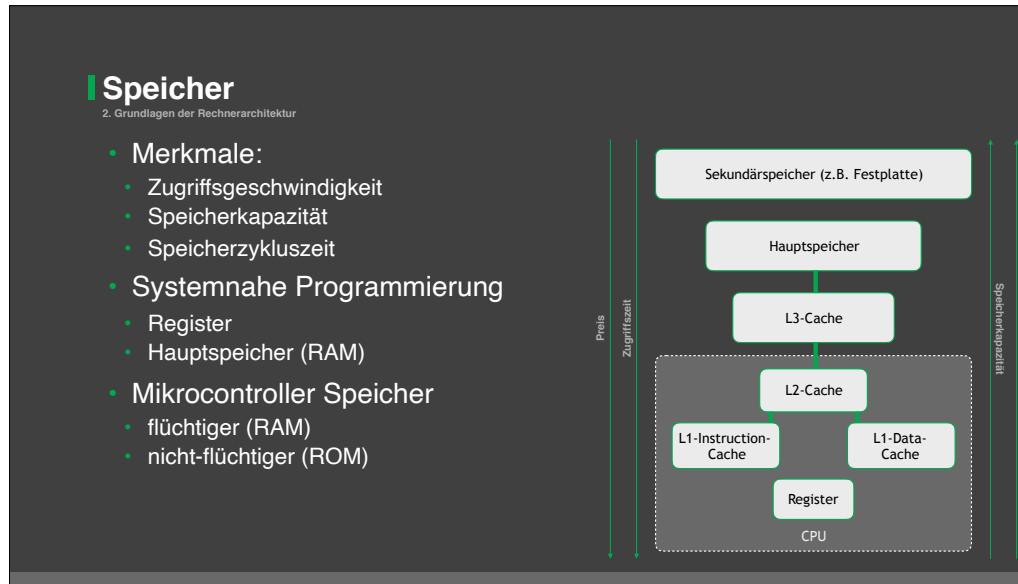
Breite des Datenbusses = Länge des Datenpuffers

Breite des Adressbusses = Länge des Speicheradressregisters

Mittels des Datenbusses werden Daten zwischen CPU und Peripherie übertragen. Je größer die Busbreite ist, desto mehr Daten können gleichzeitig übertragen werden. Aktuelle Systeme variieren zwischen 8 und 64 Bit, was auch gleichzeitig der Anzahl der Leitungen entspricht. Daraus ist leicht ersichtlich, dass Systeme mit großer Busbreite performanter sind.

Das Adressbus ist im Gegensatz zum Datenbus unidirektional. Mit Hilfe dieses Busses ist die CPU in der Lage, durch eine eindeutige Adresse, Peripherie zu adressieren. Auch hier gilt, je größer die Adressbusbreite ist, desto mehr Peripherieeinheiten können an dem Bus betrieben werden. Besitzt das System z.B. über eine 32-Bit Adressierung, können 4.294.967.296 Speicherzellen (4GB) in einem Speicherbaustein angesprochen werden. (1 Speicherzelle = 1 Byte = 8 Bit). Es ist darauf zu achten, dass die Adressierung mit der Null beginnt.

Über den Steuerbus kann der Prozessor an einer Peripherieeinheit ein Signal anlegen, welches dieser signalisiert, dass ein Zugriff auf sie erfolgen soll. Ist diese Einheit z.B. ein Speicherbaustein, kann die CPU so die gewünschte Speicherzelle mit Hilfe des Adressbusses erreichen.

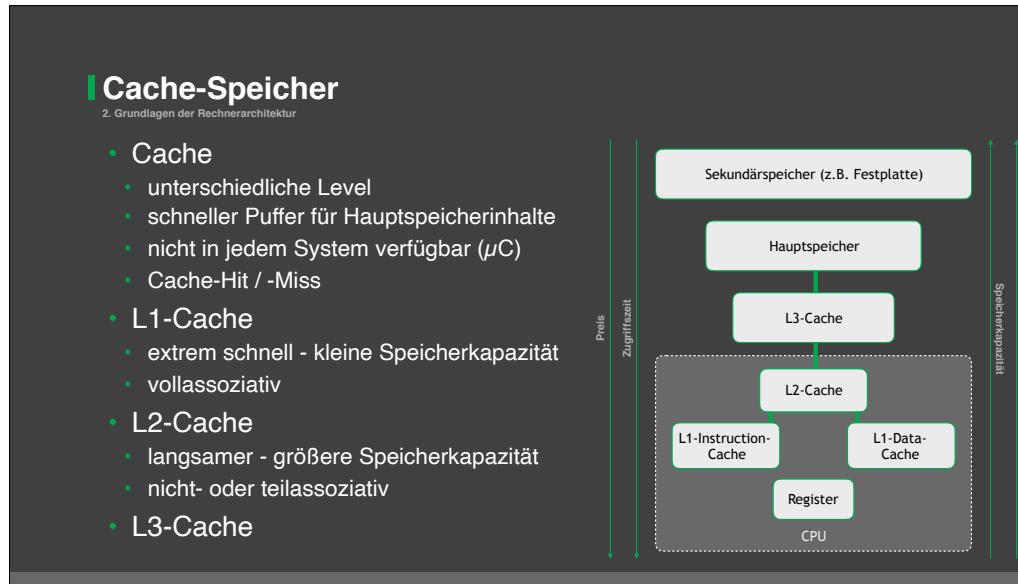


Rechner verfügen über eine Hierarchie von unterschiedlichen Speicherarten. Im Wesentlichen lassen sie sich über die Verwendungsart, die Speicherkapazität, Zugriffsgeschwindigkeit und die Speicherzykluszeit klassifizieren. Die Speicherzykluszeit setzt sich aus Zugriffszeit und allen zusätzlichen Aktivitäten zusammen, die zur korrekten Verwendung des gewählten Speichertyps notwendig sind. (z.B. vor dem erneuten Schreiben muss der Inhalt der Speicherzelle gelöscht werden)

Mit Sicht auf die systemnahe Programmierung sind die Register und der Hauptspeicher von großer Bedeutung. Sie können durch Prozessorbefehle direkt angesprochen werden. Der Prozessor legt in den Registern z.B. Operanden und die Ergebnisse der arithmetischen Operationen ab. Hingegen dient der Hauptspeicher der Speichererweiterung und beinhaltet das auszuführende Programm. Der Prozessor wird aus Geschwindigkeitsgründen versuchen die notwendige Information aus möglichst nahegelegenen Ebenen zu holen. Ist dies nicht möglich muss die Information auf den höheren Ebenen eingeholt werden, von den Sekundärspeichern. Dies ist deutlich zeitaufwändiger.

Bei der Betrachtung von embedded Systemen, die häufig mit Hilfe von Mikrocontrollern realisiert sind, ist die Unterscheidung in flüchtigen und nicht-flüchtigen (permanenten; nonvolatile memory) Speicher zu treffen. Es spielt das RAM (Random Access Memory) und das ROM (Read Only Memory) eine entscheidende Rolle, sie werden auch als Primärspeicher bezeichnet.

- RAM: Anwendungsdaten und Programme die temporär zur Laufzeit auf dem Rechner gespeichert werden. → Der Wegfall des Versorgungssystems bedeutet auch einen Datenverlust (flüchtiger Speicher).
- ROM: Essentielle Daten (Tabellen) und Programme (Initialisierungsprogramme) die für die gewünschte Funktion des Rechners verantwortlich sind. → Der Wegfall des Versorgungssystems bedeutet keinen Datenverlust (nicht-flüchtiger Speicher).



Ein schneller Zwischenspeicher wird durch den Cache repräsentiert. In ihm werden z.B. zuletzt benutzte Daten aus dem Hauptspeicher vorgehalten. Die Idee basiert auf dem Lokalitätsprinzip, nahegelegenes bzw. zuletzt verwendete Daten werden häufig noch einmal benötigt. Sind die Daten hier vorhanden muss der Prozessor sie nicht noch einmal von langsameren Speichern anfordern und abholen. Es kann von „Wartezeiten“ gesprochen werden, denn das Abholen eines Befehls aus dem Speicher dauert meist erheblich länger als dessen Ausführung.

Liegt eine angeforderte Information im Cachepreicher, so spricht man von einem Cache-Hit andernfalls von einem Cache-Miss. Bei einem Cache-Miss wird eine bestimmte Speicherportion, die ebenfalls die geforderte Information beinhaltet, von dem Hauptspeicher in den Cachespeicher geladen. Es wird einer Verdrängungsstrategie gefolgt, d.h. die am längsten nicht benutzten Speicherportionen werden ersetzt (Least Recently Used (LRU-Strategie)). Daraus geht hervor, dass gilt: Je größer der Cachespeicher desto geringer die Cache-Miss-Rate.

In modernen Prozessoren werden Cache-Hierarchien verwendet. So befindet sich der Level-1 Cache direkt auf dem Prozessor. Er hat keine große Kapazität ist aber extrem schnell und kommt in doppelter Ausführung vor. Einmal für die Daten und ein zweites Mal für die Befehle. Als nächste Ebene kommt der Level-2 Cache, welche über eine größere Kapazität verfügt aber langsamer ist. Er besitzt keine Trennung zwischen Daten und Befehlen. Ob er sich noch auf dem Prozessorchip oder Off-Chip (externen Cache) befindet ist Designabhängig. Level-3 Cache ist die logische Fortsetzung des Ganzen...

Für die systemnahe Programmierung ist es wichtig die Mechanismen des Cache-Speichers verstanden zu haben. Er wird jedoch in der Regel unabhängig von der CPU (z.B. durch das Speicherwerk (Memory Management Unit) verwaltet).

Warum ist ein kleinerer Cache-Speicher u.a. schneller? Es sind weniger Adressbits zu durchlaufen und es ist einfacher festzustellen ob die benötigten Daten sich bereits im Cache befinden.

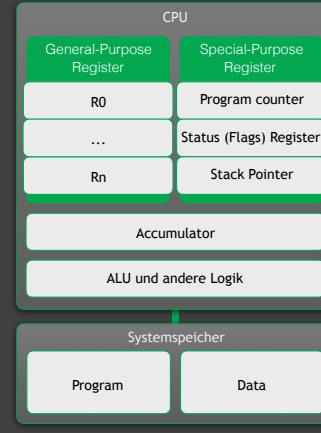
Assoziativspeicher:

Inhaltsbasierter Speicher, d.h. es wird auf den Speicherinhalt durch die Eingabe eines Speicherwertes und nicht der Speicheradresse zugegriffen. Hierzu später noch einmal mehr.

Register

2. Grundlagen der Rechnerarchitektur

- prozessorinterne Speicherplätze
- sehr viel schnellerer Zugriff
 - direkt in CPU
 - Transport über Datenbus entfällt
 - Adressrechnung entfällt
- Unterscheidung in:
 - Universal-/Allzweck-Register (General-Purpose)
 - Spezial-Register (z.B. Programmzähler, Stack Pointer)
- haben eindeutige Namen



Register sind Speicherplätze die direkt im Chip der CPU untergebracht sind. Sie können mit Daten über den Datenbus gefüllt werden oder z.B. Berechnungsergebnisse der CPU enthalten. Gleichermaßen kann der Registerinhalt über den Datenbus ausgegeben werden. Sieht man davon ab, also wird der reine Zugriff auf das Register betrachtet, entfällt der Transport über den Datenbus und die Adressrechnung. Aus diesem Grund sind sie sehr viel schneller und sehr „wertvoll“. Je mehr Register ein Controller hat, desto weniger oft muss er auf den Speicher zugreifen um Zwischenergebnisse abzulegen. Speicherzugriffe sind langsam und verzögern ggf. die Ausführungs-/Berechnungszeit.

Register haben Namen, dies macht auch aufgrund ihrer begrenzten Anzahl Sinn. Sie können bei der späteren Programmierung in Assembler wie "Variablen" aufgefasst werden. Hierzu im Verlauf der Vorlesung mehr.

Übliche Special-Purpose-Register sind:

- **Program Counter (Programmzähler, PC):** Beinhaltet die Adresse des, als nächstes auszuführenden, Maschinenbefehls, welcher aus dem Speicher zur Ausführung geholt werden muss. Er merkt sich also die Stelle an der sich das Programm zur Zeit befindet.
- **Status oder Flag Register:** Hier werden, anhand der einzelnen Bits des Registers, einzelne Informationen (Status) des Systems registriert. Jedes Bit wird nach einer Aktion der CPU aktualisiert. So ist es möglich bei (z.B.) einem Vergleich dessen Ausgang/Ergebnis festzuhalten, sodass folgende Instruktionen darauf reagieren können.
- **Stack Pointer:** Zeigt auf die Adresse einer Datenstrukturen im Speicher, welche einen Stack („Last-In-First-Out“) abbildet.
- **Accumulator:** Hält das Ergebnis einer arithmetischen oder logischen Operation. In moderneren Systemen hat sich dies etwas gewandelt, hier können ebenfalls einfache (auch mehrere) General-Purpose-Register seine Aufgabe bekommen. Da jedoch ältere Systeme davon ausgehen, dass ein einzelnes Register das Ergebnis beinhaltet, gibt es diese Bezeichnung noch immer.

Ein-/Ausgabesystem

2. Grundlagen der Rechnerarchitektur

- Kommunikation mit Umwelt
- Teilnehmer des Datenbusses
- Unterscheidung des Datentransports
 - befehlsbezogen E/A
 - speicherbezogen E/A
 - unterbrechungsgesteuerte E/A (Interrupts)

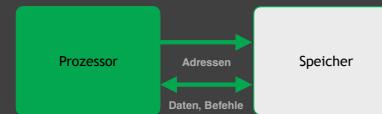
Da das Ein-/Ausgabesystem Daten übermittelt, d.h. entweder Daten von extern angeschlossenen Geräten entgegennimmt oder Daten an angeschlossene Geräte weitergibt, ist es ein Teilnehmer des Datenbusses. Die Art und Weise wie die Daten transportiert werden, hilft bei der Unterscheidung. Liegt eine befehlsbezogene E/A vor, so wird mittels einer spezifischen Adresse ein Funktionscode an die Peripherie übermittelt. Die speicherbezogene E/A (engl. memory mapped) reserviert Bereiche des Speichers (Arbeitsspeicher) für die Peripheriegeräte. Dies ermöglicht den Zugriff auf derartige Systeme in gleicher Form wie der Zugriff auf gewöhnliche Speicheradressen realisiert ist. Die unterbrechungsgesteuerte E/A triggert eine Unterbrechung der aktuellen Programmausführung auf dem CPU. Dies ermöglicht eine direkte Reaktion auf ein externes Event. Es sorgt ebenfalls für eine zeitliche Entlastung des Prozessors, denn dieser muss so keine zyklischen Wartezeiten und Statusabfragen an die Peripherie durchführen. Dieses sehr wichtige Konzept wird im Verlauf der Vorlesung noch einmal vertieft.

Es sei angemerkt, dass je nach verwendeter Hardware auch eine Datenaufbereitung stattfindet. Dies gilt für beide Richtungen, also für Ein- und Ausgabe. Um mit den Daten die am Eingang anliegen arbeiten zu können, müssen sie ggf. in eine normierte Darstellung gebracht werden. Gleichermaßen kann es notwendig sein ein definiertes Ausgabeformat (Signalpegel) einhalten zu müssen. Dies liegt außerhalb des Fokus unserer Vorlesung.

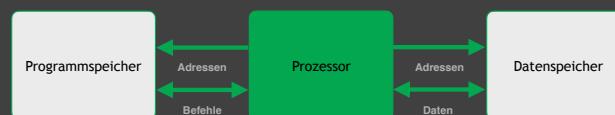
Von-Neumann vs. Harvard-Architektur

2. Grundlagen der Rechnerarchitektur

- Von-Neumann



- Harvard



Die Von-Neumann-Architektur bewahrt sowohl das Programm als auch die Daten im selben Speicher, wodurch die Implementierung einfacher ist. Diese Architektur ist in den meisten universell einsetzbaren Rechnern zu finden.

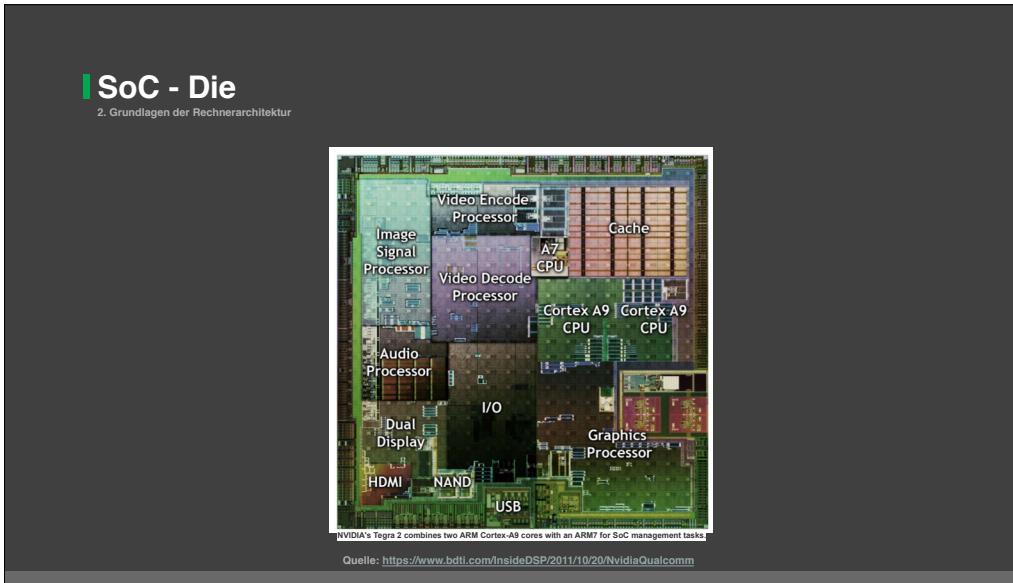
Hingegen verwaltet die Harvard-Architektur die Programmbefehle und die Daten in getrennten Speichern. Dies ermöglicht eine schnellere Arbeitsweise, ist jedoch komplexer. Diese Architektur ist in speziellen Anwendungen, bei denen sie dem System deutliche Vorteile einräumt, implementiert. Als Beispiel können hier einige Mikrocontroller oder DSPs (Digital Signal Processor) genannt werden.

Bei der Betrachtung von modernen CPUs ist zu sehen, dass diese oftmals der Architektur von Von-Neumann folgen. Jedoch ist z.B. bei dem Level-1-Cache die Harvard-Architektur zu erkennen, denn hier sind zwei Zwischenspeicher implementiert. Einmal der Level-1-Instruction-Cache und der Level-1-Data-Cache.

Mikroprozessor vs. Mikrocontroller

2. Grundlagen der Rechnerarchitektur

- Mikroprozessor
 - nackte „CPU“
- Mikrocontroller
 - CPU
 - Speicher
 - Ein-/Ausgabesystem
 - Timer
 - Interrupt
 - ...



Die: Bezeichnung für einen Halbleiter-Wafer, welcher sich noch nicht in einem Gehäuse befindet.

SoC: Ein System on Chip ist eine integrierte Schaltung, welche alle Komponenten eines elektronischen Systems (z.B. Computer) auf einem Chip vereint. Neben der Central Processing Unit (CPU) stehen z.B. eine Graphics Processing Unit (GPU), Timers (time-dependent scheduling, synchronising,...), Interrupt Controller, General-Purpose-Input-Outputs (GPIO) und USB-Schnittstellen zur Verfügung.

Der hier in der Vorlesung betrachtete Raspberry Pi ist ein solcher SoC.

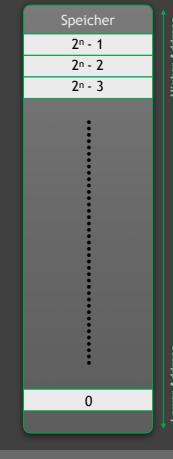
3

Speicherorganisation & Adressierungsarten

Speicherorganisation

3. Speicherorganisation & Adressierungsarten

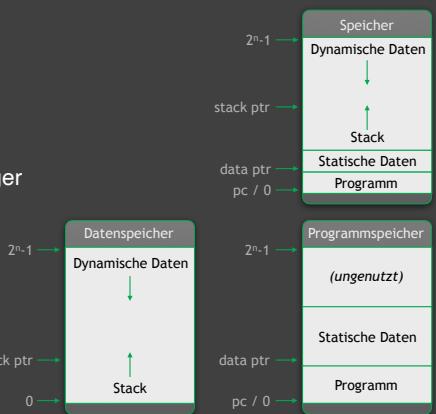
- (üblicherweise) in Bytes organisiert
 - d.h. eine Speicherzelle ist 8 Bit groß
- Speicheradressen beginnen bei 0
 - erleichtert die Arithmetik auf den Adressen
- maximale Anzahl ansprechbarer Speicherzellen
 - durch Busbreite des Adressbusses definiert
 - n -Bit Adresse = 2^n adressierbare Bytes



Speicherorganisation: Von-Neumann vs. Harvard

3. Speicherorganisation & Adressierungsarten

- Von-Neumann
 - gemeinsamer Speicher = 1 Adressraum
 - zu berücksichtigen bei Adresszuordnung:
 - Relozierbarkeit
 - Geschwindigkeit
 - relative Adressierung mittels Referenzzeiger
- Harvard
 - getrennte Speicher = 2 Adressräume
 - unterschiedliche Implementierung
 - Zugriffswege auf die Speicher geteilt oder separat
- interner und externen Bereich



Bei der Adresszuordnung, für sowohl Befehle und Daten, sollte auf die Verschiebbarkeit der verschiedenen Bereiche im Adressraum und auf Geschwindigkeit geachtet werden.
Statt der absoluten Adressierung erfolgt eine, zu den in den Referenzzeigern (Register) gehaltenen Referenzadressen, relative Adressierung.

Bei der Geschwindigkeit sei noch einmal angemerkt, dass die Register für die Übergabe von Parametern die schnellste Möglichkeit darstellen.

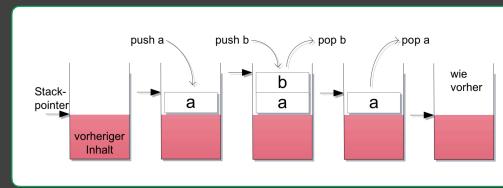
Bei der Harvard-Architektur werden die beiden Speicher in eigenen Befehlsgruppen verarbeitet, daher sind die Adressräume zu trennen. Entscheidend ist ob derselbe Zugriffsweg auf den Speicher verwendet wird. Sind sie getrennt, lassen sich Optimierungen für Technologien durchführen und die Performance erhöhen. Wo liegt der Vorteil bei einem geteilten Zugriffsweg? Hier lassen sich immer noch unterschiedliche Technologien verwenden und auch die Adressräume komplett nutzen.

Interner und externer Bereich, d.h. auch hier gibt es, egal welche Architektur eingesetzt wird, unabhängige Adressräume. Das Handling wird schwieriger, da mehrere Adressräume dieselbe Adresse haben und eine Zuordnung zu dem gewünschten Speicherbereich, in eindeutiger Form, vorgenommen werden muss. Hat natürlich auch seine Vorteile und wird in der Praxis eingesetzt.

Speicherorganisation: Stack

3. Speicherorganisation & Adressierungsarten

- Struktur LIFO (Last-In-First-Out)
- Rettung von Registerinhalten
- meist Teil des RAM-Speichers (kein eigener phys. Speicher)
- Anwendungsbereiche:
 - Unterprogrammaufrufe (Sprünge)
 - lokale Variablen
 - Interrupt Handling



Bildquellen: VoB, Prof. Dipl.-Ing. Sven-Hendrik; Vorlesungsskript Maschinenorientierte Programmierung; 17.10.2016

Rettung der Registerinhalte bei Verzweigungen (branches) und Mehrfachaufufen (Rekursion).

Bei einem Unterprogrammaufruf (Sprung), wird die Rücksprungadresse auf dem Stack gelegt. Ist das Programm abgearbeitet, erfolgt ein spezieller Befehl, der dann von dem Stack diese Adresse in den Programmzähler lädt. Es wird auch genutzt um Parameter an Unterprogramme zu übergeben.

Lokale Variablen, die z.B. nur während einer Routine Verwendung finden, werden bei Rücksprung aus dieser Routine wieder verworfen.

Im Fall eines Interrupts werden sowohl die Rücksprungadresse, als auch die Registerinhalte (inkl. Prozessorstatusregister) auf dem Stack gelagert (gerettet). Der Zustand des Systems muss nach der Abarbeitung des Interrupts wiederhergestellt werden.

Speicherorganisation: Daten-Typen / x-Bit Prozessor

3. Speicherorganisation & Adressierungsarten

- „Word“
 - natürliche Einheit eines Rechnerdesigns
 - feste Menge an Daten die als Einheit verarbeitet wird (Hardware)
- Größe eines „Word“
 - wichtige Eigenschaft einer Rechnerarchitektur (besonders für Speicher)
 - entscheidend für die Struktur und Operationsweise des Rechners
 - z.B. entsprechen meist die Register dieser Größe
 - z.B. Menge der Bits die in einer Operation vom Hauptspeicher gelesen werden kann
 - 8, 16, 32 oder 64 Bits
 - Moderne Computer: 32 oder 64 Bits
 - Embedded Systeme: 8, 16, 32 oder 64 Bits
 - DSP: viele weitere Größen
 - Achtung: Rückwärts-Kompatibilität
 - z.B. zu älteren Rechnern derselben Familie / Variationen innerhalb einer Familie

Die Word-Größe oder x-Bit geben an, wieviele Bits die Verarbeitungseinheit der CPU in einem Befehl als Einheit verarbeiten kann. Mit ihr ist die Registerbreite eng verknüpft, da (zumindest) die Universalregister in der Lage sein müssen den Inhalt der ALU zu übernehmen. Auch die Datenbusbreite entspricht oftmals diesem Wert, es kann jedoch vorkommen, dass interne und externe Datenbusbreite der CPU voneinander unterschiedlich sind.

Die kleinste Einheit eines Speichers, welche durch die Adresse adressiert (Adressbus) werden kann, entspricht meist der Word-Größe. D.h. auch die Erhöhung der Adresse um den Wert 1 führt dazu, dass auf das nächste Word im Speicher gezeigt wird. Dies ist ein Vorteil für die Befehle in Maschinen, die so gut wie ausschließlich in Word (oder mehreren Words) arbeiten. Denn so muss die Befehlsdefinition nur das Minimum an Feldern für die Adressierung bereitstellen, wodurch eine kleinere Befehlsgröße erzielt wird. Dies kann also auch das physikalische Design des Systems beeinflussen. Falls der Adressbus (z.B.) zwei Bytes weniger benötigt, benötigt er auch zwei Pins an der CPU und ebenso die zwei dazugehörigen Leitungen weniger. Es ist klar ersichtlich, dass dies natürlich erst zum tragen kommt, wenn die Rechnerarchitektur das Word nicht durch ein einfaches Byte (8 Bit) definiert.

Es sei angemerkt, dass in Systemen, die oftmals nur mit einzelnen Bytes arbeiten, das Byte die bessere Auflösung für die Adressierung darstellt. Wobei das System nicht zwingend das Byte als Word definieren muss. Dennoch kann das Word, welches z.B. mit 16 Bit definiert ist, mit nur wenigen zusätzlichen Bits in der Befehlsdefinition adressiert werden (im Vergleich zu dem oben genannten Vorteil). Bei der Adressierung eines Words muss in diesem Fall darauf geachtet werden, dass der Adresswert ein Vielfaches der Word-Größe darstellt. In klassischen Computern konnten, aufgrund der Hardware, Speicherinhalte nur in Words adressiert werden. Also nur mit Offsets, die ein Vielfaches der Word-Größe darstellen.

Wie kann dennoch bei einer Word-Größe von z.B. 16-Bit auf einem einzelnen Byte gearbeitet werden?

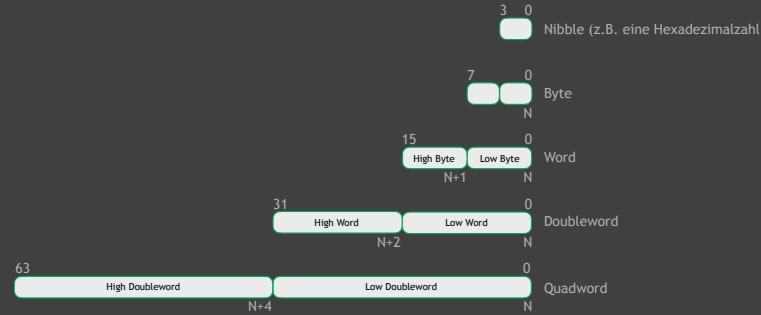
- mit Schiebe- (Shift) und Maskierungsoperationen (AND, OR, ...)

Würde bei Prozessor-Variationen (z.B. einer Rechner-Familie), welche die selbe Architektur und den selben Befehlssatz verwenden, die Word-Größe variieren, so würde die Software-Entwicklung und auch die Dokumentation sehr komplex werden. Als Beispiel sei die x86-Familie aufgeführt, die in drei verschiedenen Word-Größen veröffentlicht wurde. Damit die Software jedoch auf der gesamten x86-Familie läuft gibt es APIs die eine andere (ältere) Word-Größe softwareseitig definieren/verwenden, als die der tatsächlich verwendeten CPU. Zum Beispiel definiert die Microsoft Windows API das Word als 16 Bit-Wert, egal ob das System auf einem 32- oder 64-Bit System ausgeführt wird.

Speicherorganisation: Daten-Typen

3. Speicherorganisation & Adressierungsarten

- Operanden können folgende Daten Typen besitzen:



- Achtung: Zur Veranschaulichung wird ein 16 Bit Word als Beispiel genommen.

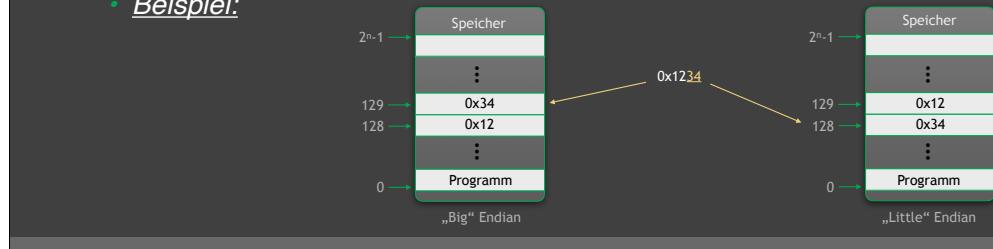
Es ist bei der jeweiligen Architektur zu prüfen wie groß die einzelnen Datentypen sind.

Zusätzlich gibt es noch die Begriffe „Double-Quadword“ (im Beispiel: 128 Bits) und das „Longword“ (meist 32 oder 64 Bits).

Ergänzung zu den Daten-Typen: Endianness

3. Speicherorganisation & Adressierungsarten

- Konventionen zur Anordnung der Bytes
- bestimmt die Ablage der Daten im Speicher (falls > 1 Byte)
- Unterscheidung durch niedrigstwertiges Byte
 - „Big“: höchste Adresse (z.B. SPARC, PowerPC)
 - „Little“: niedrigste Adresse (z.B. Intel x86)
- Beispiel:



Die von dem Prozessor bevorzugte Bytereihenfolge kann in einem Befehl gelesen bzw. geschrieben werden. Die jeweils andere benötigt mehrere Befehle (Anpassung der Bytereihenfolge vor Verarbeitung).

Es sollte besonders bei der Datenübertragung zwischen unterschiedlicher Rechnersystemen oder bei dem Auslesen von Speicherinhalten (z.B. zur Fehlersuche) auf diese Konventionen geachtet werden.

Speicherausrichtung (Memory Alignment)

3. Speicherorganisation & Adressierungsarten

- Sicht der Programmierer auf den Speicher:



- Sicht des Rechners auf den Speicher:



- Word-Größe gibt i.d.R. „**Memory Access Granularity**“ vor

- Adresse a ist k -Bit ausgerichtet, wenn gilt:

- $a \bmod (k / 8) = 0$

- Adresse a ist s-Byte ausgerichtet, wenn gilt:

- $a \bmod s = 0$

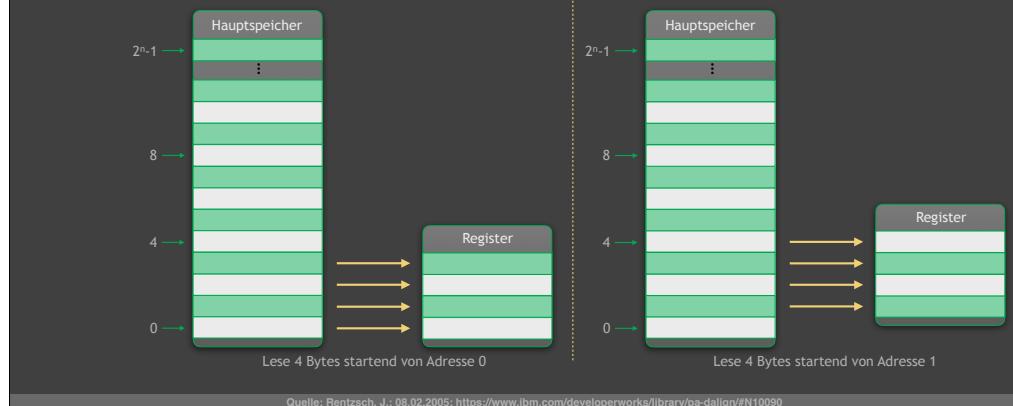
An dieser Stelle sei noch einmal angemerkt, dass die Problematik bei klassischen Computer deutlich stärker ausgeprägt ist als bei modernen. Bei den klassischen Computern kann der Speicher nur in Vielfachen der Word-Größe adressiert werden, der entsprechende Offset ist zu berücksichtigen. Manche moderne Computer können in der Lage sein den Speicher von einem individuellen Byte bis hin zu ihrer natürlichen Word-Größe zu adressieren, d.h. sie können auch mit „Misalignment“ (Unaligned Data) umgehen. Verfügen sie nur bedingt über Mechanismen um mit derartigen Daten umgehen zu können, so erfahren auch sie einen Performance-Verlust, da sie ggf. mehrere Words zum Erhalt der Information lesen und die verteilte Information zunächst zusammensetzen müssen.

In den meisten Fällen (und damit das Konzept zu verstehen) darf davon ausgegangen werden, dass der Prozessor mit seiner natürlichen Word-Größe auf den Speicher zugreift. Diese Word-Größe kann auch „Memory Access Granularity“ genannt werden, d.h. eine Word-Größe von einem Byte, kann als „Single-Byte Memory Access Granularity“ bezeichnet werden. Bei zwei Bytes kann von „Double-Byte Memory Access Granularity“ gesprochen werden, usw.

Speicherausrichtung (Memory Alignment)

3. Speicherorganisation & Adressierungsarten

- Single-Byte Memory Access Granularity

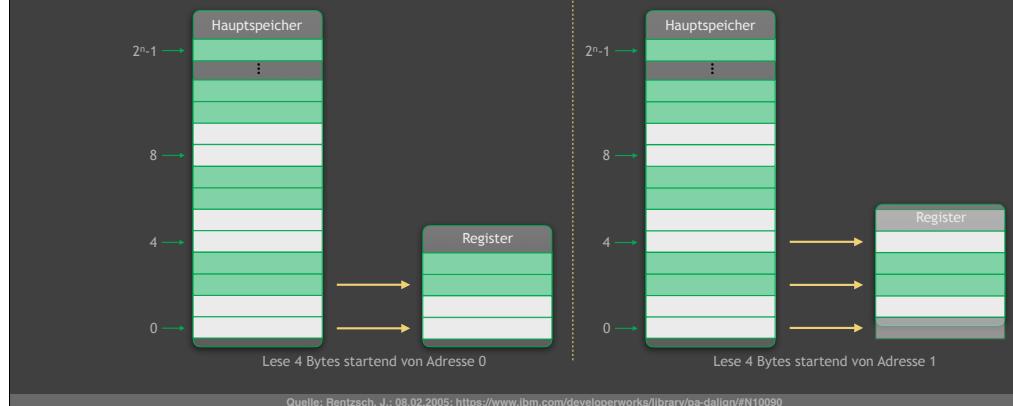


Vier Speicherzugriffe und im Prinzip genauso, wie der Programmierer sich den Speicher vorstellt.

Speicherausrichtung (Memory Alignment)

3. Speicherorganisation & Adressierungsarten

- Double-Byte Memory Access Granularity



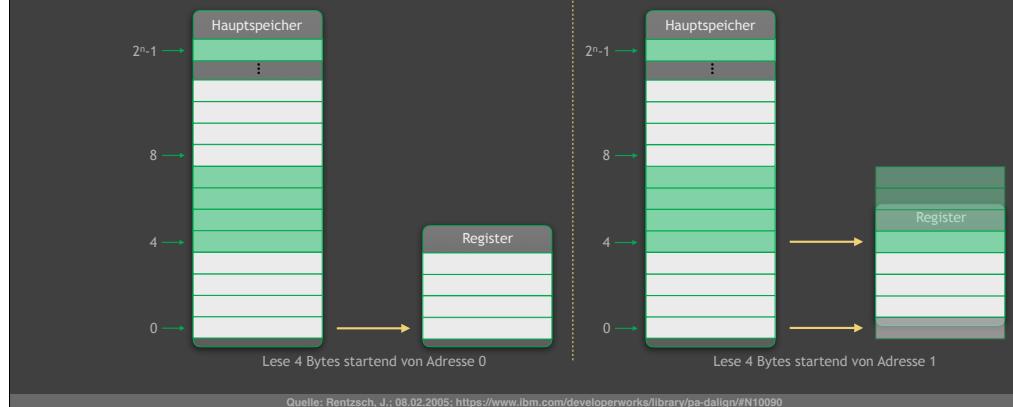
Jeder Speicherzugriff führt dazu, dass gleich zwei Bytes aus dem Speicher gelesen werden. Somit sind nur noch zwei Speicherzugriffe für die gleiche Anzahl an Bytes notwendig, was zu einer Performance-Steigerung führt.

Dennoch sollte darauf geachtet werden was passiert, wenn eine Adresse (hier 1) nicht auf die natürliche Word-Größe des Rechners passt. Hier hat der Rechner extra Arbeit zu tun, er muss einen weiteren Speicherzugriff vornehmen. Dies verlangsamt die Operationsverarbeitung. Derartige Adressen werden als „Unaligned“ bezeichnet.

Speicherausrichtung (Memory Alignment)

3. Speicherorganisation & Adressierungsarten

- Quad-Byte Memory Access Granularity

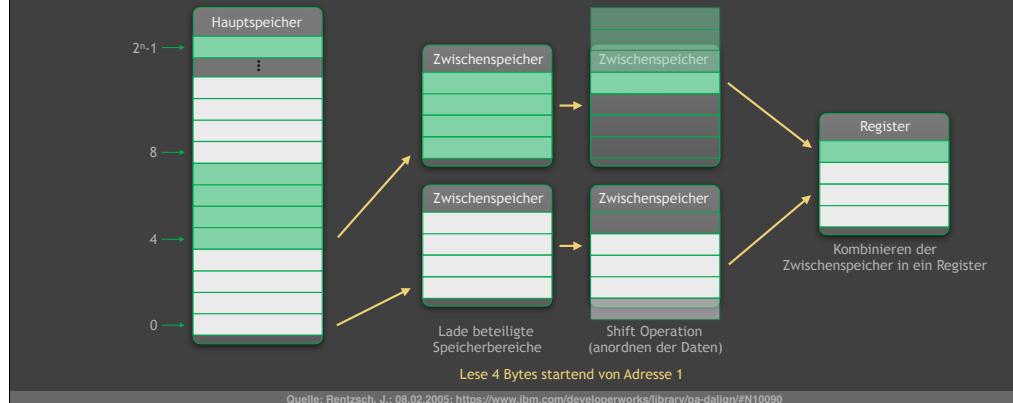


Ist die Speicheradresse „aligned“ können mit einem Speicherzugriff die geforderten vier Bytes gelesen werden. Bei einer „unaligned“ Speicheradresse, ist ein weiterer Speicherzugriff notwendig.

Speicherausrichtung (Memory Alignment)

3. Speicherorganisation & Adressierungsarten

- Korrektur bei einem „unaligned“ Speicherzugriff

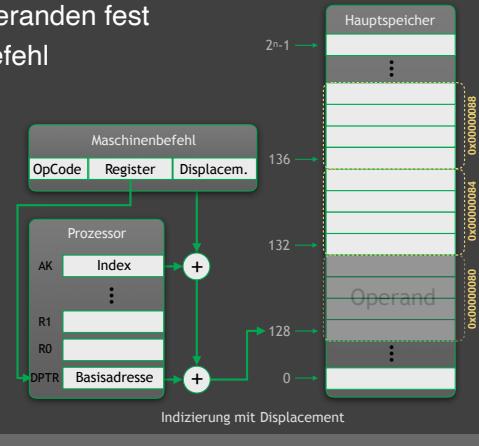


Welche Elemente der Rechner als Zwischenspeicher verwendet ist für das Beispiel egal, üblicherweise liegen Hardware-Implementierungen (Transistoren) vor. Zum Vorhalten der Daten (für das spätere Kombinieren) sind zwei ausreichend große Zwischenspeicher-Elemente notwendig. Manche Prozessoren verweigern den „unaligned“ Speicherzugriff ganz und werfen, bei der Aufforderung selbiges zu tun, eine Exception. Wird der „unaligned“ Speicherzugriff nicht unterstützt, so können die für den Sonderfall erforderlichen Transistoren anderweitig eingesetzt, um z.B. eine andere Einheit zu beschleunigen, oder aus dem Design gestrichen werden. Die Alignment-Korrektur in Software durchzuführen resultiert in geringerer Performance (im Vergleich zur Hardware-Lösung).

Adressierungsarten

3. Speicherorganisation & Adressierungsarten

- legen die Art des Zugriffs auf Operanden fest
- Information liegt im Maschinenbefehl
 - Wert des Operanden
 - Adresse des Operanden
- Unterscheidung in:
 - Immediate (Direktoperand)
 - Absolute, Direct
 - Register Direct
 - Register or Memory Indirect
 - Index



Dieser Abschnitt hat nicht den Anspruch vollständig zu sein. Die Adressierungsarten sind stark Prozessor-abhängig und lassen sich teilweise untereinander kombinieren. Hier sollen lediglich ein paar übliche Adressierungsarten aufgezeigt werden.

Immediate Mode (Direktoperand): (unmittelbare Adressierung)

Bei der Immediate Adressierung (Direktoperand) beinhaltet der Maschinenbefehl (im Programmspeicher) den Operanden als Konstante. Der Wert des Operanden ist unveränderlich. Es erfolgt kein weiterer Zugriff auf Register oder Speicher, der Operand wurde bereits mit dem Maschinenbefehl geladen. Wird in Assembler mit einer vorangestellten „#“ identifiziert.

Absolute, Direct: (absolute/direkte Adressierung)

Bei dieser Adressierung befindet sich die vollständige Speicheradresse direkt als Operand im Maschinenbefehl (und damit auch im Programmspeicher). Hierbei ist zu beachten, dass die Anzahl der Bits im Maschinenbefehl, welche für den Operanden reserviert sind, die Gesamtheit der möglichen Adressen bildet.

Register Direct: (Registeradressierung)

Der Operand ist ein Register des Prozessors, d.h. dessen Name (Nummer) ist im Maschinenbefehl gespeichert. Das Register beinhaltet wiederum den Operanden. Die Anzahl der genutzten Bits (im Maschinenbefehl) unterscheidet sich von Prozessor zu Prozessor und ist definiert durch die Anzahl der verfügbaren Register eines Rechners.

Register or Memory Indirect: ((Register-/Speicher-)indirekte Adressierung)

Der Inhalt des, im Maschinenbefehl definierten, Registers oder der Speicherzelle im Hauptspeicher wird als Speicheradresse des Operanden interpretiert. Das Register oder die Speicherzelle dienen somit als Pointer, welche auf den Operanden zeigen. In anderen Worten wird bei der Befehlausführung der Rechner erst auf eine spezielle Adresse verwiesen, dort findet er keinen Wert, sondern eine weitere Adresse (fetch operand address). Hier findet er den Operanden selbst (fetch operand value). Es sind also zwei Speicherzugriffe notwendig um den Wert des Operanden zu holen.

Index: (Indizierung)

Die Adresse des Operanden wird errechnet durch Bildung der Summe, bestehend aus einem Indexregisters (z.B. Akkumulator) und einer Basisadresse (DPTR). Letztere weist also auf die Speicherzelle bei der der erste Wert abgelegt ist. Der Index definiert den Offset, welcher zu dieser Adresse hinzugeaddiert wird um den benötigten Operanden zu adressieren. Diese Art der Adressierung ist gerade beim Zugriff auf Arrays oder zusammengesetzten Datentypen, welche in aufeinanderfolgenden Speicherbereichen abgelegt sind, sehr nützlich. Wird also der Wert des Indexregisters inkrementiert oder dekrementiert, so wird das nächste Element des Arrays adressiert.

Bei zusätzlichem Post-/Pre-increment oder -decrement wird der Inhalt des Adressregisters, abhängig von der Datenbreite (z.B. 1,2,4 oder 8) des zu adressierenden Operanden, verändert. Dies wird auch als Skalierung bezeichnet (+*1, +*2, +*4 oder +*8). Wird eine definierte Adressdistanz (Displacement) benötigt, so wird die Adresse aus dem Adressregister und dem Displacement berechnet. Der Displacement-Wert muss ebenfalls im Maschinenbefehl definiert sein.

Bei der relativen Indizierung wird der Befehlszähler (PC) als Basisadresse verwendet. D.h. die effektive Adresse wird relativ zum PC, also der des aktuellen Maschinenbefehls, errechnet. Wird diese Art der Adressierung gewählt, ist es möglich das gesamte Maschinenprogramm (inkl. Daten etc.) im Speicher zu verschieben. Die Veränderung des PC kann jedoch dafür sorgen, dass das Programm nicht mehr korrekt läuft.

Hinweise zur Adressrechnung:

1. Relativadressierung: Basis + Displacement
2. Indexadressierung: Basis + Index (+ Displacement)
3. Skalierte Adressierung: Basis + (Index * Skalierung) (+ Displacement)

Fortführende Adressierungsarten sind z.B. Segmentierung (Versteckte Basisadressen) oder Paging (Seitenadressierung). Die Segmentierung unterteilt den Speicher in eine Anzahl von Segmenten. Die Größe der Segmente kann dabei variabel sein. Die Adressierung innerhalb der Segmente startet immer wieder bei 0. Beim Paging wird der Maschinenadressraum in gleich große Kacheln („physical pages“) unterteilt (z.B. 512 Byte - 4 kByte). Die virtuelle Adresse besteht dann aus der Seiten- und Bytenummer. Bei Flash-Speichern sind oft ganze Seiten zu lesen, um dort die Information extrahieren zu können.

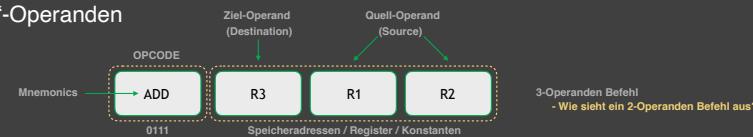
4

Maschinencode & Befehlsausführung

Maschinenbefehle

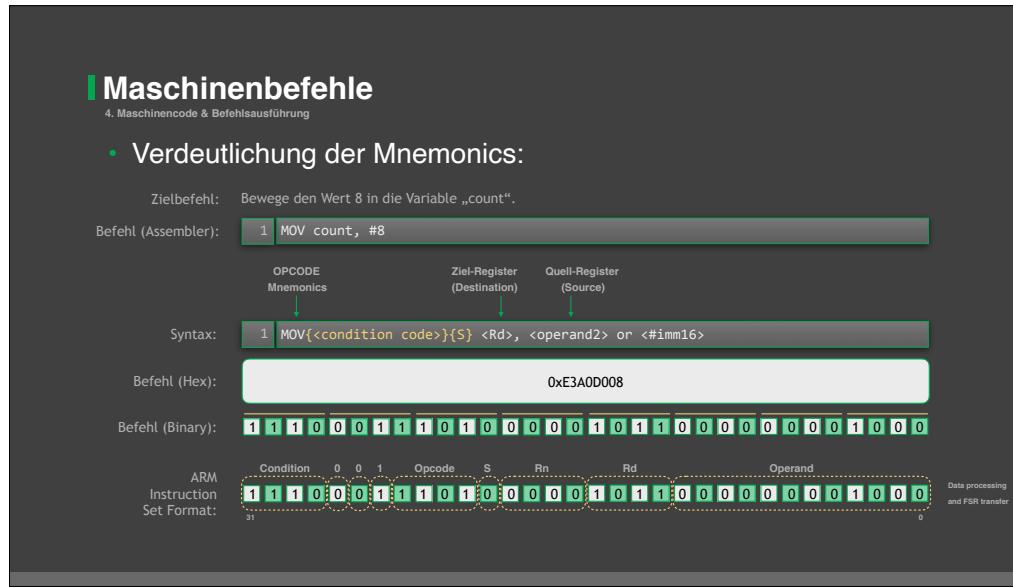
4. Maschinencode & Befehlausführung

- Befehl besteht aus:
 - OpCode (auszuführender Befehl)
 - Operanden (zur Befehlausführung benötigt)
- *Achtung:* Art der Operanden durch OpCode definiert!
 - „Adress“-Operanden
 - „Direkt“-Operanden



- *Beispiel:* Addiere den Inhalt von Register R1 zu dem Inhalt von R2 und speichere das Ergebnis in R3.

Egal ob Programmcode oder Daten, jeder Rechner kann nur mit binärem Code arbeiten. Nun ist es für uns Menschen sehr schwer unsere Anwendungen in binärer Form zu programmieren. Aus diesem Grund wird eine eindeutige symbolische Darstellung der Maschinenbefehle / Bitmuster gewählt. Symbolisch, weil die Bezeichnungen so gewählt werden, dass sie in einem „logischen“ Zusammenhang zu der Aktion auf der Hardware stehen. Da sie so im Gedächtnis bleiben sollen, werden die Symbole auch als Mnemonics (altgriechisch: „mnemonika“ = Gedächtnis) bezeichnet.



Erklärungen:

- S – Ist ein optionaler Suffix. Er ist gesetzt, wenn die Condition-Flags basierend auf dem Ergebnis der Operation aktualisiert werden sollen.
- condition code – Optionaler Code, der nur für die bedingte Code-Ausführung benötigt wird. Hierzu später mehr.
- #imm16 – Ein Wert im Bereich von 0-65535.

Erklärung des ARM Instruction Set Formats:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACCCCHGF.html>

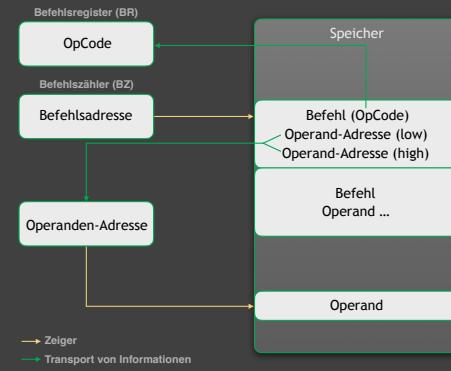
Zuordnung des MOV Befehls im ARM Instruction Set Format:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Cihcdbca.html>

Maschinenbefehle im Speicher

4. Maschinencode & Befehlausführung

- Annahme einer Von-Neumann-Architektur
- Speicher-Organisation
 - Zugriff über Adresse der Speicherzelle
 - fortlaufend nummerierte Speicherzellen
 - Aufeinanderfolgende Maschinenbefehle befinden sich in aufeinanderfolgenden Speicherzellen
 - Je nach Befehl kann anstelle einer Speicheradresse für einen Operanden auch ein direkter Wert stehen

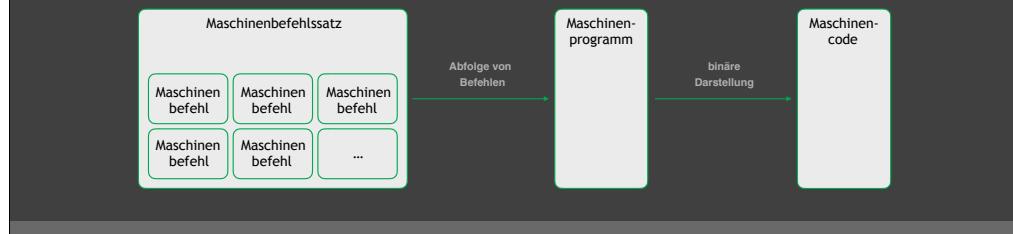


Bei allen folgenden Betrachtungen wird die Von-Neumann-Architektur zugrunde gelegt. Zur Wiederholung: Programmcode (Maschinencode) und Daten liegen im selben Speicher.

Maschinenbefehle und Begriffe

4. Maschinencode & Befehlausführung

- **Maschinen-/Computerprogramm** = Folge von Maschinenbefehlen
 - **Maschinenbefehl** = Bitmuster (OpCode), Elementaroperation, „Atomic“
- **Maschinenbefehlssatz** = Zusammenfassung aller Befehle
- **Maschinencode** = Maschinenprogramm in binärer Form
 - ausführbar von einem Rechner



Wird ein Maschinenprogramm mit Hilfe der Maschinenbefehle, welche als **Maschinensprache** oder **Maschinenbefehlssatz** zusammengefasst werden, erstellt ist dies Assemblerprogrammierung für das Zielsystem.

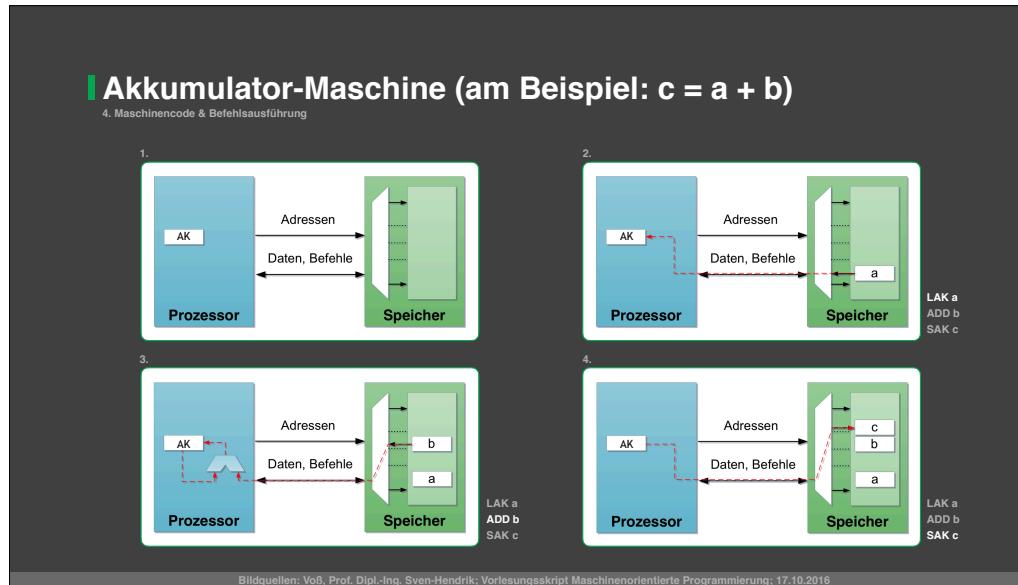
Der **Maschinenbefehlssatz** („*Instruction Set*“) ist für eine bestimmte Familie an CPUs spezifisch und damit nicht für alle, auf dem Markt existierenden, Systeme gültig. Wie z.B. bei ARM-Prozessoren kann es vorkommen, dass mehrere **Maschinenbefehlssätze** Verwendung finden. Weiter ist der **Maschinenbefehlssatz** meist in Gruppen aufgeteilt, d.h. z.B. Befehle die Daten bewegen und Befehle die arithmetische Operationen ausführen befinden sich in eigenen Gruppen.

Klassifizierung des Maschinenbefehlssatzes

4. Maschinencode & Befehlausführung

- auch als ISA (Instruction Set Architecture) bezeichnet
- ISA wird durch interne Architektur implementiert
- Klassifizierung durch:
 - Anzahl der Operationen (Befehlsvorrat)
 - Länge der Operanden
 - Typ der Operanden
 - Anzahl der Operanden
 - Zugriff auf die Operanden (Lage: Register, Stack, Memory)
- Klassifizierung in:
 - Akkumulator-Maschinen
 - Stack-Maschinen
 - Registersatz-Maschinen

Zuvor haben wir gelernt, dass die Befehlssatzarchitektur (Instruction Set Architecture (ISA)) die externe Architektur eines Rechners definiert. Dieser Maschinenbefehlssatz ist durch die interne Architektur in realen Schaltungen realisiert, also tatsächlich auf Siliziumebene.
Die Art und Weise wie auf die Operanden zugriffen wird beschreibt auch die prozessorinterne Speichernutzung und ist damit eine entscheidende Klassifizierungsmöglichkeit.

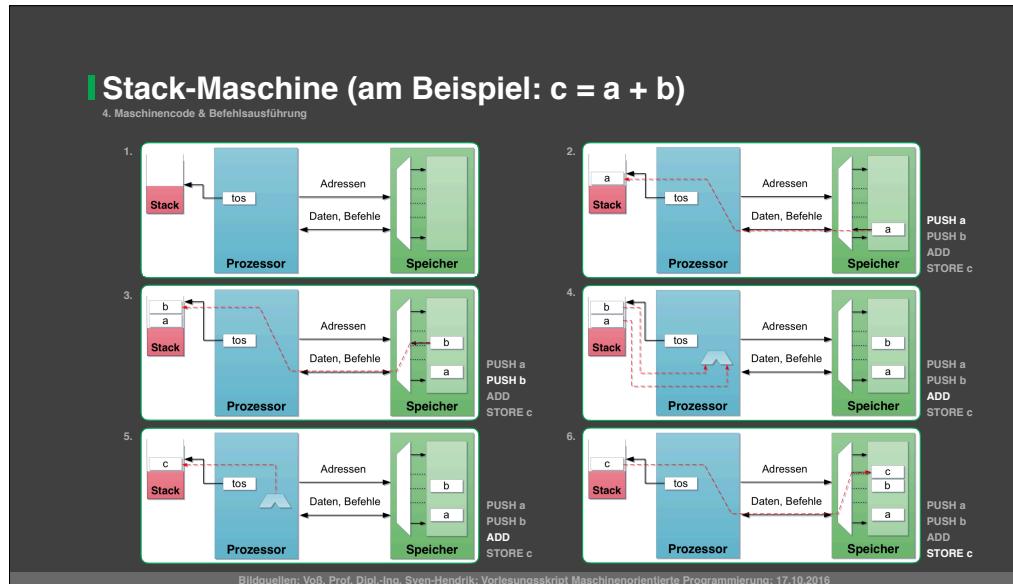


Die Akkumulator-Maschine zeichnet sich durch ein zentrales Register (dem Akkumulator-Register) aus, in welchem Rechenergebnisse „akkumuliert“ werden. Daher auch der Name. Wird also eine Rechenoperation ausgeführt, so bezieht sich diese immer nur auf einen Operanden und den Akkumulator. Diesem Vorgehen folgend muss der Befehl (dessen Codierung) bereits die Information enthalten, welches Register als Quelle oder Ziel dienen soll.

LAK $x \rightarrow$ Lade den Wert unter Adresse x in den Akkumulator

SAK $x \rightarrow$ Speichere den Wert des Akkumulators in der Speicherzelle unter der Adresse x

ADD $x \rightarrow$ Addiere den Wert aus Adresse x zu dem Akkumulator-Inhalt (d.h. das Ergebnis befindet sich im Akkumulator)



Der Zugriff auf die Operanden erfolgt immer nur vom Stackspeicher (oder „Keller“ oder „Stack“). D.h. der Prozessor weiß wie viele Operanden er je nach Befehl von dem Stapel holen muss und wie er das Ergebnis auf dem Stack abzulegen hat. Die Reihenfolge ist dabei implizit, dass letzte Element, welches auf den Stack gelegt wurde ist das Erste, welches auch den Stack wieder verlässt. Dies wird auch LIFO (Last-In-First-Out) genannt. Bei dem Vorgehen ist erkennbar, dass auch nur Objekte an oberster Position auf den Stack abgelegt werden können. Das Schreiben auf einen Stack wird mit „push“ bezeichnet, das Lesen (also das Entnehmen eines Elements) wird mit „pop“ bezeichnet. In dem gezeigten Beispiel impliziert der „STORE“ Befehl die „pop“ Operation.

Allgemein gilt:

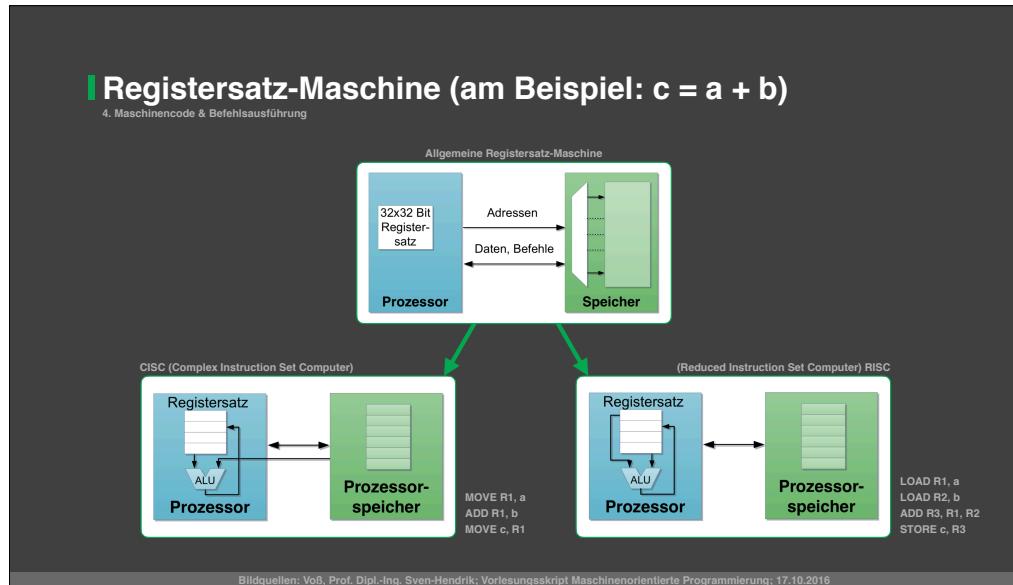
Ein Stack ist eine derart essentielle Datenstruktur, so dass sie in nahezu jedem CPU vorkommen. Bei einer reinen Stack-Maschine sind sie lediglich das Zentrum aller Operanden. Stacks werden generell als temporäre Speicher für Daten aber auch für Speicheradressen, welche bei Subroutinen-Aufrufen gespeichert werden müssen, verwendet. So ist es möglich von der Subroutine zurück zur aufrufenden Routine zu kommen. Dies ist besonders wichtig, wenn die Subroutine weitere Subroutinen aufruft („*nested subroutine calls*“). Ist die aufgerufene Subroutine eine „Blatt-Funktion“, d.h. sie ruft keine weitere Subroutine auf, so muss nicht zwingend der Stack verwendet werden. Hier kann auch das „LR (Link Register)“, welches die Return-Adresse zunächst beinhaltet, ausreichen.

Achtung!

Der Stack sollte bereits aus anderen Vorlesungen bekannt sein. Dennoch zur Wiederholung: Ein Stack ist ein dynamisches Konstrukt, welches überwacht werden muss. Der Stack ist nicht unendlich groß und droht irgendwann überzulaufen. Aus diesem Grund muss ein Mechanismus eingeführt werden, welches die Anzahl der Elemente im Stack überwacht. Hierzu existiert der sogenannte „stack pointer“, welcher immer auf das oberste Element des Stacks zeigt. In der Abbildung ist er als „Top-of-Stack“ (tos) dargestellt. Der Beginn des Stacks, also der Ort an dem er im Speicher beginnt, ist durch den „base pointer“ (ein Zeiger ist einfach eine Speicheradresse) definiert.

Man unterscheidet zwei Arten von Stacks:

„*ascending*“: Dieser Stack wächst nach oben, d.h. der Stack-Pointer wird inkrementiert.
„*descending*“: Dieser Stack wächst nach unten, d.h. der Stack-Pointer wird dekrementiert.



Bei der Registersatz-Architektur (Maschine) verfügt der Prozessor über interne Universalregister (und auch Spezialregister (hierzu später mehr)). Diese haben den Vorteil deutliche schnellere Zugriffszeiten zu ermöglichen und damit die Befehlausführung zu beschleunigen. Register sorgen ebenfalls dafür, dass die Codierung kompakter wird, da sie im Gegensatz zu Speicherzellen eine deutlich geringere Bit-Anzahl zur Adressierung benötigen. Ein Operand muss mindestens ein Prozessor-Register sein.

Unterkategorisierung in CISC und RISC:

- **CISC:** Sehr umfangreicher Befehlsatz mit entsprechend komplexer Hardware-Realisierung. Hier liegen dem High-Level-Programmierer sehr mächtige Maschinenbefehle (Mikroprogramme) vor, wodurch kürzere Programme, im Vergleich zu RISC, entstehen können. Diese Mikroprogramme agieren jedoch langsamer und brauchen bis zu zehn Takte zur Ausführung. Diese Art verfügt ebenfalls über komplexere Adressierungsmöglichkeiten, was die Ausführungs geschwindigkeit ebenfalls bremst. D.h. auch der Decoder muss einen Befehl „länger“ Interpretieren um den exakten Befehl selbst, die Adressierungsart, die Adressen und die Register feststellen zu können.
- **RISC:** Beschränkt sich auf die wirklich notwendigen Befehle, wodurch ein einfacheres Design und ein preis-optimiertes System entwickelt werden kann. Die Befehle sind vorwiegend Einzyklusbefehle, d.h. sie liegen in festverdrahteter Form auf dem Chip vor. Aufgrund ihrer Einfachheit sind diese Rechner leichter zu verifizieren. Auch der Decoder erhält ein vereinheitlichtes Datenformat, d.h. alle Befehle haben eine einheitliche Länge, was diesem ein sehr schnelles Dekodieren ermöglicht. Diese Systeme erzielen einen extrem hohen Datendurchsatz und können z.B. durch Pipelining (später mehr dazu), aufgrund der einheitlichen Befehllänge, weiter optimiert werden. Ebenso ist die RISC Architektur eine Load-Store-Architektur, was bedeutet, dass entsprechende Befehle für den Datentransport zwischen CPU und Speicher bereitgestellt werden. Jedoch können arithmetische Befehle nur auf Registern arbeiten.

Es ist wichtig zu verstehen, dass somit der Compiler bei dem RISC System in der Lage sein muss komplexere Programmteile für den Prozessor vorzubereiten. Bei CISC muss hingegen der Prozessor in der Lage sein komplexer Code zu verstehen, der Compiler kann „einfacher“ ausfallen.

In vielen Fällen läuft die Ausführung einzelner elementarer Befehle deutlich schneller ab, als ein komplexer Befehl mit der selben Wirkung. Dennoch kann nicht gesagt werden welcher Architektur-Ansatz letztendlich der bessere ist. In modernen Prozessoren findet man sogar Kombinationen aus beiden.

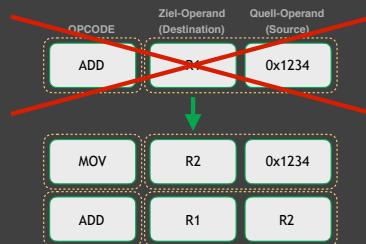
Wie ist die Ausführungszeit definiert?

Ausführungszeit = Anzahl Befehle * Anzahl Zyklen * deren Ausführungszeit

Load-Store-Architekturen

4. Maschinencode & Befehlausführung

- Datentransport zwischen Hauptspeicher und CPU über
 - dedizierte Ladebefehle (LOAD)
 - dedizierte Speicherbefehle (STORE)
 - oder z.B. das Bewegen von Speicherinhalten (MOVE)
- andere Befehle arbeiten ausschließlich auf Registern

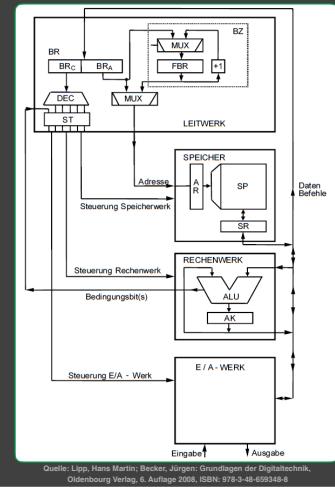


D.h. der Programmcode wird offensichtlich durch diese zusätzlichen Befehle länger. Der Vorteil ist jedoch, dass die Hauptspeicherzugriffe von den internen CPU-Vorgängen abgekoppelt werden, wodurch die Verarbeitungsgeschwindigkeit steigen kann, da die interne Wartezeit auf diese Operanden entfällt. Dies ist eine wichtige Voraussetzung für das im späteren Verlauf folgende Konzept des Pipelinings.

Befehlsausführung

4. Maschinencode & Befehlausführung

- Schema: Einfacher Rechner
- Steuerwerk (Leitwerk)
 - holt & interpretiert Befehle → elektrische Signale
 - steuert ALU & Datentransport
- Rechenwerk
 - Durchführung der Arithmetik
 - Berechnung der Bedingungsflags (Conditionflags)



Symbolik:

- AK → Akkumulator
- ALU → Arithmetic Logic Unit
- AR → Addressregister (Speicheradressregister oder Memory Adress Register (MAR); gibt die Anzahl der maximal adressierbaren Speicherstellen vor; gibt Breite des Adressbusses vor)
- BR → Befehlsregister (Instruction Register (IR))
- BRc → Codeteil von BR
- BRa → Adressteil von BR
- BZ → Befehlszähler (Programmzähler oder Program Counter (PC))
- DEC → Befehlsdecoder
- FBR → Frame Base Register
- MUX → Multiplexer
- SP → Speicher
- SR → Speicher-Register (Datenpuffer oder Memory Buffer Register (MBR); gibt die Tiefe, also die maximale Länge (in Bit) der Speicherinhalte vor; gibt Breite des Datenbusses vor)
- ST → Steuerbits-Register

Die Programmbefehle werden von dem Steuerwerk (Leitwerk) aus dem Speicher, via ihrer Speicherzellen-Adresse, geholt und interpretiert (Befehlsdecoder). Die dabei entstehenden elektrischen Signale steuern das Rechenwerk (ALU) und den Datentransport (inkl. Speicherwerk & E/A-Werk). Neben dem Rechenwerk, und der dort berechneten Bedingungsbits (z.B. Overflow-Flag, Carry, Zero, Sign) (Conditionflags oder auch Zustandsanzeige der ALU), sind das Befehlsregister und der Befehlszähler für den Instruktionszyklus von Bedeutung. Das Befehlsregister ist für die Speicherung des aktuellen Befehls zuständig. Der Befehlszähler dient der Bestimmung der Adresse des aktuellen Befehls. Hier ist bereits durch die Abbildung angegedeutet, dass durch ein Fortsetzungstoken der Programmdurchlauf realisiert wird. Ein Fortsetzungstoken (Continuations) ist ein Tupel (BZ, FBR) bestehend aus einem Befehlszähler (BZ) und einer Rahmenbasisadresse (FBR). Wichtig ist die Zustandstransformation (z.B. +1; demnach inkrementiert oder in Abhängigkeit von der Befehslänge erhöht) sobald ein Befehl ausgeführt wurde, wodurch das Fortsetzungstoken auf die nächste Speicherzelle, also auf den nächsten Befehl, zeigt.

Auf den Instruktionszyklus und die fundamentalen Register wird später eingegangen. Hier ist es erst einmal wichtig das prinzipielle Schema gesehen zu haben.

Um noch einmal den Kreis zu den vorhergehenden Folien zu schließen. Der Befehlszähler ist also maßgeblich für den Adressbus. Hingegen ist das Befehlsregister (inkl. Befehlsdekomposition, Timing und Steuerung) für den Steuerbus zuständig. Register, ALU, Speicher, usw. sind wie bereits besprochen über den Datenbus verknüpft.

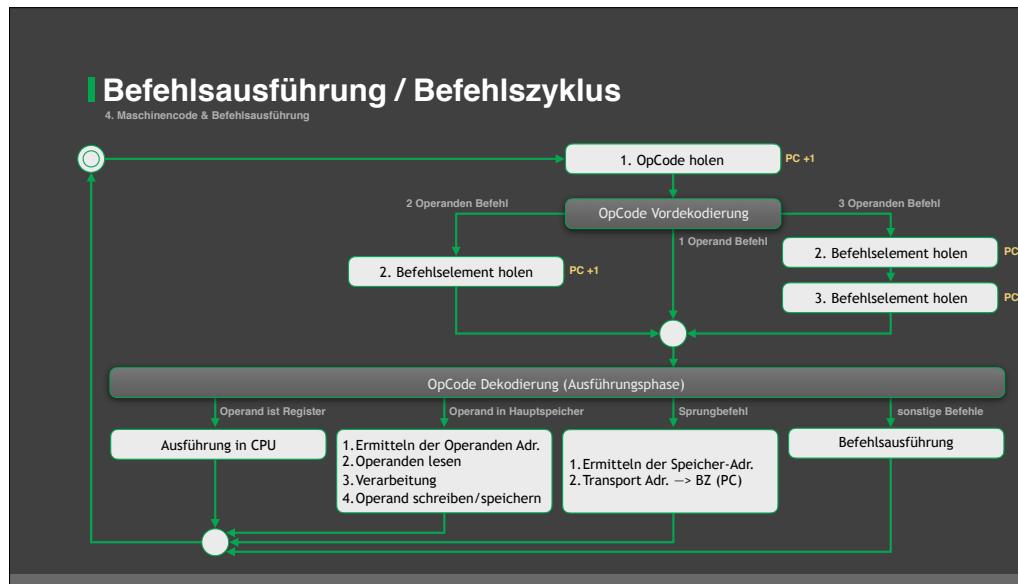
Befehlsausführung / Befehlszyklus

4. Maschinencode & Befehlsausführung

- spezielle Register im Steuerwerk
 - PC (Program Counter, Befehlszähler (BZ), Instruction Pointer (IP))
 - IR (Instruction Register, Befehlsregister (BR))
 - OR (Operand Register, Speicheradressregister (AR))
 - AC (Accumulator, Arbeitsregister/Akkumulator (AK))
- sequentielle Befehlsfolge
 - Befehle liegen hintereinander im Speicher —> bilden das Programm
- Mikroprozessoren folgen festem Schema: (z.B.)
 1. Befehlsholphase (instruction fetch, IF)
 2. Befehlsdekodierphase (instruction decode, ID)
 3. Operandenholphase (operand fetch, OF)
 4. Ausführungsphase (instruction execute, EX)

Die Befehlsabarbeitung, welche durch das Steuerwerk verwaltet wird, erfolgt über spezielle Register, die in Abhängigkeit von der aktuellen Befehlsverarbeitungsphase, Inhalte speichern und darauf basierend Aktionen anstoßen.

Der Befehlszyklus folgt genau der im Speicher abgelegten Befehlsfolge (Programm) und ist streng seriell, d.h. an dieser Stelle findet keine Parallelisierung in der Verarbeitung statt. Für die folgenden Betrachtungen wird ein Von-Neumann-Rechner zugrundegelegt und das in den Folien beschriebene Schema.



1. Befehlsholphase (IF):

Der Befehlszähler beinhaltet die Information an welcher Speicherstelle sich der Rechner zur Zeit im Programmcode befindet. Er zeigt auf die Speicheradresse auf den nächsten auszuführenden Befehl, welcher auf Basis dieser Information in das Befehlsregister geladen wird.

2. Befehlsdekodierphase (ID):

Der sich im Befehlsregister befindende Befehl wird decodiert bzw. interpretiert. Auf Basis dieser Interpretation werden weitere Aktionen eingeleitet. Hier geht auch hervor wieviele Operanden benötigt werden und wie diese zu adressieren sind.

3. Operandenholphase (OF):

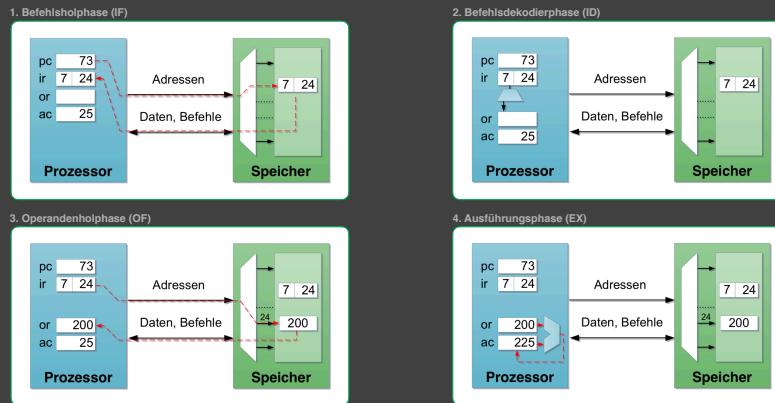
Die Operanden (ggf. mehrere) sind mit Hilfe des Speicheradressregister zu übertragen und abzulegen. Hier ist entscheidend ob ein Operand in Form eines Registers vorliegt oder ob er aus dem Hauptspeicher geladen werden muss. Muss der Operand aus dem Hauptspeicher geladen werden, so muss dessen Adresse zuvor interpretiert werden.

4. Ausführungsphase (EX):

Nachdem alle Informationen für die Ausführung des Befehls vorliegen, kann die Verarbeitung der OpCodes, welchen sich im internen Befehlsregister befindet, beginnen. Des Weiteren muss der Befehlszähler inkrementiert werden um auf die nächste Programmcode-Speicheradresse zu zeigen. Dies kann als separater Schritt erfolgen oder parallel. Dies ist in orangener Farbe ange deutet, wichtig ist, dass zum nächsten Befehlszyklus der Befehlszähler auf die korrekte Adresse zeigt.

Befehlsausführung / Befehlszyklus

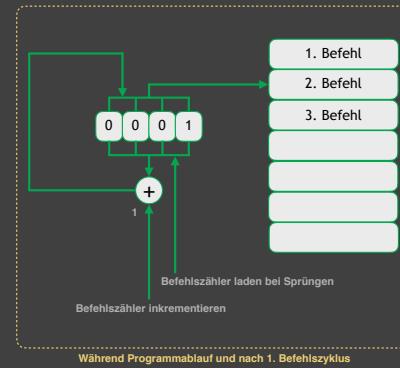
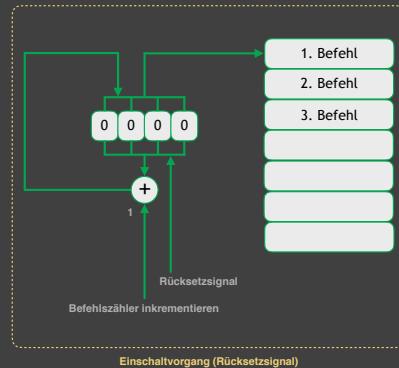
4. Maschinencode & Befehlausführung



Bildquellen: Vo8, Prof. Dipl.-Ing. Sven-Hendrik; Vorlesungsskript Maschinenorientierte Programmierung; 17.10.2016

Befehlszähler

4. Maschinencode & Befehlausführung



Aufgabe (ca. 30 Minuten)

4. Maschinencode & Befehlausführung

- Welche Maschine (Akkumulator / Stack / Register-Maschine (RISC/CISC)) liegt vor?
- Welche x-Bit-Architektur liegt vor?

- Bitte findet die Antworten für folgende Mikrocontroller
 - 80c535 (Siemens, 1995)
 - PIC16F8X (Microchip, 1998)
 - Cortex-M0 (ST Microelectronics, 2015)
- Bitte die 30 Minuten sinnvoll im Datenblatt nutzen!

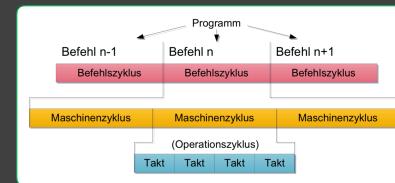
5

Ausführungszeit &
Optimierung

Befehls- vs. Maschinen- vs. Taktzyklus

5. Ausführungszeit & Optimierung

- Befehlszyklus (Instruction Cycle)
 - wird von der CPU ständig durchlaufen (z. Wdh. der Stages: IF, ID, OF, EX)
 - unterteilbar in einzelne Maschinenzyklen
- Maschinenzyklen (Machine Cycle)
 - variieren je nach Befehl
 - typische Maschinenzyklen:
 - Befehlsaufruf (OpCode-Fetch)
 - Memory Read / Write
 - Stack Pop / Push
 - Input / Output
 - Interrupt / Halt
 - unterteilbar in Taktzyklen



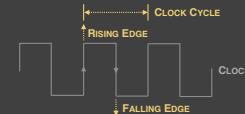
Bildquellen: VoB, Prof. Dipl.-Ing. Sven-Hendrik; Vorlesungsskript Maschinenorientierte Programmierung; 17.10.2016

D.h. wenn die Maschinenzyklen je nach Befehl unterschiedlich sind / sein können, dann ergibt sich auch eine unterschiedliche Laufzeit je Befehl.

Befehls- vs. Maschinen- vs. Taktzyklus

5. Ausführungszeit & Optimierung

- **Kernelement:** Systemtakt der CPU
 - Abläufe im CPU synchronisiert über Takt (Clock)
 - Ausführungszeit abhängig von Taktfrequenz (Clock Frequency) in Hertz (Hz)
 - Achtung: keine 1:1 Abhängigkeit; Hardware benötigt Zeit → Clock nicht beliebig wählbar!
- **Taktzyklus (Clock Cycle)**
 - definiert durch jeden Puls der Referenzquelle (z.B. Quarz)
 - Zeit zwischen den Impulsen (Clock Ticks)
 - Taktzyklus = 1 / Taktfrequenz
 - Zeit für CPU spezifische Arbeiten zu erledigen
 - bevor der nächste „Status/Daten-Snapshot“ erzeugt wird (D-Flip-Flops)
 - bestimmt Ausführungszeit eines Befehlszyklus



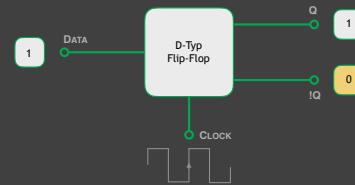
Die CPU basiert ihren Ablauf auf einem zentralen Systemtakt, welcher von einer Referenzquellen (z.B. einem Quarz) erzeugt bzw. abgeleitet wird. Die Taktpериode ist im Vergleich von konstanter Länge. D.h. auch die Ausführungszeit eines Befehlszyklus wird durch die Taktfrequenz der CPU bestimmt.

Achtung: Die Geschwindigkeit des Taktes zu erhöhen mag als eine Maßnahme zur Performance-Steigerung erscheinen. Sie ist jedoch limitiert durch die Zeit, welche ein Signal benötigt um den längsten Pfad durch alle notwendigen Logiken zu durchlaufen.

Einschub: D-Typ Flip-Flop (Wiederholung)

5. Ausführungszeit & Optimierung

- ideales Status-Speicherelement
 - für sequentiell arbeitende Schaltungen (z.B. Computer)
 - Möglichkeit zur Synchronisation der Schaltungselemente
- Snapshot des Datenwertes abhängig vom Takt
 - speichert bei LOW nach HIGH Transiente
 - gibt Datenwert auf Q-Ausgang wieder



Befehls- vs. Maschinen- vs. Taktzyklus

5. Ausführungszeit & Optimierung

- Fortsetzung: Taktzyklus (Clock Cycle)
- Ausführungszeit eines einzelnen Maschinenzyklus
 - in älteren CPUs: 4 bis 40 Taktzyklen
 - in aktuellen CPUs: 1 bis N Taktzyklen
 - unterschiedliche Befehle <> unterschiedliche Anzahl an Taktzyklen <> unterschiedliche Laufzeiten



- Messgröße: Cycles per Instruction (CPI)

Daher kann die Anzahl der „Taktzyklen pro Befehl“ als Messgröße für die Effizienz einer Architektur herangezogen werden. Je kleiner der Wert ist, desto performanter arbeitet die Architektur.

Ausführungszeit: Systemtakt

5. Ausführungszeit & Optimierung

- Woher kommt die unterschiedliche Anzahl an Taktzyklen?
 - (Früher) Maschinenbefehle als Sequenz von Mikroinstruktionen
 - Mikroinstruktionen = „Mini-Steps“ zur Durchführung komplexerer Befehle
 - nicht von „außerhalb“ der CPU erreichbar
 - gemeinsam genutzte digitale Logik —> spart Platz auf dem Chip
 - Mikrocode = Auflistung der notwendigen Mikroinstruktionen zur Ausführung eines Maschinenbefehls
- Heute:
 - mehr Transistoren finden Platz auf einem Chip
 - dedizierte Logiken für einzelne Maschinenbefehle
 - seltener Verwendung von Mikrocode
 - schnellere Ausführung (1 Maschinenbefehl pro Taktzyklus)

Beschleunigung der Programmausführung

5. Ausführungszeit & Optimierung

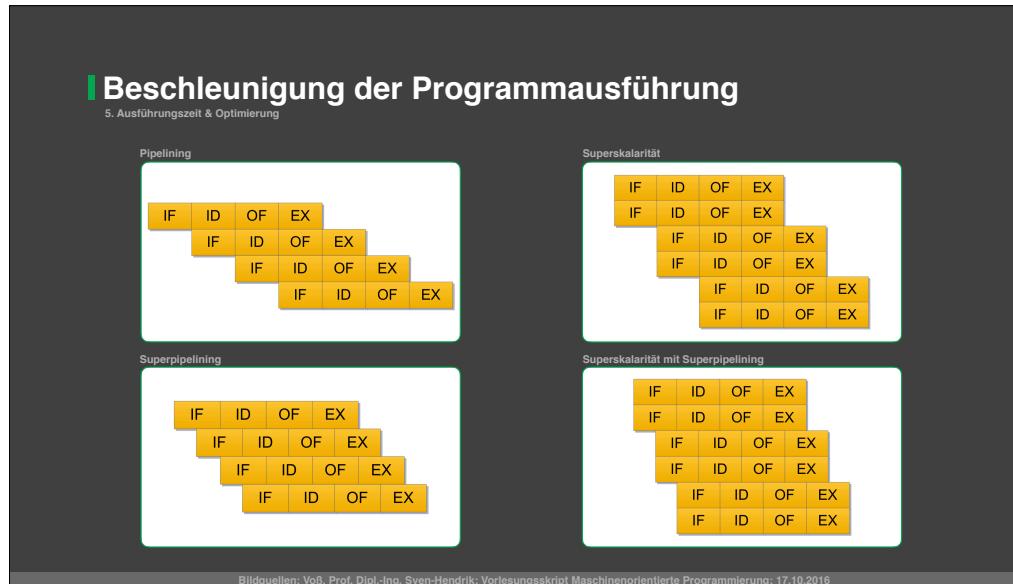
- Ist-Stand:
 

Ausführung ohne Optimierung
- Aufspaltung der Befehlsausführung
 - spezifische Reihenfolge der Stages
 - separate Aktionen / funktionale Einheiten je Stage
 - feste / deterministische Anzahl an Taktzyklus pro Stage
 - z.B. je Stage 1 clock cycle
 - funktionale Einheiten mehrfach pro Zyklus verwendbar
 - keine Hardware-Duplizierung erforderlich
 - ermöglicht „parallelisieren“

Bildquellen: VoB, Prof. Dipl.-Ing. Sven-Hendrik; Vorlesungsskript Maschinenorientierte Programmierung; 17.10.2016

Wenn ein Maschinenbefehl in einem einzelnen Taktzyklus (Single Clock-Cycle) ausgeführt wird, bedeutet dies, dass die einzelnen Stages (z.B. zur WdhL, IF, ID, OF, EX) in einer Welle von Transistor-Aktivitäten aktiv werden. Diese Welle propagiert durch die CPU von der Logik welche für den IF zuständig ist, bis hin zur Logik welche die Ausführung (EX) vornimmt. Um diese Welle nun möglichst schnell durch die CPU zu senden, muss die Zeit betrachtet werden, welche das Signal auf seinem längsten Weg durch die ganzen Logik-Elemente benötigt. Diese Zeit ist ebenfalls die maximale Taktgeschwindigkeit mit welcher das System sicher betrieben werden kann.

Diese Aufspaltung der Befehlsausführung klingt zunächst Paradox. Bei vier Stages werden vier Taktzyklen, anstelle von einem Taktzyklus benötigt, um einen Befehl auszuführen. Jedoch kann die Taktfrequenz erhöht werden, da die Anzahl der zu durchlaufenden Logik-Elemente pro Stage geringer ist. Obwohl also ein einzelne Befehlsausführung nun vier Taktzyklen benötigt, wird zu jedem Taktzyklus ein neuer Befehl gestartet und gleichzeitig eine bereits begonnene Befehlsausführung abgeschlossen. D.h. auch hier werden Befehle in jedem Taktzyklus abgearbeitet, nur jetzt mit einer höheren Taktfrequenz.



Das Pipelining (Implementierungstechnik) ermöglicht das parallele Verarbeiten bzw. die Aktivität aller Phasen / Stages. Die zeitliche Überlappung wird jedoch nicht direkt nach dem Einschalten erzielt. Zuerst muss sich die Pipeline füllen. Durch diesen erhöhten Durchsatz wird eine höhere Verarbeitungsgeschwindigkeit erzielt. Je Zeiteinheit werden somit mehr Befehlszyklusphasen abgearbeitet. Im Idealfall haben wir einen CPI von 1.

Die Superskalarität erzielt ihre Leistungsverbesserung durch parallele Funktionsseinheiten, welche in der selben Befehlszyklusphase aktiv sind. Dies ermöglicht auch bei zusätzlicher Verwendung von Pipelining eine Leistungssteigerung. Der entscheidende Vorteil zum reinen Pipelining ist, dass mehrere Instruktionen zur Verarbeitungseinheit übergeben und abgearbeitet werden können. Die Abarbeitung von einer Instruktion gleichzeitig wird „skalar“ genannt, mehrere entsprechend „superskalar“. CPI kann $1/n$ (n : Anzahl parallelen Befehlen) erreichen, im Idealfall.

Das Superpipelining erfordert eine andere Überlegung. Die Zuordnung, dass jede Phase exakt einen Taktzyklus andauert und die Verarbeitungseinheit für diese Zeit nicht zur Verfügung steht, muss überdacht werden. Wird eine zeitliche Überlappung der Befehlsabarbeitung angenommen und die Befehlspipeline in mehrere Stufen unterteilt, entsteht eine feinere Unterteilung des Fließbandes. Somit sind kürzere Durchlaufzeiten realisierbar und durch die Erhöhung der Stufen können schnellere Takte gewählt werden.

Dieses Konzept lässt sich auch mit der Superskalarität verbinden um eine weitere Beschleunigung zu erzielen.

Die Voraussetzungen sind offensichtlich:

- jede Phase dauert gleich lang (Ausführungszeit)
- alle Befehle sind in allen Phasen vertreten
- gleichlange Befehle (Ausführungsdauer)
- Pipeline ist immer voll (Leerlauf vermeiden)

Bei der Entwicklung der Pipeline ist die langsamste Stufe zu betrachten. Eine längere Pipeline hat den Nachteil, dass das Füllen und Leeren der Pipeline länger dauert. Was passiert bei einer Sprunganweisung? (Abbruch (engl. stall)). Es ist wie überall, es gilt einen guten Kompromiss zu finden.

Bei CISC-Prozessoren ist das Pipelining durch unterschiedlich komplexe (und lange) Befehle sehr aufwändig. Dies kann zu einer schlechten Pipeline-Auslastung führen, welche in keinem Performance-Gewinn endet.
Bei RISC-Prozessoren ist dies wegen dem einheitlichen Befehlssatz gut anwendbar.

Es kann, bei ungleichmäßiger Füllung, zu Konflikten (Hazards), kommen. Man unterscheidet in:

- Structural Hazard: dieselbe Hardware soll für verschiedene Zwecke eingesetzt werden (Zugriff auf Speicher, Rechenwerk)
- Data Hazard: Abhängigkeit zweier Instruktionen, wobei sich eine noch in der Pipeline befindet.
- Control Hazard: Zeitversatz bei Instruction-Fetch und darauf basierenden Entscheidungen zum Kontrollfluss (z.B. Sprünge) / Hinweis: Dies passiert, da Sprungbefehle den Befehlszähler verändern und damit die IF-Phase beeinflussen.

Konfliktlösungen werden in modernen Prozessarchitekturen immer wichtiger. Leider geht hiermit durch entstehende Verzögerungen eine Reduzierung der Performance einher.

Aufgabe (ca. 30 Minuten)

5. Ausführungszeit & Optimierung

- Wieviele Zyklen werden für die Befehle benötigt?
 - Wieviele Instruktionen gibt es?
-
- Bitte findet die Antworten für folgende Mikrocontroller
 - 80c535 (Siemens, 1995)
 - PIC16F8X (Microchip, 1998)
 - Cortex-M0 (ST Microelectronics, 2015)
 - Bitte die 30 Minuten sinnvoll im Datenblatt nutzen!

6

Polling & Interrupts

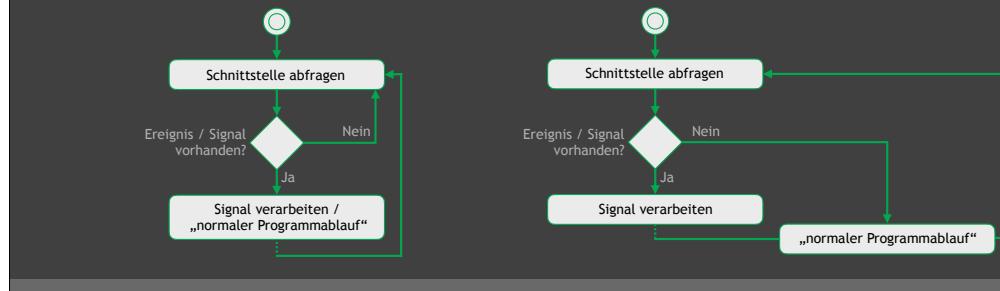
I Polling

6. Polling & Interrupts

- aktives Warten auf ein Ereignis
 - Schnittstelle wird laufend von dem Programm abgefragt
 - passive Schnittstelle
 - blockierendes vs. nicht blockierendes Polling

vs.

nicht blockierendes Polling



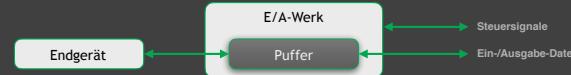
Nachteile:

- Polling kostet Ressourcen.
 - Es können Abbruchmechanismen (z.B. Timeouts) notwendig werden, falls die Schnittstelle nicht mehr antwortet.

Interrupt gesteuerte I/O

6. Polling & Interrupts

- Probleme mit I/O-Geräten:
 - Die CPU kontrolliert den Zugang zum Speicher und kann, in dem Fall, dass ein I/O-Gerät übertragen will, beschäftigt sein.
 - Überträgt das I/O-Gerät nur geringfügig langsamer als die CPU kann dies die CPU unnötig blockieren.
- Lösung → Interrupt:
 - bei Eintreten eines definierten Ereignisses
 - senden eines Unterbrechungssignals (Interrupt) zur CPU
 - kurzzeitige Unterbrechung des ausgeführten Programms zur Reaktion auf das Ereignis



I/O = Input / Output = Eingabe / Ausgabe = E/A

Beispiel: (mit Controller meinen wir das E/A-Werk)

1. Das Endgerät ist bereit zur Übertragung von Daten in den Rechner und sendet daher ein Interrupt-Signal an die CPU.
2. Die CPU überprüft den Controller-Status und sendet ein Start-Signal. Danach kann sie direkt ihre vorherige Tätigkeit fortsetzen.
3. Der Controller empfängt Daten vom Endgerät und speichert diese in seinem Puffer. Sobald der Puffer voll oder die Datenübertragung vom Endgerät beendet ist, sendet der Controller einen weiteren Interrupt an die CPU.
4. Die CPU unterbricht das gerade ablaufende Programm erneut und führt die Übertragung der Daten aus dem Controller-Puffer in den Speicher durch. Anschließend setzt sie die Ausführung des unterbrochenen Programms fort.

Optimierungen des Datentransfers

6. Polling & Interrupts

- Datentransfer via DMA-Controller
 - gerne in Mikrocontrollern verwendet
 - DMA-Controller tauscht mit CPU lediglich Steuerinformationen aus
 - Datentransport zwischen Controller und Speicher erfolgt eigenständig
 - bedienen nicht nur simple E/A-Einheiten
 - Kommunikationsbusse wie z.B. UART / SPI / I²C
 - intra-Chip Übertragungen (Multi-Core Prozessoren)
 - Memory-to-Memory Datentransfer
 - entlastet die CPU
- „*Cycle-Stealing*“ Prinzip
 - CPU wird der Speicherzugriff für einige Takte entzogen
 - im Konfliktfall hat der DMA-Controller vor der CPU den Vorrang

DMA = Direct Memory Access

Heute sind auch Burst-Mode und Transparent Modes zusätzlich zum Grundprinzip des Cycle-Stealing in DMA-Controllern implementiert.

Der Burst-Mode dient der Übertragung einer großen Datenmenge. Erlaubt der CPU dem DMA den Zugriff, erhält die CPU erst dann den Zugriff wieder zurück sobald alle Daten übertragen wurden. Dies wird auch als „Block Transfer Mode“ bezeichnet. Für die größere Datenmenge wird so der Austausch von Steuerinformationen stark reduziert.

Der Transparent-Mode ist hingegen die effizienteste Methode, wenn auch die langsamste. Hier wird nur während den Phasen, in denen ein CPU Operationen ausführt und nicht auf den Speicher angewiesen ist, auf den Speicher zugegriffen. Zugleich ist dieser Modus sehr komplex zu implementieren, da hier die Hardware rausfinden muss, wann der CPU den Speicher nicht benötigt.

Interruptklassen und -Priorität

6. Polling & Interrupts

- Interruptklassen:
 - externer Interrupt
 - außerhalb der CPU
 - interner Interrupt
 - innerhalb der CPU
 - maskierter Interrupt
 - Interrupt kann vorübergehend außer Kraft gesetzt werden
 - unmaskierter Interrupt
 - sofortiger Interrupt
- Interrupt-Prioritäten
- Bestimmung der Interrupt-Behandlung über **Interrupt-Handler**

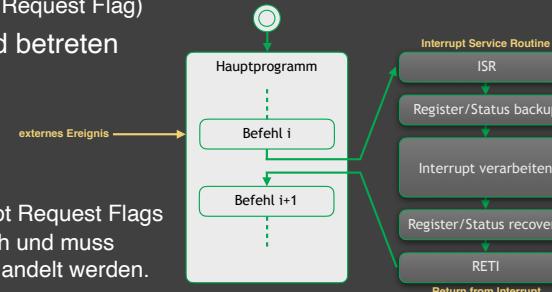
Wenn während eines Interrupts ein weiterer auftritt, wird dieser nur bearbeitet, wenn er eine höhere Priorität hat, sonst muss dieser Interrupt warten, bis der vorherige bearbeitet ist. Eine Maskierung erlaubt die Sperrung von Interrupts ab einer bestimmten Priorität.

Hinweis: In vielen Rechnern gibt es Register (Special Function Register (SFR) oder Interrupt Nable Register), die das Aktivieren und Deaktivieren einzelner, aber auch aller Interrupts ermöglichen.

Interrupt Service Routine (ISR)

6. Polling & Interrupts

- externes Ereignis unterbricht Programmausführung
- Ereigniserkennung
 - durch aktiven Steuereingang
 - erzeugt Signal: Flag (Interrupt Request Flag)
- Unterprogramm (ISR) wird betreten
 - dem Ereignis zugeordnet
 - Reaktion auf Ereignis
- **Achtung:**
 - Das Zurücksetzen des Interrupt Request Flags erfolgt nicht immer automatisch und muss daher ebenfalls in der ISR behandelt werden.



Hinweis: Da die Interrupt-Bearbeitung durch ein bestimmtes Flag ausgelöst wird, kann dieses Flag meist auch durch eine entsprechende Implementierung (in der Software) getriggert werden. Hier ist natürlich Vorsicht geboten, sofern ebenfalls Hardware-Interrupts zu diesem Flag erfolgen.

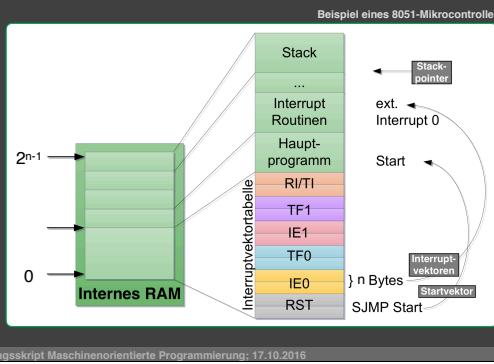
Achtung: Die Programmausführung wird nicht einfach so unterbrochen. Die Registrierung eines Interrupts erfolgt parallel zur gerade laufenden Befehlausführung. Ist die Registrierung abgeschlossen muss zwischen Unterbrechungen der gleichen Priorität unterschieden werden, dies erfolgt ebenfalls parallel. Weitere Maschinencyklen sind notwendig um den Sprung zur ISR durchzuführen.

Es kann zur Verzögerung bei der Interrupt-Verarbeitung kommen. Zum Beispiel wenn die Befehlausführung gerade begonnen hat, d.h. der erste Maschinencyklus gerade erst aktiv ist. Wir erinnern uns, dass ein Befehlszyklus aus mehreren Maschinencyklen besteht. Ein anderes Beispiel ist ein aktiver Interrupt, der eine höhere Priorität hat, als der aktuell aufgetretene. Dieser muss warten bis der höher priorisierte Interrupt abgearbeitet wurde.

Vektortabellen

6. Polling & Interrupts

- zusammenhängender, reservierter Bereich im Speicher
- jeder Interrupt ist einem Interrupt-Vektor zugewiesen
- Vektor = Zeiger
 - definiert Startadresse der ISR
 - Sprung zur ISR
 - Speicherort der ISR falls „kurz & bündig“
- Reset liegt auf Adresse 0



Die Vektortabelle steht in einem zusammenhängenden, reservierten Teil des Speichers. D.h. an dieser Stelle darf kein normaler Programmcode stehen. Ist eine ISR so kurz und bündig programmiert, dass sie innerhalb der Vektortabelle, d.h. zwischen zwei Vektoren Platz findet, kann sie direkt in der Tabelle gespeichert/abgelegt werden (kein Zeiger). Dies ist in komplexeren Programmen aber selten der Fall.

Beim Anlegen der Versorgungsspannung wird auf Adresse 0 gestartet, wodurch der Rest selbst als Interrupt-Quelle mit Vektoradresse 0 verstanden werden kann. An dieser Adresse muss ein Zeiger liegen, der die Vektortabelle überspringt und zum Start des Hauptprogramms verweist.

Polling vs. Interrupt

6. Polling & Interrupts

- Polling:
 - deterministisch
 - Laufzeit für worst- und best-case berechenbar
 - größere Reaktionszeit
 - Rechenzeitverlust
 - durch ständige Abfragen
 - viele Abfragen ohne Erfolg
 - einfache Implementierung
 - keine zusätzliche Hardware
 - leicht zu programmieren

- Interrupt:
 - nicht-deterministisch
 - Laufzeit schwer berechenbar (z.B. gleichzeitiges Auftreten aller Interrupts)
 - schnelle Reaktionszeit
 - keine Rechenzeitverluste
 - tritt Ereignis nicht ein → kein Ressourceneinsatz
 - komplexere Implementierung
 - zusätzliche Hardware
 - Flag Handling

7

Einführung des Übungsrechners

Übungsrechner: Raspberry Pi

7. Einführung des Übungsrechners

	Pi Zero	Pi Zero W	Pi 1 Model A	Pi 1 Model A+	Pi 1 Model B	Pi 1 Model B+	Pi 2 Model B	Pi 2 Model B 1.2	Pi 3 Model B
Veröffentlichungsdatum	November 2015	Februar 2017	Februar 2012	November 2014/August 2016	April-Juni 2012	Juli 2014	Februar 2015	September 2016	Februar 2016
Preisempfehlung ^a in US\$	\$[42]	19 ^[46]	29	20			35		
Platinenmaße in mm	Länge	65,0	85,6	65,0			85,6		
Gesamgröße in mm	Breite	30,0		56,0			56,0		
	Höhe	6,5	9,0	7,0, 4			9,0		
Gewicht (in g)			5,0	17,0	12,0		20,0		
SoC (Broadcom)	Type		BCM2835			BCM2836		BCM2837	
CPU	Kerne		ARM1176JZF-S			ARM Cortex-A7		ARM Cortex-A53	
	Takt in MHz	1000		700		900		1200	
	Architektur		ARMv6 (32-bit)			ARMv7 (32-bit)		ARMv8-A (64-bit)	
GPU	Familie		ARM11					ARM Cortex-A	
	Typ		Broadcom Dual Core VideoCore IV, OpenGL ES 1.1/2.0, Full HD 1080p30			250		300/400	
	Takt in MHz								
Videoausgabe	Composite Video (FBSA) ^b ; Mini-HDMI (Typ C)				Composite Video (FBSA) ^b ; HDMI (Typ A)				
Tonausgabe	HDMI (digital)				HDMI (digital); 3,5-mm-Klinkenstecker (analog)				
Arbeitspeicher LITTLE ENDIAN in MB	512	256	512(256) ^c	512(256) ^c	512		1024		
Nicht-flüchtiger Speicher	microSD-Kartenleser ^d	SD-Kartenleser ^d	microSD-Kartenleser ^d	SD-Kartenleser ^d			microSD-Kartenleser		
USB-2.0-Anschlüsse	1 (OTG)	1	2 (über Hub) ^e	2 (über Hub) ^e		4 (über Hub) ^e			

Quelle: https://de.wikipedia.org/wiki/Raspberry_Pi

Der Einplatinen-Computer „Raspberry-Pi“ bedarf heute keiner Erklärung mehr. Das geniale Konzept, der günstige Preis und der einfache Umgang mit dem System haben es bereits in der Industrie, in Lehrinstituten und auch in der „Maker“-Szene fest etabliert. Für diese Vorlesung stellt er, besonders in seiner kleinen und preisgünstigen Variante, dem „Pi Zero W“, eine sehr gute Plattform dar. So können erste Schritte innerhalb der systemnahen Programmierung auf einem aktuellen eingebetteten System gegangen werden.

Damit nicht jeder Student ein Raspberry Pi kaufen muss, kann die Entwicklungsumgebung auch auf einem virtuellen System in Betrieb genommen werden. Hierzu später mehr.

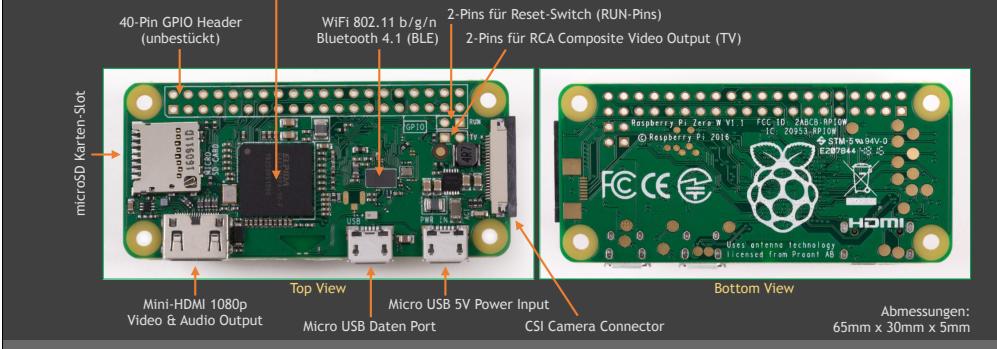
Übungsrechner: Raspberry Pi Zero W

7. Einführung des Übungsrechners

- Broadcom BCM2835 SoC (bis zu 1GHz; 512MB Onboard RAM)

- CPU: ARMv6 (ARM11JZF-S)
- GPU: VideoCore IV GPU

Fokus: Details zur GPU sind für unsere Übungen nicht von Interesse



ARM-Prozessoren

7. Einführung des Übungsrechners

- ARM steht für
 - früher: „Acorn RISC Machine“ - Unternehmen „Acorn“
 - heute: „Advanced RISC Machine“ - Unternehmen „ARM Limited“
- meist genutzte Architektur im Embedded Bereich
 - z. B. Smartphones / Tablets (Apple-Produkte und Android-Geräte)
- ARM stellt selbst keine Chips her
 - entwickelt Architektur —> vertreibt Lizenz
- 32-Bit Architekturen
 - ARMv1 (1985) bis ARMv7 (2004)
- 64-Bit Architekturen
 - ARMv8 (2011)

arm

ARM-Prozessoren			
7. Einführung des Übungsrechners			
ARM Familie	ARM Architektur	Takt	Befehlsdurchsatz
ARM1	ARMv1	4 MHz	
ARM2			
ARM3	ARMv2	8-25 MHz	0,5 DMIPS/MHz
ARM6			
ARM7	ARMv3	12-40 MHz	0,889 DMIPS/MHz
ARM7TDMI			
ARM8	ARMv4	16,8-180 MHz	0,9 DMIPS/MHz
ARM9TDMI			
ARM7EJ, ARM9E, ARM10E	ARMv5	104-1250 MHz	1,25 DMIPS/MHz
ARM11	ARMv6 ARMv6-M	200-1000+ MHz	0,6-1,54 DMIPS/MHz
ARM Cortex-M (M0, M0+, M1)			
ARM Cortex-A (A8, A9, A5, A15, A7, A12, A17) ARM Cortex-R	ARMv7-A ARMv7-R ARMv7-M	bis 2,5 GHz	1,25-3,5 DMIPS/MHz
ARM Cortex-M (M3, M4, M7)			
ARM Cortex-A (A53, A57, A72, A35, A73) ARM Cortex-R	ARMv8	1,2-3 GHz	2,3-4,1 DMIPS/MHz
ARM Cortex-M			

Quelle: <https://de.wikipedia.org/wiki/ARM-Architektur>

Hinweis:

Die Einheit Dhrystone MIPS (DMIPS) beschreibt die Rechenleistung eines Rechners. Sie wurde eingeführt um die Leistungsfähigkeit unterschiedlicher Rechner vergleichen zu können. Hierzu kommt ein Benchmark-Programm zum Einsatz, wobei dessen Iterationen pro Sekunde gemessen werden. MIPS hingegen steht für „Million Instructions Per Second“ und ist ebenfalls ein Maß für die Rechenleistung eines Prozessors.

Hinweis:

Cortex-A steht für betriebssystembasierte Anwendungen (Application)

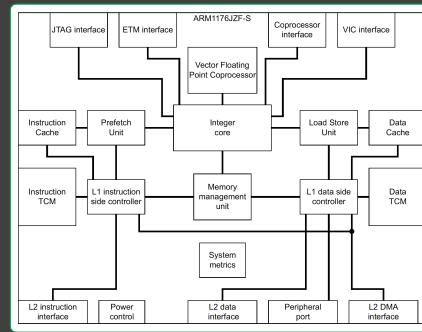
Cortex-R steht für Echtzeitanwendungen (Realtime)

Cortex-M steht für Mikrocontroller

ARM1176JZF-S (ARMv6)

7. Einführung des Übungsrechners

- 32-Bit-RISC-Architektur
- Prozessor-Features:
 - Integer Core & System Control Coprozessor (CP15)
 - interner Vector Floating-Point (VFP) Coprozessor
 - externe Coprozessor-Schnittstelle
 - 8-stages Pipeline
 - Branch Prediction mit 3-Eingänge Return Stack
 - Level 1 Tightly-Coupled Memory (TCM)
 - 64-Bit Schnittstellen zu beiden Caches
 - virtuell indizierte und physikalisch adressierte Caches
 - Instruction und Data Memory Management Units (MMUs)
 - TrustZone™ Security Extension
 - Intelligent Energy Management (IEM™)
 - High-Speed Advanced Microprocessor Bus Architecture (AMBA) Advanced Extensible Interface (AXI)
 - Level 2 Interface unterstützt priorisierte Multi-Prozessoren Setups
 - JTAG Debug-Schnittstelle inkl. trace Support (Embedded Trace Macrocell (ETM) Interface)



Bildquellen: ARM Limited; ARM1176JZF-S Technical Reference Manual; Revision: r0p7; 2009

Noch einmal zur Erinnerung: Eine 32-Bit Architektur bedeutet, dass die Maschinenbefehle 32 Bits groß sind und das auf den Speicher in 32-Bit-großen „words“ zugegriffen wird.

Die oben genannten Eigenschaften des Prozessors dienen nur einem Überblick und müssen an dieser Stelle nicht vollständig verstanden sein. Der ARM1176JZF-S erweitert die ARMv6-Architektur um weitere, hier nicht genannte, Funktionen. Diese sind für uns von untergeordneter Bedeutung und können bei Bedarf in dem zugehörigen „Technical Reference Manual“ des Prozessors nachgelesen werden.

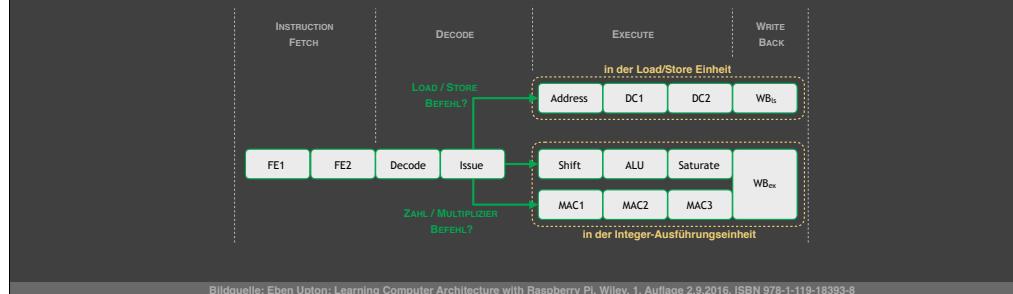
Der System Control Coprozessor ist nicht in einem physikalischen Logikblock zu finden. Er steuert und versorgt die Funktionen des ARM1176JZF-S Prozessors mit Statusinformationen. Seine Aufgaben umfassen:

- übergreifende Steuerung und Konfiguration
- Konfiguration und Verwaltung von Cache, TCM und MMU
- DMA Steuerung
- Überwachung der System-Performance

ARM1176JZF-S (ARMv6)

7. Einführung des Übungsrechners

- 8-stages Pipeline
 - Integer Execution Pfad
 - Multiply-Accumulate Pfad
 - Load/Store Pfad



Die Dekodier-Logik wählt einen der drei möglichen Pfade, sobald der Befehl bekannt ist und zur Ausführung ansteht.
Eine Ausführungseinheit (Execution Unit) ist ein Subsystem innerhalb der CPU, welches die „eigentliche Arbeit“ während der Ausführung eines Befehls erledigt.

Die Bedeutungen der einzelnen Abkürzungen der einzelnen Stages.

FE1: Die erste „Fetch Stage“. Hier wird die Adresse für den Befehl und der Befehl selbst empfangen.
FE2: Die zweite „Fetch Stage“ kümmert sich um die „Branch prediction“. Der Befehl wird dekodiert.
Decode: Der Befehl wird dekodiert.
Issue: Die Register werden gelesen und die Ausführung des Befehls wird angestoßen.
Shift: Falls Shift-Operationen notwendig sind, so werden diese innerhalb dieser Stage ausgeführt.
ALU: Alle Operationen auf Zahlenwerten werden in der ALU während dieser Stage ausgeführt.
Saturate: Das Ergebnis einer Zahlenoperationen wird „Saturated“, d.h. das Ergebnis wird überprüft ob es noch innerhalb des definierten Zahlenbereichs liegt. Ist das Ergebnis z.B. größer als der Maximalwert des Bereichs, so wird das Ergebnis auf den Maximalwert korrigiert.
MAC1: Die erste Stage für die Ausführung einer Multiplikationsoperation.
MAC2: Die zweite Stage für die Ausführung einer Multiplikationsoperation.
MAC3: Die dritte Stage für die Ausführung einer Multiplikationsoperation.
WB_{ex}: Die geänderten Registerinhalte werden in die Register zurückgeschrieben. Dies ist die letzte Stage des Integer Execution und des Multiply-Accumulate Pfades.
Address: Die Erzeugung von Adressen um auf Speicherinhalte zugreifen zu können.
DC1: Die erste Stage in der die Adresse durch die Daten-Cache-Logik verarbeitet wird.
DC2: Die zweite Stage in der die Adresse durch die Daten-Cache-Logik verarbeitet wird.
WB_{ls}: Die letzte Stage innerhalb des LOAD/STORE Pfades in der die Ergebnisse/Änderungen zurück in den Speicher geschrieben werden.

ARM1176JZF-S: Integer Core

7. Einführung des Übungsrechners

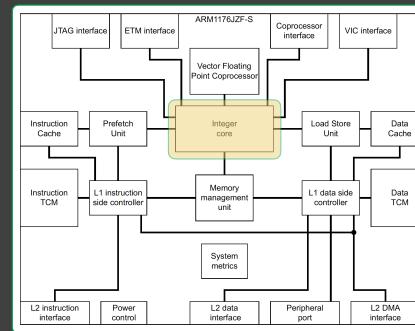
- drei Befehlssätze

Fokus →

- ARM: 32-Bit (DSP + Media Ext.)
- Thumb: 16-Bit
- Jazelle: 8-Bit-Vielfache (Java bytecode)

- Hauptkategorien der Instruktionen

- branch
- data processing
- status register transfer
- load / store / swap
- coprocessor
- exception-generating



Bildquellen: ARM Limited; ARM1176JZF-S Technical Reference Manual; Revision: r0p7; 2009

Der ARM-Befehlssatz ist durch DSP- und Media-Instruktionen erweitert. Die DSP-Erweiterung beinhaltet 16-Bit Datenoperationen und weiterführende Arithmetiken. Dabei werden z.B. Multiplikationen in nur einem Zyklus (Single-Cycle) durchgeführt. Die Media-Erweiterung dient der Vervollständigung des DSP-Befehlssatzes und ist in vier Gruppen aufgeteilt.

- Multiplikationsbefehle die z.B. auf beiden Hälften (2x16Bit) eine Multiplikation durchführen können.
- SIMD (Single Instruction Multiple Data): Hierbei befinden sich entweder zwei 16Bit-Datenwerte oder vier 8-Bit-Datenwerte in einem Register.
- Instruktionen um Bytes innerhalb eines Registers zu modifizieren
- Unsigned Sum-of-Absolute-Differences (SAD) Befehle: Werden z.B. in MPEG zur Vorhersage von Bewegungen genutzt.

Der ARM-Befehlssatz ist der am meist genutzte Befehlssatz im Umgang mit dieser Architektur und soll daher den Fokus der Vorlesung erhalten.

Der Thumb Befehlssatz besteht aus den meist genutzten 32-Bit ARM Maschinenbefehlen, welche auf 16-Bit kodiert wurden. Dadurch wurde der notwendige Speicherplatz für Instruktionen reduziert, d.h. die „code density“ kann bei Verwendung dieses Befehlssatzes gesteigert werden. Es finden mehr Instruktionen (als Teil des Programms) auf weniger Speicherplatz ihre Implementierung. Der Befehlssatz findet meist in „embedded Systems“ seine Verwendung. Hier befindet sich der Raspberry Pi auf einem schmalen Grad, da er in vielen Embedded Systems zum Einsatz kommt aber die Leistungsdaten eines „General-Purpose“-Computers besitzt.

Der Jazelle Befehlssatz erlaubt es dem ARM11 Kern direkt Java Bytecode auszuführen, d.h. ohne vorangehende Interpretation durch eine Software. Da ARM 2011 diesen Befehlssatz als „deprecated“ eingestuft hat, ist es nicht empfohlen diesen in neuen Projekten zu verwenden.

Hinweis: Nur „load, store, swap“ Instruktionen ist es erlaubt auf Daten aus dem Memory zuzugreifen.

ARM1176JZF-S: ARM Instructions

7. Einführung des Übungsrechners

(Bsp.: ADD)

- Notation:



- optionale Angaben: { ... }
- Platzhalter / Symbole: < ... >
- bedingte Ausführungsoptionen: < condition code >
- Modifizierung der „Conditional Flags“: {S}

Quelle: Eben Upton: Learning Computer Architecture with Raspberry Pi, Wiley, 1. Auflage 2.9.2016, ISBN 978-1-119-18393-8

Mit Hilfe des optionalen „S“ wird der Prozessor angewiesen die „Conditional Flags“, auf Basis des Ergebnisses der Operation, zu modifizieren. Fehlt der Suffix so werden die „Conditional Flags“ nicht beeinflusst. In anderen Worten bedeutet dies, dass eine Folge von Befehlen bedingt ihre Aktivitäten durchführen, abhängig von der ersten Instruktion, welche die „Conditional Flags“ modifiziert hat.

Achtung: Bedingte Ausführungsmöglichkeiten existieren nicht für alle Befehle.

Wie funktioniert die bedingte Code-Ausführung?

- Jede Instruktion durchläuft die Pipeline vollständig und unabhängig von der Bedingung („condition code“).
- Die Aktion des Befehls findet jedoch nur dann statt, wenn die Bedingung erfüllt ist.
- Andernfalls durchläuft der Befehl die Pipeline ohne jegliche Aktion auszuführen.

ARM1176JZF-S: Processor States vs. Processor Modes

7. Einführung des Übungsrechners

- Processor State
 - ARM state - Ausführung einer ARM Instruktion
 - Thumb state - Ausführung einer Thumb Instruktion
 - Jazelle state - Ausführung einer Jazelle Instruktion
- Processor Mode
 - Logik zur Verwaltung von System-Ressourcen
 - Logik zur Trennung von Applikationen
 - z.B. verhindert das Eingreifen einer Applikation in das Betriebssystem
 - verschiedene Modi des ARM11:
 - „privileged“ bis auf „User Mode“

Quelle: Eben Upton: Learning Computer Architecture with Raspberry Pi, Wiley, 1. Auflage 2.9.2016, ISBN 978-1-119-18393-8

Die Prozessor-Modi dienen dazu komplexe Betriebssystem, Anwendungsprogramme und „Echtzeitanforderungen“ in einem System kombinieren zu können.

„Privileged“ bedeutet, dass der volle Zugriff auf System-Ressourcen gegeben ist.

ARM1176JZF-S: Processor Modes

7. Einführung des Übungsrechners

Mode	Abkürzung	Mode bits	Beschreibung
User	usr	10000	zur Ausführung von Anwender-Applikationen
Supervisor	svc	10011	zur Ausführung des OS-Kernels; aktiv nach Reset
System	sys	11111	User-Mode + „privileged“; low-level embedded; wenig genutzt → nun obsolet
Secure monitor	mon	10110	verwendet das Feature „TrustZones“ (isolierte Speicherregionen („worlds“)); verwaltet Datenaustausch zwischen „worlds“; nicht im BCM2835 SoC implementiert!
FIQ	fiq	10001	„fast“ Interrupt-Behandlung
IRQ	irq	10010	„general-purpose“ Interrupt-Behandlung
Abort	abt	10111	für Virtuelle Speicher; andere Speicherverwaltungen
Undefined	und	11011	Software Emulation von nicht definierten Maschinenbefehlen

Quelle: Eben Upton: Learning Computer Architecture with Raspberry Pi, Wiley, 1. Auflage 2.9.2016, ISBN 978-1-119-18393-8

„Interrupts“ sind Signale von Hardware-Einheiten ausserhalb der CPU. Sie werden ausgesandt um spezielle Ereignisse zu identifizieren, bzw. dem System mitzuteilen, dass die aussendende Einheit gerade Aufmerksamkeit benötigt.

„Exceptions“ sind Ausnahmezustände/-ereignisse innerhalb der CPU. Sie benötigen in der Regel eine spezielles Handling durch die CPU selbst. Ein Beispiel hierzu ist die Division durch 0.

ARM1176JZF-S: Integer Core - Register

7. Einführung des Übungsrechners

- 33x32-Bit General-Purpose Register
 - 16 immer voll zugreifbar
 - Rest erfordern erweiterte Rechte
 - sind „banked registers“
 - dienen der Ausführungsbeschleunigung
- 7x32-Bit dedizierte Status-Register
- Register R0 bis R15 + CPSR
 - General-Purpose: R0-R12
 - Special-Purpose: R13-R15 + CPSR

Quelle: Eben Upton: Learning Computer Architecture with Raspberry Pi, Wiley, 1. Auflage 2.9.2016, ISBN 978-1-119-18393-8

Ein wichtiger Aspekt beim CPU Design ist wie viele Register innerhalb der CPU platziert werden. Je mehr Register existieren, desto weniger oft müssen die Operanden eines Befehls aus dem Speicher geholt / in den Speicher zurückgeschrieben werden. Durch weniger häufige Speicherzugriffe wird die Ausführungszeit beschleunigt. Dennoch muss die Anzahl praktikabel bleiben, es sind nicht beliebig viele Register sinnvoll.

Register Banking: Bedeutet mehrere Kopien eines Registers bei gleicher Adresse bereitzustellen. Meist erlauben die „Banked Register“ einen schnellen Context Switch, um auf Prozessorfehler und vorrangige Operationen reagieren können. Bei dem Raspberry Pi bedeutet dies, dass die Register R0 bis R7 von allen Prozessor Modi, hier existieren auch keine banked Register. Danach, also ab Register R8, wird es komplizierter. Hier hat z.B. der Fast Interrupt Mode (FIQ) private Register auf einer „private bank“.

CPSR: Current Program Status Register (hierzu später mehr)

ARM1176JZF-S: Integer Core - Register

7. Einführung des Übungsrechners

- R0-R12
 - zum Speichern von temporären Werten, Zeiger, etc.
 - R0 speichert Return-Value (Linux)
 - R7 von Interesse bei syscalls (speichert die syscall-Nummer)
 - R11 hilft uns den Stack zu überwachen (FP: Frame Pointer)
 - R12 Intra Procedural Call (IP)
 - wird von C-Libraries beim Aufruf von „dynamically linked libraries“ verwendet (z.B. printf)
- R13 (SP - Stack Pointer)
 - zeigt auf den TOS (Top-of-Stack)
 - Funktions-spezifischer Speicher —> wird nach dem Ende der Funktion frei
 - genutzt um Speicher auf dem Stack zu reservieren
 - *Achtung:* Nutzung als „General-Purpose“ —> „Crashes“ im OS



R0, R7, R11 und R12 können in einem „Bare-Metall-System“ (hierzu später mehr) bedenkenlos als „General-Purpose“-Register verwendet werden.

ARM1176JZF-S: Integer Core - Register

7. Einführung des Übungsrechners

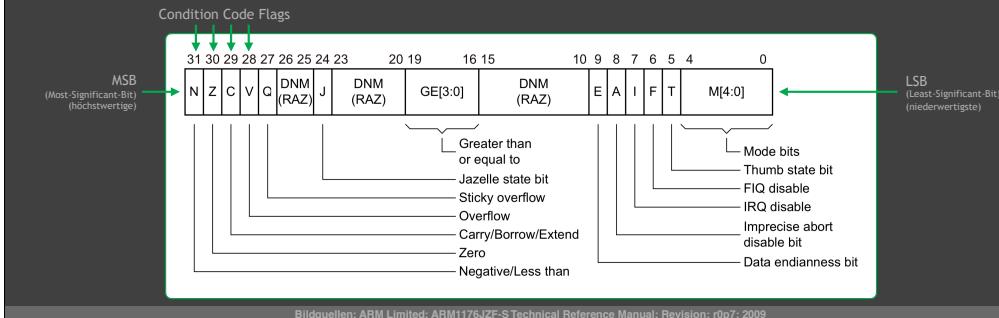
- R14 (LR - Link Register)
 - bei einem Funktionsaufruf (Subroutine) wird der Wert aktualisiert
 - „Branch with Link“ - Befehle („BL“, „BLX“)
 - erhält die Speicheradresse, die auf die nächste Instruktion zeigt
 - „return address“ → LR
 - von dort wo der Funktionsaufruf stattfand
 - erlaubt uns zum Elternteil zurückzukehren (parent)
 - Zurückkopieren von LR nach PC
- R15 (PC - Program Counter)
 - inkrementiert um 4-Byte (ARM mode) / 2-Byte (THUMB mode)
 - bei einem Branch: Zieladresse
 - *Achtung:* selbst im User-Mode änderbar!

arm

ARM1176JZF-S: Integer Core - Register

7. Einführung des Übungsrechners

- CPSR (Current Program Status Register / Flags)
 - beinhaltet Informationen über letzten ALU Operation
 - steuert das Aktivieren/Deaktivieren von Interrupts
 - setzt den Betriebsmodus des Prozessors



Bildquellen: ARM Limited; ARM1176JZF-S Technical Reference Manual; Revision: r0p7; 2009

Die Condition Code Flags (oder kurz: Condition Flags) werden durch N, Z, C und V abgebildet. Sie sind durch viele Befehle modifizierbar und deren aktueller Status kann zur bedingten Abarbeitung anderer Befehle genutzt werden.

- N: Wird zu 1 (aktiv), wenn ein Rechenergebnis negativ ist. Es ist inaktiv (0), wenn das Ergebnis 0 oder größer als 0 ist.
- Z: Wird zu 1 (aktiv), wenn das Rechenergebnis 0 ist. Andernfalls ist das Bit inaktiv (0). Aufgrund der Art und Weise wie Vergleiche berechnet werden, ist das Bit ebenfalls gesetzt, wenn zwei verglichene Operanden gleich sind.
- C: Betrifft die Addition, Shift und Subtraktion. Wird z.B. 1 (aktiv), wenn das Ergebnis der Addition den maximalen Wert (32-Bit-Wert) überschreitet. Achtung: Bei einem Shift enthält das Carry den Wert des, aus dem 32-Bit-Wert, geschobenen Bits.
- V: Ist gesetzt, wenn bei einer Addition oder Subtraktion ein vorzeichenbehäfteter Überlauf (Overflow) auftritt.

Das Q repräsentiert den „Sticky Overflow/Saturation“, d.h. es enthält die Information ob ein Ergebnis den zulässigen Wertebereich verlassen hat und korrigiert wurde, um in das Zielregister geschrieben werden zu können. Wird dieses Bit durch einen Befehl gesetzt, so muss es durch einen anderen Befehl gelöscht werden. Das Q wird nicht von bedingten Befehlen genutzt. Dieses Bit ist für diese Vorlesung „out of scope“.

Das T und J repräsentiert den „Thumb“ bzw. „Jazelle State“ und ist entsprechend aktiv, wenn der Prozessor sich in dem jeweiligen Zustand befindet. Ist keins der beiden Bits gesetzt, so befindet sich der Prozessor im „ARM State“.

Das E beschreibt die „endianess“ der aktuellen CPU Operation. Der Bit-Wert 1 steht für „little endian“ und 0 steht entsprechend für „big endian“. Dieses Bit muss durch eigene Maschinenbefehle aktiv gesetzt werden, diese lauten entsprechend „SETEND LE“ bzw. „SETEND BE“.

Das A erlaubt die Unterscheidung ob ein Fehler auf einer virtuellen Speicherseite auftrat oder ob es sich um einen Fehler in einem externen Speicher handelt. Auch dieses Bit spielt für diese Vorlesung keine große Rolle.

Die I und F Bits stellen Masken für die Interrupts da. Dazu später mehr.

Die Einträge DNM (RAZ) stehen für Bits, welche undefiniert oder für die Verwendung in neueren ARM-Architekturen reserviert sind.

ARM1176JZF-S: Interrupt, Exception, Vector Table

7. Einführung des Übungsrechners

- verschiedene Klassen
 - haben eigenen Prozessor Mode
 - haben eigene „banked Registers“
 - behandelt durch Interrupt-Handler (Wdh.)
- beim Auftreten eines Interrupts:
 1. CPU wechselt Prozessor Mode (abhängig von dem Interrupt / Exception)
 2. speichert PC in LR des neuen Modes („banked Register“)
 3. speichert CPSR in SPSR des neuen Modes („banked Register“)
 4. setzt PC auf Adresse in der Vektortabelle
 5. Ausführung des Interrupt-Handlers

SPSR: Saved Program Status Register

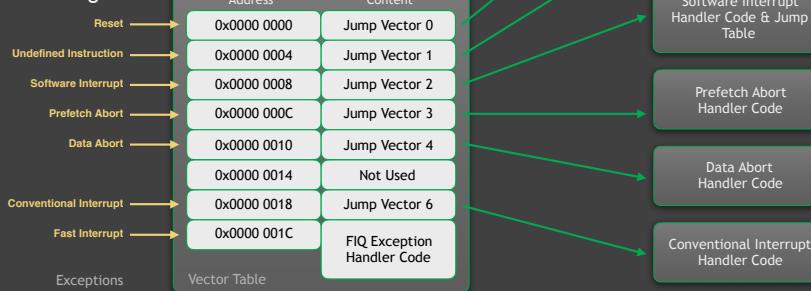
Die Verwendung von „banked Register“ hilft der CPU, auf einfache Art und Weise, das Minimum an Zuständen abzuspeichern, um später wieder an dem ursprünglichen Programmabschnitt weitermachen zu können.

ARM1176JZF-S: Interrupt, Exception, Vector Table

7. Einführung des Übungsrechners

- Vector-Table

- jeder Eintrag ist ein 32-Bit unbedingter Sprungbefehl
- Sprungbefehl führt CPU zum Handler
- 8x 32-Bit lang



Quelle: Eben Upton: Learning Computer Architecture with Raspberry Pi, Wiley, 1. Auflage 2.9.2016, ISBN 978-1-119-18393-8

Der Handler Code muss alle Operationen beinhalten um die Exception erfolgreich zu behandeln. Wie dies im einzelnen abläuft haben wir bereits zuvor besprochen.

ARM1176JZF-S: „Physical“ Interrupts

7. Einführung des Übungsrechners

- Interrupts von „außerhalb“
- 2 Typen + 2 zugehörige Signale → 2 Einträge in Vector Table
 - regular (IRQ)
 - fast (FIQ)
- FIQ Besonderheiten:
 - 2 Mechanismen zur Minimierung der „Interrupt Servicing Latency“
 - 1. befindet sich am Ende der Vector Table → 2 Möglichkeiten
 - Sprungbefehl zum Handler
 - Handler Code direkt beginnend am Eintrag (Verhindern von z.B. Pipeline Stalls)
 - 2. hat Kopien („banked Registers“) der Register R8 bis R14 (statt R13 + R14)
 - Kopien direkt verwendbar ohne die Register der Anwendung zu zerstören
 - keine Zeiteinbußen für das Sichern der Register auf dem Stack

Im Vergleich zu den IRQ ist die Antwortzeit auf einen FIQ schneller und deterministischer. Der Grund hierfür ist der Versuch die Speicherzugriffe zu verringern. Befindet sich der Handler Code bereits im Cache, beginnt und endet die Behandlung der Exception ohne jeglichen Speicherzugriff.

Was ist ein Pipeline Stall?

Ein „Pipeline Stall“ ist eine Verzögerung in der Ausführung eines Befehls aufgrund von einem Hazard, welcher vor der weiteren Ausführung erst behoben werden muss.

IRQ: Interrupt Request

ARM1176JZF-S: Software Interrupt (SWI)

7. Einführung des Übungsrechners

- keine „ungeplante“ Unterbrechung der CPU
- eher ein Unterprogramm-Aufruf („managed call“)
 - um den Supervisor Mode vorsichtig/verwaltet zu betreten
 - z.B. zur Kommunikation mit dem OS-Kernel
- SWI sind nummeriert
 - Nummer als Operand übergeben
 - Handler Code muss sich um die Nummer kümmern —> keine Unterscheidung in der Vector Table
 - ggf. Aufruf von weiteren Sub-Handler
- Aufruf / Auslösen eines SWI:



Ein SWI ist die einzige Möglichkeit um vom User Mode in den Supervisor Mode zu gelangen. Dieses Vorgehen, also der Umweg über die Vector Table, verhindert, dass jede Art von Code im „privileged“ Modus ausgeführt werden kann. Die Limitierungen welche im User Mode herrschen, wie z.B. die Konfiguration der MMU (Memory Management Unit), untermauern in einem Betriebssystem die Auffassung von Prozessisolierung.

ARM1176JZF-S: Interrupt Prioritäten

7. Einführung des Übungsrechners

- weitere Exception während ein anderer Handler aktiv ist
- 2 Möglichkeiten zur Verhinderung:
 - Interrupts deaktivieren bei Handler Ausführung
 - Vergabe von Prioritäten

ARM 11 Interrupt Prioritäten

Exception	Priorität
Rest	1
Data Abort	2
Fast Interrupt (FIQ)	3
Conventional Interrupt (IRQ)	4
Prefetch Abort	5
Software Interrupt (SWI)	6
Undefined Instruction	6

Quelle: Eben Upton: Learning Computer Architecture with Raspberry Pi, Wiley, 1. Auflage 2.9.2016, ISBN 978-1-119-18393-8

CPSR: Bit F auf 1 deaktiviert die Fast Interrupts. Bit I auf 1 deaktiviert die Conventional Interrupts.

Eine Exception mit einer höheren Priorität (kleinere Zahl) darf nicht durch einen Exception niedriger Priorität (größere Zahl) unterbrochen werden. So kann z.B. der Handler für den Reset durch keine andere Exception unterbrochen werden.

Wenn ein Interrupt Handler seine Ausführung startet, werden alle Interrupts mit der selben Priorität automatisch deaktiviert. Dies verhindert, dass auch dieser Interrupt Handler auf dem selben Prioritätslevel unterbrochen werden kann.

Interrupts werden nicht von Software im User Mode deaktiviert. Dies würde die Autorität des Betriebssystems untergraben. Jedoch können Software Interrupts aus der User-Mode-Anwendung ausgelöst werden. Da Sie die niedrigste Priorität haben, dürfen alle anderen Interrupts nach wie vor ihre Dienste verrichten, sofern sie nicht von anderer Stelle deaktiviert wurden.

Der Software Interrupt hat das selbe Prioritätslevel wie auch der undefinierte Befehl. Sie beide können nicht gleichzeitig auftreten und daher ist dies zulässig. In allen ARM Prozessoren ist der SWI definiert, d.h. dieser Befehl ist niemals undefiniert.

ARM1176JZF-S: Bedingte Befehlsausführung

7. Einführung des Übungsrechners

- „*Conditional Instruction Execution*“
- alle 32-Bit ARM Befehle
 - erlauben die bedingte Befehlsausführung
 - beinhalten ein 4-Bit Feld für „*Condition Codes*“
- ARM Architecture
 - 15 „*Condition Codes*“
 - entsprechen Kombinationen der „*Condition Flags*“
 - *Achtung:* Kein Bit-für-Bit Vergleich!
 - „*Condition Code Field*“ wird ausgewertet wenn CPU Befehl dekodiert

Bedingte Sprungbefehle („Conditional Branch Instruction“) werden oft genutzt um die Ablauf des Programms zu alternieren, je nach dem welche Gegebenheiten vorliegenden. Stattdessen sind manchmal Befehle, die eine bedingte Befehlsausführung ermöglichen, die bessere Wahl.

Achtung: Hier wird nicht Bit für Bit verglichen, sondern jeder 4-Bit Binärwert hat seine eigene Bedeutung.

ARM1176JZF-S: Bedingte Befehlsausführung

7. Einführung des Übungsrechners

Code	Suffix	Meaning (for cmp or subs)	Flags Tested
%0000	eq	Equal.	Z==1
%0001	ne	Not equal.	Z==0
%0010	cs or hs	Unsigned higher or same (or carry set).	C==1
%0011	cc or lo	Unsigned lower (or carry clear).	C==0
%0100	mi	Negative. The mnemonic stands for "minus".	N==1
%0101	pl	Positive or zero. The mnemonic stands for "plus".	N==0
%0110	vs	Signed overflow. The mnemonic stands for "V set".	V==1
%0111	vc	No signed overflow. The mnemonic stands for "V clear".	V==0
%1000	hi	Unsigned higher.	(C==1) && (Z==0)
%1001	ls	Unsigned lower or same.	(C==0) (Z==1)
%1010	ge	Signed greater than or equal.	N==V
%1011	lt	Signed less than.	N!=V
%1100	gt	Signed greater than.	(Z==0) && (N==V)
%1101	le	Signed less than or equal.	(Z==1) (N!=V)
%1110	a1 (or omitted)	Always executed.	None tested.

Quelle: <https://community.arm.com/processors/b/blog/posts/condition-codes-1-condition-flags-and-codes>

Der Befehl wird also nur dann ausgeführt, wenn der aktuelle Zustand der „Condition Flags“ im CPSR Register mit dem „Condition Code“ des Befehls übereinstimmen. Zur Erinnerung: das CPSR Status Register hat 4 Flags (N, Z, C und V) die hierzu verwendet werden.

ARM1176JZF-S: Bedingte Befehlsausführung

7. Einführung des Übungsrechners

- Spezifizierung in Assembler-Code:
 - 2-Charakter Suffix an Mnemonic
 - Beispiele:

	OPCODE Mnemonics	2-Charakter Suffix	
1	MOV R0, #4		wird immer ausgeführt
2	MOVEQ R0, #4	EQ	ausgeführt wenn Z=1 (EQUAL)
3	MOVNE R0, #4	NE	ausgeführt wenn Z=0 (NOT EQUAL)
4	MOVMI R0, #4	MI	ausgeführt wenn N=1 (NEGATIVE)

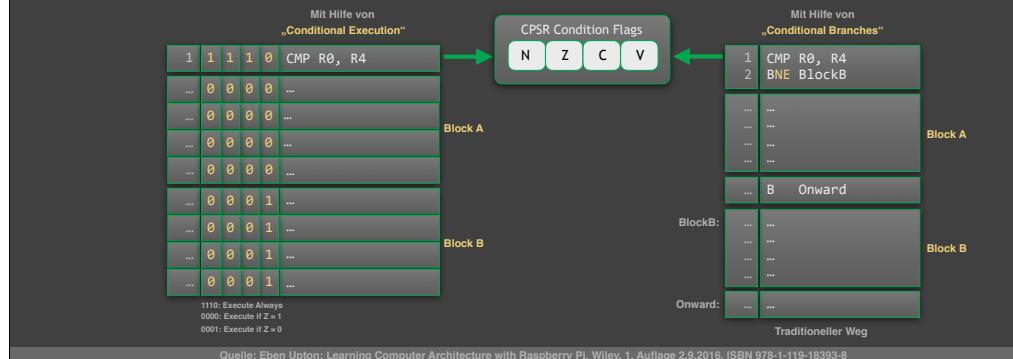
Alle der Operationen kopieren den Wert 4 in das Register R0.

Zeile 2 wird ausgeführt, wenn vorher ein Vergleich (oder eine andere Operation) das Ergebnis 0 generierten.
Zeile 4 wird ausgeführt, wenn eine vorangehende Operation ein negatives Ergebnis lieferten.

ARM1176JZF-S: Bedingte Befehlsausführung

7. Einführung des Übungsrechners

- Warum ist die bedingte Befehlsausführung so mächtig?
- z.B. if R0 = R4 dann führe BLOCK A aus, sonst BLOCK B



Beide Seiten führen das selbe Programm aus!

Der erste Befehl vergleicht die Register R0 und R4. Sind die Registerwerte gleich, so setzt CMP das Z-Flag auf 1, anderenfalls auf 0. Dieser Befehl ist also entscheidend für die darauffolgende Befehlausführungen.

Ein Beispiel für eine Sequenz ohne „Conditional Execution“:

```
CMP r3, #0
BEQ next
ADD r0, r0, r1
SUB r0, r0, r2
next
...
```

Die gleiche Sequenz mit „Conditional Execution“ erlaubt das Entfernen eines Befehls:

```
CMP r3, #0
ADDNE r0, r0, r1
SUBNE r0, r0, r2
...
```

Nochmal zurück zu dem dargestellten Beispiel: Die rechte Seite ermöglicht es zwei Befehle wegzulassen. Dies an sich ist bereits ein Vorteil, jedoch hat es einen weiteren großen Vorteil. In modernen Prozessoren, wie auch dem ARM werden Sprünge vorausgesagt. Wäre diese Voraussage falsch, so würde dies ggf. unsere Pipeline zunächst zum Erliegen bringen und die Ausführung an Geschwindigkeit verlieren. Alles was getan werden kann um Sprünge zu vermeiden kann der Performance zugute kommen.

Es ist wichtig im Kopf zu behalten, dass Befehle mit „Conditional Execution“ nicht übersprungen werden, auch wenn ihr Code nicht zutrifft. Sie durchlaufen die Pipeline und konsumieren dabei einen Clock-Cycle. Jedoch werden sie nicht ausgeführt und ändern somit auch nichts im System. Der Benefit von „Conditional Execution“ macht sich besonders bezahlt, wenn nur kleine Code-Blöcke ursprünglich übersprungen werden sollen. Diese Sprünge kosten viel mehr Zeit als das Lesen, aber nicht Ausführen eines Blocks. Hierzu gibt es einen „Block-Size-Threshold“ (abhängig von der Architektur) über welcher sich die Implementierung eines Sprungs eher eignet. Diese Grenze liegt meist bei 3-4 Befehlen, ist also nicht sehr groß.

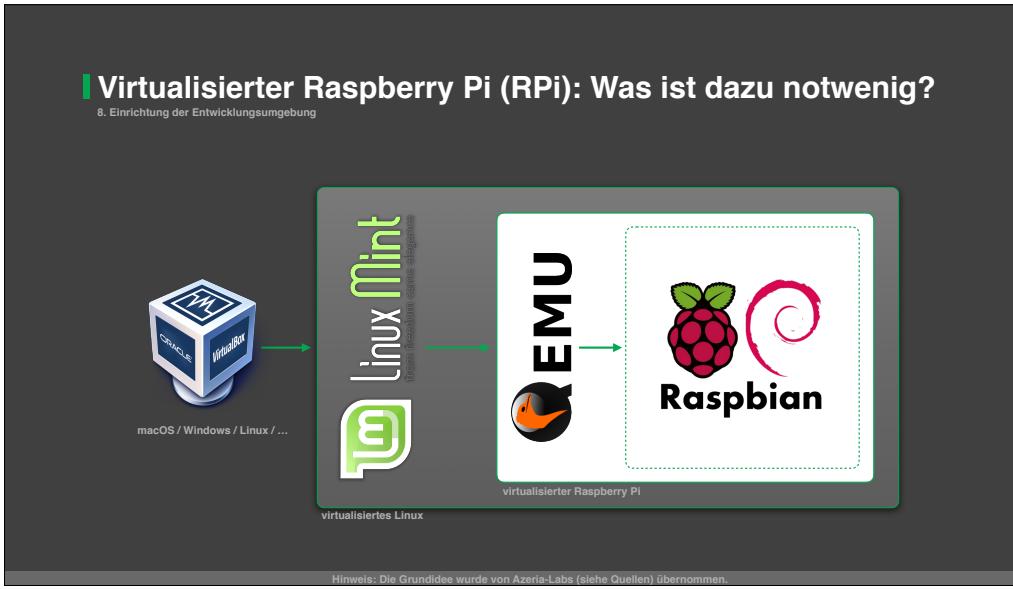
8

Einrichtung der Entwicklungsumgebung

Entwickeln auf dem Übungsrechner

8. Einrichtung der Entwicklungsumgebung

- Möglichkeiten:
 1. Vorhandensein einer Raspberry Pi Hardware
 2. Virtualisierung eines Raspberry Pi
- Vorteile der Virtualisierung:
 - Sandbox, d.h. Spielwiese, ohne Risiko die Hardware zu beschädigen
 - kostengünstig
- Nachteile der Virtualisierung:
 - langsamere Ausführung
 - keine Anbindungsmöglichkeit von externer Hardware über GPIOs
- Erste Schritte: Wir beginnen mit dem virtualisierten Raspberry Pi!



VirtualBox ermöglicht uns, unabhängig von dem Grundsystem (Mac, Windows PC, Linux PC), einen Rechner zu virtualisieren. Leider unterstützt die Software „VirtualBox“ nicht die Virtualisierung eines Rechners auf Basis der ARM-Architektur, weshalb QEMU eingesetzt werden muss. QEMU ist ebenfalls eine Virtualisierungssoftware und verfügt über passende Kernel, die uns das Erstellen eines ARM-basierten Rechners ermöglichen und wir somit Raspbian, also unseren Raspberry Pi, sinnvoll aufsetzen können.

Hinweis:
Wer ein Linux oder Windows verwendet, kann versuchen durch QEMU direkt einen Raspberry Pi zu virtualisieren. Auf dem Mac ist es nur durch inoffizielle Quellen „Homebrew“ möglich. Durch das „Vorschalten“ von VirtualBox haben alle Studenten die gleiche Betriebssystembasis, was die Problembehandlung im Kurs vereinfacht.

Virtualisierter RPi: Aufsetzen der VirtualBox

8. Einrichtung der Entwicklungsumgebung

- Download des aktuellen Linux-Mint-Images
 - <https://linuxmint.com/download.php>
 - Cinnamon 64-bit-edition
- Erstellen einer neuen VirtualBox
 - Typ: Linux
 - Version: Ubuntu (64-bit)
 - Hauptspeicher: 4096 MB
 - virtuelle Größe: fest auf min. 20 GB (besser 25 GB); Typ: normal (VDI)
- Linux-Mint-Image einbinden und installieren
 - nach Installation Linux Mint Update durchführen (via Terminal-Befehle)
 - `sudo apt-get update`
 - `sudo apt-get upgrade`

Natürlich wird vorausgesetzt, dass VirtualBox bereits auf dem Rechner installiert wurde.

Virtualisierter RPi: Aufsetzen von QEMU / Raspbian (1)

8. Einrichtung der Entwicklungsumgebung

- Download des Raspbian-Images (in Linux-Mint)
 - <http://downloads.raspberrypi.org/raspbian/images/raspbian-2017-04-10/>
 - Jessie wird empfohlen - neuere Versionen können ebenfalls funktionieren
- Download des letzten QEMU Kernels (in Linux-Mint)
 - <https://github.com/dhruvvyas90/qemu-rpi-kernel>
 - Download als Repositories als ZIP
 - hier verwendet „kernel-qemu-4.4.34-jessie“
- Ordner im Home-Verzeichnis auf Linux-Mint erstellen
 - mkdir ~/*qemu_vms* oder mkdir /home/<user_name>/*qemu_vms*
- Kernel + Image im Verzeichnis ablegen
 - Dateien entpacken: unzip <name>.zip
 - ZIP-Archive und überflüssige Elemente löschen

```
dhw@dhbw-VirtualBox: ~/qemu_vms $ ls -l
total 302408
drwxr-xr-x 1 dhbw dhbw 302408 Sep 14 18:05 kernel-qemu-4.4.34-jessie
drw-r--r-- 1 dhbw dhbw 1072745678 Feb 15 13:00 raspbian.img
drwxr-xr-x 1 dhbw dhbw 35188 Sep 15 2017 README.md
drwxr-xr-x 1 dhbw dhbw 35188 Sep 15 2017 Reg-SCREEN.png
-rwxr-xr-x 1 dhbw dhbw 228 Jan 31 17:48 runPi.sh
-rwxr-xr-x 1 dhbw dhbw 37 Jan 31 18:13 sshPi.sh
drwxr-xr-x 1 dhbw dhbw 4096 Sep 14 18:05 tools
dhw@dhbw-VirtualBox: ~/qemu_vms $
```

Der Name des „raspbian.img“ ist zu Beginn ein Anderer. Der Verzeichnis-Screenshot zeigt bereits die fertige Installation und dient als Orientierung. Bitte beachtet auch die Vergabe der Zugriffs- und die Besitzrechte der Dateien.

Hinweis: Aufgrund der Virtualisierung kann es schwierig sein die Tilde (-) auf der Tastatur zu finden. Sie kann weggelassen werden, wenn ihr die Operationen, beginnend von eurem Home-Verzeichnis, ausführt.

Hinweis: Zum Löschen kann der Linux-Befehl „rm“ (remove) verwendet werden. Beispiel: „rm <name>.zip“ oder, wenn gleich ein ganzer Ordner gelöscht werden soll, „rm -rf tools“.

Hinweis: Zum Verschieben kann der Befehl „mv“ (move) verwendet werden. Beispiel: „mv Downloads/<raspbian-img>.zip qemu_vms“ hierzu müssen wir zuvor in das Home-Verzeichnis gewechselt sein.

Um bei der Anpassung der Rechte nicht den ganzen Pfad angeben zu müssen wechselt ihr am besten in das erstelle Verzeichnis. (*cd ~/*qemu_vms*/*)

Linux-Befehl zum Setzen der Zugriffsrechte: chmod 755 *kernel-qemu-4.4.34-jessie* (Wobei 755 natürlich nur als Beispiel dient. Die 7 (binär 111) setzt die ersten 3 Flags (rwx). Die 5 die nächsten 3 Flags (r-x). Usw.)

Linux-Befehl zum Setzen der Besitzrechte: chown -R dhbw:dhbw *tools* (Wobei das Attribut -R alle Dateien des Ordners „*tools*“ mit bedient und nur bei Ordnern anzugeben ist.)

Guest-Additions der VirtualBox in Linux-Mint installieren:

```
sudo apt-get install virtualbox-guest-additions-iso
sudo apt-get install virtualbox-guest-utils
```

Virtualisierter RPi: Aufsetzen von QEMU / Raspbian (2)

8. Einrichtung der Entwicklungsumgebung

- Installation des QEMU Grundsystems
 - sudo apt-get install qemu-system
- Raspbian-Image vorbereiten
 - Home-Verzeichnis betreten: cd /home/<user_name> oder cd ~
 - QEMU-Ordner betreten: cd qemu_vms
 - Raspbian-Image betrachten: fdisk -l <raspbian-downloaded>.img

```
shihmed@shihmed-VirtualBox:~/qemu_vms$ fdisk -l raspbian.img
Medium raspbian.img: 10 GiB, 10727456768 Bytes, 20952064 Sektoren
Einheiten: sectors von 1 * 512 = 512 Bytes
Sektorengröße (logisch/physisch): 512 Bytes / 512 Bytes
I/O Größe (minimal/optimal): 512 Bytes / 512 Bytes
Typ der Reduziermechanik: dos
Medienkennung: 0xA02e4a57
```

Gerät	Boot	Start	Ende	Sektoren	Größe	Id	Typ
raspbian.img1	8192	92159	83968	41M	c w95 FAT32 (LBA)		
raspbian.img2	92160	20952663	20859904	106 83	Linux		

- Offset im Beispiel: img2 startet bei Sektor 92.160, aufgrund der Sektogröße von 512 sind diese beiden Werte zu multiplizieren —> Resultat: 47.185.920

Auch hier bitte darauf achten, dass dies das fertige System ist. Das heruntergeladene Image sollte eine Mediumgröße von ca. 4,1GB besitzen.

Virtualisierter RPi: Aufsetzen von QEMU / Raspbian (3)

8. Einrichtung der Entwicklungsumgebung

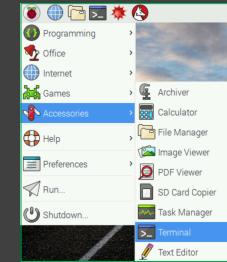
- Fortsetzung: Raspbian-Image vorbereiten
 - Raspbian-Image einbinden und modifizieren
 - sudo mkdir /mnt/raspbian
 - sudo mount -v -o offset=47185920 -t ext4 /home/<user_name>/qemu_vms/<raspbian-downloaded>.img /mnt/raspbian
 - Raspbian-Image für QEMU modifizieren und Änderungen speichern
 - sudo vi /mnt/raspbian/etc/ld.so.preload
 - In der Datei sind alle Einträge durch eine „#“, vor der jeweiligen Zeile, auszkommentieren.
 - sudo vi /mnt/raspbian/etc/fstab
 - Falls in dieser Datei Einträge mit „mmcblk0“ vorhanden sind, müssen diese ersetzt werden. Aus „dev/mmcblk0p1“ wird „/dev/sda1“ und aus „dev/mmcblk0p2“ wird „/dev/sda2“.
 - modifiziertes Raspbian-Image aushängen
 - Home-Verzeichnis betreten: cd /home/<user_name> oder cd ~
 - Image aushängen: sudo umount /mnt/raspbian

Anstelle des Editors „vi“ können natürlich andere Editoren gewählt werden, wie z.B. „nano“. Da wir uns hier nicht auf die Benutzung der Editoren konzentrieren, solltet ihr einen euch bekannten Editor nutzen.

Virtualisierter RPi: Aufsetzen von QEMU / Raspbian (4)

8. Einrichtung der Entwicklungsumgebung

- Emulation mit QEMU starten
 - qemu-system-arm -kernel /home/<user_name>/qemu_vms/kernel-qemu-4.4.34-jessie -show-cursor -cpu arm1176 -m 256 -M versatilepb -serial stdio -append "root=/dev/sda2 rootfstype=ext4 rw" -hda /home/<user_name>/qemu_vms/<raspbian-downloaded>.img -redir tcp:5022::22 -no-reboot
- SSH-Service auf dem Raspberry Pi
 - Terminal öffnen und SSH-Service starten
 - sudo service ssh start
 - sudo update-rc.d ssh enable
 - in Linux-Mint via SSH auf den Raspberry Pi zugreifen
 - ssh pi@127.0.0.1 -p 5022 (default password: raspberry)



Falls kein Mauszeiger erscheint, kann versucht werden das Menü mithilfe der „Win“-Taste (bitte bei einem Mac-System selbst die Taste herausfinden) zu öffnen.

Das Update der „update-rc.d“ dient lediglich dazu um sicherzustellen, dass der SSH-Service nach jedem Neustart der Emulation wieder aktiv ist.

Virtualisierter RPi: Aufsetzen von QEMU / Raspbian (5)

8. Einrichtung der Entwicklungsumgebung

- innerhalb der QEMU Emulation den Raspberry Pi konfigurieren
 - sudo raspi-config
 - In diesem Konfigurationsmenü können viele nützliche Dinge, wie z.B. die Zeitzone, das Tastaturlayout, etc. angepasst werden. Dies sollte entsprechend durchgeführt werden.
 - Wichtig: Hier kann unter Menüpunkt 3 (Boot Options) —> B1 (Desktop / CLI) —> B2 (Console Autologin) die GUI deaktiviert werden. Die GUI bremst das System nur unnötig aus, wir benötigen lediglich das Terminal / die Konsole für das systemnahe Programmieren.
 - QEMU Emulation (Raspberry Pi) beenden
- Größe des Raspbian-Images / der QEMU Emulation anpassen
 - Schritte in Linux-Mint durchführen
 - ohne Größenanpassung —> voller Speicher des Raspbian-Systems
 - Home-Verzeichnis betreten: cd /home/<user_name>/qemu_vms
 - Kopie des Images erstellen: cp <raspbian-downloaded>.img raspbian.img
 - Größe anpassen: qemu-img resize raspbian.img +6G

Sollte der Befehl „qemu-img“ fehlschlagen, so müssen ggf. via „sudo apt-get install qemu-utils“ weitere, notwendigen Pakete installiert werden.
Die QEMU Emulation kann mit dem Befehl: „sudo shutdown -h now“ beendet werden.

Virtualisierter RPi: Aufsetzen von QEMU / Raspbian (6)

8. Einrichtung der Entwicklungsumgebung

- QEMU Emulation mit erweitertem Image starten
 - sudo qemu-system-arm -kernel /home/<user_name>/qemu_vms/kernel-qemu-4.4.34-jessie -show-cursor -cpu arm1176 -m 256 -M versatilepb -serial stdio -append "root=/dev/sda2 rootfstype=ext4 rw" -hda /home/<user_name>/qemu_vms/<raspbian-downloaded>.img -redir tcp:5022::22 -no-reboot -hdb raspbian.img
- Fortsetzung: Größe des Raspbian-Images / der QEMU Emulation anpassen
 - Partition überarbeiten: sudo cfdisk /dev/sdb
 - Größe anpassen: sudo resize2fs /dev/sdb2
 - Partition überprüfen: sudo fsck -f /dev/sdb2
 - System herunterfahren: sudo shutdown -h now

In dem Schritt „Partition überarbeiten“ muss zunächst die Partition „sdb2“ gelöscht und danach eine neue Partition mit dem gesamten verfügbaren Speicherplatz erstellt werden (primary). Danach kann die Änderung mit „Write“ auf die Festplatte geschrieben und im Anschluss das Tool mittels „Quit“ beendet werden.

Virtualisierter RPi: Aufsetzen von QEMU / Raspbian (7)

8. Einrichtung der Entwicklungsumgebung

- QEMU Emulation mit korrigiertem Image starten
 - sudo qemu-system-arm -kernel /home/<user_name>/qemu_vms/kernel-qemu-4.4.34-jessie -show-cursor -cpu arm1176 -m 256 -M versatilepb -serial stdio -append "root=/dev/sda2 rootfstype=ext4 rw" -hda /home/<user_name>/qemu_vms/raspbian.img -redir tcp:5022::22 -no-reboot
- Festplattengröße der Emulation überprüfen
 - Speicherplatz prüfen: df -h

Hinweis nochmal durchführen nach reboot!
Größe anpassen: sudo resize2fs /dev/sda2

Optimierung der Netzwerkschnittstellen

8. Einrichtung der Entwicklungsumgebung

- Zugriff auf alle Ports erhalten
 - auf dem Host (Linux Mint):
 - notwendigen Pakete installieren: `sudo apt-get install uml-utilities`
 - tunctl konfigurieren: `sudo tunctl -t tap0 -u <user>`
 - Shared Network Interface einrichten: `sudo ifconfig tap0 172.16.0.1/24`
 - Shared Network Interface prüfen: `ifconfig tap0`
 - QEMU Emulation mit Shared Network Interface starten
 - `sudo qemu-system-arm -kernel /home/<user_name>/qemu_vms/kernel-qemu-4.4.34-jessie -show-cursor -cpu arm1176 -m 256 -M versatilepb -serial stdio -append "root=/dev/sda2 rootfstype=ext4 rw" -hda /home/<user_name>/qemu_vms/raspbian.img -net nic -net tap,ifname=tap0,script=no,downscript=no -no-reboot`
 - In der RPi Emulation muss das Netzwerk konfiguriert werden:
 - `sudo ifconfig eth0 172.16.0.2/24`
 - Sollte diese Konfiguration nach einem Neustart verloren gegangen sein, so ist diese noch permanent in den Netzwerk-Konfigurationen der RPi-Emulation zu hinterlegen.

Ergänzungen zu GDB (optional)

8. Einrichtung der Entwicklungsumgebung

- innerhalb der Raspberry Pi Virtualisierung durchzuführen
- Version >7.7 sinnvoll
 - installierte Version herausfinden
 - GDB Version prüfen: `gdb --version`
 - manuelle Installation einer neueren Version:
 - Temp-Verzeichnis betreten: `cd /tmp`
 - Download der Version: `wget https://ftp.gnu.org/gnu/gdb/gdb-X.XX.tar.gz`
 - Download entpacken: `tar vxzf gdb-X.XX.tar.gz`
 - Ordner betreten: `cd gdb-X.XX`
 - Quellen aktualisieren: `sudo apt-get update`
 - notwendige Pakete installieren: `sudo apt-get install libreadline-dev python-dev texinfo -y`
 - GDB-Paket konfigurieren: `./configure --prefix=/usr --with-system-readline --with-python && make -j4`
 - GDB installieren: `sudo make -j4 -C gdb/ install`
 - GDB Version prüfen: `gdb --version`

Achtung: Der Schritt, welcher die Konfiguration des GDB-Paketes auslöst, hat, aufgrund der Virtualisierung, ca. 12h in Anspruch genommen. Dieser sollte also gewählt und ggf. über Nacht angestoßen werden. In der hier vorgestellten virtuellen Maschine wurde Version „gdb-7.12“ verwendet.

Erweiterung von GDB durch GEF (optional)

8. Einrichtung der Entwicklungsumgebung

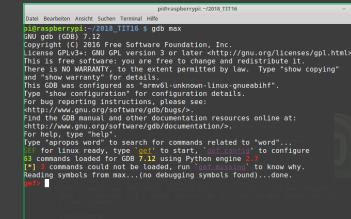
- GEF (GDB Enhanced Features)
 - erweitert GDB um nützliche Tools / Commands
 - verbessert die Darstellung
 - erweiterbar durch GDB-Python-API

• Installation via Skript

- Home-Verzeichnis betreten: cd /home/pi/
- Skript-Download & Ausführung:
wget -q -O- https://github.com/hugsy/gef/raw/master/scripts/gef.sh | sh

• Starten von GDB zeigt nun GEF in der Konsole

- „**gef>**“ statt „**gdb>**“ vor Eingabe-Cursor



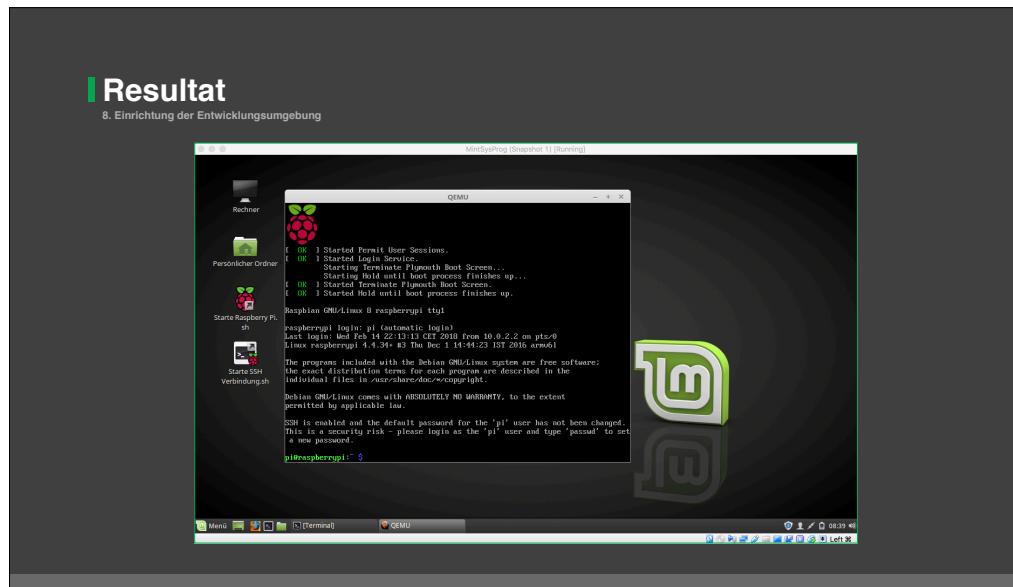
```
pi@raspberrypi: ~ $ 2018.11.16 $ gdb max
GNU gdb (GDB) 7.12
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change it and/or redistribute it
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "armv6l-unknown-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
For Linux ready, type "gdb" to start.
* ( gef ) gef: Python engine 1.1
[*] 3 commands could not be loaded, run "gef help" to know why.
Reading symbols from max... (no debugging symbols found)...done.
gef >
```

QED

Sollten beim Starten von „gdb“ bzw. „gef“ noch Elemente als „Missing“ deklariert sein, so können diese mit dem Befehl „gef missing“ aufgelistet werden. Zur Installation der fehlenden Pakete muss der Debugger verlassen werden (mittels Befehl „quit“). Danach können die vorgeschlagenen Befehle zur Nachinstallation im Terminal ausgeführt werden.

Resultat

8. Einrichtung der Entwicklungsumgebung



Die virtuelle Maschine kann mit dem gesamten Setup bereitgestellt werden. Dies erleichtert den Einstieg ungemein, dennoch ist es natürlich vorteilhaft ein derartiges System selbst aufsetzen zu können.

Die auf dem Desktop erkennbaren Links sind im Hintergrund verknüpft mit zwei Shell-Skripten. Ein Shell-Skript startet die QEMU Raspberry Pi Emulation, wodurch nicht jedesmal erneut das gezeigte, lange Kommando eingegeben werden muss. Die zweite Verknüpfung startet mittels Skript die SSH-Verbindung zum Raspberry Pi und sollte sinnvollerweise erst gestartet werden, wenn dieser auch betriebsbereit ist.

Eclipse Projekt einrichten

8. Einrichtung der Entwicklungsumgebung

- innerhalb der Linux-Virtualisierung
 - bitte zunächst den gewünschten Ordnerpfad anlegen
 - vorbereitete Tools zur Verwendung herunterladen:
 - git clone <https://github.com/raspberrypi/tools.git> -depth=1
 - eclipse C/C++ herunterladen, entpacken und ausführen
 - Zusatzpakete inst.: Help → Eclipse Marketplace → TM Terminal
 - eclipse Projekt anlegen
 - File → New → C/C++ → C Project → Executable Project → Empty Project (Cross GCC Toolchain)
 - Toolchain angeben (Pfad und Prefix wie oberster Bildausschnitt
 - Projekteinstellungen (siehe Bilder rechts) vornehmen

The image contains three separate windows from the Eclipse IDE:

- Properties for MINTED.sproj:** Shows the "Build" tab with configurations for "Debug" and "Release". It lists "Toolchain" (arm-none-linux-gnueabihf), "Linker" (ld), and "Optimizer" (none).
- Properties for MINTED.sproj:** Shows the "File Types" tab with "Assembly Source File" selected.
- Properties for MINTED.sproj:** Shows the "Settings" tab with "Toolchain" set to "arm-none-linux-gnueabihf" and "Linker" set to "ld".

Das “-depth=1” bewirkt, dass nur die letzte Version heruntergeladen wird.

Quelle: <https://sebastianfoerster86.wordpress.com/2015/06/13/remote-debugging-mit-eclipse/>
 Quelle: <https://mcuoneclipse.com/2016/05/05/assembly-files-in-eclipse-cdt-projects/>

Eclipse Debugging einrichten
8. Einrichtung der Entwicklungsumgebung

- Debugging einrichten
 - einen sinnvollen Namen vergeben
 - Debugging einrichten nachdem der erste Build erfolgreich war
 - Run → Debug Configurations...
 - Doppelklick auf „C/C++ Remote Application“
 - Neue Remote-Verbindung: „Connection → New“ (siehe kleines Bild)
 - Netzwerk / Login-Daten des Raspberry Pi oder der Raspberry Pi Virtualisierung eingeben
 - sonstige Debug-Einstellungen (siehe Bilder rechts) vornehmen
- auf dem Raspberry Pi:
 - GDB-Server installieren: `sudo apt-get install gdbserver`
 - GDB-Server stoppen: `sudo pkill gdbserver`

Quelle: <https://sebastianfoerster86.wordpress.com/2015/06/13/remote-debugging-mit-eclipse/>

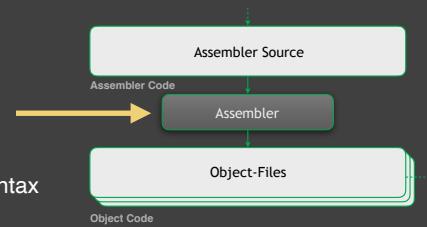
9

**ARM Assembler &
GNU Toolchain**

GNU Assembler (GAS)

9. ARM Assembler & GNU Toolchain

- Assembler
 - einfach portierbar (Plattformverfügbarkeit)
 - konfigurierbar / anpassbar
 - einfache & allgemeine Syntax
 - nicht auf eine Architektur beschränkt
 - leichte Unterschiede zur ARM-Developer-Syntax
 - dennoch gleiche Fähigkeiten



- Assembler Programm besteht aus vier Grundelementen:
 - assembler directives
 - labels
 - assembly instructions
 - comments

Klassische Struktur eines Assembler Programms

„Hello World“ Programm in C Code

```

1 #include <stdio.h>
2 int main() {
3     printf("Hello World\n");
4     return 0;
5 }

```

„Hello World“ Programm in ARM Assembler Code

```

1      .data
2 str:  .asciz  "Hello World\n" @ define a string
3
4      .text
5      .globl main
6      /* Beginning of the main-Function. */
7 main:  stmfd   sp!,{lr}    @ push return address on stack
8         ldr    r0, =str      @ load pointer to format string
9         bl     printf       @ printf("Hello World\n");
10        mov   r0, #0        @ move return code into r0
11        ldmfd  sp!,{lr}    @ pop return address from stack
12        mov   pc, lr       @ return from main

```

Quelle: Larry D. Pyeatt: Modern Assembly Language Programming with the ARM Processor, Newnes, 1. Auflage 2016

Labels:

Sind sogenannte „BESCHRIFTUNGEN“ oder „ETIKETTEN“ die eine bestimmte Programmstelle markieren. Sie können überall im Code eingesetzt werden um auf Daten, eine Funktion oder einen Codeblock zu verweisen. In der GNU-Syntax enden sie immer mit einem Doppelpunkt („::“).

Comments:

Hier wird zwischen zwei Typen unterschieden:

- Multi-Line: Sie starten wie z.B. in C/C++ mit einem „/**“ und enden mit einem „*/“. Alle Zeichen, die sich zwischen den beiden Symbolen befinden werden als Kommentar interpretiert und vom Assembler ignoriert.
- Single-Line: Sie starten mit einem „@“ oder einem „//“ und enden mit der aktuellen Zeile. Endet ein Assembler-Programm mit „.s“ statt „.S“ (großgeschrieben) so ist die Verwendung „//“ nicht erlaubt.

Assembler Directives:

Assemblerdirektiven sind keine Befehle oder Daten, es sind Anweisungen für den Assembler selbst. Beispiel hierfür sind z.B. Macros, das Einbinden von Dateien, das Reservieren von Speicher innerhalb einer gewünschten/definierten Region und viele weitere. Alle Direktiven starten mit einem „.“, der von einer Folge von (üblicherweise) Kleinbuchstaben ergänzt wird. Eine komplette Dokumentation aller Direktiven ist in dem Paket „GNU Binutils“ zu finden.

Assembly Instructions:

Sind die Maschinenbefehle, die von der CPU ausgeführt werden.

„hello.s“ unter der Lupe																																																																							
8. ARM Assembler & GNU Toolchain																																																																							
<table border="1"> <thead> <tr> <th>Addr</th><th>Code</th><th>Label</th><th>Source</th></tr> </thead> <tbody> <tr><td>1</td><td></td><td></td><td>.data</td></tr> <tr><td>2</td><td>0000 48E656C6C</td><td>str:</td><td>.asciz "Hello World\n" @ define a string</td></tr> <tr><td>2</td><td>6F20576F</td><td></td><td></td></tr> <tr><td>2</td><td>726C640A</td><td></td><td></td></tr> <tr><td>2</td><td>00</td><td></td><td></td></tr> <tr><td>3</td><td></td><td></td><td></td></tr> <tr><td>4</td><td></td><td></td><td>.text</td></tr> <tr><td>5</td><td></td><td></td><td>.globl main</td></tr> <tr><td>6</td><td></td><td></td><td>/* Beginning of the main-Function. */</td></tr> <tr><td>7</td><td>0000 00402DE9</td><td>main:</td><td>stmfd sp!,{lr} @ push return address on stack</td></tr> <tr><td>8</td><td>0004 0C009FE5</td><td></td><td>ldr r0, =str @ load pointer to format string</td></tr> <tr><td>9</td><td>0008 FEF7FFEB</td><td></td><td>bl printf @ printf(„Hello World!\n“);</td></tr> <tr><td>10</td><td>000C 0000A0E3</td><td></td><td>mov r0, #0 @ move return code into r0</td></tr> <tr><td>11</td><td>0010 00408D8E</td><td></td><td>ldmfd sp!,{lr} @ pop return address from stack</td></tr> <tr><td>12</td><td>0014 0EF0A0E1</td><td></td><td>mov pc, lr @ return from main</td></tr> <tr><td>12</td><td>00000000</td><td></td><td></td></tr> </tbody> </table>				Addr	Code	Label	Source	1			.data	2	0000 48E656C6C	str:	.asciz "Hello World\n" @ define a string	2	6F20576F			2	726C640A			2	00			3				4			.text	5			.globl main	6			/* Beginning of the main-Function. */	7	0000 00402DE9	main:	stmfd sp!,{lr} @ push return address on stack	8	0004 0C009FE5		ldr r0, =str @ load pointer to format string	9	0008 FEF7FFEB		bl printf @ printf(„Hello World!\n“);	10	000C 0000A0E3		mov r0, #0 @ move return code into r0	11	0010 00408D8E		ldmfd sp!,{lr} @ pop return address from stack	12	0014 0EF0A0E1		mov pc, lr @ return from main	12	00000000		
Addr	Code	Label	Source																																																																				
1			.data																																																																				
2	0000 48E656C6C	str:	.asciz "Hello World\n" @ define a string																																																																				
2	6F20576F																																																																						
2	726C640A																																																																						
2	00																																																																						
3																																																																							
4			.text																																																																				
5			.globl main																																																																				
6			/* Beginning of the main-Function. */																																																																				
7	0000 00402DE9	main:	stmfd sp!,{lr} @ push return address on stack																																																																				
8	0004 0C009FE5		ldr r0, =str @ load pointer to format string																																																																				
9	0008 FEF7FFEB		bl printf @ printf(„Hello World!\n“);																																																																				
10	000C 0000A0E3		mov r0, #0 @ move return code into r0																																																																				
11	0010 00408D8E		ldmfd sp!,{lr} @ pop return address from stack																																																																				
12	0014 0EF0A0E1		mov pc, lr @ return from main																																																																				
12	00000000																																																																						
<p>Seite 1</p> <table border="1"> <thead> <tr> <th>DEFINED SYMBOLS</th> </tr> </thead> <tbody> <tr><td>hello.s:2 .data:0000000000000000 str</td></tr> <tr><td>hello.s:9 .text:0000000000000000 main</td></tr> <tr><td>hello.s:9 .text:0000000000000000 \$a</td></tr> <tr><td>hello.s:14 .text:0000000000000018 \$d</td></tr> </tbody> </table> <p>UNDEFINED SYMBOLS</p> <p>printf</p>			DEFINED SYMBOLS	hello.s:2 .data:0000000000000000 str	hello.s:9 .text:0000000000000000 main	hello.s:9 .text:0000000000000000 \$a	hello.s:14 .text:0000000000000018 \$d	Seite 2																																																															
DEFINED SYMBOLS																																																																							
hello.s:2 .data:0000000000000000 str																																																																							
hello.s:9 .text:0000000000000000 main																																																																							
hello.s:9 .text:0000000000000000 \$a																																																																							
hello.s:14 .text:0000000000000018 \$d																																																																							

Der GAS konvertiert den String aus Zeile 2 in seine binäre Darstellung. Wird eine ASCII-Tabelle zur Hand genommen, so ist leicht erkennbar, dass die hexadezimale „48“ das „H“ als Zeichen repräsentiert. Das erste Zeichen wird in der „.data“-Sektion des Programms an Adresse 0x0000 gespeichert.

Ab der Zeile 4 wechselt GAS in die „.text“-Sektion und beginnt die Maschinenbefehle in ihre binäre Form zu wandeln. Auch hier landet der erste Befehl an der Adresse 0x0000, jedoch wie bereits erwähnt in der „.text“-Sektion, welche eine andere Sektion als die „.data“-Sektion ist und dies somit keinen Konflikt darstellt.

Dieses Ergebnis speichert der GAS in einem speziellen Format, der „object“-Datei, welche die Dateiendung „.o“ besitzt. Es ist auf der zweiten Seite auffällig, dass der Assembler „printf“ nicht finden kann. Darum kümmert sich im späteren der Linker.

Erstellung eines ausführbaren Programms („main“)

9. ARM Assembler & GNU Toolchain

• 1. Möglichkeit: GCC (GNU C Compiler)

```
1 gcc -o hello hello.s
```

- „hello.s“ als Quelle (in-file)
 - erkennt „s“ und „S“ automatisch
 - gibt diesen Dateityp an GAS (Assembler) weiter → erzeugt object-Datei (Endung „o“)
 - erzeugt daraus das Programm „hello“ („o“ = out-file)
 - automatisch mit Hilfe des Linkers → nimmt Standardbibliotheken hinzu
 - C-Standardbibliotheken erwarten „main“-Funktion

• 2. Möglichkeit: GAS + Linker

```
1 as -o hello.o hello.S  
2 ld -o hello -dynamic-linker /lib/ld-linux.so.2 ... -lc hello.o /usr/lib/crt0.o ...
```

- erzeugt ebenfalls das Programm „hello“
 - nicht automatisch durch GCC
 - größeren Einfluss auf die einzelnen Schritte
 - Detailangaben für Linker erforderlich

Der nächste Schritt, um ein ausführbares Programm zu erhalten, wäre es also den Linker zu starten, welcher die erzeugte „hello.o“ mit der C Standard Bibliothek verknüpft. Diese beinhaltet nämlich die fehlende Funktion „printf“. Der Linker sorgt ebenfalls dafür, dass notwendiger „Start-Up“-Code, welcher für den Einstieg in die „main“ notwendig ist mit dem Programm verknüpft.

Programmstart „main“ versus „_start“

9. ARM Assembler & GNU Toolchain

- „main“
 - eine Funktion / logischer Beginn eines C-Programms
 - nach Ende der Funktion wird zurückgesprungen
 - verwendet C-Standardbibliotheken (kümmern sich um Plattformspezifika)
- „_start“
 - keine Funktion / kein Aufruf —> nur ein Einsprungspunkt —> kein Zurück
 - verwendet keine C-Standardbibliotheken für den Aufruf / Rücksprung
 - Beginn eines Prozesses —> Kernel springt unverzüglich dorthin
 - spezifisch für Linux

```
1 .text
2 .global main
3 main: ...
        @ do something
```

```
1 .text
2 .global _start
3 _start: ...
        @ do something
```

Die Direktiven „global“ und „globl“ sind identisch, trotz unterschiedlicher Schreibweise.

Soll im Fall von „_start“ das Programm sauber beendet werden, muss dies dem Kernel mitgeteilt werden. Daher muss der entsprechende Syscall durch (ggf. mehrere) Maschinenbefehle implementiert werden.

Linux: _start vs. main

<https://www.uninformativ.de/blog/postings/2011-03-31/0/POSTING-de.html>

Bei Linux sind alle Bindings vorhanden und „_start“ ist bereits (durch Linux selbst) definiert. Daher kommt es, bei Verwendung von „_start“ als Einstiegspunkt in das eigene Programm, zur Fehlermeldung, dass „_start“ bereits definiert wurde. Hier kann entsprechend auf „main“ ausgewichen werden. Da in einem Bare-Metal-System kein Linux-Betriebssystem verwendet wird, sollte hier auf „_start“ zurückgegriffen werden. Zu dieser Systemart später mehr.

Erstellung eines ausführbaren Programms („`_start`“)

9. ARM Assembler & GNU Toolchain

- 1. Möglichkeit: GCC (GNU C Compiler)

```
1 gcc -nostdlib -o example example.s
```

- „`example.s`“ als Quelle (in-file)
 - erkennt „`s`“ und „`S`“ automatisch
 - gibt diesen Dateityp an GAS (Assembler) weiter → erzeugt *object-Datei* (Endung „`.o`“)
 - erzeugt daraus das Programm „`example`“ („`-o`“ = out-file)
 - automatisch mit Hilfe des Linkers → aber es werden keine Standardbibliotheken hinzugenommen
 - durch „`-nostdlib`“ wird **keine** „`main`“-Funktion erwartet

- 2. Möglichkeit: GAS + Linker

```
1 as -o example.o example.s  
2 ld -o example example.o
```

- erzeugt ebenfalls das Programm „`example`“
 - nicht automatisch durch GCC
 - größeren Einfluss auf die einzelnen Schritte
 - **keine** zusätzlichen Angaben für Linker erforderlich (Programmabhängig)

Die Erzeugung eines ausführbaren Programms, welches als Programmstart „`_start`“ besitzt ist hier zum Vergleich abgebildet.

■ Dateiendung „.s“ versus „.S“

9. ARM Assembler & GNU Toolchain

- „.s“ (lowercase)
 - identifiziert Assembler-Code der **keine Vorverarbeitung** (Preprocessing) benötigt
 - üblich bei erzeugtem Assembler-Code (z.B. durch Compiler)
 - kann direkt an den Assembler weitergegeben werden
- „.S“ (uppercase)
 - identifiziert Assembler-Code der eine **Vorverarbeitung** (Preprocessing) benötigt
 - üblich bei handgeschriebenem Assembler-Code
- **Achtung:**
 - Eine Vorverarbeitung durch den C-Preprocessor ist dann notwendig, wenn der Assembler-Code Direktiven aus C enthält. Diese sind z.B. „#include<...>“, „#define“, „ifdef“ und C-Style-Kommentare (z.B. „//“).

Direktiven - Sections

9. ARM Assembler & GNU Toolchain

- unterschiedliche Sektionen in der Programmdatei
 - für Daten und Instruktionen
 - haben eigene Adresszähler
 - weiter unterteilbar —> nummerierte Subsections
- durch Direktiven selektiert
 - Instruktionen und Daten landen in der selektierten Sektion
 - bis zur erneuten Selektion
 - ein Label bekommt den Adresszählerwert der aktuellen Sektion

Direktiven - Sections

9. ARM Assembler & GNU Toolchain

- „**.data subsection**“

- Assembler platziert Instruktionen / Daten in der gewählten Data-Subsection
- falls keine „subsection“ angegeben ist → Default: 0
- üblich für:
 - globale Variablen
 - Konstanten mit Labels

- „**.text subsection**“

- Assembler platziert nachfolgende Angaben in der gewählten Text-Subsection
- falls keine „subsection“ angegeben ist → Default: 0
- üblich für:
 - Instruktionen
 - Konstanten

Direktiven - Sections

9. ARM Assembler & GNU Toolchain

- „**.bss subsection**“
 - bss = Block Started by Symbol
 - definiert Datenspeicherblöcke, die bei Programmstart mit 0 initialisiert werden sollen
 - Assembler platziert nachfolgende Angaben in der gewählten bss-Subsection
 - falls keine „subsection“ angegeben ist → Default: 0
 - üblich für:
 - globale Variablen die mit 0 initialisiert werden müssen
 - statische Variablen
 - verbraucht keinen Platz im Object- oder Executable-File
 - sorgt nur dafür, dass beim Laden des Programms zusätzlicher Speicher reserviert wird
 - somit existiert die Section erst bei Programmausführung
- „**.section name**“
 - eigene Sektionen
 - müssen dem Linker mitgeteilt werden

Der Vorteil bei der Verwendung der bss-Section ist, dass so die Programmgröße reduziert werden kann und dadurch ein schnelleres Laden des Programms erzielbar ist.

Direktiven - Allocating Space

9. ARM Assembler & GNU Toolchain

- Speicherzuweisung für statische Variablen und Konstanten
 - optional: inklusive Initialisierung mit Hilfe von „expressions“
 - Expressions sind durch Komma „,“ separiert
 - Fall: keine Expression übergeben
 - erhöht Adresszähler nicht
 - kein Speicher wird reserviert
- „.byte expression(s)“
 - z.B.:

	Addr	Code	Label	Source
1	0000	414200	ch:	.byte 'A', 'B', 0
 - erwartet keine oder mehrere Expressions
 - 1 Expression = 8 Bit (Byte)
 - jede Expression wird im darauffolgenden Byte abgelegt

Expressions können simple Integer sein aber auch C-Style-Expressions.

Direktiven - Allocating Space

9. ARM Assembler & GNU Toolchain

- „.hword expression(s)“ oder „.sword expression(s)“

- beide beim ARM-Prozessor exakt gleich

• z.B.:

	Addr	Code	Label	Source
1	0000	01000200	i:	.hword 1,2

- erwartet keine oder mehrere Expressions

- 1 Expression = 16 Bit (half-word; 2 Bytes)

- „.word expression(s)“ oder „.long expression(s)“

- beide beim ARM-Prozessor exakt gleich

• z.B.:

	Addr	Code	Label	Source
1	0000	01000000	j:	.word 1,2
1		02000000		

- erwartet keine oder mehrere Expressions

- 1 Expression = 32 Bit (word; 4 Bytes)

Achtung:

Warum liegt die „01“ bei dem „.word“ ganz vorne und wir lesen nicht „00000001“?

„01“ befindet sich an der Adresse 0000 und damit an der niederwertigsten Stelle. Da der Prozessor vier Bytes verarbeitet und das höchstwertige Byte im Beispiel die Adresse 0003 hat, ist die Darstellung korrekt. D.h. der Prozessor verarbeitet exakt den gewünschten Wert „00 00 00 01“.

Direktiven - Allocating Space

9. ARM Assembler & GNU Toolchain

- „.ascii "string"(,"string",...)“

• z.B.:

	Addr	Code	Label	Source
1	0000	48E56C6C	fmt:	.ascii „Hello\n”
1		6F0A		

- erwartet keine oder mehrere Zeichenketten

- jeder Character im String ist 8 Byte groß (ASCII)

- „.asciz "string"(,"string",...)“ oder „.string "string"(,...)“

• beide beim ARM-Prozessor exakt gleich

• z.B.:

	Addr	Code	Label	Source
1	0000	48E56C6C	fmt2:	.asciz „Hello\n”
1		6F0A00		

- erwartet keine oder mehrere Expressions

- jeder Character im String ist 8 Byte groß (ASCII)

- fügt jedem String eine ASCII NULL am Ende hinzu

Das „z“ in „.asciz“ steht für „zero“, also der angehängten ASCII NULL. Warum ist diese ASCII NULL notwendig? Sie terminiert den String und hilft so das Ende eines Strings zu bestimmen. Natürlich lässt sich damit auf einfache Art (durch Zählen) die Anzahl der sich im String befindenden Character ermitteln.

Direktiven - Allocating Space

9. ARM Assembler & GNU Toolchain

- „.float num(,num,...)“ oder „.single num(,num,...)“
 - beide beim ARM-Prozessor exakt gleich

• z.B.:	Addr	Code	Label	Source
	1	0000 ????????	n:	.float 1.222

- erwartet keine oder mehrere Floating-Point-Zahlen
 - 1 Expression = 32 Bit (single precision nach IEEE; 4 Bytes)

• „ .double num(,num,...) “

• z.B.:	Addr	Code	Label	Source
	1	0000 ????????	m:	.double 1.222

- erwartet keine oder mehrere Floating-Point-Zahlen
 - 1 Expression = 64 Bit (double precision nach IEEE; 8 Bytes)



Quelle: https://en.m.wikipedia.org/wiki/Single-precision_floating-point_format

Wie genau ein Gleitkommawert für die Abbildung im Speicher berechnet und repräsentiert wird soll hier nicht beschrieben werden. Hierzu bitte weitere Literatur konsultieren.

Direktiven - Filling and Alignment

9. ARM Assembler & GNU Toolchain

- aufgrund zuvor diskutiertem Memory-Alignment notwendig
- Zugriff auf Mis-Aligned/Unaligned Data benötigt mehr Zeit
- Direktiven zur Sicherstellung des Memory-Alignments

Direktiven - Filling and Alignment

9. ARM Assembler & GNU Toolchain

- „.align abs-expr, abs-expr, abs-expr“
 - passt den Location Counter an das gewünschte Alignment an
 - erste Expression:
 - spezifiziert die Anzahl der low-order Zero Bits die der Location Counters nach dem Vorrücken haben muss
 - z.B. „.align 3“ → falls notwendig: Hinzufügen von Füll-Bytes (Padding) bis der Location Counter ein Vielfaches von acht ist.
 - zweite Expression (optional):
 - Wert mit dem aufgefüllt wird / Wert der in den Padding Bytes gespeichert wird
 - falls keine Angabe erfolgt → Standardwerte:
 - Data-Sektion: 0
 - Code-Sektion: NOP-Befehl
 - dritte Expression (optional):
 - maximale Anzahl der Bytes die übersprungen (aufgefüllt) werden sollen
 - falls das Durchführen des Alignments mehr Bytes beansprucht → keine Durchführung des Alignments
 - die zweite Expression kann durch zwei Kommas übersprungen werden
 - z.B. „.align 3,,7“

„abs-expr“ steht für absolute Expression, also eine Ganzzahl wie z.B. 4 oder ein Hexadezimal-Wert wie z.B. 0x04.

Der Location Counter ist die aktuelle Position im Adressraum der aktuellen Sektion. Oder anders gesagt: der Adresszähler der Sektion.

Erklärung zu low-order Zero Bits:

Angenommen wir wählen den Wert 2, so müssen innerhalb der Adresse immer die niederwertigsten zwei Bit auf 0 bleiben.

Einige Beispiele:

0000 0012 → nicht ok, da die 2 also das 2. Bit gesetzt ist ... hier muss aufgefüllt werden

0000 0004 → ok, da die 2 niederwertigsten Bits auf 0 gesetzt sind ... hier muss nicht aufgefüllt werden

NOP = No Operation

Achtung:

Bei manchen Systemen stellt die erste Expression die Anzahl der Bytes oder Words dar. Hier bei dem ARM-Prozessor ist es die Anzahl der Low-Order Bits. Bitte mit Vorsicht verwenden, falls die Software ggf. auf ein anderes System portiert werden soll oder dieses kleine Detail den Programmierern nicht bekannt ist. Diese Inkonsistenz beruht darauf, dass GAS die einzelnen Verhalten der nativen Assembler der Systeme emulieren muss.

Direktiven - Filling and Alignment

9. ARM Assembler & GNU Toolchain

- „.balign[w1] abs Expr, abs Expr, abs Expr“
 - passt den Location Counter an das gewünschte Alignment an
 - erste Expression:
 - Byte-Mehrfaches für das Speicheralignment
 - z.B. „.balign 4“ —> falls notwendig: Hinzufügen von Füll-Bytes (Padding) bis der Location Counter ein Vielfaches von vier ist.
 - zweite Expression (optional):
 - siehe „.align“
 - dritte Expression (optional):
 - siehe „.align“
 - Varianten:
 - „.balignw“: Füll-Pattern ist ein 2-Byte Word
 - z.B. „.balignw 4, 0x368d“ richtet sich an 4-Byte aus
 - werden dabei 2 Bytes übersprungen, so werden diese mit 0x368d aufgefüllt
 - werden dabei 1 oder 3 Byte(s) übersprungen, so ist der Füllwert undefiniert
 - „.balignl“: Füll-Pattern ist ein 4-Byte Word

ARM empfiehlt die Verwendung von „.balign“ anstelle von „.align“, da Letzteres von System zu System unterschiedlich sein kann. (Quelle: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0742b/chr1398259885678.html>)

Direktiven - Filling and Alignment

9. ARM Assembler & GNU Toolchain

- „**.skip size, fill**“ und „**.space size, fill**“
 - „**size**“
 - reserviert ein Speichergebiet von dieser Größe
 - „**fill**“ (optional)
 - initialisiert das Gebiet mit dem selben Wert
 - falls keine Angabe erfolgt —> Standardwerte:
 - Data-Sektion: 0

Direktiven - Filling and Alignment

9. ARM Assembler & GNU Toolchain

Unaligned Data

1	Addr	Code	Label	Source
2	0000	00000000	i:	.word 0
2	0004	01000000	j:	.word 1
2	0008	48656C6C	fmt:	.asciz "Hello\n"
2		6F0A00		
3	000F	414200	ch:	.byte 'A','B',0
4	0012	00000000	ary:	.word 0,1,2,3,4
5		01000000		
6		02000000		
7		03000000		
8		04000000		

0x12 = 18 → nicht ohne Rest durch 4 teilbar!

Aligned Data

1	Addr	Code	Label	Source
2	0000	00000000	i:	.word 0
2	0004	01000000	j:	.word 1
2	0008	48656C6C	fmt:	.asciz "Hello\n"
2		6F0A00		
3	000F	414200	ch:	.byte 'A','B',0
4	0012	0000		.align 2
5	0014	00000000	ary:	.word 0,1,2,3,4
6		01000000		
7		02000000		
8		03000000		
9		04000000		

- Es ist gute Praxis eine Alignment-Direktive nach der Verwendung von *chars (Bytes)* oder *half-words (2 Bytes)* zu verwenden.

Direktiven - Conditions and Include

9. ARM Assembler & GNU Toolchain

- Bedingte Assemblierung
 - „.if exp“ → wahr → Expression != 0
 - „.ifdef symbol“ → Assemblierung falls Symbol definiert wurde
 - „.ifndef symbol“ → Assemblierung falls Symbol nicht definiert wurde
 - „.else“ → falls vorherige if-Direktive nicht eintritt
 - „.endif“ → schließt die if-Direktiven ab
- Einbinden weiterer Quelldateien
 - „.include "file"“ → bindet das angegebene Source-File ein
 - Achtung:
 - Auf Pfade achten!
 - Pfade können mit dem GAS Parameter „-I“ gesteuert werden
 - sehr ähnlich zu C/C++

10

Hands On: ARM Assembler auf dem Übungsrechner

- Generelle Leseempfehlung:
- <http://bob.cs.sonoma.edu/IntroCompOrg-RPi/sec-using-book.html>

Sonderfälle berücksichtigen

10. Hands On: ARM Assembler auf dem Übungsrechner

- Befehle aktualisieren nicht immer die Flags
 - **MOV:** 1 MOV <Rd>, #expr or MOV <Rd>, <Rm>
 - legt #expr oder <Rm> in <Rd>
 - **MVN:** 1 MVN <Rd>, <Rm>
 - führt eine bitweise, logische NOT-Operation auf den Wert von <Rm> aus und legt das Ergebnis in <Rd> ab
 - **NEG:** 1 NEG <Rd>, <Rm>
 - führt eine Multiplikation mit -1 und dem Wert von <Rm> durch und legt das Ergebnis in <Rd> ab
 - für MOV (#expr), MVN und NEG müssen <Rm> in <Rd> im Bereich R0 bis R7 liegen
 - für MOV <Rd>, <Rm> können die Register R0 bis R15 genutzt werden
 - nur bei der Verwendung von R0 bis R7 werden die N, Z Flags aktualisiert und C, V Flags zurückgesetzt

Quelle: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/BABJAJIB.html>

MOV (Rd,#expr) und MVN Befehle aktualisieren die N und Z Flags. Sie berühren die C oder V Flags nicht.
NEG Befehle aktualisieren die N, Z, C, und V Flags.

Beispiele

```
MOV r3,#0
MOV r0,r12 ; does not update flags
MVN r7,r1
NEG r2,r2
```

Incorrect examples

```
MOV r2,#256 ; immediate value out of range (max. 8 Bit = 255)
MOV r8,#3 ; cannot move immediate to high register
MVN r8,r2 ; high registers not allowed with MVN or NEG
NEG r0,#3 ; immediate value not allowed with MVN or NEG
```

THANK YOU!



Vielen Dank für eure Aufmerksamkeit!