Zahlensysteme

$f(a,b,N) = \sum_{i=0}^{N-1} (a_i * b^i)$

Basis b -> Wertebereich jeder Ziffer

Ziffer -> Tupel mit Elemente a_i

N -> Anzahl der Stellen

a) Dezimalsystem

Basis $b = 10$

Ziffer $a_i \in \{0,1,2,3,4,5,6,7,8,9\}$

$2021_{10} = f((2,0,2,1),10,4) = \sum_{i=0}^{3} (a_i * 10^i)$

$= 2*10^3 + 0*10^2 + 2*10^1 + 1*10^0 = 2000 + 000 + 20 + 1 = 2021$

b) Binärsystem

Multiplikation

$0 \cdot 1 = 0$
$1 \cdot 0 = 0$
$0 \cdot 0 = 0$
$1 \cdot 1 = 1$

```
1 0 0 0 · 1 0 1 0
          0 0 0 0
        1 0 0 0
      0 0 0 0
    1 0 0 0
  1 0 1 0 0 0 0
```

Basis $b = 2$

Ziffer $a_i \in \{0,1\}$

$1111100101_2 = f((1,1,1,1,1,1,0,0,1,0,1),2,11) = \sum_{i=0}^{10} (a_i * 2^i)$

$= 1*2^{10} + \ldots + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

n Bits -> 2^n Zustände (Wertebereich von 0 bis $2^n - 1$)

c) Hexadezimalsystem

Basis $b = 16$ **Ziffer**

Ziffer $a_i \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$

$= \{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\}$

0x5AF3$_{16} = f(a,16,N) = \sum_{i=0}^{N-1} (a_i * 16^i)$

$= 5*16^3 + 10*16^2 + 15*16^1 + 3*16^0 = 23283_{10}$

ABER wie zurück umwandeln? $23283_{10} = \ ?_{16}$

$23283 : 16 = 1455$ **Rest 3**

$1455 : 16 = 90$ **Rest 15**

$90 : 16 = 5$ **Rest 10**

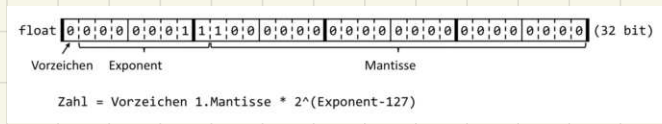
$5 : 16 = 0$ **Rest 5**

-> $5 - 10 - 15 - 3$ = **0x5AF3**$_{16}$

Festkommazahl-Darstellung
Vorteil: sehr verständlich, einfache Umrechnung
Nachteil: begrenzte Reichweite, unnötige Genauigkeit

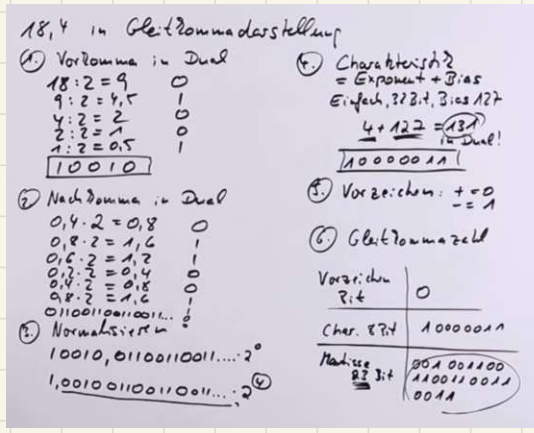


Gleit-/Fließkommazahl-Darstellung
Vorteil: große Reichweite
Nachteil: ungleichmäßige Verteilung



-> Binär-Darstellungsarten

Art	Beschreibung	Vor-/Nachteil	Beispiel
Vorzeichenbit Most Signifikant Bit (Hochwertiges Bit)	Ein extra Bit nur für das Vorzeichen "0" ⇒ + "1" ⇒ −	⊕ Analog zu Dezimalsystem ⊖ Doppelte Null { +0 = 0000, −0 = 1000 } ⊖ neue Arithmetik nötig	1010.1010
Einerkomplement	Alle Bits invertieren alle 0 ⇒ 1 alle 1 ⇒ 0	⊕ intuitiv ⊖ Doppelte Null { +0 = 0000, −0 = 1111 } ⊖ neue Arithmetik nötig	1101.0101
Zweierkomplement	Einerkomplement +1 Wertebereich -2^{n-1} bis $2^{n-1}-1$ 8 Bits { max = $01111111_2 = 127_{10}$, min = $10000000_2 = -128_{10}$ }	⊕ volle Ausnützung des Zahlenraums ⊕ Eindeutigkeit der Null ⊕ Arithmetik wieder verwendbar ⊖ Zahl nicht direkt ablesbar	1101.0110



-> Binärarithmetik

Subtraktion -> umgewandelt -> Addition von negativer Zahl im Zweierkomplement-Format

1) Als erstes wird das Zweierkomplement des Subtrahenden berechnet: $-(0010.1010)_2 = (1101.0110)_2$

2) Und dann auf den Minuenden addiert: $(0011.0011)_2 + (1101.0110)_2 = (0000.1001)_2$

Byte-Reihenfolge

24	16	8	0	
76	54	32	10	=0x76543210

Little Endian

N+3	76
N+2	54
N+1	32
Adresse N	10

Big Endian

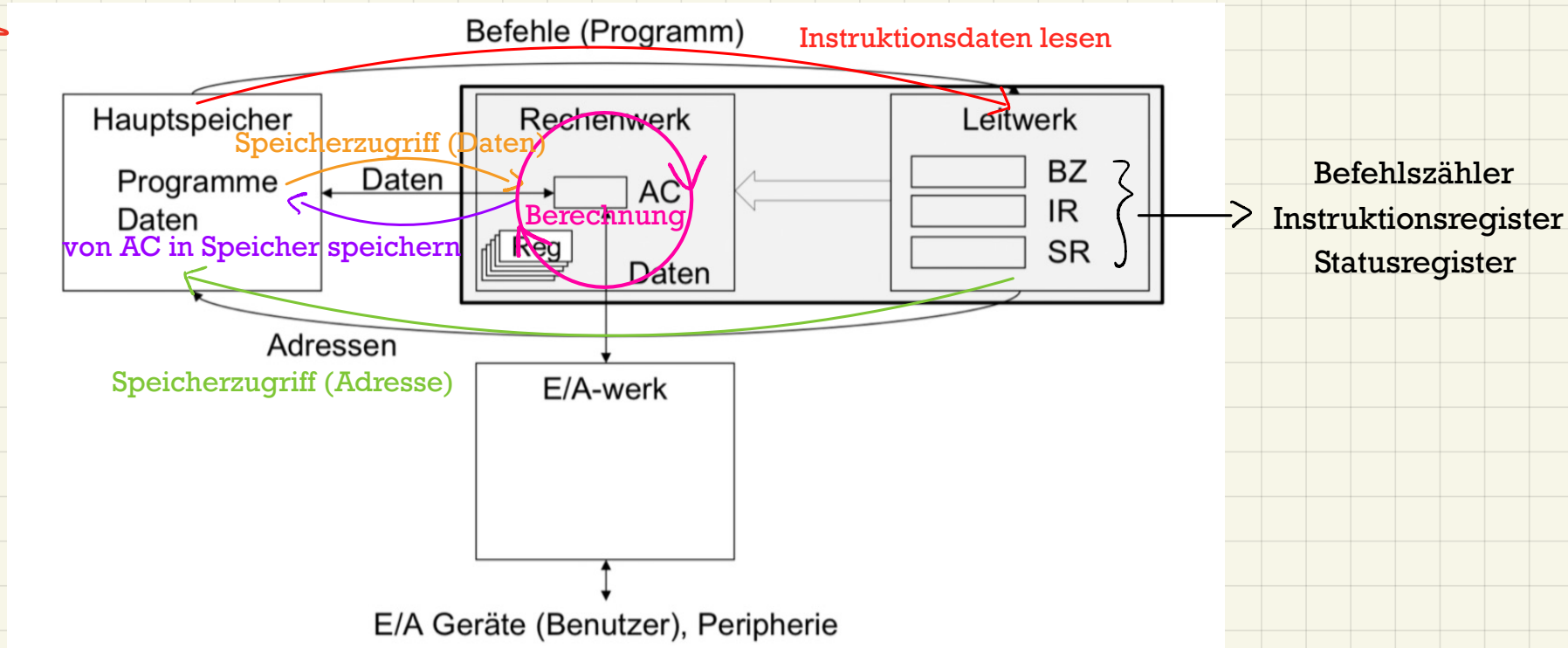
N+3	10
N+2	32
N+1	54
Adresse N	76

Speicherbedarf & Wertebereich

1 Byte = 8 Bits

Typname in C	Bits	Vorzeichen	min	max
char	8	ja	-128	127
unsigned char		nein	0	255
short	16	ja	-32.768	32.767
unsigned short		nein	0	65.535
int	32	ja	-2.147.483.648	2.147.483.647
unsigned int		nein	0	4.294.967.295
long long	64	ja	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
unsigned long long		nein	0	18.446.744.073.709.551.615

Von-Neumann Architekturkonzept und Übersicht



1. Rechnersruktur ist unabhängig vom bearbeiteten Problem (Programmsteuerung!)
2. Rechner besteht aus vier Werken - durch Bussystem kommunizieren
   - Rechenwerk (Arithmetic Logic Unit): arithmetische Operationen ausführen
   - Leitwerk (Control Unit): Daten lesen, Befehl ausführen, Daten schreiben
   - Speicherwerk (Memory Unit): Programme und Daten speichern; Random Access Memory (RAM) als Speicher für von Neumann; Jederzeit Zugriff auf jede Speicherzelle
   - Ein-/Ausgabewerk (I/O Unit): mit Umgebung kommunizieren (Tastatur, Lesegerät, Drucker, Magnetband)
3. Hauptspeicher besteht aus Zellen gleicher Größe - werden durch fortlaufende Nummern/Adressen bezeichnet
4. Programm und Daten stehen beide in Hauptspeicher und können durch die Maschine verändert werden
5. Die Maschine benutzt Binärcodes: Zahlen werden dual dargestellt
6. Das Programm besteht aus einer Folge von Befehlen (Instruktionen)
   - nacheinander gespeichert (aufsteigende Adressen)
   - nacheinander ausgeführt (sequentiell)
7. Von der Abfolge kann nur durch (un)bedingte Sprungbefehle abgewichen werden (Programmfortsetzung aus einer anderen Zelle)

Adresse und Datenzugriffe mit Addition (a, b) —> (a+b)

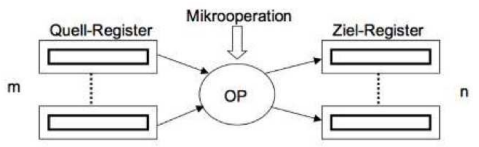
1. Natürliche Zahlen a und b liegen in Hauptspeicher vor
2. die erste Instruktion laden (-> Speicherzugriff der Adresse) und deren Instruktionsdaten lesen
   - Daten an Aderesse "a" laden (-> Speicherzugriff der Adresse und Daten)
3. die zweite Instruktion laden (-> Speicherzugriff der Adresse) und deren Instruktionsdaten lesen
   - Daten an Adresse "b" laden (-> Speicherzugriff der Adresse und Daten)
4. die dritte Instruktion laden (-> Speicherzugriff der Adresse) und deren Instruktionsdaten lesen
5. Berechnung - Instruktion ausführen (Addiere a und b und deren Ergebnis im AC (Akkumulator) speichern
6. Speichern von Register AC in Adresse "c" in Speicher

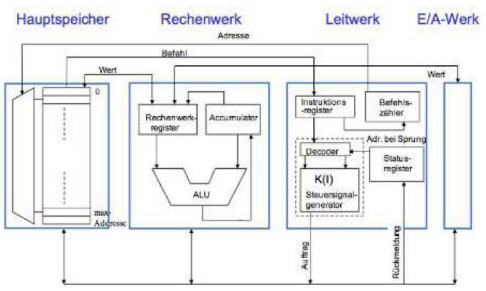
Funktionale Schichten

- Ein Computer ist in folgende Abstraktionsschichten aufgeteilt. Dabei benötigt eine gegebene Schicht die jeweils nächst untere.
- Benutzerprogramm Schicht
  - Es wird das vom Benutzer verwendete Programm in einer höheren Programmiersprache geschrieben
  - Meist selbst in weitere Schichten unterteilt.
  - Wird auf die Von Neumann Schicht abgebildet.
- Von Neumann Schicht (Maschinenbefehlsebene / conventional machine level)
  - Definiert durch den Maschinenbefehssatz und dessen Objekte / Operationen.
  - Wird auf die Mikroarchitektur Schicht abgebildet.
- MIkroarchitektur Schicht (RTL / Register Transfer Level)
  - Definiert durch den Mikrobefehlssatz und dessen Objekte.
  - Wird räumlich auf die Gatter Schicht abgebildet (Übersetzung in Hardware)
- Gatter Schicht (gate level)
  - Definiert durch Boolesche Funktionen. Diese werden durch elementare Schalter realisiert.
  - Wird durch Schaltungen auf die Bauelemente Schicht abgebildet.
- Bauelemente Schicht (component level)
  - Definiert durch idealisierte Bauelemente (Widerstände, Transistoren…)
  - Wird durch reale Bauelemente realisiert.
- Physikalische Schicht (physical level)
  - Definiert durch reale Bauelemente

2.2. Mikroarchitektur Schicht

- Die Mikroarchitektur Schicht besteht aus div. Mikroinstruktionen.
  - Es können Teile dieser parallel ausgeführt werden
  - Mikroinstruktionen werden im Mikroprogrammspeicher abgelegt
  - Jede Instruktion besteht aus einer Menge an in einem Takt ausführbaren Mikrooperationen
    - Mikrooperationen sind atomar (nicht weiter aufteilbar)



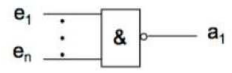


- Leitwerk
  - Objekte und Instruktionen höherer Schichten werden dargestellt als:
    - Variablen: Register & Speicherzellen
    - Zuweisungen: Transportbefehle
    - Funktionen: Schaltnetze
- Mikroinstruktionen
  - Können durch verschiedene Arten von Mikroinstruktionsformaten effizient gespeichert werden.
    - Horizontal: uncodiert, Steuerbits können direkt gelesen werden.
    - Vertikal: Sinnvolle Kombinationen zusammengefasst. Decodierung nötig.
    - Quasihorizontal: optimal codiert, es müssen nur bestimmte Teile decodiert werden.
- Speicher
  - Wird realisiert durch n 1-bit Flip Flops
  - Somit wird ein Speicher zu einem Schaltwerk, da dieser einen internen Zustand besitzt.
- Tri-State-Bus
  - Ein Bus ist eine einfache Möglichkeit, mehrere Komponenten über einen Zeitmultiplex (untersch. Signale zu untersch. Zeit auf einem Bus) miteinander kommunizieren zu lassen.
    - Zu einem Zeitpunkt ist maximal ein Register auf den Bus geschaltet
  - Hat die Zustände: ON, OFF, NONE

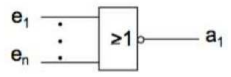
2.3. Gatter Schicht

- Die Gatter Schicht besteht aus mehreren Booleschen Funktionen.
  - Diese sind somit Schaltnetze mit nur einem Ausgang, und 1-bit Speicher
- Schaltglieder
  - Jede Boolesche Funktion kann in konjunktiver- oder disjunktiver Normalform dargestellt werden.
  - Auch mit nur `NOR` oder `NAND` möglich. (meist verwendet)

NAND: $a_1 = \neg(e_1 \wedge e_2 \wedge \ldots \wedge e_n)$

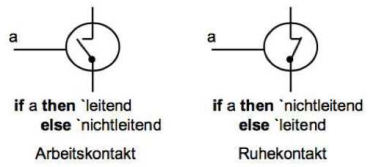


NOR: $a_1 = \neg(e_1 \vee e_2 \vee \ldots \vee e_n)$



2.4. Bauelemente Schicht

- In der Bauelemente Schicht werden Gatter durch elektrische Schaltungen realisiert.
- Es werden 2 Arten von Schaltern verwendet:
  - Arbeitskontakt: Wenn a fließt Strom
  - Ruhekontakt: Wenn nicht a fließt Strom
  - Diese lassen sich mit Feldeffekttransistoren realisieren (keine beweglichen Teile)

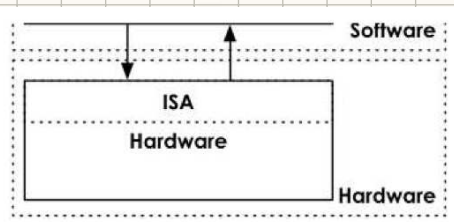


2.5. Physikalische Schicht

- Die Bauelemente Schicht wird meist durch Schaltungen auf Silizium realisiert. (MOS metal oxide semiconductor)
  - Ein Transistor ist die Realisierung eines Schalters

ISA (Instruction Set Architecture)

- Eine ISA beschreibt die Sicht des Programmierers auf den Prozessor
  - Diese spezifiziert: Datentypen; Register; Befehle / Adressierung (sowie die Befehlscodierung / das Befehlsformat); Ausführungsmodi (privilegierte Funktionen / Interrupts)
- Vollständige Abstraktion des Prozessors
  - Somit wird die Implementierung der ISA für den Programmierer irrelevant

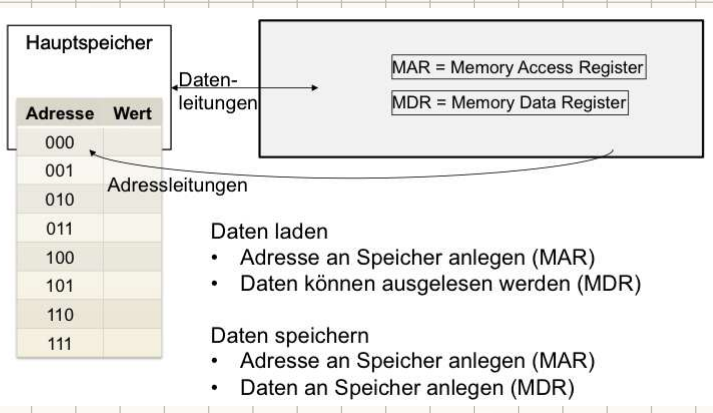


CISC vs RISC

CISC (Complex Instruction Set Computer) (Intel x86)
Definition: viele komplexe Instruktionen
Vorteil: kompakte Maschinencodedarstellung
geringer Speicherbedarf des Quellcodes
Nachteil: höhere Komplexität um dies auf Prozessor umzusetzen
viele ungenutzte Funktionen

RISC (Reduced Instruction Set Computer) (M1 Chip - ARM)
Definition: wenige und sehr einfache Instruktionen
Vorteil: einfacher zu implementieren wegen weniger Befehlen
Aufgrund des reduzierten Befehlssatz wird weniger Chipplatz benötigt
Nachteil: schwieriger programmierbar, wird meist ein Compiler verwendet

Speicherzugriff auf Daten



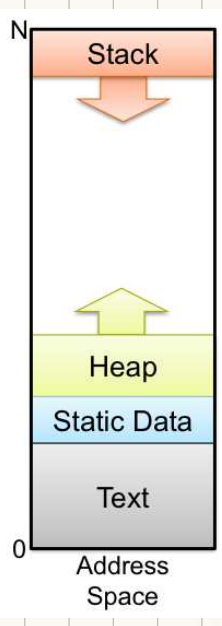
DMA (Direct Memory Access)
-> direkt Zugriff von I/O auf Speicher

Datentypen

- Datentypen sind die Definition der Daten, welche vom Prozessor verarbeitet werden können.
  - Bestehen im Hauptspeicher nur aus Bytefolgen, ohne Typinformationen (diese folgen aus dem Verwendungskontext)
- Der Datentyp wird durch die angewendeten Befehle definiert
  - Diese interpretieren die Bytefolge immer gleich
- Nicht nativ unterstützte Datentypen müssen auf elementare (unterstützte) zurückgeführt werden.
  - Bsp.: String → Folge von Zeichen → Folge von Bytes (durch Compiler)

Typischer Speicherverwaltung

- Von einem Programm verwendeter Speicher wird in verschiedene Bereiche unterteilt. Dabei werden die Bereiche Code und Static vor dem Start des Programms zugewiesen.
  - Code — Maschinencode des Programms
  - Static — statische (zur Compilezeit bekannte) Variablen (Konstanten)
  - Heap — Dynamisch allokierte Datenstrukturen
  - Stack — Stack für Funktionsaufrufe und lokale Variablen
- statische Speicherallokation
  - Durch den Compiler vordefinierter Speicher, welcher sich über die gesamte Laufzeit des Programms nicht ändert (Read Only)
  - Daten können durch padding an den Speicher ausgerichtet werden (Vermeidung von memory misalignment)
- dynamische Speicherallokation
  - Kann die Speicherzuweisung nicht vom Compiler vorgenommen werden, muss dieser entweder auf dem Stack, oder dem Heap bereitgestellt werden
  - Stack
    - Jeder Funktionsaufruf bekommt einen eigenen Block auf dem Stack. (erlaubt Rekursion)
    - Lokale variablen und Rückgabewerte werden dort gespeichert.
    - Kann beim Verlassen einer Funktion schnell wieder freigegeben werden
  - Heap
    - Für Daten, die über eine Funktion hinaus existieren müssen.
    - Muss explizit Allokiert und freigegeben werden. (Freigabe auch durch Garbage Collector)
    - Bei Allokation und Freigabe von vielen Objekten ist eine Fragmentierung des Heaps möglich



Adressierungsarten

ein konkretes Schema zur Angabe eines Maschinenbefehl-Operanden

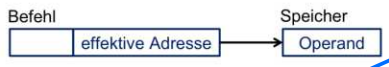
- **Unmittelbare Adressierung**
  - Der Operand steht als Konstante im Befehl

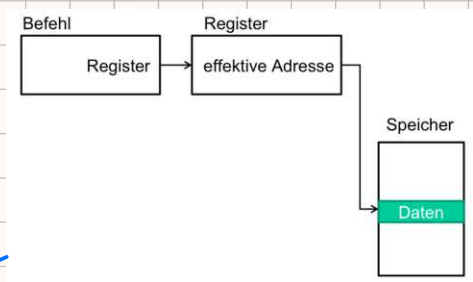


- **Register-Adressierung**
  - Der Operand steht in einem Register
  - Die Registeradresse steht im Befehl

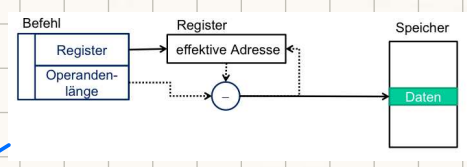


- **Speicheradressierung (direkte Adressierung)**
  - Der Operand steht im Hauptspeicher
  - Die Adresse steht (als Konstante) im Befehl

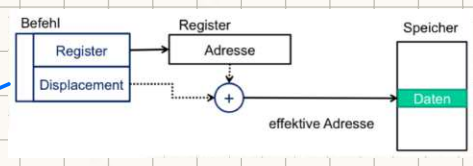


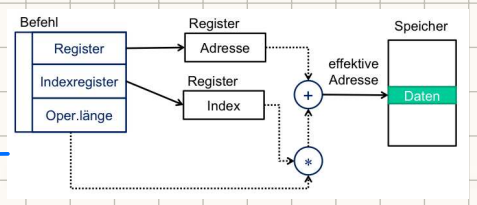


- **Dynamische Adressierung**
  - Man berechnet die effektive Adresse des Operanden - Berechnung erfolgt zur Laufzeit
  - Erhöhte Flexibilität, weil Rechenprozess die Adressen von Operanden, Ergebnissen, Sprungzielen zur Laufzeit ändern können



- **Registerindirekte Adressierung**
  - Die effektive Adresse des Operanden steht in einem Register
  - Das Register steht im Befehl
  - **Registerindirekt mit Prä-/Post (dekrementieren/ inkrementieren)**
    - Die im Register stehende Adresse wird vor/ nach ihrer Verwendung aktualisiert
  - **Registerindirekt mit Displacement (basisrelative Adressierung)**
    - Zu der im Register stehenden Adresse wird ein konstanter (pos./ neg.) Wert addiert
  - **Indizierte Adressierung mit Skalierungsfaktor**
    - Index wird mit Konstante multipliziert

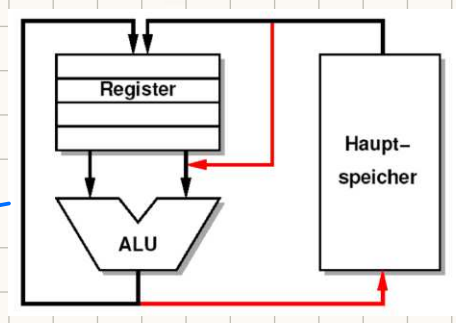




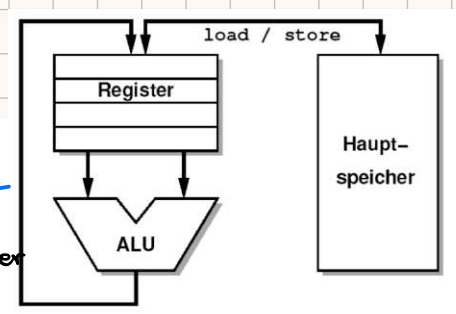
Adressierungsformen

- **Nulladressform: [ADD]**
  - **Ausführungsmodell: Kellermaschiene** (Java VM)
  - Das oberste Stack-Elemente werden aus dem Stack entfernt
  - Das Ergebnis wird wieder auf den Stack gelegt

- **Einadressform: [x = A]**
  - **Ausführungsmodell: Akkumulatormaschiene**
  - Operand ist immer ein Register (Akkumulator)
  - Das Ergebnis wird wieder im Akkumulator abgelegt
  - Nachteil: Unflexibel, häufiges Laden & Speichern des Akkumulators erforderlich



- **Zweiadressform: [x = x+y]**
  - **Ausführungsmodell: Register-Speicher-Modell**
  - Das Ergebnis eines Befehls überschreibt immer den ersten Operanden
  - Nur einer der Operanden kann im Hauptspeicher stehen
  - Ergebnis überschreibt entweder Register- oder Speicheroperanden



- **Dreiadressform: [x = y+z]**
  - Operanden und das Ergebnis werden explizit adressiert
  - **Speicher-Speicher-Modell** - frühere CISC-Prozessoren
    - Operanden und Ergebnis dürfen im Hauptspeicher stehen
    - Nachteil: Befehle benötigen sehr viel Speicherplatz und Speicherzugriffe, langsam
  - **Register-Register-Modell** - fast alle RISC-Architekturen (Ausnahme: LOAD-/STORE)
    - Operanden und Ergebnis dürfen nur im Register stehen
    - Nachteil: Weniger Auswahlmöglichkeiten, weil Operanden & Ergebnisse nur in Registern stehen dürfen

Vorteil: volle Flexibilität für Programmierer

Befehlsformat

Einheitliches Befehlsformat
- Alle Befehle haben eine feste Länge

Vorteile: einfach & schnell zu dekodieren

Nachteile: - unflexibel, nur bei LOAD / STORE Architekturen praktikabel
- kein Platz für große Konstanten im Befehl

Variables Befehlsformat
- Befehle können unterschiedlich lang sein; Typisch für CISC ISAs

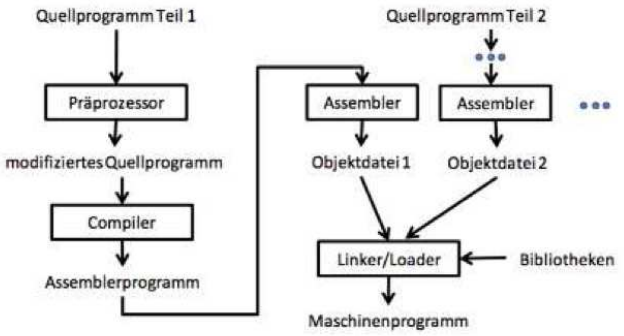
Vorteile: extrem flexibel; alle Adressierungsarten sind möglich

Nachteile: - Dekodierung der Befehle ist sehr zeitaufwändig
- Probleme mit Speicherverwaltung

Ausführungsmodus des Betriebssystems

- Werden u.a. zur Unterstützung von Mehrbenutzersystemen verwendet
- Systemmodus
  - Bietet vollen Zugriff auf alle Funktionen des Prozessors.
  - Vom Betriebssystem verwendet.
- Benutzermodus
  - eingeschränkter Zugriff auf die Befehle des Prozessors (keine privilegierten Befehle möglich).
  - Speicher nur über Adressabbildung verfügbar (virtuelle Adressen)

5. Übersetzung von Hochsprachen

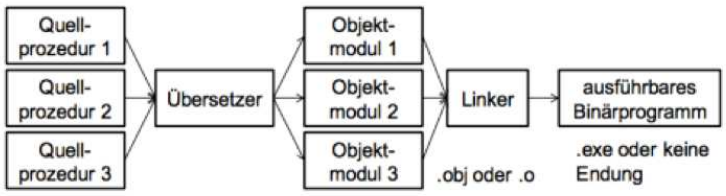


5.1. Assemblierer

- Ein Assemblierer übersetzt ein Programm in Maschinensprache. Somit wird dieses vom Prozessor ausführbar.
- Zwei-Pass-Assemblierer
  - Es muss 2 Mal über den Quellcode gegangen werden, da sonst Marken (Labels) nicht aufgelöst werden können (Vorwärtsreferenzen)
  - 1. Pass Sammeln aller Symbol- und Makrodefinitionen ⇒ Makro Erweiterung
  - 2. Pass Generierung des Maschinencodes & Bereitstellen der Informationen für den Linker
    - In diesem Schritt wird auch Fehlererkennung und -behandlung ausgeführt.
- Aufrufkonventionen (calling conventions)
  - Eine Aufrufkonvention legt bei einem Funktionsaufruf fest:
    - Wo Parameter liegen
    - Wo der Rückgabewert gespeichert werden soll
    - Welche Register / Teile des Stacks überschrieben werden können (muss Aufrufer, oder Aufgerufene die Register sichern?)

5.2. Linker

- Ein Linker verbindet mehrere voneinander getrennt übersetzte Prozeduren zu einer.
  - Somit kann ein Programm auf mehrere Dateien aufgeteilt werden.



Beim linken auftretende Probleme	Lösung
• Sprünge sind nicht mehr korrekt • Adressen von Variablen sind nicht mehr korrekt • Lösen von bei der getrennten Übersetzung aufgetretenen externen Referenzen	1. Erstellen von Tabelle aller Objektmodule 2. Anfangsadresse im Zielcode für jedes Modul bestimmen 3. Jeweilige Anfangsadressen auf bestehende Speicherreferenzen addieren 4. Aktualisieren von Aufrufen zu externen Prozeduren

- Programmbibliotheken
  - Programmbibliotheken sind eine Sammlung von fertigen Funktionen, welche im Quellcode verwendet werden können.
  - Statische Bibliotheken (*.a / *.lib)
    - Bibliotheksroutinen werden in in den Zielcode kopiert.

Vorteile	Nachteile
• Muss nicht nachgeladen werden ○ schnelleres Laden	• Größere ausführbare Datei

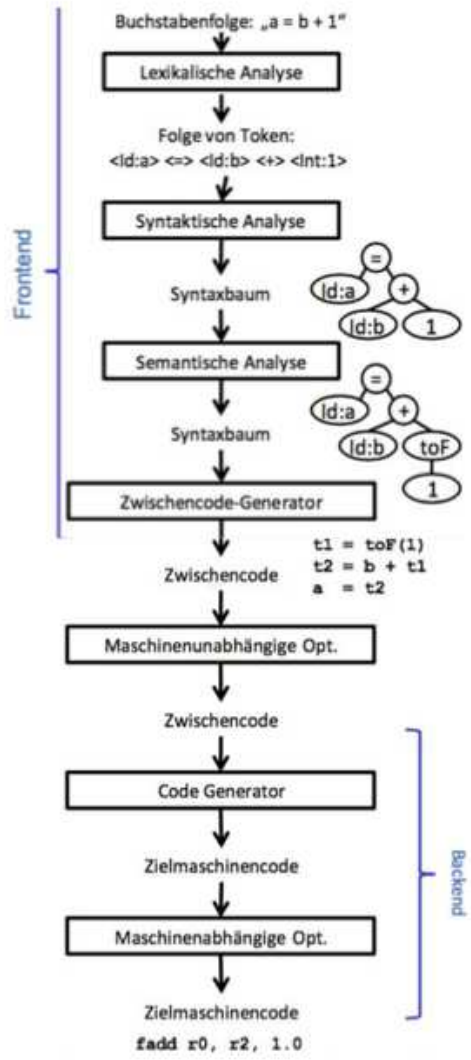
- Gemeinsame Bibliotheken (*.so / *.dll)
  - Vom Betriebsystem mit der Anwendung geladen, falls nicht schon vorhanden.
  - Adressbindung beim Laden / Ausführen (Runtime) des Programms.

Vorteile	Nachteile
• Kleine ausführbare Datei • Gemeinsam nutzbar (spart Speicher) • Austauschbar	• Versionskonflikte • Fehlt eventuell auf dem System • Langsameres Laden

- Dynamisch ladbare Bibliotheken (*.so / *.dll)
  - Erst geladen, wenn eine Funktion aus der Bibliothek aufgerufen wird.
  - Adressbindung während der Programmausführung (Runtime).

5.3. Compiler

- Ein Compiler übersetzt ein Programm in Maschinensprache und verwendet dabei meist eine Form von Assembly als Zwischenergebnis.
  - Führt statische Fehleranalyse durch.
    - Erkennt syntaktische und semantische Fehler
  - Erzeugt optimierten Maschinencode
    - über mehr als eine Anweisung
  - Schnellere Programmausführung
- Phasen eines Compilers
  - Lexikalische Analyse
    - Lesen der Buchstabenfolge
    - Auftrennen des Textes in einzelne Token
  - Syntaktische Analyse
    - Überprüft das Programm auf Syntaktische Fehler
    - Erzeugt Syntaxbaum, der das Programm darstellt
  - Semantische Analyse
    - Hinzufügen von Typinformationen
    - Überprüft auf semantische Konsistenz, Typkonsistenz
  - Zwischencode Generierung
    - Erzeugt Zwischencode, welcher aus einer Folge von Anweisungens besteht.
    - Äquivalent zur Ausführungsreihenfolge der Hochsprache
  - Maschinenunabhängige Code Optimierung
    - Verbesserung des Zwischencodes, sodass dieser kürzer, schneller wird.
  - Code Generierung
    - Erzeugung von Assembly, oder direkt Maschinencode
    - Zuweisung von Speicheradressen und Register.
  - Maschinenabhängige Code Optimierung
    - Der Zwischencode wird auf die von der Ziel ISA angebotenen Befehle angepasst.



- Optimierungstechniken
  - Basisblöcke
  - Register Allokation

5.4. Interpreter

- Ein Interpreter lädt den Quelltext eines Programms und führt diesen direkt aus. Eingaben des Benutzers werden ebenfalls an den Interpreter weiter gegeben.
  - Langsamerer Ausführung (keine globale Optimierung)
  - Bessere Fehlermeldungen (Quelltext und Laufzeitinformationen verfügbar)

Assembly x86

Assembler: eine Hardwarenahe Programmiersprache
- für jeden Befehl genau einen Maschinenbefehl erzeugt
- durch einen Assemblierer zu einem ausführbaren Maschinenprogramm übersetzt
- Instruction-Bestandteile: Marke | Opcode | Operanden | Kommentar
- Operanden-Quellen: Teil der Instruktion (Immediate); Aus Register/ Speicher; Vom I/O Port

Vorteil
- Rechengeschwindigkeit -> Programme können sehr effizient geschrieben werden
- bieten direkten Zugriff auf die Hardware

Nachteile
- sehr aufwendig (niedrige Abstraktion)
- fehleranfällig

Arithmetische Operatoren

Länge muss übereinstimmen

mov ziel, quelle

`ADD ziel, quelle`	`ziel := ziel + quelle`
`SUB ziel, quelle`	`ziel := ziel - quelle`
`NEG ziel`	`ziel := -ziel`
vorzeichenlose Zahlen:	
`MUL quelle`	`EDX:EAX := EAX * quelle`
`DIV quelle`	`EAX := (EDX:EAX) / quelle  EDX := (EDX:EAX) % quelle`
vorzeichenbehaftete Zahlen:	
`IMUL quelle`	`EDX:EAX := EAX * quelle`
`IMUL ziel, quelle`	`ziel := ziel * quelle`
`IMUL ziel, quelle, konst`	`ziel := quelle * const  (const max 8 bit)`
`IDIV quelle`	`EAX := (EDX:EAX) / quelle  EDX := (EDX:EAX) % quelle`

multiply (AX := AL * *r/m8*).
multiply (DX:AX := AX * *r/m16*).
multiply (EDX:EAX := EAX * *r/m32*).
multiply (RDX:RAX := RAX * *r/m64*).

Dividend	Divisor	Quotient	Remainder
AX	r/m8	AL	AH
DX:AX	r/m16	AX	DX
EDX:EAX	r/m32	EAX	EDX
RDX:RAX	r/m64	RAX	RDX

Symbolische Adresse (Labels)

-> als Platzhalter für Werte (Deklaration mit EQU)

```
value:
  EQU 15     ; 32 bit integer
MOV EAX, value
```

-> Zeilen in Code symbolisieren

```
start:
  MOV EAX, 42
...
JMP start
```

80386 Stack
- Wachstumsrichtung von höherer Adresse zu niedrigerer (von „oben" nach „unten")
- ESP zeigt auf zuletzt abgelegtes Element （最后存入的元素）



Reservieren von nicht initialisiertem Speicherbereich

```
section .bss
    RESB 1     ; Reserviere 1 Byte
    RESW 1     ; Reserviere 2 Byte
    RESD 1     ; Reserviere 4 Byte
    RESQ 1     ; Reserviere 8 Byte
```

Reservieren und Initialisieren von Speicher

```
section .data
    DB 123          ; Reserviere 1 Byte mit Wert "123"
    DW 1337         ; Reserviere 2 Byte mit Wert „1337“
    DD 0xCDFFEE     ; Reserviere 4 Byte mit Wert „0xCDFFEE“
    DQ 0XDEADBEEF   ; Reserviere 8 Byte mit Wert „0XDEADBEEF“
```

Ein-Operand-Befehl mit Immidiate

Compiler kann Daten-Größe des Ein-Operand-Befehls nicht inferieren

Operandenlänge muss vor Immidaite angegeben werden `PUSH DWORD 123`

Attribut	Bytes
BYTE	1
WORD	2
DWORD	4
QWORD	8

Stack

- Der Hauptspeicher wird als Stack angelegt. Dabei zeigt das Register ESP immer auf das oberste Element des Stacks.
- Wird bei Funktionsaufrufen genutzt:
  - function calls genau erklären
- Wird zum Sichern von Registern am Anfang einer Funktion benutzt:
  - Die gesicherten Register müssen am Ende wieder in umgekehrter Reihenfolge vom Stack geladen werden.

```
PUSH EAX
PUSH EBX
…
…
POP EBX
POP EAX
```

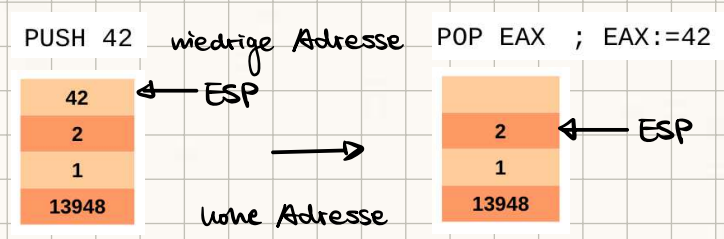
Speichern von EAX, EBX auf dem Stack.

Holen von EBX, EAX vom Stack. (umgekehrte Reihenfolge)

```
Stack:      EBX   <- ESP
            EAX
            …

Stack:      EBX
            EAX
            …     <- ESP
```

- **PUSH foo**:
  - Verringere ESP um Größe von „foo“ (entspricht Allokation)
  - Speichere „foo“ an Adresse ESP
- **POP bar**:
  - Speichere Inhalt an Adresse ESP in „bar“
  - Erhöhe ESP um Größe von „bar“ (entspricht Freigabe)



Speicherzugriff

- Bei der Adressierung einer Speicherzelle mit dem MOV Befehl sind folgende Formen möglich
- Register Indirect Mode

```
MOV EAX, [EBX]
```

- Based Mode

```
MOV EAX, [EBX + 42]                 Register + Konstante
```

- Based Indexed Mode

```
MOV EAX, [EBX + ESI]                Register + Register
```

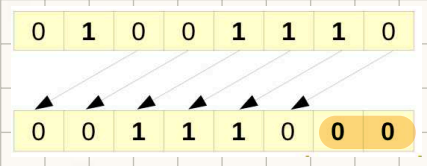
- Based Indexed Mode with Displacement

```
MOV EAX, [EBX + ESI + 42]           Register + Register + Konstante
```

- Base and Scaled Index with Displacement
  - Scale kann nur 2, 4, 8 sein

```
MOV EAX, [EBX + ESI * 4 + 42]       REG + REG * Scale + Konstante
```

4 Bytes 42 Bytes Verschiebung



Schiebeoperator

beim Shiften: das herausgeschiftete Bit landet in Carry-Flag

-- nach Verschiebung werden „Fehlende“ Bits mit 0 aufgefüllt

`SHL r/m, imm8/cl` - Linksschieben um n Bits von vorzeichenloser Ganzzahl entspricht (Multiplikation mit 2^n)

`SHR r/m, imm8/cl` - Rechtsschieben um n bits von vorzeichenloser Ganzzahl entspricht (Division mit 2^n)

SHL -> insert 0

SRL -> insert 0

CF = MSB(DEST)
OF = MSB(DEST) xor CF

```
1  MOV EAX, 7     ; EAX = 7
2  SHL EAX, 1     ; EAX = EAX*2 = 7*2
3                 ; EAX = 14
4
5  MOV EAX, 35    ; EAX = 35
6  SHR EAX, 2     ; EAX = EAX/ 4 = 35/ 4
7                 ; EAX = 8
```

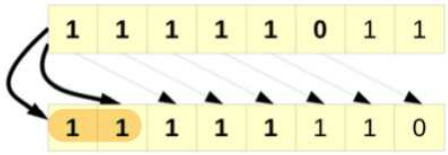
-- nach Verschiebung werden „Fehlende“ Bits mit 1 aufgefüllt

`SAL` Linksschieben um n bits -> vorzeichenbehaftetes Multiplikation mit 2^n

`SAR` Rechtsverschiebung um n bits -> vorzeichenbehaftetes Division mit 2^n

SAL -> insert 0

SAR = 2



SAR { falls MSB ist 1 => insert 1
falls MSB ist 0 => insert 0

SAR CF = LSB(DEST) OF = 0
SHR CF = LSB(DEST) OF = MSB(DEST)

```
1  MOV EAX, -35   ; EAX = -35
2  SAR EAX, 2     ; EAX = EAX/ 4 = -35/ 4 = -8.75
3                 ; Rundung nach "unten" für negative Zahlen
4                 ; EAX = -9
```

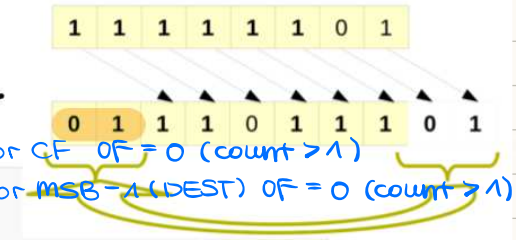
-- nach Rotation werden „Fehlende“ Bits mit herausgeschossenen Bits aufgefüllt

`ROR r/m, imm8/cl` Rechtsrotieren um n Bits

`ROL r/m, imm8/cl` Linksrotieren um n Bits

ROR = 2

count = 1

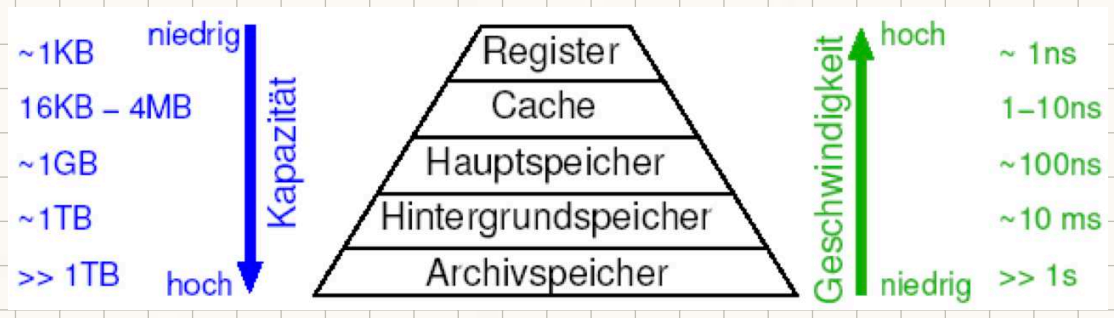


ROL CF = MSB(DEST) OF = MSB(DEST) xor CF OF = 0 (count > 1)
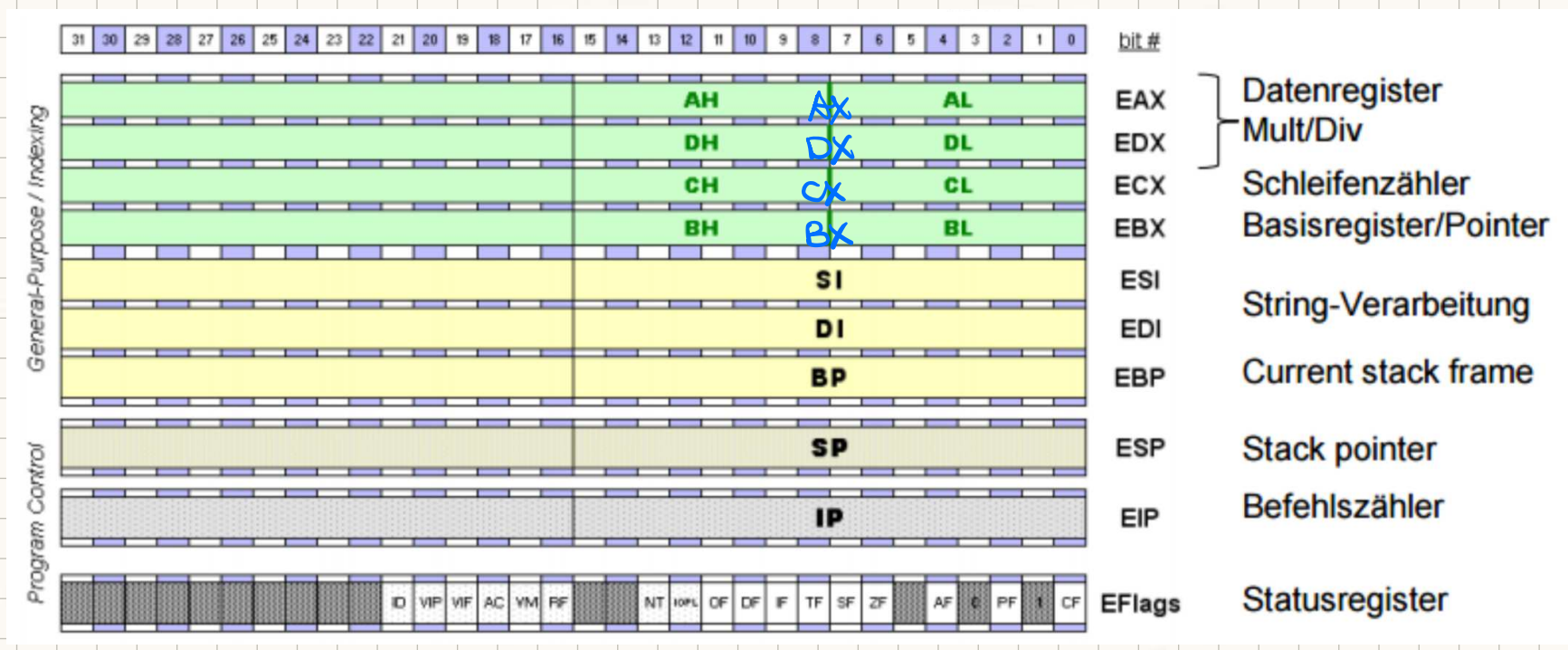
ROR CF = LSB(DEST) OF = MSB(DEST) xor MSB-1(DEST) OF = 0 (count > 1)

```
1  MOV AL, 99h    ; AL = 10011001b
2  ROL AL, 1      ; AL = 00110011b Rotate to the left
3
4  MOV AL, 99h    ; AL = 10011001b
5  ROR AL, 1      ; AL = 11001100b Rotate to the right
```

Speicherpyramide

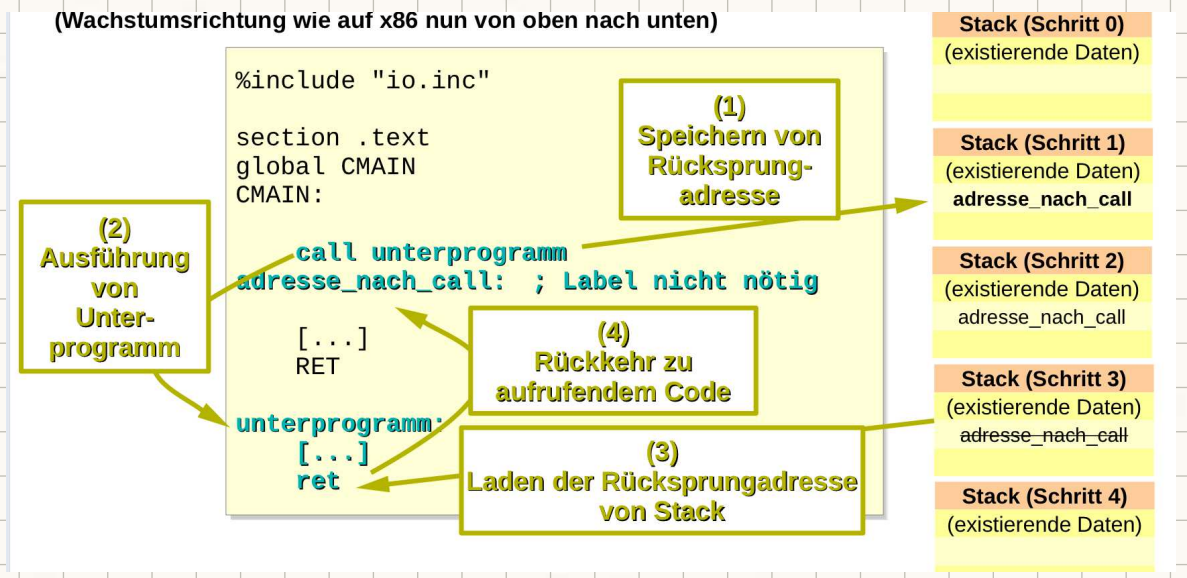


Registerübersicht



Unterprogrammaufruf

- **Rücksprungsadresse wird auf Stack gespeichert**
- **Jede Rückkehr von Unterprogramm räumt seinen Teil des Stacks auf**



unbedingte Sprung

`jmp r/m` -> nächste Instruktion an Adresse des Ziels

```
JMP 0x1234                  ; Sprung an bestimmte Adresse
                            ; Z.B. Startadresse vom Betriebssystem
JMP label1

MOV EAX, 0x4567             ; Laden von Programmadresse
JMP EAX                     ; Ansprung von Programm an Adresse 0x4567

label1:
        JMP [EBX]           ; Springe zu 32-bit Adresse in Speicherzelle
                            ; welche durch EBX adressiert ist
```

Spezialbefehl CMP

`CMP EAX, EBX` **-> zwei Registerinhalte vergleichen ohne Ändern**

bedingte Sprung J[cc]

- **Programmablauf - unabhängig von Werten im Register**
- **Bedingung richtet sich nach bestimmten Bits im Statusregister**

Instruktion	Bedeutung	Flags
JC	carry	CF = 1
JNC	no carry	CF = 0
JE, JZ	equal/ zero	ZF = 1
JNE, JNZ	not equal/ not zero	ZF = 0
O	(signed) overflow	OF = 1
JNO	(signed) not overflow	OF = 0
JGE, JNL	(Signed) greater or equal/ not less	(SF xor OF) = 0
JG, JNLE	(signed) greater/ not less or equal	((SF xor OF) or ZF) = 0
JLE, JNG	(Signed) less or equal/ not greater	((SF xor OF) or ZF) = 1
JL, JNGE	(signed) less / not greater or equal	(SF xor OF) = 1

Zeroflag (unabhängig von Vorzeichen)

Ergebnis gleich 0

- ZF = 1, wenn EAX = EBX
- ZF = 0, wenn EAX != EBX

Carryflag (vorzeichenlos)

Übertragsbit ((n+1)-tes Bit bei n-bit Zahlen)

- CF = 1, wenn EAX < EBX (ohne Vorzeichen)
- CF = 0, wenn EAX >= EBX (ohne Vorzeichen)

Overflowflag (vorzeichenbehaftet)

Überlaufbit generiert, wenn vorzeichenbehafteter Überlauf vorliegt

- OF = 1, wenn EAX-EBX nicht repräsentierbar (Overflow)
- OF = 0, wenn EAX-EBX repräsentierbar

Signflag (vorzeichenbehaftet...)

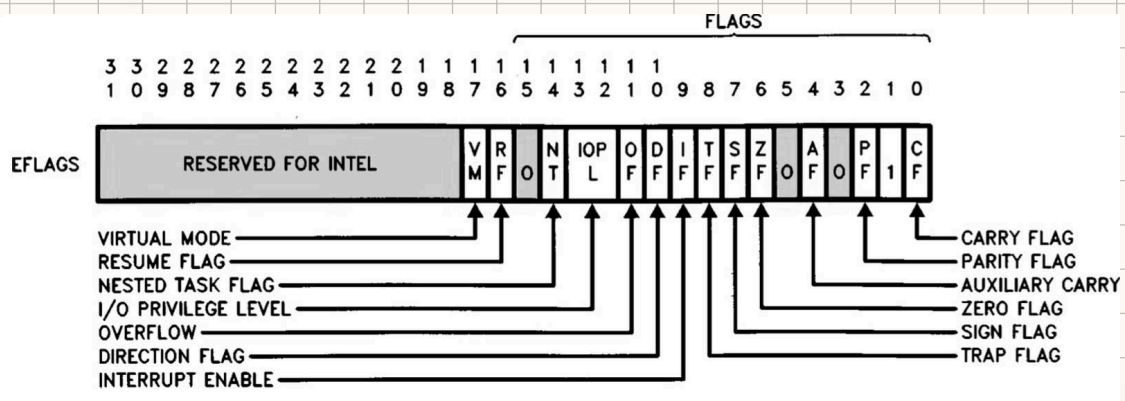
SF = 1 wenn EAX − EBX < 0
(höchste Bit = 1)

Mit 4 Bit:
1111 - 0101 = 1010
Hier ist das SF gesetzt aber das CF nicht

(+) + (+) = - -> OF
(-) + (-) = + -> OF

Statusregister

字母小写 ⇒ 0　字母大写 ⇒ 1



für AND, OR, TEST, XOR　CF = 0, OF = 0

MOV, PUSH, POP setzen keine Flags !

Overflow Flag → signed Overflow $[-2^{N-1}, 2^{N-1}-1]$
Carry Flag → unsigned Overflow $[0, 2^{N-1}]$

char letter -> 1 Byte

$\text{char } vorname[64]$ -> 64 Byte

$\text{char* } zeiger = 0$ -> 32 Byte (um einen Pointer zu kennzeichnen (Pointer is so groß wie eine Speicheradresse)

Logische Operator

`AND r/m, r/m/imm` 逻辑与指令 - 两个数都是1, output才是1; 否则都是0

`OR r/m, r/m/imm` 逻辑或指令 - 两个数中的任意一个是1甚至两个都是1, output就是1; 否则就是0

`XOR` 异或电路 - 两个input相同的时候, output为0； 否则output为1 `XOR dx, dx` 等效于 ~~`XOR dx, 0`~~ MOV

`XOR Ziel & Quelle_1, Quelle_2` -> Ausgabebit ist gesetzt, wenn Eingabebits unterschiedlich sind

`TEST Quelle_1, Quelle_2` → wie "AND" (ohne Ändern)

Parameterübergabe auf Stack `[EBP+X]`

method (parameter 1, parameter 2, parameter 3)

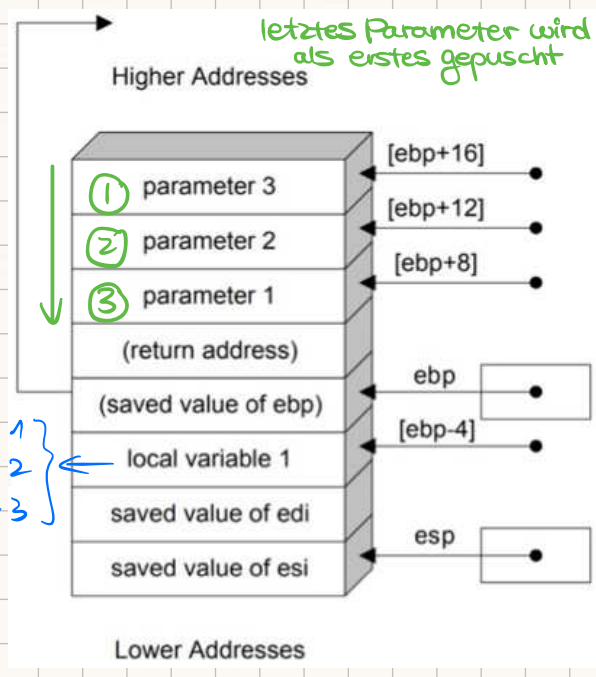
1) Speichern der Parameter auf Stack
2) Aufruf der Funktion (push ebp, mov ebp, esp)
2.1) Sichern des Basiszeiger und laden mit Stackzeiger
2.2) Ausführen der Funktion wie gewohnt
3) Zugriff auf Parameter ab `[EBP+8]` (äquivalent zu [esp + 12])

`[EBP]` -> Backup von EBP Register `[EBP+4]` -> 32 Bit Rücksprungadresse

4) Maschinencode von Unterprogramm ausführen
5) Wiederherstellung von Basiszeiger
6) Rückkehr
7) Säubern des Stacks ~~von~~ entfernen Parametern → ADD ESP, 4 * 3

** Lokale Variablen sind über `[EBP - x ]` adressierbar

push parameter 3
push parameter 2
push parameter 1
call method



lokal variable 1
lokal variable 2
lokal variable 3

Variierendes ESP - falls Daten auf Stack hinterlegt oder vom Stack genommen werden

Basepointer EBP - um „Basisadresse“ auf Stack („Anfangswert“ von ESP) zu Beginn von aktuellem Unterprogramm zu sichern

Schleifen

- Durch bedingte Sprünge können Schleifen implementiert werden:

```
for (int i = 0; i < 10; i++) {
    ...
}
```

```
MOV EAX, 0          int i = 0
JMP cond
loop:
    …
    INC EAX         i++
cond:
    CMP EAX, 10     i < 10
    JL loop
```

```
; Aufruf
        push dword 4            ; p4
        push dword 3            ; p3
        push dword 2            ; p2
        push dword 1            ; p1
        call fn_stack           ; stack:[retaddr] p1 p2 p3  p4
        add esp, 16             ;       ^esp+0^   +4 +8 +12 +16
        ...

fn_stack:
        push ebp
        mov ebp, esp            ; stack:EBP [retaddr] p1 p2 ...
        mov eax, [ebp + 8]      ; p1
        add eax, [ebp + 12]     ; p1 + p2

        add eax, [ebp + 16]     ; + p3
        add eax, [ebp + 20]     ; + p4
        pop ebp
        ret
```

Fakultät berechnen

$$n! = fak(n) = \begin{cases} n \times fak(n-1) & \text{für } n \geq 1 \\ 1 & \text{für } n = 0 \end{cases}$$

Rekursion mit Abbruchbedingung

```
fakultaet:
    ; IN: EAX: Parameter fuer Fakultaet
    ; OUT: EAX: Ergebnis
    PUSH EBX
    PUSH EDX

    CMP EAX, 0      ; Abbruchbedingung
    JE fakultaet_n_null ; Springe, wenn n=0

    MOV EBX, EAX    ; Sichern von n, EBX=n
    DEC EAX         ; n-1
    CALL fakultaet  ; EAX=fakultaet(n-1)
    MUL EBX         ; fakultaet(n-1)*n
    JMP fakultaet_return    ; Rueckkehr

fakultaet_n_null:
    MOV EAX, 1      ; Rueckgabewert 1
                    ; Achtung: Hier kein RET!
                    ; Weiter mit POP EDX, ...
fakultaet_return:
    POP EDX
    POP EBX
    RET     ; Ergebnis in EAX
```

Schleifen mit Abbruchbedingung

```
fakultaet:
    ;  IN: EAX: Parameter fuer Fakultaet
    ; OUT: EAX
    PUSH ECX
    PUSH EDX

    MOV ECX, EAX    ; n in Zaehlerregister
    MOV EAX, 1

fakultaet_nochmal:
    CMP ECX, 0          ; Abbruchbedingung
    JE fakultaet_return ; Springe, wenn n=0

    MUL ECX
    DEC ECX
    JMP fakultaet_nochmal ; Schleife

fakultaet_return:
    POP EDX
    POP ECX
    RET     ; Ergebnis in EAX
```

Array-Reihenfolge umkehren

```
revert:
    ; input:
    ;   EBX: start address of dwords
    ;   ECX: number of dwords
    PUSH EBX
    PUSH ECX
    PUSH EDX

    ; compute pointer to top element
    MOV EDX, ECX
    DEC EDX
    SHL EDX, 2
    ADD EDX, EBX
    SHR ECX, 1

body:
    PUSH DWORD [EBX]
    PUSH DWORD [EDX]
    POP DWORD [EBX]
    POP DWORD [EDX]
    ADD EBX, 4
    SUB EDX, 4
    LOOP body

    POP EDX
    POP ECX
    POP EBX
    RET
```

Aufgabe 5

„erster Parameter bei kleinster Adresse“

a) Übergabe über Register

```
fn_a:
  add eax, ebx
  add eax, ecx
  add eax, edx
  ret
```

+ sehr schnell und simpel
- Anzahl der Parameter begrenzt
- Laut Calling Convention sind Register flüchtig -> Rekursion so nicht möglich

b) Übergabe über feste Speicherstellen

```
fn_b:
  mov eax, [arg1]
  add eax, [arg2]
  add eax, [arg3]
  add eax, [arg4]
  ret
```

+ kein Stack nötig
+ einfaches Debugging
- keine Rekursion möglich (Überschreiben)
- Speicherverschwendung, da Speicher auch ohne Funktion belegt ist

c) Übergabe über Stack

```
fn_c:
  push ebp
  mov ebp, esp
  mov eax, [ebp + 8]  ;arg1
  add eax, [ebp + 12] ;arg2
  add eax, [ebp + 16] ;arg3
  add eax, [ebp + 20] ;arg4
  pop ebp
  ret
```

+ beliebige Anzahl an Parametern
+ Rekursion möglich
- Adressierung etwas umständlicher
- Anker ebp für weitere lokale Argumente

Anzahl der negativen Werte zählen

```
count_negative:
    PUSH EBX
    PUSH ECX

    XOR EAX,EAX

body:
    CMP EBX,0
    JE end
    CMP DWORD [EBX+4],0
    JNS next
    INC EAX

next:
    MOV EBX,[EBX]
    JMP body

end:
    POP ECX
    POP EBX
    RET
```

$c = a \bmod 256$

① **mov ecx, 0**
mov cl, al

② **and a, 255**

8.8 Festkommazahl multipliziert pi

```
mul_pi:
    mov bx, 804
    mul bx
    mov al, ah
    mov ah, dl
    ret
```

8.8 Festkommazahl dividiert pi

```
div_pi:
    mov dh, 0
    mov dl, ah
    mov ah, al
    mov al, 0
    mov bx, 804
    div bx
    ret
```

Information in Baumartigen Speicherstruktur finden

Startadresse	Schlüsselwert (32-Bit-Zahl)
Startadresse + 4	Zeiger auf Information
Startadresse + 8	Zeiger auf linken Teilbaum
Startadresse + 12	Zeiger auf rechten Teilbaum

```
find:
    CMP EBX,0
    JE not_found
    CMP [EBX],ECX
    JE found
    JL left

right:
    MOV EBX,[EBX+12]
    JMP find

left:
    MOV EBX,[EBX+8]
    JMP find

not_found:
    MOV EAX,0
    RET

found:
    MOV EAX,[EBX+4]
    RET
```

Multiplikation in Festkommarechnung

Wenn man Vor- und Nachkommaanteil konkateniert und multipliziert, ist das Multiplikationsergebnis um die Nachkommastellen nach links verschoben. Dies muss nach der Multiplikation wieder korrigiert werden.

Beispiel mit 8+8 Vorkomma/Nachkommastellen:

a= 1.5 = (0000 0001 · 1000 0000 = 384)

b= 3.25 = (0000 0011 · 0100 0000 = 832)

$384 \cdot 832 = 319488 = 00000100111000000000000$

Verschiebung um n=8 Stellen nach rechts (Division durch 256)

0000 0100 · 1110 0000 = 4.875

Das heißt, Festkommamultiplikation geht mit normaler Ganzzahlmultiplikation, wenn man danach eine Korrektur durch Rechtsverschiebung bzw. Teilen durch die passende Zweierpotenz durchführt.

Absolutwert

```
absolut:
    PUSH EBX ;
    CMP EAX,0
    JGE end
    NEG EAX
end:
    POP EBX ;
    RET
```

Elemente mit dem nachfolgenden Element tauschen

```
MOV EAX,[EBX+8*ECX]
XCHG [EBX+8*ECX+8],EAX
MOV [EBX+8*ECX],EAX
MOV EAX,[EBX+8*ECX+4]
XCHG [EBX+8*ECX+12],EAX
MOV [EBX+8*ECX+4],EAX
```

Mittelwert von Listen

```
%include "io.inc"
section .data
list: dd 1,2,3,4,5,5,0
satz: db "Hallo!",0
char: db "l"

section .text
global CMAIN
CMAIN:
    ;write your code here
    mov edx, 0xFFFFFF
    push dword list
    sub esp, 4
    call mittelwert
    pop eax
    add esp, 4
    PRINT_DEC 4, eax
    ret
mittelwert:
    push ebp
    mov ebp, esp
    mov eax, [ebp + 12]
    cmp dword [eax] , 0
    je zero
    xor ebx, ebx
    xor ecx, ecx
list_loop:
    cmp dword [eax + 4 * ebx] , 0
    je divide
    add ecx, [eax + 4 * ebx]
    inc ebx
    jmp list_loop
 divide:
    mov edx, 0
    mov eax, ecx
    div ebx
    mov [ebp + 8], eax
    pop ebp
    ret
zero:
    mov dword [ebp + 8], 0
    pop ebp
    ret
ret
```

2

Anzahl an entsprechende Buchstaben in String

```
%include "io.inc"
section .data
satz: db "Hallo, AAAAlle",0

section .text
global CMAIN
CMAIN:
    mov ebp, esp; for correct debugging
    push byte 'a' ;A ; dec esp by 4
    push dword satz ; dec esp by 4
    call count_char
    add esp, 8
    PRINT_DEC 4, eax
    ret

count_char:
    push ebp
    mov ebp, esp
    mov ecx, [ebp+8]
    mov dl, [ebp+12]
    xor eax, eax
    xor ebx, ebx
string_loop:
    cmp byte [ecx+ ebx],0
    je ende
    cmp [ecx + ebx], dl
    jne weiter
    inc eax
weiter:
    inc ebx
    jmp string_loop
ende:
    pop ebp
    ret
```

3

größtes Element einer Matrix

```
%include "io.inc"
section .data
    matrix: dd 1,2,3,0,2,-10,9,3,0,1,3,-4,0,0
section .text
global CMAIN
CMAIN:
    mov eax, matrix
    call matrix_norm
    PRINT_DEC 4, eax
    ret


matrix_norm:; addr in eax, ret val eax
    mov ebx, eax
    mov eax, 0
    mov ecx, 0
m_loop:
    cmp dword [ebx + 4 * ecx], 0
    jne weiter
    cmp dword [ebx + 4 * ecx + 4], 0
    je ende
    jne increment
weiter:
    mov edx, [ebx + 4 * ecx]
    cmp edx, 0
    jge positive
    neg edx
positive:
    cmp edx, eax
    jle increment
    mov eax, edx
increment:
    inc ecx
    jmp m_loop
ende:
    ret
```

5

Buchstaben in Groß/ Klein umwandeln

```
%include "io.inc"
section .data
satz: db "hALLO!",0
section .text

global CMAIN
CMAIN:
    mov eax, satz
    call undo_caps
    PRINT_STRING satz
    ret

undo_caps:
    xor ecx, ecx

str_loop:
    cmp byte [eax + ecx], 0
    je ende
    mov bl, [eax + ecx]
    cmp bl, 65  ;A
    jl increment
    cmp bl, 90  ;Z
    jg kl_or_sz
    add bl, 32  ; GROSS->klein
    mov [eax + ecx], bl
    jmp increment

kl_or_sz:
    cmp bl, 97  ;a
    jl increment
    cmp bl, 122 ;z
    jg increment
    sub bl, 32  ;klein->GROSS
    mov [eax + ecx], bl

increment:
    inc ecx
    jmp str_loop

ende:
    ret
```

7

Palindrom Test

```
%include "io.inc"
section .data
satz: db "abccba",0
section .text
global CMAIN
CMAIN:
    mov eax, satz
    call palindrom
    PRINT_DEC 4, eax
    ret

palindrom:
    mov ecx, 0 ; length
    mov edx, 0 ; start index

length_loop:
    cmp byte [eax + ecx], 0
    je got_length
    inc ecx
    jmp length_loop

 got_length:
    cmp ecx, 0
    je null

 p_loop:
     cmp ecx, edx
     jle eins
     mov bl, [eax + edx]
     cmp bl, [eax + ecx-1]
     jne null
     inc edx
     dec ecx
     jmp p_loop

null:
    mov eax, 0
    ret
eins:
    mov eax, 1
    ret
```

8

String umdrehen

```
%include "io.inc"
section .data
satz: db "Hallo!",0
section .text
global CMAIN
CMAIN:
    mov eax, satz
    call reverse
    PRINT_STRING satz
    ret


reverse:
    mov ecx, 0

length_loop:
    cmp [eax+ecx], byte 0
    je got_length
    inc ecx
    jmp length_loop

got_length:
    cmp ecx, 0
    je ende
    mov ebx, 0
    dec ecx

rev_loop:
    cmp ebx, ecx
    jge ende
    mov dl, [ebx]
    xchg dl, [ecx]
    mov [ebx], dl
    inc ebx
    dec ecx
    jmp rev_loop

ende:
    ret
```

9

Bubble-Sort

```
%include "io.inc"
section .data
list: dd 4,3,2,1,0
section .text
global CMAIN
CMAIN:
    mov eax, list
    call bubble_sort
    PRINT_DEC 4, [list]
    NEWLINE
    PRINT_DEC 4, [list+4]
    NEWLINE
    PRINT_DEC 4, [list+8]
    NEWLINE
    PRINT_DEC 4, [list+12]
    NEWLINE
    ret

bubble_sort:
    mov ecx, 0
count_loop:
    cmp dword [eax+4*ecx],0
    je sort
    inc ecx
    jmp count_loop
sort:
    cmp ecx, 1
    jle ende
    dec ecx
    mov ebx, 0
sort_step:
    mov edx, [eax + 4 * ebx]
    inc ebx
    cmp dword [eax + 4 * ebx], 0
    je sort
    cmp edx, [eax + 4 * ebx]
    jle sort_step
    xchg edx, [eax + 4 * ebx]
    mov [eax + 4 * ebx -4], edx
    jmp sort_step
ende:
    ret
```

8

ggT

```
%include "io.inc"
section .text
global CMAIN
CMAIN:
    mov eax, 5
    mov ebx, 20
    call gcd
    PRINT_DEC 4, eax
    ret

gcd:
    ;eax:a
    ;ebx:b
    cmp ebx, 0
    je ende
    mov edx, 0
    div ebx
    mov eax, ebx
    mov ebx, edx
    jmp gcd
ende:
    ret
```

10

Anzahl der gesetzten Bits

```
push EAX  ; Zu sichern
mov ECX, 0  ; zähler als Rückgabewert
mov EAX, 0  ; Platzhalter für Bitsequenz
mov AL, 0b0000 0001 ; oder 0x01
loopbits:
test AL, DL
jnz count
increment:
shl AL, 1
cmp AL, 0
jnz loopbits
jmp exitbits

count:
add ECX,1 ; Increment bei Match
jmp increment

exitbits:
pop EAX  ; Wiederherstellen
ret
```

GesamtAnzahl der gesetzten Bits gerade oder ungerade

```
;push EAX  ; $N$
;push EBX  ; Base-Addresse
push ECX  ; Frei
;EDX  ; Rückgabewert

mov EDX, 0  ; Zählregister
mov ECX, -1 ; Offset
loop1:
inc ECX ; Offset+1
push ECX ; Rückgabewert von checkbyte!
push EDX  ; Abspeichern aktueller Zählstand ; Daten für checkbyte
mov EDX, 0  ; mit Null initialisiert; nicht nötig
mov DL, [EBX+ECX]  ; DL ist enthält die Bitsequenz N+Offset
call checkbyte
pop EDX  ; Wiederherstellen aktueller Zählstand
add EDX, ECX  ; Aufaddieren des Ergebnis
pop ECX  ; Wiederherstellen des Offset

cmp ECX, EAX ; Vergleiche ob beim letzten Byte
jl loop1

;Berechnung der Parität:
and EDX,1  ; 1 falls parität

; Ergebnis befindet sich bereits in EDX
pop ECX   ; Wiederherstellen des Wertes
;pop EBX  ; Wert wird nicht verändert!
;pop EAX  ; Wert wird nicht verändert!
ret    ; Routine
```

Logische Verknüpfungen

$a \wedge b$ -> Und-Verknüpfung: Gesamtaussage ist wahr, wenn beide Teilaussagen wahr sind -> Konjunktion

$a \vee b$ -> Oder-Verknüpfung: Gesamtaussage ist dann wahr, wenn mindestens eine der beiden Teilaussagen wahr ist -> Disjunktion

$a \equiv b$ -> Äquivalenz-Verknüpfung/ Negiertes Exklusives Oder (XNOR): Gesamtaussage ist dann wahr, wenn beide Aussagen den gleichen Wahrheitswert haben

$a \neq b$ -> Antivalenz/ $a \otimes b$ -> Exklusives Oder (XOR) a $\oplus$ b: Gesamtaussage ist wahr, wenn genau eine Teilaussage wahr ist

$a \Rightarrow b$ -> Implikation: Gesamtaussage ist nur dann falsch, wenn erste Teilaussage wahr und zweite Teilaussage falsch ist

Kommutativität: $a \wedge b \equiv b \wedge a$

Assoziativität: $(a \wedge b) \wedge c \equiv a \wedge (b \wedge c)$

Und, Oder, Äquivalenz und Antivalenz sind kommutativ und assoziativ

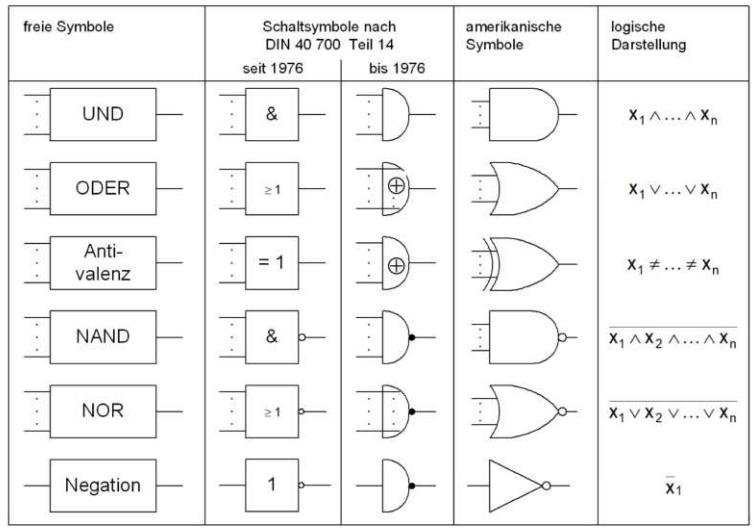
Implikation ist nicht kommutativ und nicht assoziativ

Addition ≡ Oder-Verknüpfung $a + b \equiv a \vee b$

Multiplikation ≡ Und-Verknüpfung $a * b \equiv a \wedge b$

Tautologie: Aussage, die immer wahr ist z.B. $a \vee \neg a$

Kontradiktion: Aussage, die immer falsch ist z.B. $a \wedge \neg a$



komplementär	Idempotenz	Absorption	De Mogansche Regeln
$a + \neg a = 1$	$a + a = a$	$(a + b) * a = a$	$(\neg a^* \neg b) = \neg(a + b)$
$a * \neg a = 0$	$a * a = a$	$(a * b) + a = a$	$(\neg a + \neg b) = \neg(a * b)$

*(a + b) * c = (a * c) + (b * c)*
*(a * b) + c = (a + c) * (b + c)*

Distributivität einer Operation bezüglich einer anderen

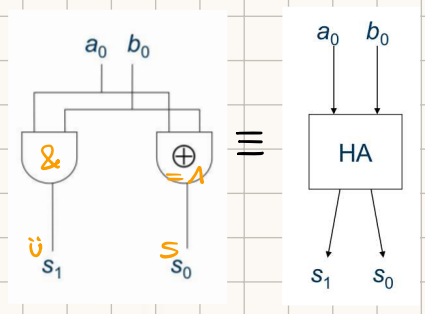
Eingabe x_1	Eingabe x_2	AND $x_1 \wedge x_2$	NAND $\overline{x_1 \wedge x_2}$	OR $x_1 \vee x_2$	NOR $\overline{x_1 \vee x_2}$	XOR $x_1 \oplus x_2$	XNOR $x_1 \equiv x_2$	Implikation $x_1 \Rightarrow x_2$	Implikation $x_2 \Rightarrow x_1$
0	0	0	1	0	1	0	1	1	1
0	1	0	1	1	0	1	0	1	0
1	0	0	1	1	0	1	0	0	1
1	1	1	0	1	0	0	1	1	1

a b Übertrag S_1/C_0 Summe S_0

Sum = X ⊕ Y
Carry = X · Y

n Eingänge, m Ausgänge => $(2^m)^{2^n}$ Schaltfunktionen

Halbaddierer

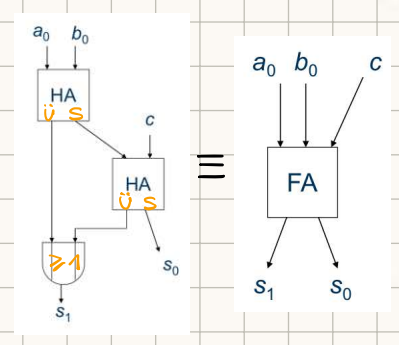


A	B	S	C_0
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$S = A \oplus B$
$C_0 = A \cdot B$

```
1  library IEEE;
2  use ieee.std_logic_1164.all;
3
4  entity halfadder is
5
6    port (
7      a, b       : in  std_logic;
8      sum, cout : out std_logic);
9
10 end entity halfadder;
11
12 architecture structural of halfadder is
13
14 begin  -- architecture structural
15
16   sum  <= a xor b;
17   cout <= a and b;
18
19 end architecture structural;
```

Volladdierer



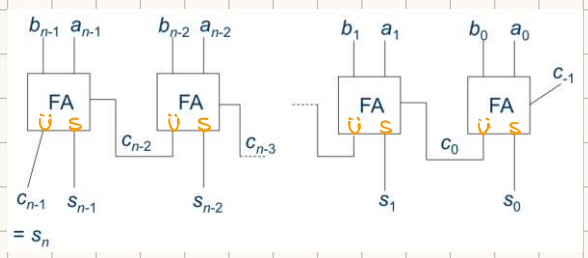
A	B	C_i	S	C_0
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	+ 0 = 1	+ 1 = 0	0	1
1	+ 1 = 0	0	0	1
1	+ 1 = 0	1	1	1

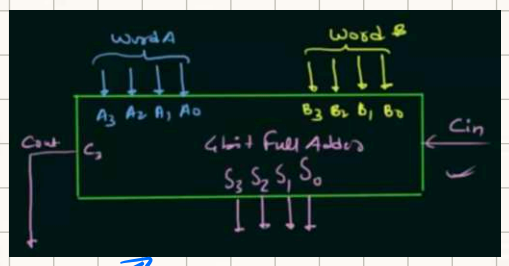
$S = A \oplus B \oplus C_i$

$C_0 = AB + A\bar{B}C_i + \bar{A}BC_i$
$= AB + C_i(A\bar{B} + \bar{A}B)$
$= AB + C_i(A \oplus B)$

```
23 library IEEE;
24 use ieee.std_logic_1164.all;
25
26 entity fulladder is
27
28   port (
29     a, b, cin : in  std_logic;
30     sum, cout : out std_logic);
31
32 end entity fulladder;
33
34 architecture structural of fulladder is
35
36   component halfadder is
37     port (
38       a, b       : in  std_logic;
39       sum, cout : out std_logic);
40   end component halfadder;
41
42   signal temp_sum, cout1, cout2 : std_logic;
43 begin  -- architecture structural
44
45   x : halfadder port map(a, b, temp_sum, cout1);
46   y : halfadder port map(cin, temp_sum, sum, cout2);
47   cout <= cout1 or cout2;
48 end architecture structural;
```

(Carry Rippel) Addierer






```vhdl
library IEEE;
use ieee.std_logic_1164.all;

entity adder is

  port (
    a, b     : in  std_logic_vector(3 downto 0);
    cin      : in  std_logic;
    sum      : out std_logic_vector(3 downto 0);
    overflow : out std_logic);

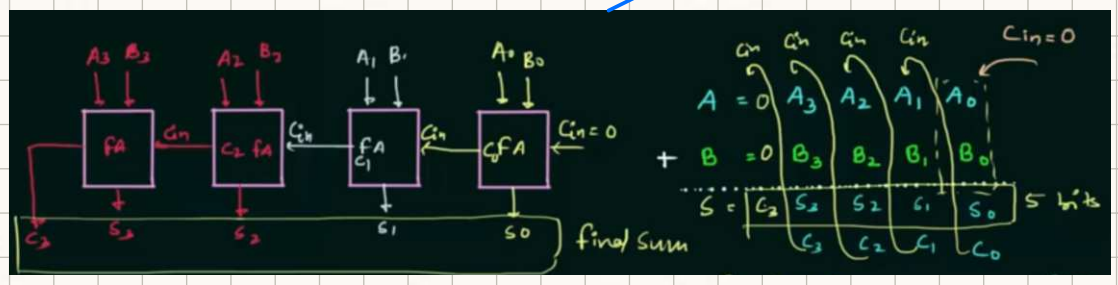
end entity adder;

architecture structural of adder is
  component fulladder is
    port (
      a, b, cin : in  std_logic;
      sum, cout : out std_logic);
  end component fulladder;

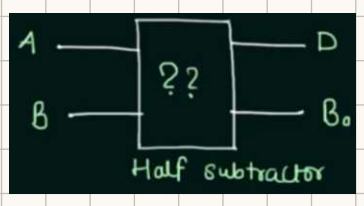
  signal carry    : std_logic_vector(2 downto 0);
  signal sum_temp : std_logic_vector(3 downto 0);
begin  -- architecture structural
  add1 : fulladder port map(a(0), b(0), cin, sum_temp(0), carry(0));
  add2 : fulladder port map(a(1), b(1), carry(0), sum_temp(1), carry(1));
  add3 : fulladder port map(a(2), b(2), carry(1), sum_temp(2), carry(2));
  add4 : fulladder port map(a(3), b(3), carry(2), sum_temp(3), overflow);

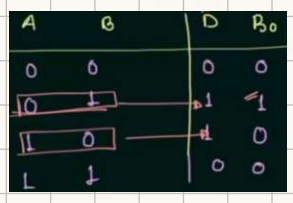
  sum <= sum_temp(3) & sum_temp(2) & sum_temp(1) & sum_temp(0);
end architecture structural;
```

4-Bit Volladdierer



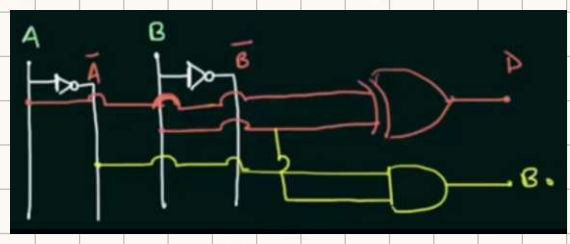
Halb Subtrahierer



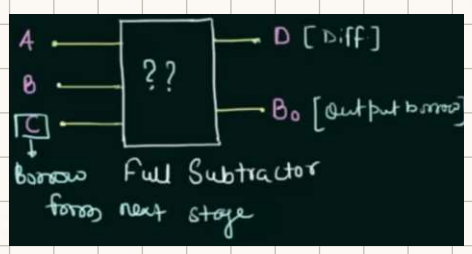


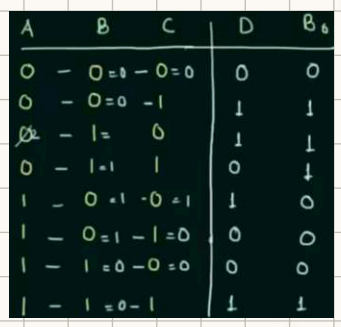
$$D = A \oplus B$$

$$B_0 = \bar{A} B$$



Voll Subtrahierer





$$D = A \oplus B \oplus C$$

$$B_0 = BC + \bar{A}C + \bar{A}$$

4-Bit Subtrahierer

```vhdl
library IEEE;
use ieee.std_logic_1164.all;

entity sub is

  port (
    a, b : in  std_logic_vector(3 downto 0);
    diff : out std_logic_vector(3 downto 0));

end entity sub;

architecture structural of sub is

  component adder
    port (
      a, b     : in  std_logic_vector(3 downto 0);
      cin      : in  std_logic;
      sum      : out std_logic_vector(3 downto 0);
      overflow : out std_logic);
  end component;

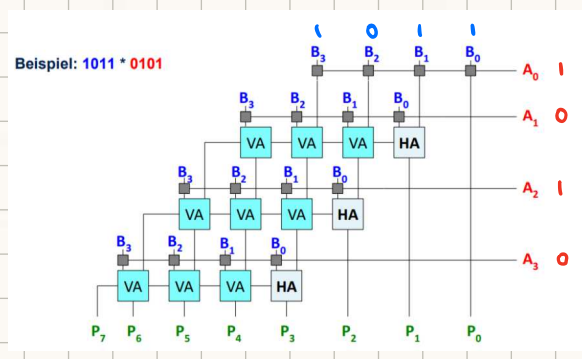
  signal b_comp : std_logic_vector(3 downto 0);
begin  -- architecture structural

  b_comp <= not b;

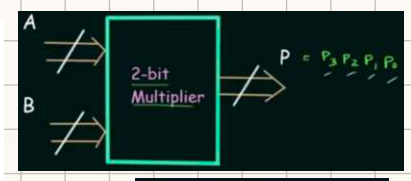
  adder_comp : adder port map(a, b_comp, '1', diff, open);

end architecture structural;
```

Multiplizierer



2-Bit Multiplizierer

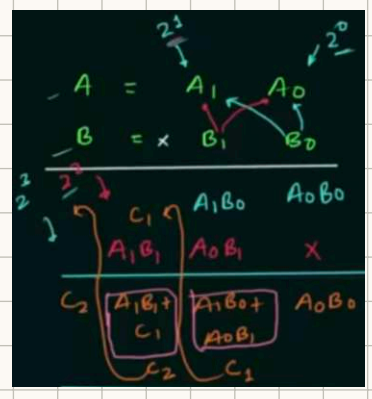


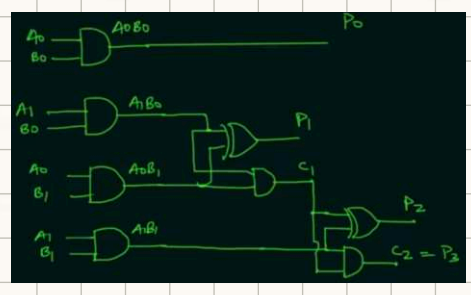
$$P_0 = A_0 B_0$$

$$P_1 = A_1 B_0 + A_0 B_1$$

$$P_2 = A_1 B_1 + C_1$$

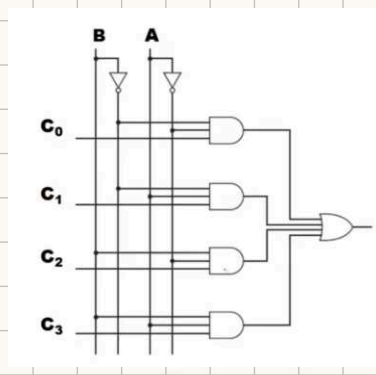
$$P_3 = C_2$$





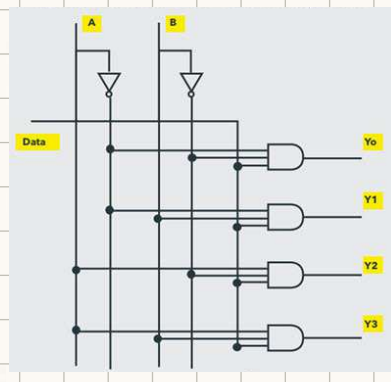
Multiplexer

- n Inputs
- k Steuereingänge
- 1 Output

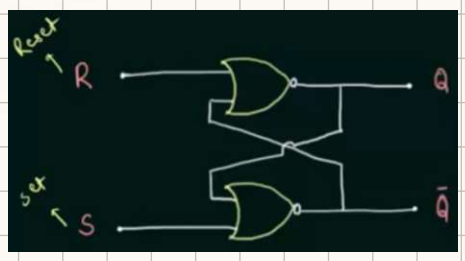


Demultiplexer

- 1 Input
- k Steuereingänge
- n Output



RS-FlipFlop (NOR-Gatter)

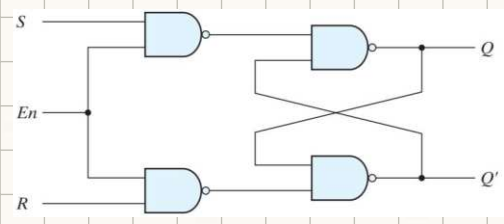


S	R	Q	$\bar{Q}$
0	0	Memory (as before)	
0	1	0	1
1	0	1	0
1	1	Not used	

NOR $Q = \bar{Q} = 0$

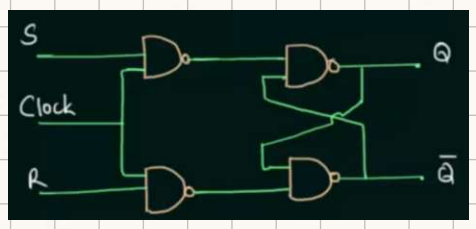
NAND $Q = \bar{Q} = 1$

RS-Latch



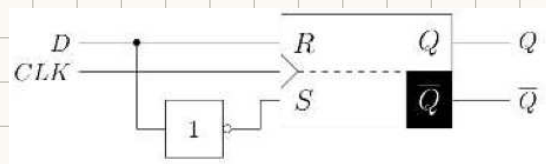
En	*S*	*R*	Next state of *Q*
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

RS-FlipFlop (mit clk)



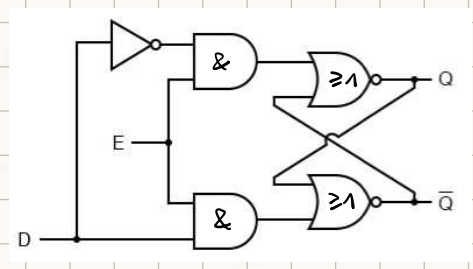
CLK	S	R	Q_{n+1}
0	x	x	Q_n (memory)
1	0	0	Q_n (memory)
1	0	1	0
1	1	0	1
1	1	1	Not used.

D-FlipFlop (mit clk)



clk	D	Q
0	X	memory
1	0	0
1	1	1

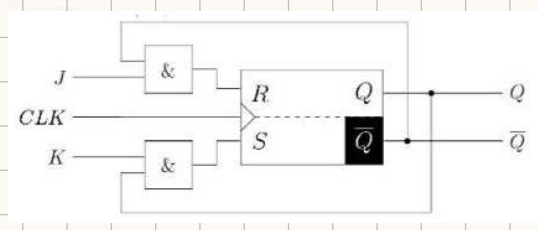
D-Latch



E	D	Q
0	X	memory
1	0	0
1	1	1

Q copys D

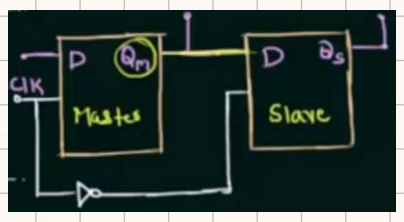
JK-FlipFlop

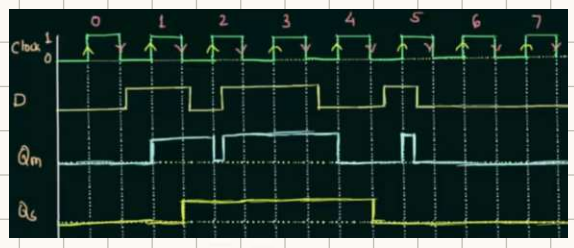


clk	J	K	Q
0	x	x	memory
1	0	1	0
1	1	0	1
1	1	1	$\bar{Q}$

$Q = 0, 1, 0, 1 \ldots$
immer wechselnd

Master Slave D-FlipFlop





D bei D-FlipFlop

Master Q_m übernimmt **bei steigender Flanke** den Ausgang des Schaltnetzes

Slave Q_s übernimmt **bei fallender Flanke** den Ausgang des Masters

VHDL - CRASHCOURSE

= Very High-Speed Integrated Circuit Hardware Description Language

Hardware BESCHREIBUNGS Sprache
- nicht interpretiert
- nicht sequenziell ausgeführt

↳ Beschreibung des Endprodukts, dann Umsetzung in die Hardware
→ zur Programmierung von FPGA's

FPGA: (= Field Programmeable Gate Arrays)
= ermöglicht (beim Start des Chips) neuen Code zu laden, r interne Gater steuert
- Schneller als normale CPUs, aber langsamer als fest verdrahtete Schaltungen (ASICs)
- Verwendung in Prototypen (Bevor Schaltung als Chips produziert wird)

Grundlagen

- Basiseinheit: signal <name> : <datentyp> [:=default];
- Datentypen:
  - std_logic → signal, entweder 0,1,Z,L,H,U,W,X
  - std_logic_vector → viele std_logic hintereinander (Strang von Leitungen / Bits)
  - unsigned → 1 Vektor an std_logic, die aber eine vorzeichenlose Zahl darstellen
  - signed → -"- vorzeichenbehaftete Zahl -"-

- Vektorendeklaration:
  - datentyp (x downto Y)
  - datentyp (x upto Y)
- Einzelauswahl (Signal an Index c)
  - <signalname> (c)
- Subvektor
  - <signalname> (c downto / upto d)
- Vektoren verbinden
  "&" ← z.B. "00" & "100" = "00100"

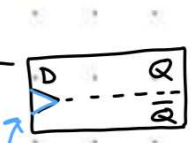
Operationen auf Signale

- Zuweisung: A <= B
- Bedingte Zuweisung: (immer alle Fälle abdecken!)

```
A <= <wert1> when <Bedingung1> else
     <wert2> when <Bedingung2> else
     <wert3>
```

- Bedingungen:
  - A = B, A /= B, <Bedingung1> and/or/xor/.../ <Bedingung2>
  - Bei Zahlen: A>B, A<B, A>=B, ...
  - A <= B and/xor/or ... C;
  - A <= not B
  - Bei Zahlen A<B+C, A-C, B*C, A / 2^n

Kleines Beispiel:



D-LATCH in VHDL:

```
latch template2:
process (D, clock)
begin
  if clock = '1' then Q <= D;
  end if;
end process;
```

Clock :)

```
signal a : std_logic_vector(3 downto 0);
```

a_3	a_2	a_1	a_0

```
signal a : std_logic_vector(0 to 3);
```

a_0	a_1	a_2	a_3

index

$$\begin{cases} a(1) \Rightarrow a_1 = 1 \\ \text{not } a(1) \Rightarrow a_1 = 0 \end{cases}$$

VHDL Übersicht

Die wichtigsten VHDL Syntax-Elemente auf einen Blick

Entity:

```
library ieee;
use ieee.logic_1164.all;
use ieee.numeric_std.all;

entity MyModule is
  generic (
      S : std_logic_vector (3 downto 0) := "0101";
      N : integer := 8
      );
  port (
      clk    : in  std_logic;
      reset  : in  std_logic;
      dataIn : in  std_logic_vector (3 downto 0);
      result : out std_logic_vector (N-1 downto 0)
      );
end;
```

Architecture:

```
architecture rtl of MyModule is
      -- Signal- und Typdefinitionen vor "begin" !
      constant ci  : integer := 7;
      signal   tmp : unsigned (4 downto 0);

      type T_STATE is (Init,Count,Open,Alarm);
      signal Zustand :  T_STATE;
begin
      -- Prozesse
      -- Signal- bzw. Portzuweisungen
      ...
end;
```

Prozesse:

Asynchroner Prozess

```
process (sig1,dataIn) -- Sensitivitätsliste!
      variable my_var : std_logic_vector (3 downto 0);
begin
      -- Anweisungen, z.B. if, case
end process;
```

Synchroner Prozess *(mit synchronem Reset)*

```
process -- Keine Sensitivitätsliste!
      variable my_other_var : unsigned (3 downto 0);
begin
      wait until rising_edge (clk);
      if reset = '1' then
            -- Resetverhalten
      else
            -- Anweisungen, z.B. if, case
      end if;
end process;
```

Synchroner Prozess *(mit asynchronem Reset)*

```
process (clk, reset) -- Sensitivitätsliste!
begin
      if reset = '1' then
            -- Resetverhalten
      elsif rising_edge (clk)
            -- Anweisungen, z.B. if, case
      end if;
end process;
```

Generell: *Entweder Sensitivitätsliste oder Wait-Anweisung*
Für Synthese: max. 1 Wait-Anweisung je Prozess

Instanziierung:

```
my_module_instance: entity work.MyModule
  generic map (
      S => "1101",
      N => 4
      )
  port map (
      clk    => clk_sig,  -- die Port-Namen aus
      reset  => reset,    -- der Entity stehen links
      dataIn => data_inp,
      result => res_out
      );
```

Bedingte Ausführung *(nur in Prozessen)*:

```
if Bedingung then
      ...
elsif Bedingung then
      ...
else
      ...
end if;

case Ausdruck is
   when "000" =>
      ...
   when "010" | "111" =>
      ...
   when others =>
      ...
end case;
```

Schleifen *(nur in Prozessen)*:

```
for i in 0 to 7 loop
      ...
end loop;

while Bedingung loop
      ...
end loop;
```

Datentypen:

integer	32 bit, vorzeichenbehaftet
std_logic std_logic_vector unsigned, signed	: 9-wertige Datentypen (u.a.: **0,1,U,X,-**)

Typumwandlung:

Integer nach (un)signed:	**to_(un)signed**(val,width)
(Un)signed nach Integer:	**to_integer**(val)
Std_logic_vector nach (un)signed:	(**un**)**signed**(val)
(Un)signed nach std_logic_vector:	**std_logic_vector**(val)

Signale vs. Variablen:

Variablen: Zuweisung erfolgt sofort (Verhalten wie z.B. in C/C++)

Signale/Out-Ports: Zuweisung erfolgt mit Unterbrechung/Ende des Prozesses
(nur die jeweils zuletzt ausgeführte Zuweisung wird "sichtbar")

Syntax der Zuweisung: `my_var := "1010";  my_sig <= "1010";`
(*Variablen* `:=` *Signale, Ports* `<=`)

Operatoren:

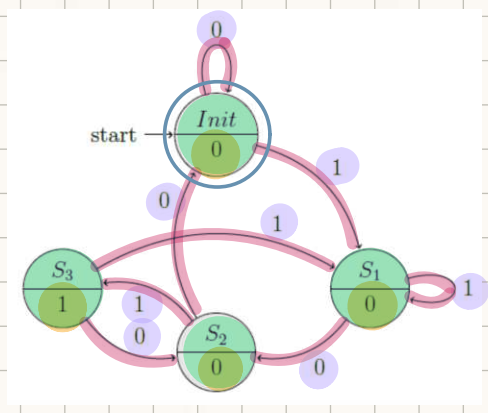
Symbol	Bedeutung	Datentypen	Rangfolge
abs	Absolutwert	integer, (un)signed	1 (hoch)
not	Invertierung	(un)signed, std_logic(_vector)	
**	Potenzierung	integer	
* /	Multiplikation, Division	integer, (un)signed	2
mod,rem	Modulo , Rest	integer, (un)signed	
+ -	Vorzeichen	integer, (un)signed	3
+ -	Addition, Subtraktion	integer, (un)signed	4
&	Vektoren zusammenfügen	(un)signed, std_logic(_vector)	
sll,sla,sra, srl,rol,ror	Schieben, Rotieren	(un)signed, std_logic_vector	5
= /=	Gleich, Ungleich	integer, (un)signed, std_logic(_vector)	6
< <= > >=	Kleiner/Größer	integer, (un)signed	
and or nand nor xor	Logische Verknüpfungen	(un)signed, std_logic(_vector)	7 (niedrig)

Konstanten *(Beispiel: Zahlenwert 254):*

Integer: `254  10#254#  16#FE#`
Vektor: `"11111110"  x"FE"`

Endlicher Automat

Zustandsdiagramm



mathematische Darstellung

Initialzustand $s_0 = Init$

Bedingung (Eingaben) $I = \{0, 1\}$

Zustandsübergangsfunktion

$\delta : S \times I \Longrightarrow S$
- $\delta(0, Init) \Longrightarrow Init$
- $\delta(1, Init) \Longrightarrow S_1$
- $\delta(0, S_1) \Longrightarrow S_2$
- $\delta(1, S_1) \Longrightarrow S_1$
- $\delta(0, S_2) \Longrightarrow Init$
- $\delta(1, S_2) \Longrightarrow S_3$
- $\delta(0, S_3) \Longrightarrow S_2$
- $\delta(1, S_3) \Longrightarrow S_1$

Zustände $S = \{Init, S_1, S_2, S_3\}$

Ausgaben/ outputs $O = \{0, 1\}$

Ausgabefunktion

$\lambda : S \Longrightarrow O$
- $\lambda(Init) \Longrightarrow 0$
- $\lambda(S_1) \Longrightarrow 0$
- $\lambda(S_2) \Longrightarrow 0$
- $\lambda(S_3) \Longrightarrow 1$

$Init = 00$

$S_1 = 01$

$S_2 = 10$

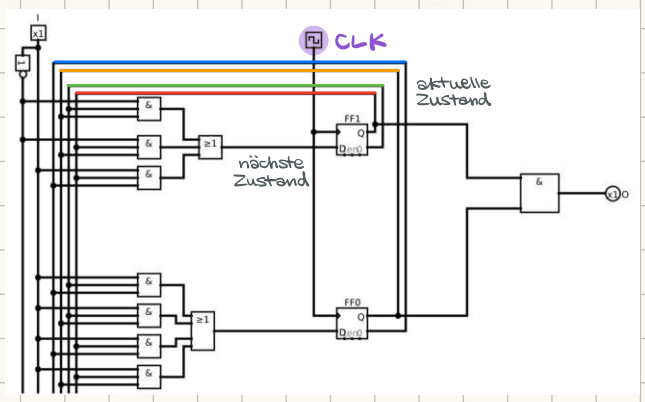
$S_3 = 11$

Wahrheitstabelle

I	FF1	FF0	FF1'	FF0'	O
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	0	1	1

Entsprechende Schaltung

$FF1$ $\overline{FF1}$ $FF0$ $\overline{FF0}$



Boolesche Terme

$$FF1' = \overline{I} \cdot \overline{FF1} \cdot FF0 + \overline{I} \cdot FF1 \cdot FF0 + I \cdot FF1 \cdot \overline{FF0}$$

$$FF0' = I \cdot \overline{FF1} \cdot \overline{FF0} + I \cdot \overline{FF1} \cdot FF0 + I \cdot FF1 \cdot \overline{FF0} + I \cdot FF1 \cdot FF0$$

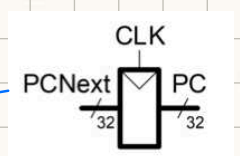
$$O = FF1 \cdot FF0$$

=> Output ist immer vom alten Zustand

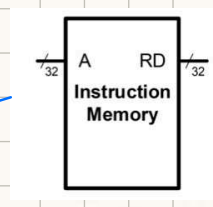
RISC-V

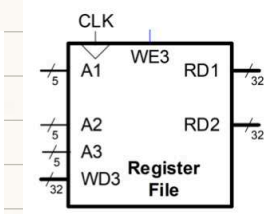
Prozessor-Komponente

Program Counter (PC)
Adresse des nächsten Befehls



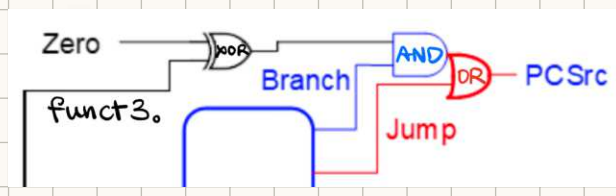
Instruktionsspeicher
A: Adresse
RD (Read Data): Gelesene Instruktion



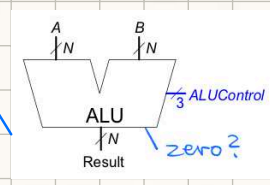


Registerbank
2 Ports zum Lesen von Operanden (RD1, RD2; zugehörige Adressen A1 und A2)
1 Port zum Schreiben des Resultats (WD3 und Adresse A3)
Es wird nur geschrieben wenn WE3=1 ist (Write Enable)

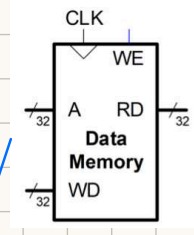
Name	Nummer	Beschreibung
`zero`	x0	Konstante 0 (kann nicht überschrieben werden)
`ra`	x1	Rücksprungadresse
`sp`	x2	Stack-Zeiger
`gp`	x3	Zeiger auf globale Daten
`tp`	x4	Zeiger auf thread-lokale Daten
`t0-2`	x5 – 7	Temporäre Register*
`s0/fp`	x8	Stack-Frame-Zeiger (optional)**
`s1`	x9	Gesicherte Register**
`a0-1`	x10 – 11	Funktionsargumente/Rückgabewerte
`a2-7`	x12 – 17	Funktionsargumente
`s2-11`	x18 – 27	Gesicherte Register**
`t3-6`	x28 – 31	Temporäre Register*



Arithmetic Logic Unit (ALU)
32-Bit Ausgang mit Ergebnis
1-Bit Ausgang, der angibt, ob Ergebnis 0 ist



Datenspeicher
A: Adresse zum Lesen/Schreiben (Wortadressierung)
RD (Read Data): Gelesenes Datum
WD (Write Data): Zu schreibendes Datenwort
Geschrieben wird nur bei positiver CLK-Taktflanke und wenn WE=1



R-Typ: Register-zu-Register Operationen
I-Typ: Short Immediates und Ladebefehle
U-Typ: Befehle für Long Immediates
S-Typ: Abspeichern von Datenworten
B-Typ: Bedingte Sprünge
J-Typ: Unbedingte Sprünge

PCSrc

Instruktion	Opcode	Befehlstyp / ImmSrc	RegWrite	ALUSrc	MemWrite	$ResultSrc_{1:0}$	$ALUOp_{1:0}$	$ALUControl_{1:0}$	Branch	Jump
LW	0000011 (3)	I-Typ / 00	1	1	0	01	00	000 (ADD)	0	0
SW	0100011 (35)	S-Typ / 01	0	1	1	--	00	000 (ADD)	0	0
OR	0110011 (51)	R-Typ / --	1	0	0	00	10	011 (OR)	0	0
ORI	0010011 (19)	I-Typ / 00	1	1	0	00	10	011 (OR)	0	0
XOR	0110011 (51)	R-Typ / --	1	0	0	00	10	XOR	0	0
XORI	0010011 (19)	I-Typ / 00	1	1	0	00	10	XOR	0	0
AND	0110011 (51)	R-Typ / --	1	0	0	00	10	010 (AND)	0	0
ANDI	0010011 (19)	I-Typ / 00	1	1	0	00	10	010 (AND)	0	0
BEQ	1100011 (99)	B-Typ / 10	0	0	0	--	01	001 (SUB)	1	0
BNE	1100011 (99)	B-Typ / 10	0	0	0	--	01	001 (SUB)	1	0
BGE	1100011 (99)	B-Typ / 10	0	0	0	--	10	101 (SLT)	1	0
JAL	1101111 (111)	J-Typ / 11	1	--	0	10	00 ?	---	--	1

RegWrite => nur bei sw und Branch wird 0, sonst 1 (Resultat abspeichern)

ALUSrc => nur bei sw, lw, I-Typ wird 1 (Speicheradresse berechnen)

MemWrite => nur bei sw wird 1 (Daten im Speicher ablegen), sonst 0

ALUOp => bei lw, sw 00, beq 01, alle andere 10

ResultSrc =>
- I-Typ, R-Typ 00
- lw 01 (gelesene Daten speichern)
- jal 10 (PC im Register speichern)
- B-Typ und sw --

ImmSrc =>
- I-Typ 00
- sw 01 Adresse berechnen
- B-Typ 10 Sprungadresse berechnen
- jal 11 Sprungadresse berechnen

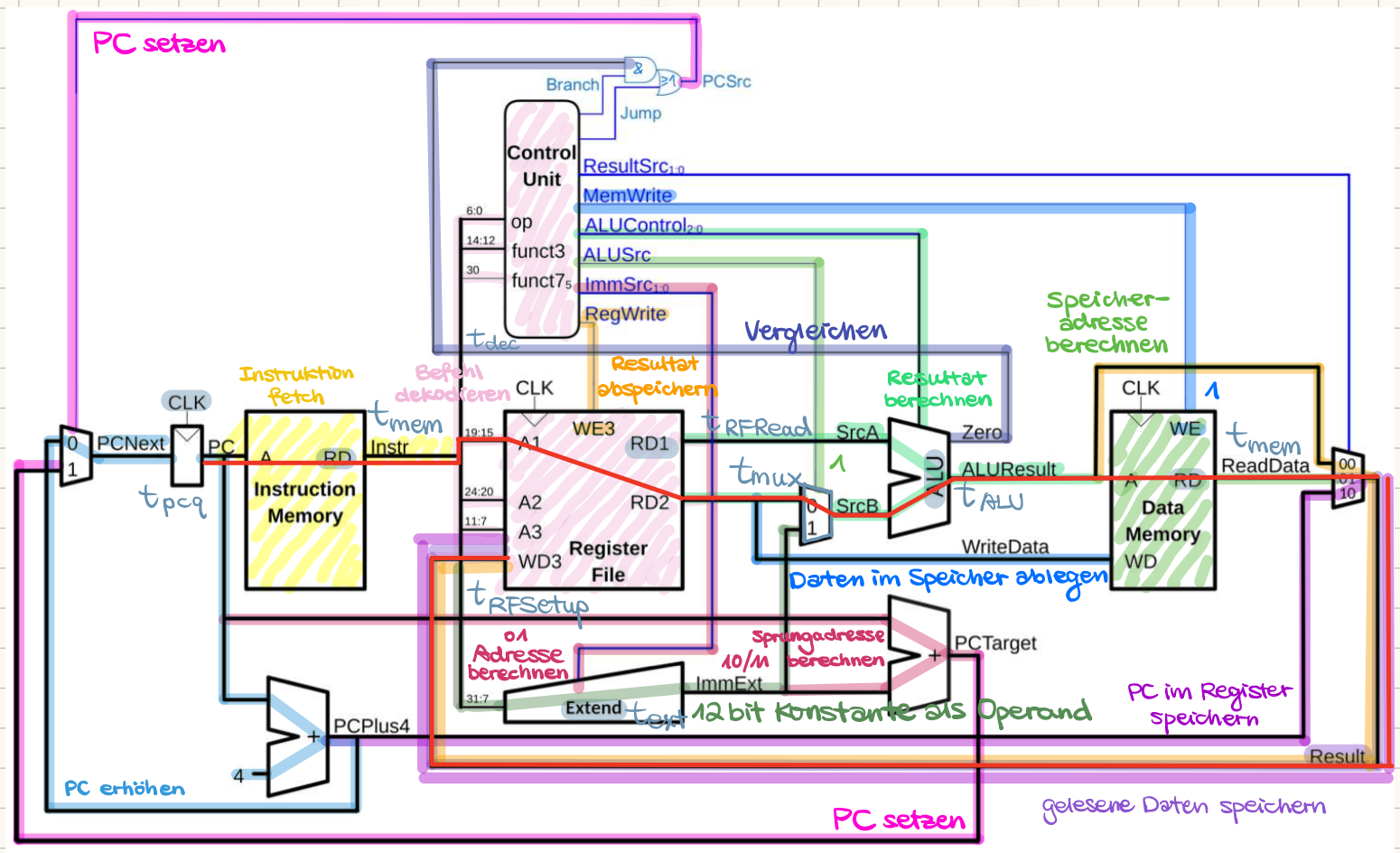
PCSrc => bei B- und J-Typ 1 (PC setzen)

SINGLE-CYCLE PROZESSOR

- + Einfach: Jede Instruktion wird in einem Takt ausgeführt
- - Getrennter Speicher für Instruktionen und Daten
- - Takt muss lang genug für langsamste Instruktion (lw) sein, auch wenn andere Instruktionen viel schneller arbeiten -> Performanceverlust
- - 2 Addierer notwendig (ALU, 2x für PC-Berechnung)

Single Cycle RISC-V

kritischer Pfad



Propagation Delay

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	40
AND gate	t_{AND}	10
OR gate	t_{OR}	10
ALU	t_{ALU}	100
Addierer	t_{add}	90
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Memory setup	$t_{MemSetup}$	150
Register file read	t_{RFRead}	170
Register file setup	$t_{FRSetup}$	60

Single Cycle RISC-V Beispiel für lw rd, imm(rs1)

Die blauen Linien des kritischen Pfads zeigen an welche Komponenten involviert sind. Der kritische Pfad beginnt mit dem Laden des PC bei steigender Taktflanke (t_{pcq}). Dann wird die neue Instruktion aus dem Instruction Memory ausgelesen (t_{mem}) und anschließend rs1 aus der Registerbank geladen (SrcA) (t_{RFRead}). Parallel zum Lesen der Registerbank wird das Immediate in der Extend Unit berechnet (t_{ext}) und als SrcB durch den Multiplexer vor der ALU geleitet (t_{mux}). Die ALU addiert beide Datenwörter um die Adresse zu berechnen (t_{ALU}). Aus dem Datenspeicher wird mit dieser Adresse dann gelesen (t_{mem}) und das Ergebnis durch den Result-Multiplexer geleitet (t_{mux}). Zum Schluss wird dann Result in die Registerbank geschrieben ($t_{RFSetup}$). Das Setup des Schreibens nimmt ebenfalls etwas Zeit in Anspruch bevor auf der steigenden Flanke dann geschrieben werden kann.

$$T_{c_single} = t_{pcq} + t_{mem} + \max[t_{RFRead}, t_{dec} + t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

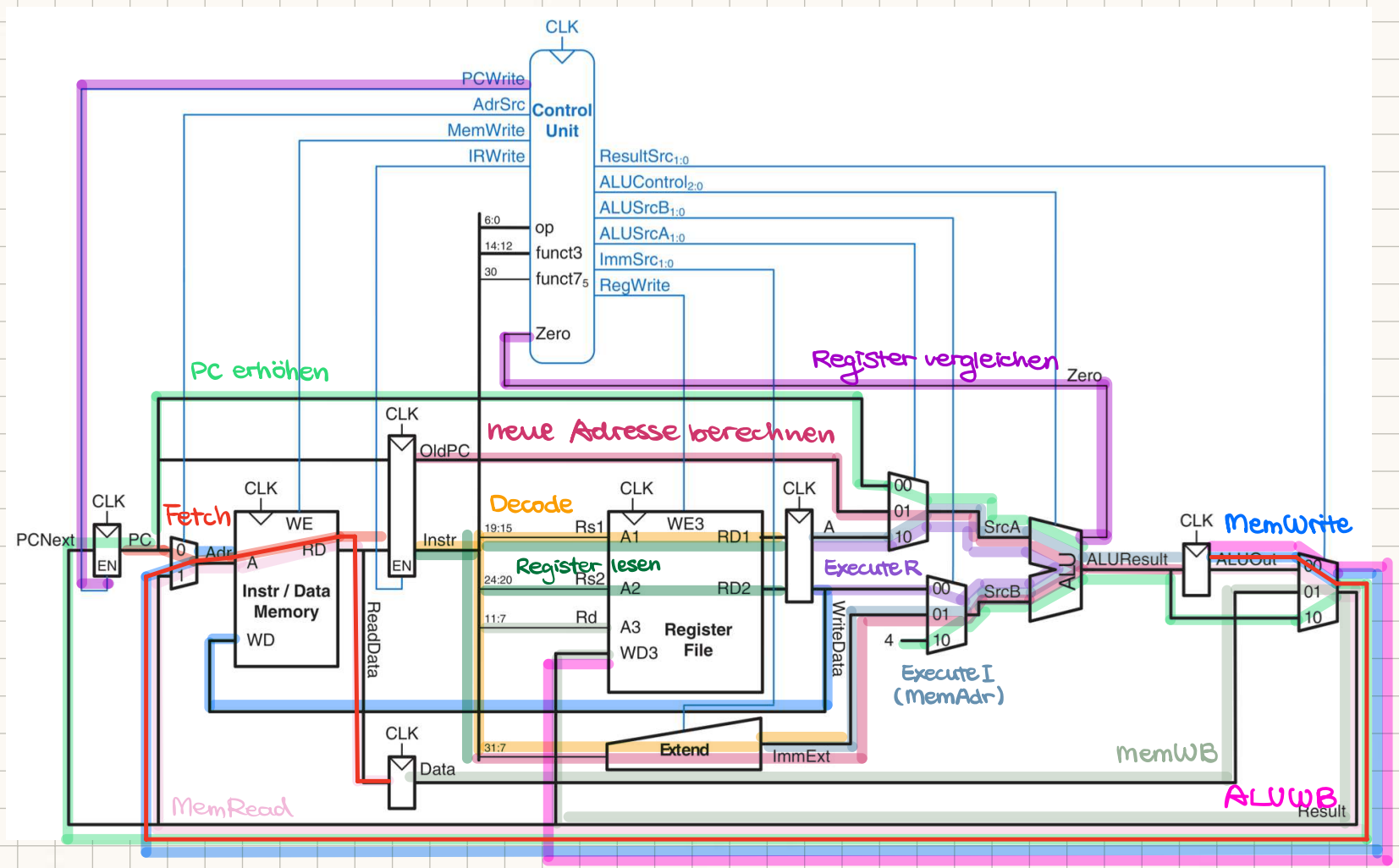
$$= t_{pcq} + 2t_{mem} + t_{RFRead} + t_{ALU} + t_{mux} + t_{RFsetup}$$

MULTICYCLE PROZESSOR

- Aufteilung einer Instruktion in Teilschritte -> In einem Takt wird nur ein Teilschritt ausgeführt
- Ausführung einer Instruktion benötigt mehrere Takte
- Höhere Taktgeschwindigkeit, da kürzere kritische Pfade
- Einfache Instruktionen sind schneller fertig
- Gemeinsamer Instruktions- und Datenspeicher
- Wiederverwendung von HW, z.B. nur ein Addierer erforderlich

Multi Cycle RISC-V

kritischer Pfad



Multi Cycle RISC-V Beispiel für lw rd, imm(rs1)

$$T_{c_multi} = t_{pcq} + t_{dec} + 2t_{mux} + \max[t_{ALU}, t_{mem}] + t_{setup}$$

Befehl Load Word
lw rd, imm(rs1)
① Fetch + PC setzen
② Decode
③ Adresse berechnen (MemAdr)
④ Wert laden (MemRead)
⑤ Abspeichern (MemWB)
5 Taktzyklen

Befehl Store Word
sw rs2, imm(rs1)
① Fetch + PC setzen
② Decode
③ Adresse berechnen (MemAdr)
④ Wert speichern (MemWrite)
4 Taktzyklen

I-Typ Befehl
addi rd, rs1, imm
① Fetch + PC setzen
② Decode (Execute I)
③ Operation ausführen
④ Resultat speichern (ALUWB)
4 Taktzyklen

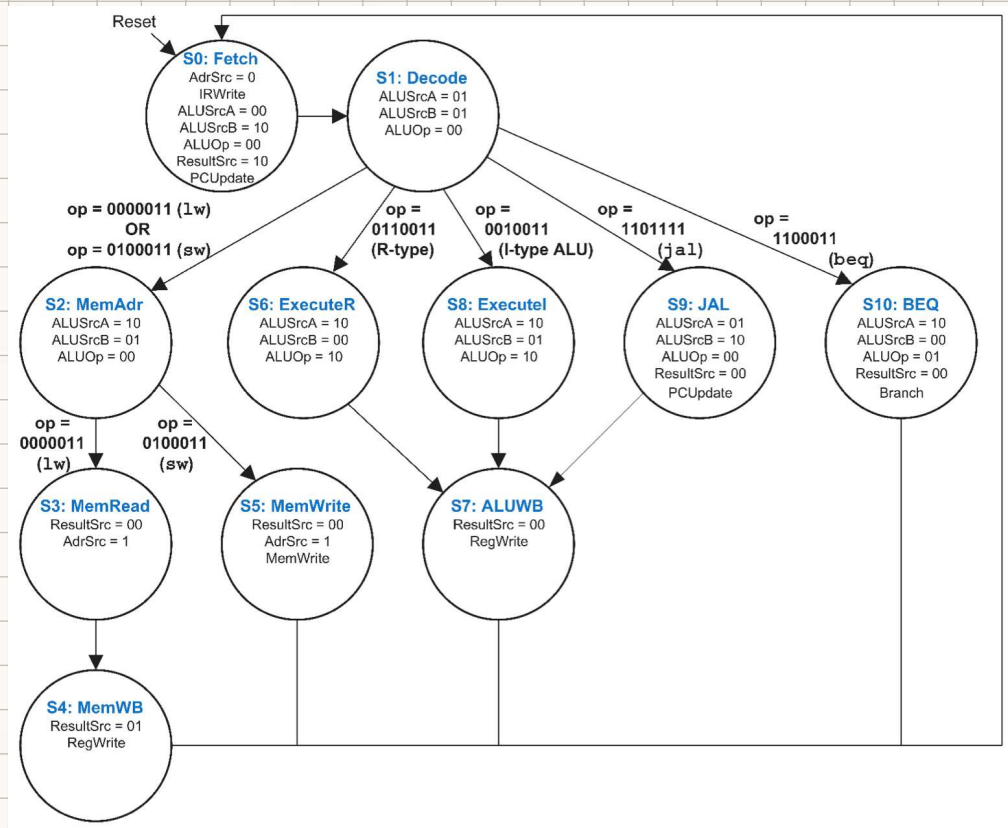
R-Typ Befehle
add rd, rs1, rs2
① Fetch + PC setzen
② Decode (Execute R)
③ Operation ausführen
④ Resultat speichern (ALUWB)
4 Taktzyklen

JAL Befehl
jal rd, label
① Fetch + PC setzen
② Decode
③ Sprung ausführen (JAL)
④ Resultat speichern (ALUWB)
4 Taktzyklen

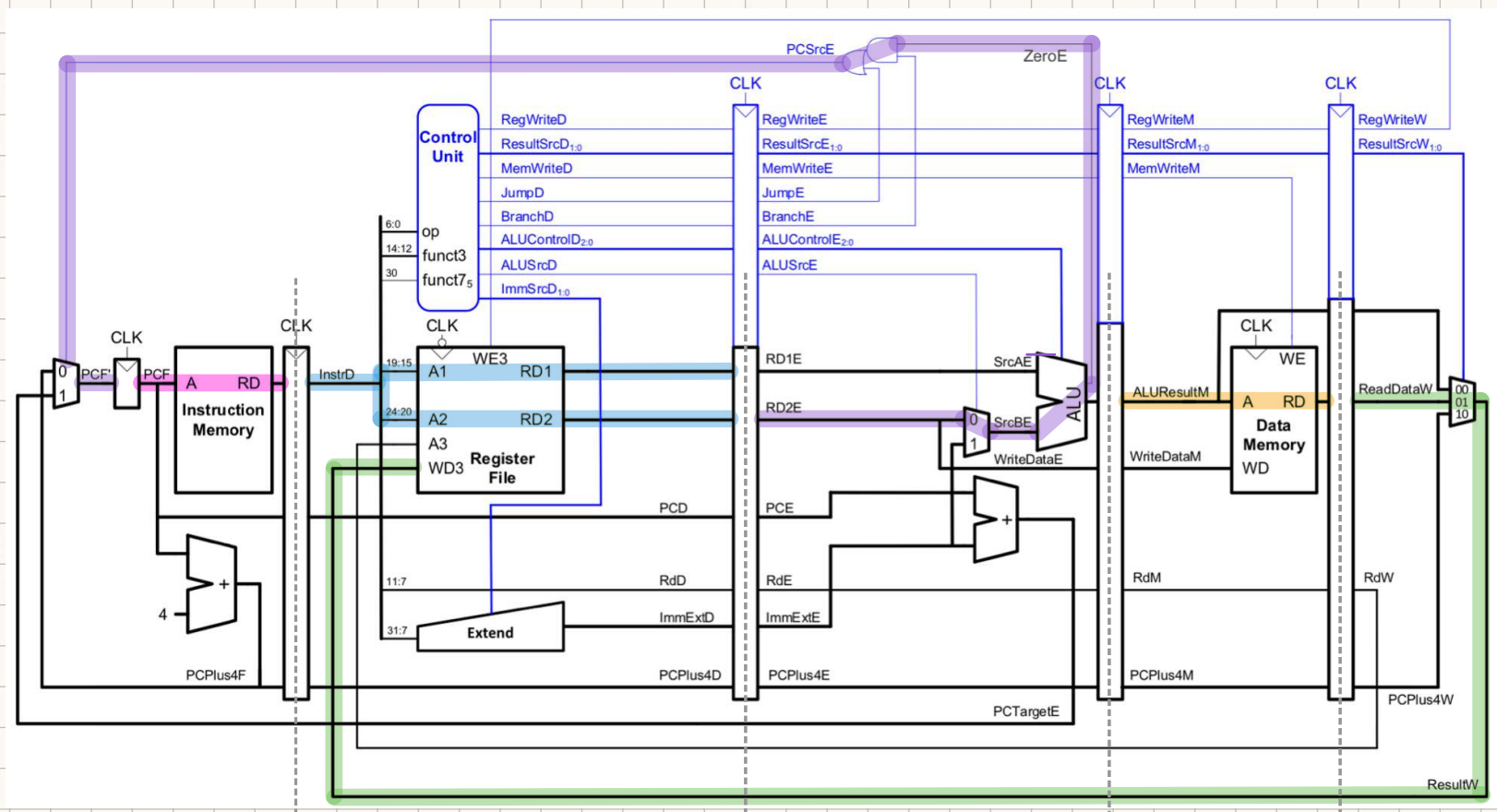
Branch Befehl
beq rs1, rs2, label
① Fetch + PC setzen
② Decode + Sprungsadresse berechnen
③ Sprung ausführen (BEQ)
3 Taktzyklen

Zustand	Datenpfad
Fetch	Instr ← Mem[PC]; PC ← PC + 4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1 op rs2
ExecuteI	ALUOut ← rs1 op imm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	ÜBUNG

Endlicher Automat des sequentiellen Steuerwerks



Pipelined RISC-V



1) Fetch — Befehls-Holphase

$t_{fetch} = t_{pcq} + t_{memRead} + t_{Setup} = 290ps$

2) Decode — Dekodierphase/Lesen von Operanden aus Registern

$t_{decode} = t_{pcq} + t_{RFRead} + t_{Setup} = 190ps$

3) Execute — Ausführung/Adressberechnung

$t_{exexute} = t_{pcq} + t_{mux} + t_{ALU} + t_{AND} + t_{OR} + t_{mux} + t_{Setup} = 290ps$

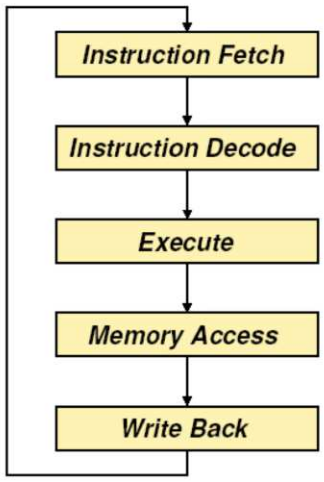
4) Memory — Speicherzugriff

$t_{memory} = t_{pcq} + t_{memRead} + t_{Setup} = 290ps$

5) Write Back — Abspeicherphase

$t_{writeback} = t_{pcq} + t_{mux} + t_{RFSetup} = 120ps$

Erinnerung: ein einfacher Befehlszyklus (leicht modifiziert)



Programmausführung & Takt-Anzahl

Singlecycle-Prozessor

→ Takt-Anzahl = Anzahl der ausgeführten Instruktionen

Multicycle-Prozessor

→ Takt-Anzahl = Summe der jeweils benötigten Takten

- Branch → 3 Takte
- Load Word → 5 Takte
- I-Typ, R-Typ, J-Typ, Store Word → 4 Takte

Pipeline-Prozessor

→ Takt-Anzahl = Anzahl der ausgeführten Instruktionen (inklusiv NOPs) + 4 Takte (bis alles fertig)

Pipelining-Speedup

Abarbeitungszeit eines Befehls ohne Pipelining: t
k Pipelinestufen, gleiche Laufzeit der Stufen
→ Laufzeit einer Stufe der Pipeline: t/k

m (m ≥ 1) auszuführenden Instruktionen →
- Laufzeit mit Pipeline = $t + (m-1) * \frac{t}{k}$
- Laufzeit ohne Pipeline = $m * t$

Beschleunigung um Faktor →

$$\frac{Laufzeit\ ohne\ PL}{Laufzeit\ mit\ PL} = \frac{m*t}{t+(m-1)*\frac{t}{k}} = k - \frac{k*(k-1)}{m+k-1}$$

Ergebnis: Für m >> k nähert sich der Speedup also der Anzahl k der Pipelinestufen

Pipeline-Konflikte (entstehen durchs parallele Ausführen)

① **Datenkonflikte** (data hazards) entstehen durch Datenabhängigkeiten

→ Beispiel

Ursprüngliche Befehlsfolge

```
S1 addi s2, s1, 5
S2 add  s1, s2, s3
S3 sub  s4, s5, s6
S4 xor  s7, s5, s3
S5 or   s6, s5, s5
```

→

Zeitschritt	F	D	E	M	W
1	addi				
2	add	addi			
3	sub	add	addi		
4	xor	sub	add	addi	
5	or	xor	sub	add	addi
6		or	xor	sub	add

Data Hazard in Zeitschritt 3 zwischen S1 (addi) und S2 (add):
s2 wird in S2 gelesen, aber s2 wurde von S1 noch nicht geschrieben

→ Lösen

Methode 1: Einfügen von NOPs (no operation)

(min. 3 andere Befehle)

*für B-/J-Typ Befehle: 2 NOPs

Zeitschritt	F	D	E	M	W
1	addi				
2	NOP	addi			
3	NOP	NOP	addi		
4	NOP	NOP	NOP	addi	
5	add	NOP	NOP	NOP	addi
6		add	NOP	NOP	NOP

Methode 2: Befehle umordnen

Neue Befehlsfolge

```
S1 addi s2, s1, 5
S3 sub  s4, s5, s6
S4 xor  s7, s5, s3
S5 or   s6, s5, s5
S2 add  s1, s2, s3
```

→

Zeitschritt	F	D	E	M	W
1	addi				
2	sub	addi			
3	xor	sub	addi		
4	or	xor	sub	addi	
5	add	or	xor	sub	addi
6		add	or	xor	sub

② **Steuerkonflikte** (control hazards) entstehen durch Änderung des Kontrollflusses

→ Beispiel

Ursprüngliche Befehlsfolge

```
S1 beq s4, zero, label
S2 add s1, s2, s3
S3 sub s5, s6, s7
…
S4 label: xor s5, s6, s7
```

→

Zeitschritt	F	D	E	M	W
1	beq				
2	add	beq			
3	sub	add	beq		
4	xor	sub	add	beq	

control hazards
- Sprungbedingung wird erst in Execute ausgewertet (im Bsp: falls s4 == 0, dann wird ALU-Ausgang ZeroE = 1 und dann PCSrcE = 1 und damit PCF' = PCTargetE)
- Ausführung des add- und sub-Befehls wird hier fehlerhafterweise begonnen, auch wenn gesprungen wird

→ Lösen

Methode 1: Einfügen von NOPs (NOP = no operation)

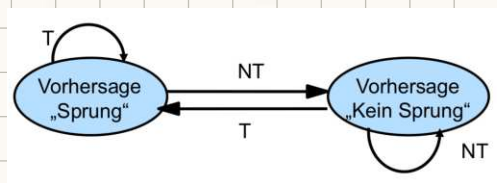
Methode 2: Sprungrichtungsvorhersage: „Raten" der nächsten Instruktion und spekulativer Sprung (z.B. durch Analysen der Sprungrichtung-Häufigkeit)
- **Schleifen:** Wahrscheinlichkeit hoch, dass gesprungen wird
- **if-then-else:** 2/3 der Ausführungen gehen in den ELSE-Teil

Sprungvorhersage (Branch Prediction)

Definition: Um Steuerkonflikte und das damit verbundene Pausieren der Pipeline zu vermeiden, kann vorhergesagt werden, wohin der Sprung gehen wird
Bei bedingter Sprung werden spekulativ die Verzögerungsphasen mit den folgenden Befehlen gefüllt
Nach Auswertung der Sprungbedingung wird die momentane Pipeline fortgeführt oder verworfen (-> "Branch Miss Prediction")

2 Typen von Sprungvorhersge

- **Statische Sprungvorhersage**
  - Es wird immer der gleiche Weg eines Sprungs ausgeführt (z.B. immer true)
  - im Durchschnitt in 50% aller Fälle richtig
- **Dynamische Sprungvorhersage**
  - Es wird eine Vorhersage abhängig von den vorherigen Sprüngen an dieser Adresse getroffen
  - als endlicher Automat betrachtet - bei einer falschen Vorhersage den Zustand wechselt (1-bit Vorhersage)



$ImmSrc_{1:0}$	ImmExt	Instr. Type
00	{{20{instr[31]}}, instr[31:20]}	I-Type
01	{{20{instr[31]}}, instr[31:25], instr[11:7]}	S-Type
10	{{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	B-Type
11	{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}	J-Type

(a) Extend Unit

RISC-V Instruction Set Summary

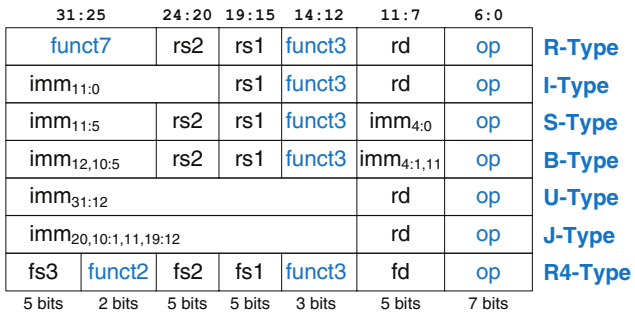


Figure B.1 RISC-V 32-bit instruction formats

- `imm`: signed immediate in $\mathbf{imm}_{11:0}$
- `uimm`: 5-bit unsigned immediate in $\mathbf{imm}_{4:0}$
- `upimm`: 20 upper bits of a 32-bit immediate, in $\mathbf{imm}_{31:12}$
- `Address`: memory address: **rs1** + SignExt($\mathbf{imm}_{11:0}$)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({$\mathbf{imm}_{12:1}$, 1'b0})
- `JTA`: jump target address: PC + SignExt({$\mathbf{imm}_{20:1}$, 1'b0})
- `label`: text indicating instruction address
- `SignExt`: value sign-extended to 32 bits
- `ZeroExt`: value zero-extended to 32 bits
- `csr`: control and status register

Table B.1 RV32I: RISC-V integer instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	`rd = [Address]31:0`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]15:0)`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	`[Address]7:0 = rs27:0`
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	`[Address]15:0 = rs215:0`
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	`[Address]31:0 = rs2`
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	`rd = rs1 − rs2`
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	`rd = rs1 << rs24:0`
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	`rd = (rs1 < rs2)`
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	`rd = (rs1 < rs2)`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	`rd = rs1 >> rs24:0`
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>> rs24:0`
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	`rd = rs1 & rs2`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

*Encoded in $instr_{31:25}$, the upper seven bits of the immediate field

Literal: Variable oder deren Negation
Monom: Verundung von Literalen (wobei jede Variable nur max. einmal vorkommt)
Minterm: vollständiges Monom, wenn jede Variable entweder als positives oder als negatives Literal vorkommt
Polynom: Veroderung von Mintermen
Implikant: Monom, das die Funktion impliziert
Primimplikant: Implikant, der nicht weiter verkürzt werden kann

Quine-McCluskey

Dezimalzahlen	A	B	C	D	Ausgangswert Q
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	X
3	0	0	1	1	0
4	0	1	0	0	X
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	1

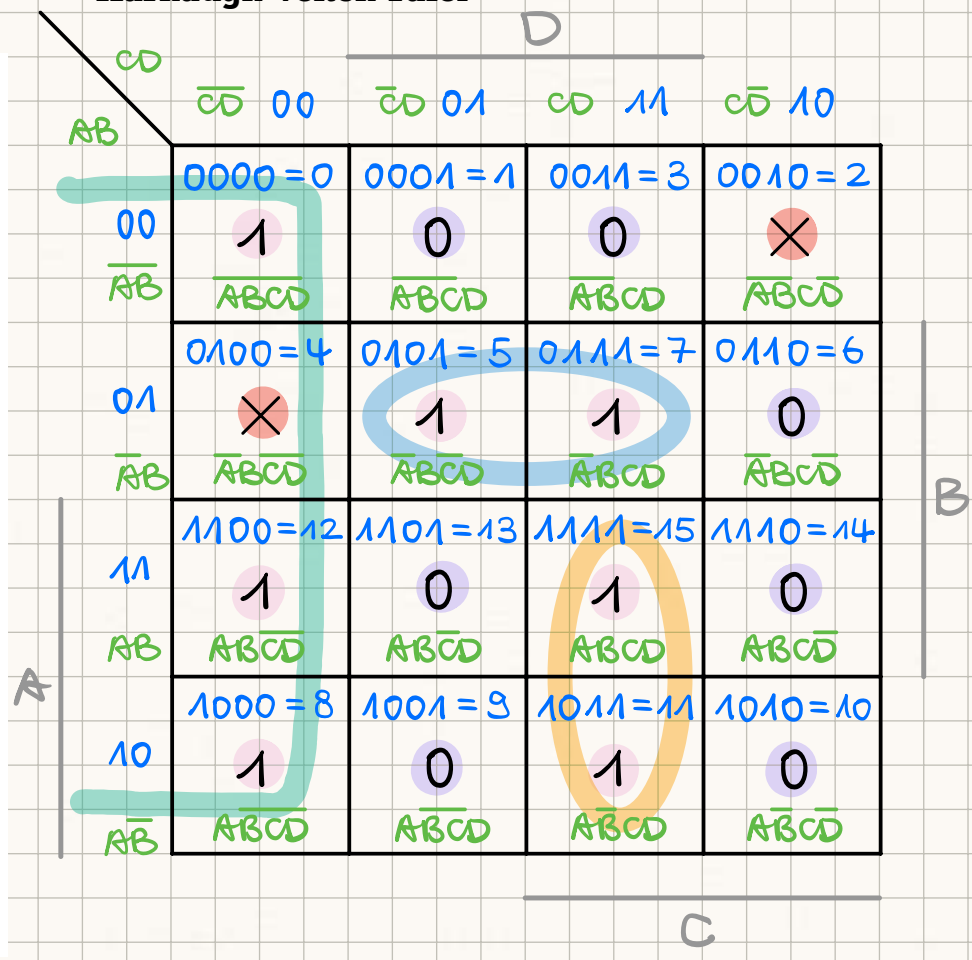
Bei n Eingangsvariablen gibt es 2^n Felder
Die Anzahl der Felder im Block entspricht einer 2-er Potenz, also 2, 4, 8, 16

Blöcke können nur horizontal, vertikal oder quadratisch auftreten.
Die Bereiche können über die Seiten des KV-Diagramms hinaus reichen
Bleiben am Ende einzelne „1"-Felder übrig werden diese allein eingekreist

Die Ausgangsvariable Q ist an den beiden mit einem X gekennzeichneten Stellen frei wählbar und die zugehörigen Eingangskombinationen heißen „frei wählbare Ausdrücke" oder „don´t care positions". Sie können in redundanten logischen Verknüpfungen zur Vereinfachung logischer Ausdrücke in KV-Tafeln verwendet werden. Dazu werden die Kreuze zusätzlich zu den Einsen eingetragen und können zum Bilden von Gruppen herangezogen oder auch nicht berücksichtigt werden.

Minterme	A	B	C	D
2	0	0	1	0
0, 4, 8, 12	-	-	0	0
4, 5	0	1	0	-
5, 7	0	1	-	1
7, 15	-	1	1	1
11, 15	1	-	1	1

Karnaugh-Veitch Tafel



Minterm (DNF): alle "1" $(\bar{A} \wedge \bar{B} \wedge \bar{C} \wedge \bar{D}) \vee \ldots$

Maxterm (KNF): alle "0"

$MP = (\bar{C} \wedge \bar{D}) \vee (\bar{A} \wedge B \wedge D) \vee (A \wedge C \wedge D)$

$= \bar{C}\bar{D} + \bar{A}BD + ACD$

$X = (\bar{A} \wedge B \wedge \bar{C} \wedge \bar{D}) \vee (\bar{A} \wedge \bar{B} \wedge C \wedge \bar{D})$

$= \bar{A}B\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D}$

Reduktionsregel

1. Entferne aus der Primimplikantentafel PIT alle wesentlichen Primimplikanten und alle Minterme, die von diesen überdeckt werden
2. Entferne aus der Primimplikantentafel PIT alle Minterme, die einen anderen Minterm in PIT dominieren -> Bei **Spaltendominanz** wird die **dominierende** Spalte gestrichen
3. Entferne aus PIT alle Primimplikanten, die durch einen anderen, nicht teureren Primimplikanten dominiert werden -> Bei **Zeilendominanz** wird die **dominierte** Zeile gestrichen

dominierend (-> hat mehr "1"); **dominiert** (hat weniger "1")

Primmimplikantentafel (PIT)

unnötige X kann zuerst gestrichen werden

(hier Zeilendominanz)

Primmimplikanten	0	2 (x)	4 (x)	5	7	8	11	12	15
$\bar{A}B\bar{C}\bar{D}$		x							
$\bar{C}\bar{D}$	x		x			x		x	
$\bar{A}B\bar{C}$			x	x					
$\bar{A}BD$				x	x				
BCD					x				x
ACD							x		x

dominiert dominiert dominiert dominiert

wesentlicher Primmimplikant

Minimalpolynom $MP = \bar{C}\bar{D} + \bar{A}BD + ACD = 0 + 4 + 12 + 8 + 5 + 7 + 15 + 11 = 62$

zyklische Überdeckungsproblem

$$
\begin{aligned}
m(0,2): &\quad \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_5\\
m(0,4): &\quad \overline{x}_1\overline{x}_2\overline{x}_4\overline{x}_5\\
m(0,8): &\quad \overline{x}_1\overline{x}_3\overline{x}_4\overline{x}_5\\
m(2,3): &\quad \overline{x}_1\overline{x}_2\overline{x}_3x_4\\
m(4,5): &\quad \overline{x}_1\overline{x}_2x_3\overline{x}_4\\
m(4,20): &\quad \overline{x}_2x_3\overline{x}_4\overline{x}_5\\
m(3,7): &\quad \overline{x}_1\overline{x}_2x_4x_5\\
m(5,7): &\quad \overline{x}_1\overline{x}_2x_3x_5\\
m(24,28): &\quad x_1x_2\overline{x}_4\overline{x}_5\\
m(8,9,24,25): &\quad x_2\overline{x}_3\overline{x}_4\\
m(20,22,28,30): &\quad x_1x_3\overline{x}_5
\end{aligned}
$$

$$
\begin{aligned}
p_1 &= m(0,2)\\
p_2 &= m(0,4)\\
p_3 &= m(2,3)\\
p_4 &= m(4,5)\\
p_5 &= m(3,7)\\
p_6 &= m(5,7)
\end{aligned}
$$

	0	2	3	4	5	7
m(0, 2)	X	X				
m(0, 4)	X			X		
m(2, 3)		X	X			
m(4, 5)				X	X	
m(3, 7)			X			X
m(5, 7)					X	X

Nach Petrick's Methode müssen alle Zeilen summiert (verodert) werden die die gleichen Minterme überdecken. Aus diesen Summanden wird anschließend das Produkt (Verundung) gebildet und ausmultipliziert. Da die Primimplikanten p_i insbesondere Boolsche Variable (in Abhängigkeit der x_j) sind, können wir die Regeln der Boolschen Algebra verwenden um die folgende Berechnung zu vereinfachen. Die Identitäten $x \cdot x = x$ und $x + xy = x$ sind besonders hilfreich.

$$
\begin{aligned}
&\underbrace{(p_1+p_2)(p_1+p_3)}\underbrace{(p_2+p_4)(p_3+p_5)}\underbrace{(p_4+p_6)(p_5+p_6)} =\\
&(p_1+p_1p_3+p_1p_2+p_2p_3)(p_2p_3+p_2p_5+p_3p_4+p_4p_5)(p_4p_5+p_4p_6+p_5p_6+p_6) =\\
&\underbrace{(p_1+p_2p_3)}(p_2p_3+p_2p_5+p_3p_4+p_4p_5)(p_4p_5+p_6) =\\
&(p_1p_2p_3+p_1p_2p_5+p_1p_3p_4+p_1p_4p_5+p_2p_3+p_2p_3p_5+p_2p_3p_4+p_2p_3p_4p_5)(p_4p_5+p_6) =\\
&(p_1p_2p_5+p_1p_3p_4+p_1p_4p_5+p_2p_3)(p_4p_5+p_6) =\\
&p_1p_2p_4p_5+p_1p_2p_5p_6+p_1p_3p_4p_5+p_1p_3p_4p_6+p_1p_4p_5+p_1p_4p_5p_6+p_2p_3p_4p_5+p_2p_3p_6 =\\
&p_1p_2p_5p_6+p_1p_3p_4p_6+p_1p_4p_5+p_2p_3p_4p_5+p_2p_3p_6
\end{aligned}
$$

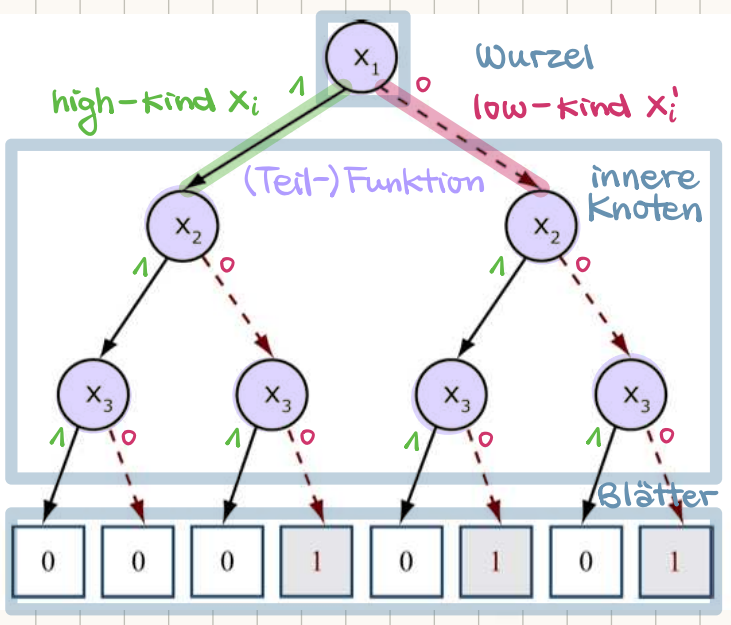
© 2023 Fachschaft für Praktische Informatik Praktische Systeme in Rostock vorbehalten

Die grün markierten Monome in der letzen Zeile haben minimale Kosten. Es gibt also 2 Lösungen für das Überdeckungsproblem. Die möglichen Minimalpolynome sind also wie folgt.

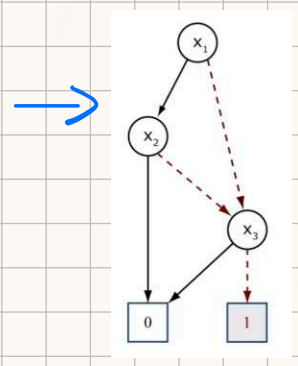
(i) $f(x_1,x_2,x_3,x_4,x_5) = x_2\overline{x}_3\overline{x}_4 + x_1x_3\overline{x}_5 + \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_5 + \overline{x}_1\overline{x}_2x_3\overline{x}_4 + \overline{x}_1\overline{x}_2x_4x_5$

(ii) $f(x_1,x_2,x_3,x_4,x_5) = x_2\overline{x}_3\overline{x}_4 + x_1x_3\overline{x}_5 + \overline{x}_1\overline{x}_2\overline{x}_4\overline{x}_5 + \overline{x}_1\overline{x}_2\overline{x}_3x_4 + \overline{x}_1\overline{x}_2x_3x_5$

Binäres Entscheidungsdiagramm (BDD)

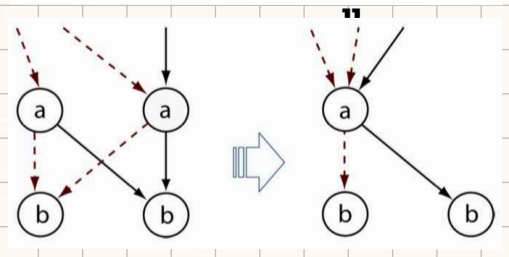


nach Reduktion

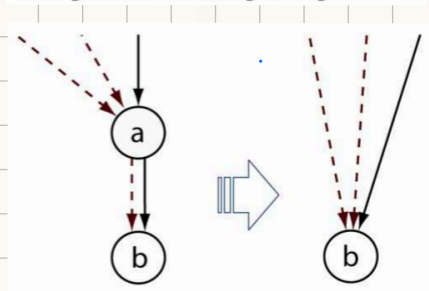


Reduktionsregel

Regel 1: Isomorphismus
Führe ismomorphe Knoten zusammen

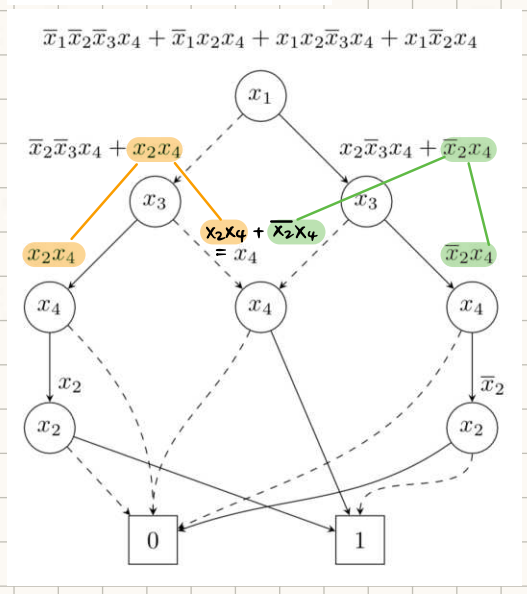


Regel 2: Elimination
Entferne Knoten, dessen beide Kinder zum gleichen Nachfolger zeigen



Beispiel

(b) $x_1 < x_3 < x_4 < x_2$



Shannon Zerlegung

$f_v = (x_j \wedge f_{high(v)}) \vee (x_j' \wedge f_{low(v)})$

$x_i = 1$ $\quad$ $x_i = 0$

Kofaktor von f

- in Bezug auf $x_i = 1$: $f_{x_i=1} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$
- in Bezug auf $x_i = 0$: $f_{x_i=0} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$

Speicheraufbau

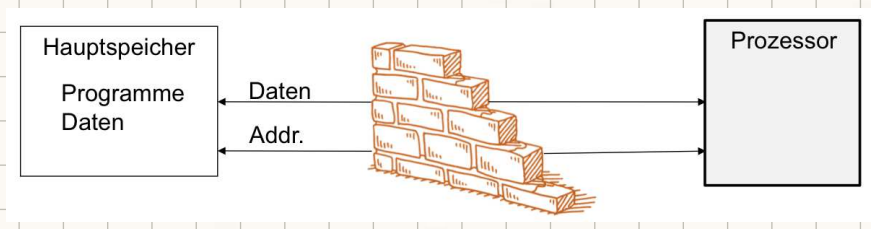
Der Von-NeuMann-Flaschenhals (The Memory Wall)

Konsequenz

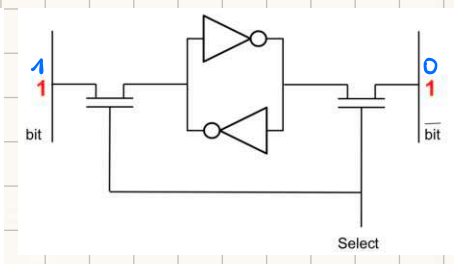
- Befehle (die auf Speicher zugreifen) müssen warten
- Prozessoren sind im Leerlauf; Reduzierte Ausnutzung des Prozessors

Verbesserung der Speichergeschwindigkeit

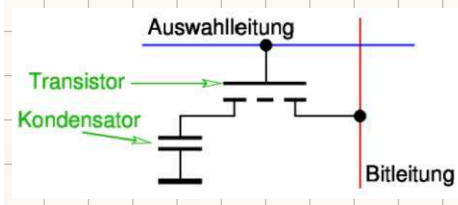
- Latenz (Dauer eines Zugriffes) reduzieren
- Bandbreite (Durchsatz) vergrößern



Statisches Random Access Memory (SRAM): Prinzip ähnlich zu Flip-Flops
-> Speicherinhalt bleibt bei anliegender Spannung erhalten
SRAM-Zelle Aufbau: typischerweise aus 6 Transistoren
Schreiben: Schreiben erfolgt wenn Select auf 1, Inverter schwingen sich ein
Lesen: Inverter versuchen einen Ausgang auf 0 zu ziehen, Spannungsabfall registrieren
Vorteil: CMOS Technologie (einfach integrierbar); behält Wert solange Strom anliegt; schneller Zugriff
Nachteil: Relativ groß und teuer; Großer Energiebedarf wenn aktiv; nur kleine Kapazität möglich; unempfindlicher
Verwendung: für Register und Caches



Dynamisches Random Access Memory (DRAM): Ladung im Kondensator
-> Speicherzustand muss immer wieder neu (periodisch) aktualisiert (refresh) werden
DRAM-Zelle Aufbau: Kondensator (Ladung/ Information halten); Transitor; Auswahlleitung; Bitleitung
Schreiben: Auswahlsignal setzen, 0 oder 1 an Bitleitung, Kondensator laden oder entladen
Lesen: Auswahlsignal setzen, (Kondensator lädt sich) oder (Kondensator entlädt sich) an Bitleitung
Probleme: Lesen zerstört die Information; Kondensator verliert Ladung
Lösung: Nach Lesen werden Daten sofort wieder geschrieben; Regelmäßiges "Auffrischen" des Zustands
Vorteil: einfach zu bauen; Kann sehr groß und „preiswert" gebaut werden; große Kapazitäten leicht möglich
Nachteil: braucht Kondensator (schwierig integrierbar); großer Energiebedarf wenn aktiv und passiv; langsamer Zugriff; braucht refresh
Verwendung: für Hauptspeicher (benötigt Speicherkontroller (oft in Prozessoren integriert); Parallelität wichtig (mehrere Kontroller))

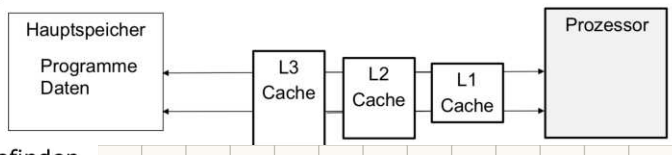


Cache

Definition: Ein Cache ist ein schneller Zwischenspeicher vom Daten-Ausschnitt aus dem Hauptspeicher

Eigenschaften

- Enthält immer nur eine Kopie der Daten des Hauptspeichers, Ausschnitt kann mit der Zeit wechseln
- Wird automatisch (transparent) verwaltet: keine expliziten Adressen
  - Transparent: Zugriff auf Daten unabhängig davon, ob sie sich im Cache oder im Hauptspeicher befinden
  - Hardware entscheidet welche Daten im Cache gelagert werden
    - Software muss dabei helfen -> sonst: höhere Ausführungszeiten / ineffektive Caches



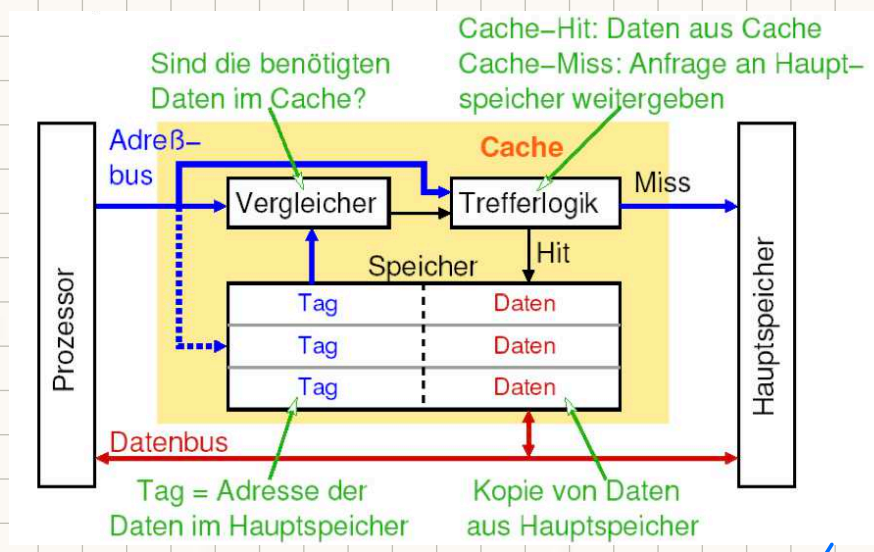
Speicherzugriffsmuster

Zeitliche Lokalität (temporal locality)

- Wenn ein Programm auf Adresse x zugreift, ist es wahrscheinlich, dass es kurz darauf wieder auf x zugreifen wird
- Caches behalten auf kurzem zugegriffene Daten im Speicher

Räumliche Lokalität (spatial locality)

- Wenn ein Programm auf Adresse x zugreift, ist es wahrscheinlich, dass es auch auf Adressen in der Nähe von x zugreift
- Caches behalten benachbarte Daten im Speicher



Aufbau

- Ein Cache besteht aus einer größeren Menge von Cache-Zeilen (-> meist 32 - 128 Bytes)
- immer ganze Zeilen geladen oder geschrieben -> Laden größerer Datenblöcke ist weniger aufwendig
- die zusätzlich geladenen Bytes werden mit hoher Wahrscheinlichkeit auch benötigt (räumliche Lokalität)

Cache-Typen

Zugriffsadresse: 0x000C /4 → 0x0003 (Adresse / Blocknummer) % 32 → 0x0003 (Cache Index)

Direkt abbildender Cache („Direct Mapped Cache")

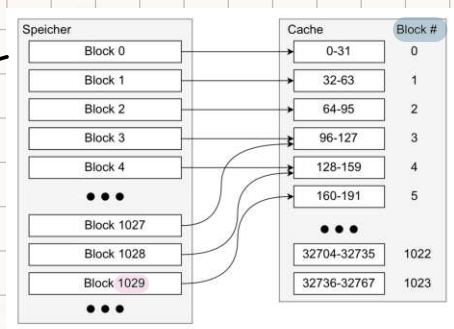
Jede Adresse passt genau zu einer Cachezeile — Index beschreibt eindeutige Zeile

Index Bits bestimmen die genaue Lage der Daten im Cache

Vorteil: benötigt keine Übersetzung der Zieladresse -> schneller Zugriff auf den Cache-Daten

Nachteil: Unflexibel, Häufige Verdrängung von Cachezeilen

Blocknummer mod (#Blöcke in Cache)

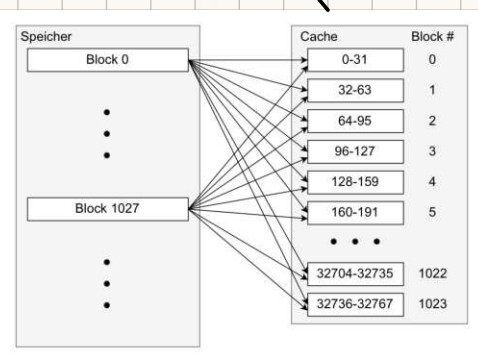


Voll-assoziativer Cache („Fully Associative Cache")

Jede Adresse kann auf jede Cache Zeile abgebildet werden — jede Cachezeile möglich

Vorteil: keine Adresskonflikte, größte Flexibilität, nur ein „Cache Set", keine Index bits

Nachteil: sehr aufwendig/ langsam -> Bei einem Zugriff muss die gesuchte Adresse mit allen Tags verglichen werden; Es können keine Speicherblöcke im Cache stehen, die auf den selben Index abgebildet werden



Kompromiss -> Mengenassoziativer Cache („Set-Associative Cache")

Jede Adresse kann auf einen Teil der Cache Zeilen abgebildet werden

Ein Speicherblock kann in genau einer Adresse gespeichert werden.

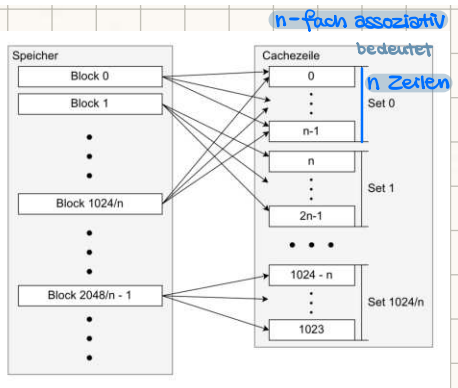
Vorteil: Mehr Flexibilität, Mehrere Cache-Sets, weniger Index bits, Handhabbare Komplexität

Nachteil: Indexberechnung notwendig

Index beschreibt Menge aus N Zeilen => $\log_2(N)$ Index bits

Blocknummer mod (#Menge (Sets) in Cache)

n-fach assoziativ bedeutet n Zeilen



Klassifizieren der "Cache Misses"

Compulsory (cold) Misses
- Erster Zugriff auf eine Adresse (Diese ist noch nicht im Cache und muss somit nachgeladen werden)
- Können nur durch Prefetching vermieden werden - sonst kann nicht vermeiden

Conflict Misses
- Daten in einem Cache wurden überschrieben, da ein anderer Block auf das gleiche Cache Zeile abgebildet wurde
- Können durch höhere Assoziativität des Caches reduziert werden. Dies macht den Cache jedoch langsamer
- Treten nur in mengenassoziativem oder direkt abbildendem Cache auf

Capacity Misses
- Die benötigten Daten eines Programms ist größer als der Cache
- Können durch größeren Cache reduziert werden. Dies macht den Cache jedoch langsamer.
- Treten in voll-assoziativem Cache auf

Update Strategie

Ersetzungsstrategien
- Falls ein Cache voll läuft muss entschieden werden, welche alten Daten mit den neuen überschrieben werden
- Beide Strategien nutzen dabei die zeitliche Lokalität der Speicherzugriffe aus

① **LRU (Least Recently Used)**
- Es wird immer die älteste Cache Zeile ersetzt
- Realisiert über eine Priority Queue, in der immer das letzte Element ersetzt wird

② **LFU (Least Frequently Used)**
- Es wird immer die am seltensten benutzte Cache Zeile (neue Blöcke werden bevorzugt eliminiert)

③ Random (keine Berücksichtigung von Lokalität)

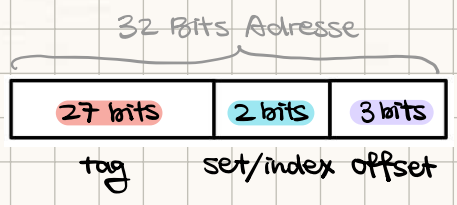
④ FIFO (keine Berücksichtigung von Nutzungsfrequenz)

Aktualisierungsstrategie: Was passiert, wenn der Prozessor die Daten schreiben?

Bei Cache-Hit: Wann erfolgt Aktualisierung des Hauptspeichers?
- **Durchschreibe-Strategie (Write Through)**
  - Schreibzugriffe werden im Cache und Hauptspeicher aktualisiert
  - Schreibtreffer (Write Hit): Änderung wird auch im Cache aktualisiert, aber nicht schneller beim schreiben
  - Schreibfehler (Write Miss): Der in den Hauptspeicher geschriebene Block wird in den Cache geladen
- **Rückschreibe-Strategie (Write Back)**
  - Schreibzugriffe werden nur im Cache aktualisiert
  - Rückschreiben auf den Hauptspeicher, sobald die Cache Zeile überschrieben wird
    - weniger Hauptspeicherzugriffe; Inkonsistenzen zwischen Cache und Hauptspeicher

Bei Cache-Miss: wird Daten in den Cache eingelagert?
- **Write-Allocation:** Speicher wird in den Cache geladen und aktualisiert
- **No-Write-Allocation/ Write Around:** keine Aktion des Caches

Beispiel für Cache-Zugriff-Schema



offset bits = $\log_2 8 = 3$ bits
sets = $64/(8 \cdot 2) = 4$
sets/index bits = $\log_2 4 = 2$ bits
tag bits = $32 - 3 - 2 = 27$ bits

Zugriff-Adresse	Tag (27 bits)		Set/ index (3 bits)	Offset (3 bits)	1st
0x0000	00000000 00000000 00000000 000	A	00	000	Miss
0x0004	00000000 00000000 00000000 000	B	00	100	Hit
0x1008	00000000 00000000 00010000 000	C	01	000	Miss
0x2008	00000000 00000000 00100000 000	D	01	000	Miss
0x3008	00000000 00000000 00110000 000	E	01	000	Miss

1st Hit Rate = 1/5 = 20 %

Zugriff-Schema

1. Adresse liegt am Cache an
2. Berechnung des Index
3. Berechnung des Tags
4. Suche durch alle Cachezeilen im Cache-Set
5. Ist einer der Tags = Tag der Adresse?
   - ☐ Ja: Gebe Datum an Offset in der Cachezeile zurück
   - ☐ Nein:
     - Fals voll, lösche eine der Cachezeilen im Cache Set
     - Lade neue Cachezeile aus dem Hauptspeicher
     - Ersetze Tag

1 Cache Zeile (8 Byte Block)

2-Fach assoziativ

Set	LRU	Tag (Way 1)	Tag (Way 0)
00 (= 0)	1		A
01 (= 1)	x0	D	C E
10 (= 2)			
11 (= 3)			

⇒ insgesamt 8 Cachezeile

64 Bytes Cache

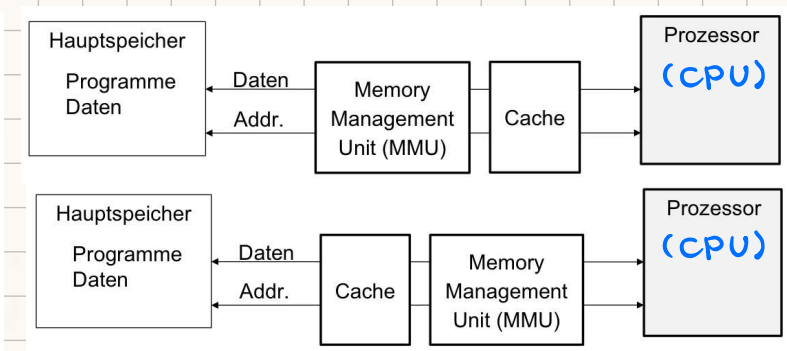
4 Cachemenge (4 Sets)

Virtueller Speicher

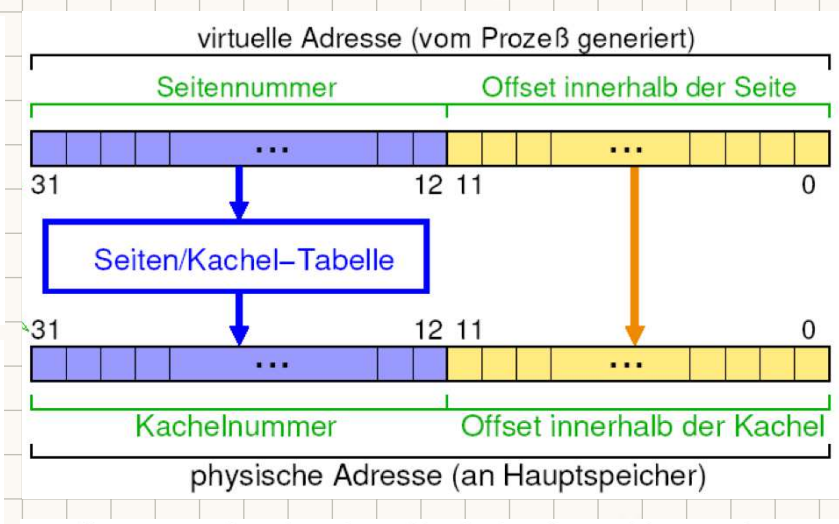
Memory Management Unit (MMU)
- Die MMU übersetzt bei jedem Speicherzugriff die virtuelle Adresse in eine physikalische im Hauptspeicher
- Realisiert durch Umsetzungstabellen, welche vom Betriebssystem bereitgestellt werden

Virtually Indexed Cache
- Cache zwischen Prozessor und MMU
- virtuelle Adressen verwenden - daraus berechnen sich Tag/Index/Offset
- **Vorteil:** Direkter Zugang zu Cache ohne Übersetzung, gut für kleine Caches
- **Nachteil:** Mehrere Prozesse verwenden die gleichen virtuellen Adressen

Physically Indexed Cache
- Cache zwischen MMU und Hauptspeicher
- physikalische Adresse verwenden - daraus berechnen sich Tag/Index/Offset
- **Vorteil:** Übersteht Prozess/Kontextwechsel, Gut für grosse Caches
- **Nachteil:** Jeder Zugriff muss zuerst durch die MMU



Paging: Adressumsetzung-Prinzip
- Innerhalb der MMU werden die virtuelle Adressen über die Seitentabelle des Betriebssystems in physikalische Adresse übersetzt
- Seitennummern (hohe Bits) der virtuellen Adresse werden durch die Seitentabelle in Kachelnummer übersetzt, können mehrere Adressen in einen Block (Seite) zusammengefasst werden.
- Offset (niedrige Bits) der virtuellen Adresse ist gleich dem der physischen Adresse (muss nicht übersetzt)



Vorteile
- Das Betriebssystem kann einem Prozess mehr Speicher zuweisen, als verfügbar ist
  - Virtuelle Adressen können auch auf den Hintergrundspeicher abgebildet werden.
  - Nicht verwendete Blöcke können ausgelagert werden.
- Jeder Prozess kann die gleichen Virtuellen Adressen verwenden
- Ein zufälliges Layout der Daten im Hauptspeicher erhöht die Sicherheit

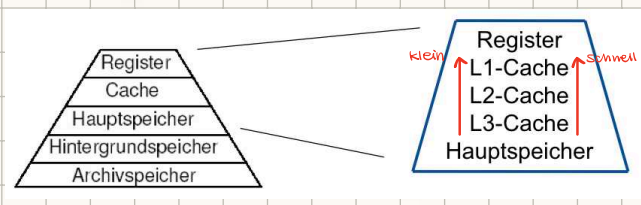
Nachteile
- Jeder Speicherzugriff muss übersetzt werden

Vorteil von Verwendung der Seitengrößen in virtuellen Speicherverwaltung
kleinere Seitengrößen: mehr Flexibilität, weniger Speicherverschnitt
größere Seitengrößen: kleinere Seitentabelle, schnellere Übersetzung, weniger TLB Misses

Cache Hierarchie

Inklusive Cache Hierarchie
- Alle in einem Cache enthaltenen Blöcke sind auch in einem darunterliegenden (größeren) Cache enthalten
- Hauptspeicherzugriffe legen Kopien in allen Cache Levels ab

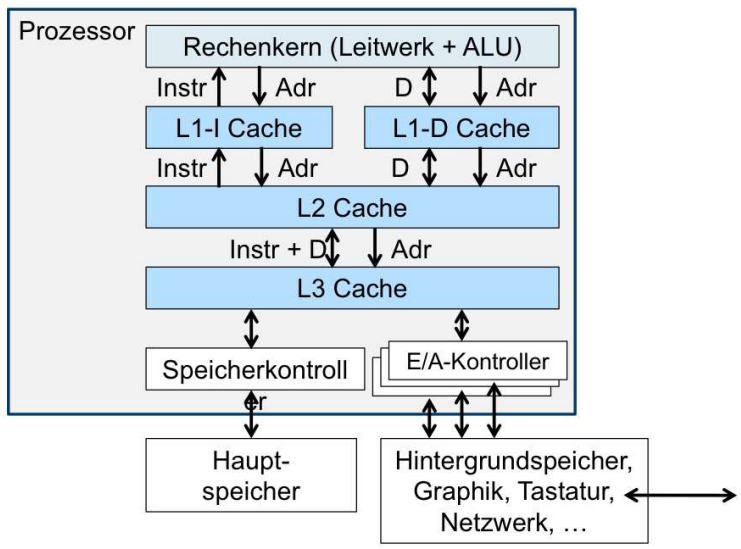
Exklusive Cache Hierarchie
- Keine zwei Cache Blöcke enthalten den gleichen Speicherblock
- Falls ein Cache voll wird, verdrängt dieser den betreffenden Block in den nächst tieferen Speicher (L1 → L2 → L3)



Von-Neumann Flaschenhals

2 verschiedene Speicherzugriff-Typen können sich gegenseitig behindern und zu unnötigen Leistungseinbußen führen
- zum Laden der Maschinenbefehle
- zum Laden/ Speichern der Daten in Befehlsausführung

Wird durch das Trennen von Daten und Code gelöst
- **Harvard Architektur**
  - Code und Daten werden getrennt gespeichert
  - Es kann parallel auf beide Speicher zugegriffen werden
- **Von Neumann Architektur**
  - L1 Cache wird in Code und Daten getrennt
  - Da beide unterschiedliche Anforderungen haben, kann weiter optimiert werden
    - Code: sequentielle Lesezugriffe
    - Daten: komplexe Lese- und Schreibzugriffe



Prefetching

Definition: Um auch die räumliche Lokalität von Speicherzugriffen auszunutzen, werden mehr Blöcke in den Speicher geladen, als in der momentanen Anfrage benötigt werden

Hardware Prefetching
- Hardware untersucht alle Zugriffsadressen und entdeckt regelmäßige Zugriffe
- Sobald der Cache erkennt, aufeinanderfolgende Speicherblöcke (Stream) zugegriffen wird -> lädt diese auf Verdacht nach
- **Vorteil:** einfach zu implementieren, gut für reguläre Zugriffe
- **Nachteil:** nicht hilfreich für komplexe Zugriffsmuster

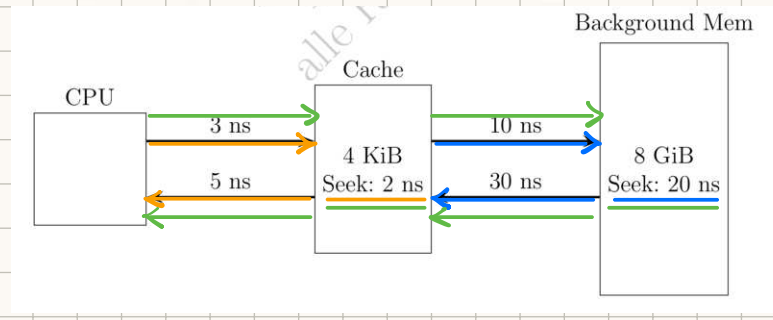
Software Prefetching
- Ein Compiler fügt prefetching Instruktionen in den Maschinencode ein, um den Cache besser nutzen zu können
- Durch Software Prefetching kann präziser angewandt werden, da nicht vom Cache erkannt werden muss, wann ein Stream Zugriff vorliegt
- **Vorteil:** auch unregelmäßige Zugriffe können verbessert werden
- **Nachteil:** fehlende Transparenz - benötigt Codeänderungen

Terminologie

Hit Latency: Antwortzeit, wenn Wert im Cache — **Hit Rate:** $\frac{\text{Hits}}{\text{Number Requests}}$

Miss Latency: Antwortzeit, wenn Wert nicht im Cache — **Miss Rate:** $\frac{\text{Misses}}{\text{Number Requests}}$

Frequenz (f):	Frequenz des Prozessors
Instruction Count (IC):	Anzahl an auszuführenden Instruktionen
Cycles per Instruction (CPI):	(Durchschnitts-) Anzahl an Zyklen pro Instruktion
Memory Access Rate:	(Durchschnitts-) Anzahl an Speicherzugriffen pro Instruktion



Hit-Latency = 3ns + 2ns + 5ns = 10ns

Miss-Penalty = 10ns + 20ns + 30ns = 60ns

Miss-Latency = Hit-Latency + Miss-Penalty = 70 ns

Miss-Penalty: die zusätzlich von der Speicherhierarchie benötigte Antwortszeit, wenn einCache-Miss auftritt

Beispielrechnung

Sie haben ein Programm mit 10^9 Instruktionen, dass auf einem in-order Prozessor mit einer Frequenz von 3 GHz läuft. Der Prozessor braucht durchschnittlich 3 Zyklen pro Instruktion (exklusive Speicherzugriff) und pro Instruktion fallen durchschnittlich 1.5 Speicherzugriffe an. Es gibt einen 2 KB Cache mit einer Latenz von 5 ns und einer Miss-Rate von 20%. Der Hintergrundspeicher ist 2 GB und hat eine Latenz von 50 ns. Berechnen Sie die Ausführungszeit des Programms ohne und mit Cache.

$$\text{CPU Time} = \text{IC} \times \left(\frac{\text{CPI}}{f} + \frac{\text{Memory Accesses}}{\text{Instruction}} \times \text{Average Memory Access Time}\right)$$

Datenverarbeitungszeit — Speicherzugriffszeit

$$\text{Average memory access time} = \text{Hit rate} \times \text{Hit latency} + \text{Miss rate} \times \text{Miss latency}$$



Ohne Cache ≡ 100% Miss-Rate

$$\text{CPU Time} = 10^9 \text{ ins} \times \left(\frac{3 \frac{\text{cycles}}{\text{ins}}}{3 \text{ GHz}} + 1.5 \frac{\text{accesses}}{\text{ins}} \times \left(0\% \times 5 \frac{\text{ns}}{\text{access}} + 100\% \times 500 \frac{\text{ns}}{\text{access}}\right)\right)$$

$$\text{CPU Time} = 10^9 \text{ ins} \times \left(\frac{3 \frac{\text{cycles}}{\text{ins}}}{3 \text{ GHz}} + 1.5 \frac{\text{accesses}}{\text{ins}} \times \left(500 \frac{\text{ns}}{\text{access}}\right)\right)$$

$$\text{CPU Time} = 10^9 \text{ ins} \times \left(\frac{3 \frac{\text{cycles}}{\text{ins}}}{3 \times 10^9 \text{ Hz}} + 1.5 \frac{\text{accesses}}{\text{ins}} \times \left(500 \times 10^{-9} \frac{\text{s}}{\text{access}}\right)\right)$$

$$\text{CPU Time} = \left(\frac{3 \text{ cycles}}{3 \frac{\text{cycles}}{\text{s}}} + 750 \text{ s}\right) = 751 \text{ s} = 12 \text{ minutes } 31 \text{ seconds}$$

Mit Cache

$$\text{CPU Time} = \left(\frac{3 \text{ cycles}}{3 \text{ Hz}} + 1.5 \text{ accesses} \times \left(80\% \times 5 \frac{\text{s}}{\text{access}} + 20\% \times 500 \frac{\text{s}}{\text{access}}\right)\right)$$

$$\text{CPU Time} = \left(\frac{3 \text{ cycles}}{3 \frac{\text{cycles}}{\text{s}}} + (6 \text{ s} + 150 \text{ s})\right)$$

$$\text{CPU Time} = (1 \text{ s} + (6 \text{ s} + 150 \text{ s}))$$

$$\text{CPU Time} = 157 \text{ s} = 2 \text{ minutes } 37 \text{ seconds}$$

Nachteil des bisherigen Pipeline-Konzeptes

- Befehle werden weiterhin in der ursprünglichen Reihenfolge ausgeführt
- Wartezyklen blockieren alle nachfolgenden Instruktionen

→ Beispiel

Div	R0 ← R1,R2
Add	R0 ← R0,R4
Load	R5 ← [R6]

- Die Division blockiert alle nachfolgenden Operationen
- Die Addition ist datenabhängig
- Die Ladeoperation ist unabhängig, ist aber trotzdem blockiert

:) Idee

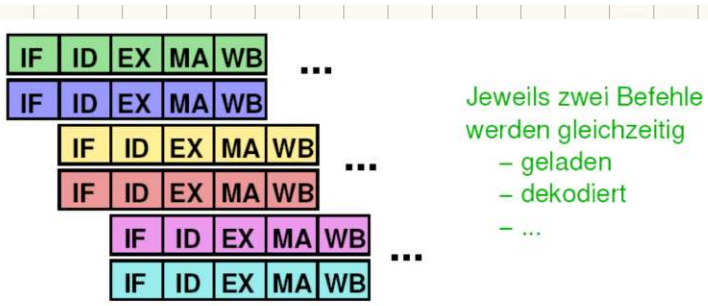
Dynamisches Scheduling (Out of Order Execute) OoO

Definition: (Daten-)unabhängige Instruktionen können parallel ausgeführt werden, obwohl vorherige Operationen blockiert sind

Vorteil: Bessere Nutzung von vorhandenen Ressourcen; Bessere Nutzung von Parallelität im Programm; die Möglichkeit, mehrere gleiche Befehle anzustoßen (wenn die Hardware dafür da ist)

Superskalarprinzip

Definition: mehrere einzelne unabhängige Maschinenbefehle können gleichzeitig ausgeführt werden



Superskalarität & Pipelining

Gemeinsamkeiten: beide erhöhen die Leistung
- Bessere Ausnutzung von Einheiten
- Nutzung von Parallelität
- Vor allem in Kombination mit Out of Order Execution

Unterschiede:
- Pipelining: mehrere verschiedene Stufen gleichzeitig
- Superskalar: mehrere gleiche Einheiten gleichzeitig

Parallelismus Kategorien

Parallelität kann auf den folgenden Ebenen erzielt werden:

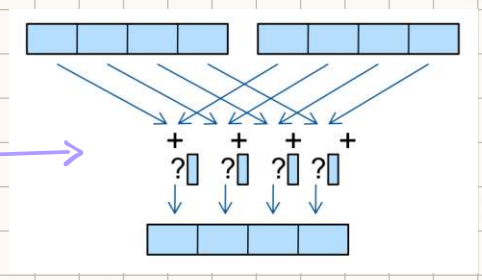
Bitebene (Bit Level Parallelism BLP)	• Die Wortbreite einer Architektur (16, 32, 64 bit) kann als parallelismus interpretiert werden • Es werden jeweils n Bits gleichzeitig verarbeitet.
Instruktionsebene (Instruction Level Parallelism ILP)	• Mehrere aufeinanderfolgende Maschinenbefehle können gleichzeitig ausgeführt werden. • Bsp.: Pipelining
Datenebene (Data Level Parallelism DLP)	• Es kann ein Befehl auf mehrere Datenteile gleichzeitig angewendet werden. (unabhängig voneinander) • Bsp.: Parallele Addition, SIMD Instruktionen
Thread Ebene (Thread Level Parallelism TLP) (Hardware-Thread)	– Synchronisation • Gleichzeitige Ausführung mehrerer Befehlsströme. • Jeder Thread arbeitet unabhängig von einander. • Muss explizit vom Programmierer verwendet werden

Finn's Classification - explizite Parallelität

SISD (Single Instruction Single Data) – 1 Leitwerk, 1 Rechenwerk
- Klassische Sequentielle Abarbeitung nach VNM

(Voraussetzung: mehrere Operationen gleiches Types, unabhängig ausführen können)

SIMD (Single Instruction Multiple Data) – 1 Leitwerk, n Rechenwerke
- Mehrere Daten auf einmal innerhalb des gleichen Befehls verarbeiten
- **Bsp.:** Vektoradditionen
- **Vorteil:** Höhere Rechenleistung durch Parallelität; Weniger Aufwand für Programmabarbeitung
- **Nutzung:** Durch vektorisierende Compiler realisiert (dieser erkennt, ob Vektoroperationen eingesetzt werden können (VLIW)); Verwendung von Funktionsbibliotheken, welche Vektorbefehle verwenden



MISD (Multiple Instructions ~~Multiple~~ Single Data) – n Leitwerke, 1 Rechenwerk - Hat wenig Sinn!

MIMD (Multiple Instructions Multiple Data) – n Leitwerke, n Rechenwerke
- Mehrere Hardwarethreads
- Mehrere Ausführungseinheiten und mehrere Befehlszähler
- Jeder dieser führt getrennte Instruktionsströme aus
  - Unkoordiniert: Parallele Ausführung
  - Keine direkten Abhängigkeiten
- IMMER gleicher Speicher

Multithreading

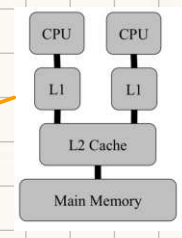
- Mehrere Hardware Threads in einem Prozessor
- Jeder hat eigenen Befehlszähler und Register
- ABER: Alle anderen Ressourcen werden geteilt

- **Einfaches Multithreading**
  - Im Extremfall werden pro Takt Threads umgeschaltet
  - Die Ressourcen werden in jedem Takt von nur 1 Thread gleichzeitig genutzt
- **Simultaneous Multithreading (SMT)**
  - Befehle eines Threads werden gepuffert und nach Bedarf an die verfügbaren Prozessoren zugeordnet
  - Ressourcen können gleichzeitig genutzt werden
  - Passt zu Out of Order Execute und Superskalarität
  - Erhöhte Buchhaltungskosten

Mehrkern Architektur (Parallelisierung)

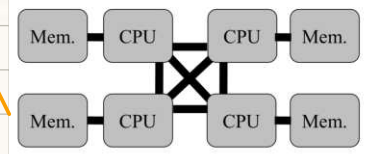
- Mehrere Hardware Threads im Prozessor
  - getrennte Ressourcen (getrennte Kerne) verwenden
  - Replikation von Leit- und Rechenwerk
  - Gemeinsamer Speicher - Teile der Cachehierarchie werden geteilt
  - Üblicherweise auch gemeinsames E/A Werk
  - Durch parallele Ausführung voller Speed-Up möglich
  - Jeder Kern(Core) hat die gleiche Komplexität

Mehrere Prozessoren/Sockel pro System
- Wie sieht der Zugriff der verschiedenen CPUs auf den Speicher aus?



UMA (Uniform Memory Access)
- Zentraler gemeinsamer Speicher
- Speicherzugriff haben bei jedem Prozessor die gleiche Latenz

NUMA (Non-uniform Memory Access)
- verteilt über mehrere Speicher
- Lokale Zugriff schneller
- Datenplatzierung wichtig

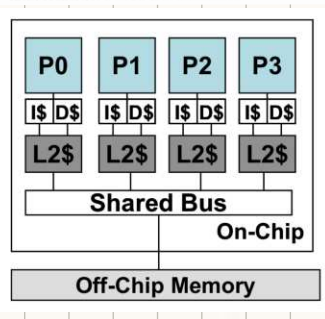


Konsequenzen von Mehrkern Architektur

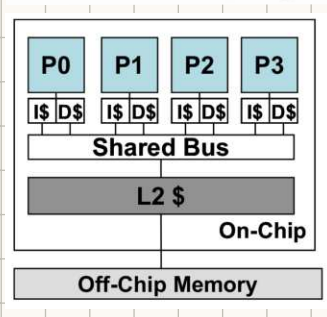
-> Einfluss auf Speicher
- Mehrere Kerne greifen auf gemeinsamen Bus zu
- Man braucht mehrfache Datenmenge
  - > Zugrffsgeschwindigkeit des DRAM weiterhin langsam
- Starke Speicheranbindung nötig -> Direkte Anbindungen
- Speicherzugriff muss geregelt ablaufen - vom Speichercontroller zuständig

-> Einfluss auf Caches
- Mehrere Kerne müssen in einer Speicherhierarchie eingebunden werden
  - Welche Caches sollen von den Kernen geteilt werden?

Ansatz 1
- Jeder Kern mit eigenem L2 Cache verbunden
- dann jeweils mit dem Shared Bus verbunden
- **Vorteil:** Schnellerer Zugriff
- **Nachteil:** L2 schlecht genutzt



Ansatz 2
- Kerne mit Shared Bus verbunden
- dann gemeinsamer L2 Cache darunter
- **Vorteil:** L2 besser genutzt
- **Nachteil:** Hohe Latenz/Verzögerung



-> Einfuss auf ISA
- Kommunikation zwischen HW-Threads
  - Datenaustausch via Speicher
    - -> Problem: Races -> Welcher Zugriff gewinnt?
  - Neue Assemblerbefehle notwendig
    - Atomare Operationen
    - **Konsistenzmodelle** - um Zugriffskonflikte zu vermeiden
      - Vertrag zwischen Programmierer und Maschine
      - Sequential Consistency: Ergebnis soll nach jeder Ausführung dasselbe bleiben
      - Bestimmten wie, wann und wo Daten sichtbar sind
      - Problem: Teuer und langsam; Meiste Modelle nicht sequential konsistent und verstecken die Nebeneffekte
- Synchronisation
  - Lock: Falls Lock frei -> Besetzen; Falls Lock besetzt -> Warten
  - Unlock: Freigabe des Locks
  - Schützen von einzelnen Speicherbereichen -> NT: Zeitverlusten (im schlimmsten Fall zu Deadlocks führen)

Cacheprobleme: Wie werden Daten zwischen den einzelnen Caches ausgetauscht, wenn jeder Kern einen eigenen Cache hat?

Snooping - Zur Regelung des Datenaustausches
- Gemeinsamer Bus zwischen Caches
  - Alle Teilnehmer können mithören und Daten anpassen
1. Write Invalidate: Schreibzugriffe führen zur Invalidierung anderer Zeilen
2. Write Update: Neue Werte direkt in die anderen Caches reingeschrieben

Protokoll
- Jede Cachezeile muss ihren Status laufend mitverfolgen
- Automat wird nach jedem Zustand aktualisiert

Aktion	P0	P1	P2	P3
-	IIII	IIII	IIII	IIII
R_0(01001110)	IEII	IIII	IIII	IIII
R_2(01001110)	ISII	IIII	ISII	IIII
R_1(01001110)	ISII	ISII	ISII	IIII
R_2(10111010)	ISII	ISII	ISIE	IIII
W_2(01001110)	IIII	IIII	IMIE	IIII
R_3(00000000)	IIII	IIII	IMIE	EIII
W_0(00000000)	MIII	IIII	IMIE	IIII
W_3(10111010)	MIII	IIII	IMII	IIIM

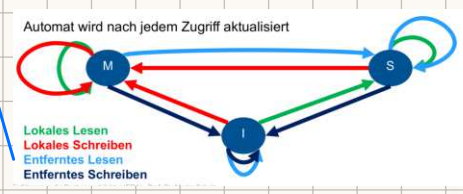
Read I -> E falls mehrere shared -> S

Write diese Cachezeile S/I/E -> M

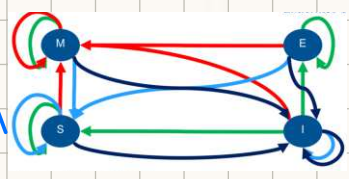
andere Cachezeile -> I (veraltet)

Write: M —und dann Read→ S (beide Prozesser)

① MSI Protokoll
- Automat mit 3 Zustände
  - Modified: Cachezeile lokal geändert, Speicher nicht kohärent
  - Shared: Wert in mindestens einem weiteren Cache, kohärent
  - Invalid: Cacheblock nicht mehr gültig



② MESI Protokoll
- Automat mit 4 Zustände
  - Modified: Cachezeile lokal geändert, Speicher nicht kohärent
  - Exclusive: Wert in einem Cache, unverändert
  - Shared: Wert in mindestens zwei Caches, kohärent
  - Invalid: Cacheblock nicht gültig oder veraltet



③ Alternative: Verzeichnisbasierte (Directory) Protokolle
- Statusbits in einem zentralen Verzeichnis
- Verzeichnis/Directory merkt sich, wo Kopien liegen
  - Gezielte Invalidierung statt Broadcast
  - Verringerter Verkehr
  - Kein zentraler Bus mehr nötig
- Erhöhter Hardwareaufwand

jeder der 4 Buchstaben ist eine Cachezeile 11:32

und wenn P0 W bloß 00 verändert, dann müssen auch bloß die 00 Cachzeilen der anderen Prozessoren verändert werden 11:33

False Sharing
- Cachezeilen wandern zwischen Caches, obwohl es nicht nötig ist

->> Viele (unnötige) Cache Misses & Geschwindigkeitseinbußen

z.b. dadurch lösbar, dass man nicht den ganzen Array in eine Cachezeile lädt

Paralleler Speedup: Leistungsgewinn durch Parallelisierung

$$\text{Speedup}(n) = \frac{t_{p=1}}{t_{p=n}}$$

$$t = t_{\text{sequential}} + t_{\text{parallel}}$$

n	Anzahl Prozessoren
$t_{p=n}$	Gesamtausführungszeit für n Prozessoren
t_{parallel}	Ausführungszeit paralleler Programmteil
$t_{\text{sequential}}$	Ausführungszeit sequenzieller Programmteil

$$Speedup_{Amdahl}(n) = \frac{T}{t_s + \frac{t_p}{n}}$$

$$Speedup_{Gustafson}(n) = \frac{t_s + n \cdot t_p}{T}$$

Amdahl's law: theoretische Speedup bei konstanter Problemeöße

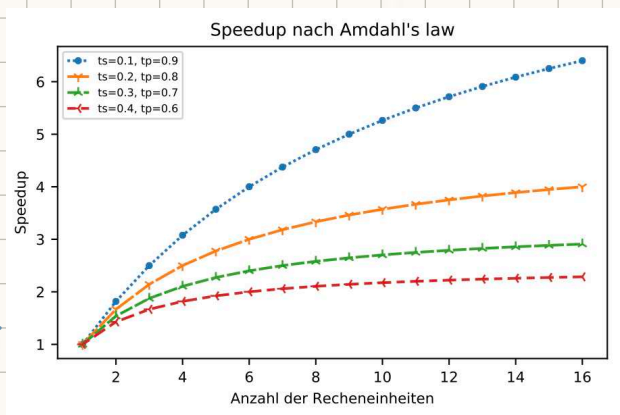
konstante Last

$$t_{p=n} = t_{\text{sequential}} + t_{\text{parallel},\ p=n} \qquad t_{\text{parallel},\ p=n} = \frac{t_{\text{parallel},\ p=1}}{n}$$

$$\text{Speedup}_{\text{Amdahl}}(n) = \frac{t_{p=1}}{t_{p=n}} = \frac{t_{\text{sequential}} + t_{\text{parallel},\ p=1}}{t_{\text{sequential}} + \frac{t_{\text{parallel},\ p=1}}{n}} = \frac{1}{t_{\text{sequential}} + \frac{t_{\text{parallel},\ p=1}}{n}} \Rightarrow \left(\frac{1}{s + \frac{1-s}{n}}\right)$$

$$\lim_{n\to\infty} \text{Speedup}_{\text{Amdahl}}(n) = \lim_{n\to\infty} \left(\frac{1}{s + \frac{1-s}{n}}\right) = \left(\frac{1}{s + \frac{1-s}{\infty}}\right) = \left(\frac{1}{s}\right)$$

↳ resultierend asymptotische beschränkte Speedup



Beispielaufgabe 1

Sie haben ein Programm, dass auf dem 1-core Supercomputer *AncientMUC* 60 Sekunden Ausführungszeit hat. Sie lassen dasselbe Programm auf dem neueren 10-core *TinyMUC* laufen und messen eine Ausführungszeit von 40 Sekunden. Welcher Prozentsatz des Programms ist parallelisiert? Gehen sie von idealer Parallelisierung aus (keine Overheads).

$$\frac{t_{p=1}}{t_{p=10}} = \frac{60\ \text{s}}{40\ \text{s}} = \frac{60\ \text{s} - t_{\text{parallel},\ p=1} + t_{\text{parallel},\ p=1}}{60\ \text{s} - t_{\text{parallel},\ p=1} + \frac{t_{\text{parallel},\ p=1}}{10}}$$

$$60\ \text{s} \times \left(60\ \text{s} - \tfrac{9}{10} t_{\text{parallel},\ p=1}\right) = 40\ \text{s} \times 60\ \text{s}$$

$$3600\ \text{s}^2 - 54\ \text{s} \times t_{\text{parallel},\ p=1} = 2400\ \text{s}^2$$

$$t_{\text{parallel},\ p=1} \approx 22.2\ \text{s} \rightarrow \frac{22.2\ \text{s}}{60\ \text{s}} \approx 37\%\ \text{parallel}$$

Beispielaufgabe 2

Sie lassen dasselbe Program nun auf dem *HyperMUC* laufen, der eine exzellente Anzahl ($= \infty$) an Rechenkernen aufweist. Was für eine Ausführungszeit erwarten Sie? Gehen sie von idealer Parallelisierung aus (keine Overheads).

$$37\%\ \text{parallel} \rightarrow 63\%\ \text{sequenziell}$$

$$\lim_{n\to\infty} \text{Speedup}_{\text{Amdahl}} = \frac{1}{s} = \frac{1}{63\%} \approx 1.59$$

$$\frac{60\ \text{s}}{1.59} \approx 37.7\ \text{s}$$

Gustafson's law: bei Problemgröße proportional zur Prozessoranzahl

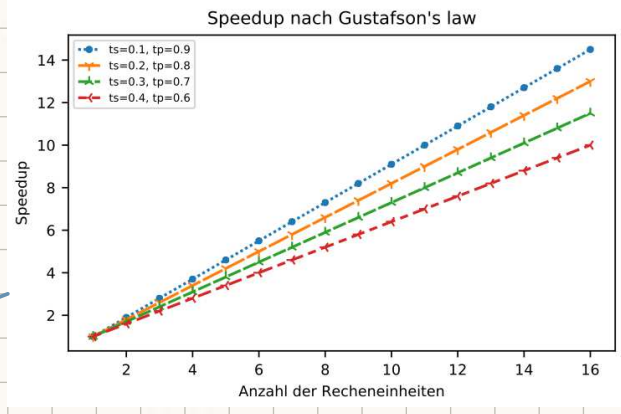
n-fache Last für n-fache Prozessoren

$$t_{p=n} = t_{\text{sequential}} + t_{\text{parallel},\ p=n} \qquad n \times t_{\text{parallel},\ p=n} = t_{\text{parallel},\ p=1}$$

$$\text{Speedup}_{\text{Gustafson}}(n) = \frac{t_{p=1}}{t_{p=n}} = \frac{t_{\text{sequential}} + n \times t_{\text{parallel},\ p=n}}{t_{\text{sequential}} + t_{\text{parallel},\ p=n}} = t_{\text{sequential}} + n \times t_{\text{parallel},\ p=n}$$

$$\lim_{n\to\infty} \text{Speedup}_{\text{Gustafson}}(n) = \lim_{n\to\infty} t_{\text{sequential}} + n \times t_{\text{parallel},\ p=n}$$

↳ resultierend unbeschränkte Speedup



Berechnung der Rechenleistung

SuperMUC-NG Phase 1 mit ∼ 26.9 PetaFLOPS

- Grundannahme:
  - 2 SIMD-Einheiten (AVX512) / Core: 2 x 8 double Operationen pro Rechenzyklus
  - mit FMA Units (Fused Multiply Add): $c := a * b + c$ 2 Operationen gleichzeitig
  - Prozessortakt 2.7GHz
- Pro Rechenkern: **2 x 8 x 2 x 2.7 = 86.4 GFLOPS**

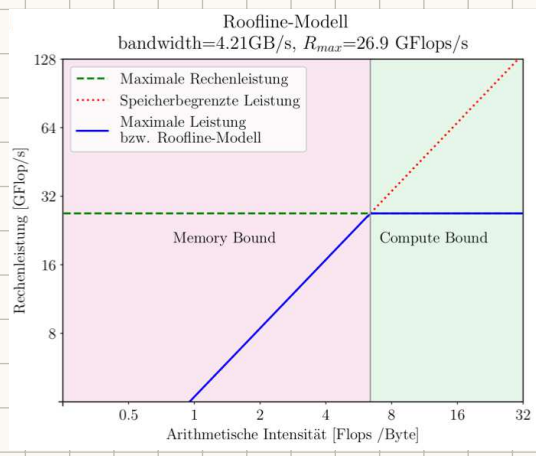
- Pro Server-Knoten:
  - 24 Kerne pro Prozessor
  - 2 Prozessor pro Rechenknoten

  ⇒ **86.4 x 24 x 2 = 4.1472 TFLOPS**
- 6480 Rechenknoten im System
- Gesamtleistung SuperMUC-NG Phase 1: **4.1472 x 6480 = 26.873 PetaFLOPS** Theoretische Maximalleistung

Roofline-Modell

-> Modellierung der maximalen Gesamtleistung
X-Achse: Arithmetische Intensität (in FLOPs pro Byte; abhängig vom Algorithmus)
Y-Achse: Rechenleistung (in FLOPS; abhängig vom Rechensystem)
Diagonale: Max Speicherbandbreite

-> Limitierungen identifizieren
Durch Berechnung Limitiert: bessere Ausnutzung der Architektur / des Systems
TLP, ILP, SIMD (Upgrade!)
Durch Speicher Limitiert: bessere Nutzung der Speicherhierarchie
Caches, Prefetching, Latency Hiding (Upgrade?)

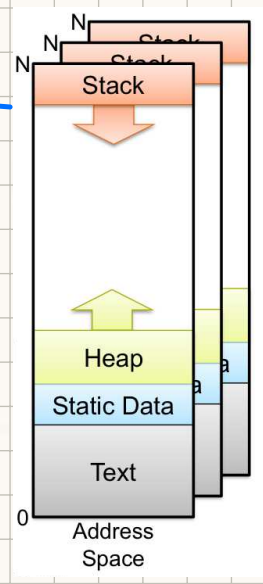


Programmierung vom Multikern-System

Hardware Threads
- Ausführungseinheit im Prozessor
- Befehlszählregister
- Stapelregister
- Rechenregister
- Leitwerk und ein oder mehrere Rechenwerke

Software Threads
- Programmierabstraktion
- Beschreibt eine Ausführung
- Beschreibt auch Status der Ausführung
- Es kann beliebig viele im System geben

Mehrbenutzersysteme – in parallelen Systemen
- Prozesse benötigen zusätzlichen Adressraum
- Jeder Prozess braucht immer mind. einen Software Thread
- Scheduler im Mehrkernsystem: alle Software Threads verwalten; Software Threads einem von mehreren Hardware Threads zuordnen



Paralleles Programmieren mit mehreren Threads
Kernpunkt: Nutzung mehrerer Software Threads gleichzeitig in einem Prozessor
Nebenläufige Programme: Einfach nur getrennte Aufgaben; lose Kooperation
Parallele Programme: Eine gemeinsame Aufgabe auf mehrere Threads verteilt; enge Kooperation

Aufgaben auf mehrere Prozesse oder mehrere Threads?

Vorteil - Mehrere Prozesse
- Bessere Isolierung -> Getrennte Ressourcen und Binärcode

Vorteil - Mehrere Threads - gemeinsamen Adressraum verwalten
- Einfache Kommunikation zwischen Threads
- Durch den Scheduler möglich zwischen einzelnen Threads schnell zu schalten
- Threads können Ressourcen/ Dateien/ Netzanbindung gemeinsam verwenden

-> Innerhalb eines Prozesses kann es mehrere Threads geben, die jeweils ein eigenes Stack haben

Kommunikation zwischen Threads
- Threads greifen auf sind eng gemeinsamen Speicher zu
- Teil eines Adressraums, Teil eines Prozesses
- Kommunikation via Lese/Schreibbefehle
- Kohärenz durch CC-Protokoll
- Geteilte Daten z.B. gemeinsame Metadaten (Konfigurations- und Steuerungsdaten, Daten bei parallelen Berechnungen, Producer/Consumer)

-> Notwendig Speicherkonsistenz aufrecht zu erhalten

Synchronisation zwischen Threads
- Notwendig, wenn mehrere Threads auf gemeinsame Ressourcen(Variable/ Speicherzelle) zugreifen
- Ohne Synchronisation kann es zu Races kommen

Races bei gemeinsamen Speicher
- Unklar, in welcher Reihenfolge Zugriffe ausgeführt werden
  - Tritt auch bei Sequential Consistency auf
  - Tritt auch bei bestehendeer Cache Kohärenz auf
- Zugriffe müssen synchronisiert werden -> Garantierte Reihenfolge
  - Serialisierung (ein Zugriff nach dem Anderen)
  - Beschreibung von Abhängigkeiten

RAW – Read after Write
WAR – Write after Read
WAW – Write after Write

Locks: Threads können Locks anfordern; Implementiert durch Flags im gemeinsamen Speicher

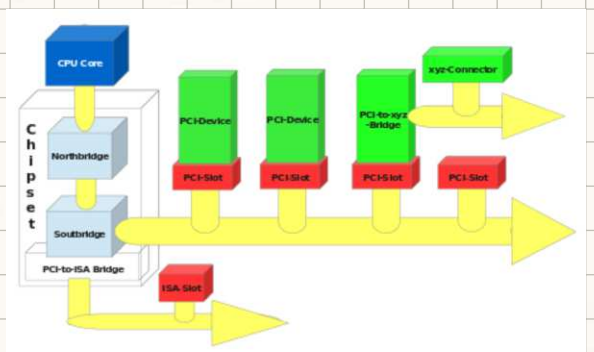
Spin-Locks: falls Lock vergeben, warte in einer aktiven Schleife auf Lock
Vorteil: Schnelle Reaktionszeit
Nachteil: Blockiert Hardware-Threads, kostet Ressourcen

Yielding-Locks: falls Lock vergeben, führe anderen Hardware Thread aus
Vorteil & Nachteil: Gegenteil von Spin-Locks)

Interne Bussystem

PCI – Peripheral Component Interconnect
- Verallgemeinerung von internen Bussen
- Bus-Standard zur Verbindung von Peripheriegeräten mit dem Chipsatz
- Alle möglichen Karten (Netzwerk, Grafik) anschließbar
- **PCI-Kontroller** - befindet sich im „Chip Set"
  - Kontrolliert welche der E/A-Werke mit dem Prozessor kommunizieren darf
  - Kontrolliert von welchem Peripheriegerät die Daten auf die Host-Bridge gelangen dürfen
- **Host Bridge** - Verbindung zwischen PCI-Kontroller und den anderen Komponenten
- **PCI-Express**
  - Verwendung von mehreren Lanes
  - PCI Bus mit PCI Slots, wo man externe Geräte anschließen kann

Systemarchitektur mit PCI



Beschleuniger (Beispiel: GPUs)

Graphics Processing Units (GPUs) - Extremes SIMD
- Ursprünglich für Bildschirmansteuerung entwickelt
- Ermöglicht moderne Grafikarbeitsplatzmaschinen
- Vorteile: reduziert die Komplexität des Prozessors; erhöht Energieeffizienz
- Nachteile: verringerte Softwareflexibilität (SIMD muss genutzt werden); Zusätzlicher Aufwand bei der Programmierung; Speicheranbindung benötigt hohe Bandbreiten

CPU Verbindung

PCI-Karten Nachteil: Weit von der CPU weg; kein kohärenter Speicher; hohe Datentransferkosten
Versuche zur besseren Einbindung: Sachen in Chip reingeholt; Prozessornahe Busse: Spezielle GPU Anbindungen
Probleme: Teuer, Portablilität eingeschränkt

Externe Anbindung via Netzwerke

Netzwerke: Verbindung mehrerer Rechensysteme
- WAN – Wide Area Network (Verbindung über weite Entfernung)
- LAN – Local Area Network (Lokales Netzwerk - typisch innerhalb von Organisationen)
- SAN – Storage/ System Area Network (Anbindung zum Speicher)

-> Unterschiede in Geschwindigkeit, Sicherheitsanforderungen und Interoperabilitätsanforderungen

Ethernet
- Familie an Netzwerktechnologie, die Software und Hardware für kabelgebundene Datennetze spezifiziert
- Heute meist „Point to Point" (Switched Networks) verbunden
- Vorteile: günstig wegen Massenproduktion; gute Leistung; weit verbreitet & verfügbar; einfach zu nutzen; skalierbar; über verschiedene Größenordnungen einsetzbar
- TCP/IP: Ethernet braucht Protokoll, um zwischen Rechnern zu kommunizieren

Paralleles Programmieren in (HPC) Clustern
- einzelne Knoten durch Netzwerk verbunden -> "Shared Nothing Architecture"
  - Problem: Keine gemeinsamen Ressourcen (unabhängige Knoten, OS Instanzen und Programme)
- Kommunikation via Nachrichten - MPI (Message Passing Interface)

Programmierung von E/A System

Hauptaufgabe eines E/A Systems
- Logische Abstrakte Sicht auf E/A Geräte (Hardware Details und Fehlerbehandlung verstecken)
- Effiziente Abarbeitung (auch bei großen Datenmengen)
- Unterstützung Mehrbenutzerbetrieb (Kontrolle der Daten bei Eingabe/Ausgabe -> Wem gehören die Daten?)

Einbindung von E/A an den Rechner
CPU braucht Zugrff auf die Ein-/Ausgabegeräte

Direct I/O
- Eingabe/ Ausgabe direkt aus der CPU
- Spezielle Befehle im Befehlssatz
- Separate Adressen, die I/O Geräte ansprechen
- Separate Busse für I/O Operationen

"Memory Mapped I/O" (Abbildung von E/A in Speicher)
- Geräte werden über einen Speicherbereich angesprochen
- Speicher - Teil des normalen Speicherbereiches
  - Bestimmte physikalische Adressen sprechen E/A Geräte an
- Vorteil: Kann durch virtuelle Speicherverwaltung kontrolliert werden -> Wenn Benutzer/Betriebssystem ein E/A Gerät benötigt, wird dieser nur in dessen Adressbereich abgebildet (nur für bestimmte Benutzer zugänglich)
- Nachteil: Zugriff beschränkt sich auf Standard Lese und Schreibezugriff

Kommunikation/ Synchronisation

Direct I/O
1. Prozessor stößt E/A an
2. E/A Modul führt Operation aus und setzt ein Statusbit
3. Prozessor prüft Bit regelmäßig
4. Sobald Bit gesetzt -> E/A komplett

Vorteil: schnell
Nachteil: verschwendet Ressourcen

Interrupt-Driven
1. Prozessor stößt E/A an
2. E/A Modul führt Operation aus
3. Prozessor arbeitet weiter
4. E/A Modul löst Interrupt aus
5. Prozessor liest Daten aus

Vorteil: weniger Ressourcenverschwendung

Direct Memory Access (DMA)
- Prozessor beschreibt Transfer
- unabhängiger Kontroller führt ihn aus
- Interrupt bei Abschluss

DMA Operation
- Beschrieben durch DMA-Deskriptor:
  - Lese/Schreiboperation
  - Geräteadresse
  - Startadresse
  - Länge vom Transfer
- Ablauf
  - Prozessor schreibt Deskriptor in den DMA Kontroller
  - Prozessor stößt DMA Transfer an
  - Prozessor und DMA Kontroller arbeiten unabhängig weiter
- Konsequenz: E/A Werk konzeptionell direkt mit Speicher verbunden

Hintergrundspeicher

Hintergrundspeicher – Festplatten
- Erweiterung vom Hauptspeicher
- Dauerhafte Speicherung größerer Datenmengen
- Magnetische Aufzeichnung
  - Magnetisierbare Schicht (Trägermaterial)
  - Zwei Magnetisierungsrichtungen (0 und 1)
  - Verwendung eines Magnetkopfes um zu lesen/ schreiben
  - Schreibstrom und Lesestrom (nur bei Magnetisierungsrichtung-Wechsel gibt Signal)
- Aufbau
  - Schreib/Lesekopf: sehr geringer Abstand zur Pla?enober?äche
  - 2 Köpfe pro Platte des Plattenstapels

Redundanz - Daten mehrfach gespeichert
- Fehler können über die Zeit auftreten (Abnutzung, Head Crashes) -> Daher wichtig Datenverlust zu vermeiden!
- **RAID (Redundant Array of Independent Disks)**
  - Festplatten zusammengeschaltet und als „eine" Festplatte dem System bereitgestellt
  - **RAID 0:** Disk Stripping: Keine Redundanz, Festplattenkapazität voll ausgelastet
  - **RAID 1:** Festplatteninhalte „gespiegelt" -> existieren mehrfach
  - **RAID 3:** Striping mit zusätzlicher Parity Disc (falls eine Disk ausfällt, können anhand dessen die Daten wieder berechnet werden)
  - **RAID 5:** Parity Disc verteilt
- Parity-Platte
  - Parity-Bit merkt sich auf Grundlage den anderen Platten einen Wert
  - Fehlendes Datum kann aus dem Parity und den noch vorhandenen Platten zurückgerechnet werden

Weitere Problemstellen bei NAND SSD
- **Nur blockbasierter Zugriff möglich**
  - Nur ganze Blöcke gelesen oder gelöscht
  - Nötig: Seitentabelle im SSD Speicher
  - Schnell nachschlagen, wo was ist
- **Anzahl Löschvorgänge begrenzt**
  - Man kann nur begrenzt auf den gleichen Block schreiben
  - Führt zu Bad Blocks

-> Nötig: Wear-Leveling
- Lösch im Kontrollor und Dateisystem möglichst verteilen
- Lebenszeit des Speichers erhöhen

SSD: Solid State Drives
- NV-RAMs (Non-volatile RAM) - RAM Speicher behält die Daten
- Alternative zu Standard HDD Hintergrundspeicher
- NAND Flash: Schneller, kleiner (gut für Speicher)
- NOR Flasch: Zuverlässiger, Einzelzugriff möglich (gut für BIOS)
- Im Vergleich zu HDD (Hard Disk Drive)
  - Vorteil: SSD in allen Hinsichten gegenüber HDD besser
  - Nachteil: teuer

Von Neumann Architektur

- **Was sind die 4 Bestandteile der Von Neumann Architektur?**
  - Leitwerk, Rechenwerk (ALU), Ein / Ausgabewerk, Speicher
- **Warum wir der Speicher in der Von Neumann Architektur in eine Speicherhierarchie aufgespaltet?**
  - Um Zugriffe auf häufig verwendete Daten zu beschleunigen. Da Hauptspeicher Zugriffe sind langsam
- **Was ist eine ISA?**
  - Schnittstelle zwischen Hard- und Software
  - Definiert die Befehle, die ein Prozessor ausführen kann
- **Was ist der Unterschied zwischen einem Datentyp und einem Datenformat?**
  - Datentyp: Interpretation der Daten durch ein Programm
  - Datenformat: Wie stehen die Daten im Speicher
- **Was ist die Nulladressform?**
  - Alle Operanden eines Befehls liegen auf dem Stack
  - Ergebnis wird wieder auf den Stack gelegt
- **Was ist die Einadressform?**
  - Es kann nur ein Operand angegeben werden
  - Andere Operanden werden wieder vom Stack gelegen
- **Was ist Little Endian?**
  - niederwertige Bytes werden zuerst gestpeichert
- **Was ist eine Register-Register Maschine?**
  - Alle Operanden müssen aus Registern kommen und werden wieder in Register gespeichert
- **Welche Modi gibt es innerhalb eines Betriebssystems?**
  - Benutzermodus: Hier laufen alle vom Benutzer ausgeführten Programme
  - Systemmodus: 1) Superset des Benutzermodus. Bietet zugriff auf tiefer liegende Funktionen des Betriebssystems
    2) Ein Programm im Benutzermodus kann durch System Calls in den Systemmodus wechseln
- **Was sind virtuelle Adressen?**
  - Vom Betriebssystem werden virtuelle Adressen an die Programme vergeben, welche dann auf reale, im Speicher verfügbare Adressen abgebildet werden
- **Warum werden Virtuelle Adressen verwendet?**
  - Es kann mehr Speicher, als tatsächlich physikalisch verfügbar an Programme vergeben werden
  - Jedes Anwendungsprogramm kann die selben Adressen verwenden (einfachere Kompilierung)
- **Was ist eine Familie von Instruktionssätzen ?**
  - ISAs innerhalb einer Familie sind untereinander kompatibel. Es kann ein altes Programm somit auch auf einer neueren ISA ausgeführt werden
  - Kompatibilität auf Hardware Ebene
- **Was passiert, wenn im Benutzermodus das system Flag Interrupt enable gesetzt wird?**
  - Es kommt zu einer Ausnahme vom Betriebssystems. Da ein Benutzerprogramm dieses Flag nicht schreiben darf
- **Warum sollte ein Prozessor in einem solchen Fall eine Ausnahme werfen?**
  - Damit ein Betriebssystem, selbst wenn dieses im Benutzermodus läuft (Hypervisor) die Ausnahme behandeln kann
- **Wo liegt die Verbindung zwischen einer PUSH und einer CALL Operation in der x86 ISA?**
  - Ein CALL ist ein PUSH und ein JMP. Denn es wird die Rücksprungadresse und der momentane Frame Pointer auf den Stack gelegt. Dann wird zum Unterprogramm gesprungen
  - Somit können mehrere CALL Aufrufe ineinander geschachtelt werden
- **Was ist das Problem an der Beschreibung einer Von Neumann Architektur in einer Hochsprache?**
  - Parallel Abläufe können nur schwer dargestellt werden
  - Deswegen wird eine Sprache, wie VHDL verwendet werdendet
- **Was ist der Unterschied zwischen Makros und einem Funktionsaufruf mit CALL?**
  - Makros: Der Code einer Unterroutine wird direkt in das Assembler Programm kopiert. Instruktionen werden dupliziert
  - CALL: Sprung zu einem Unterprogramm. Overhead durch Zugriffe auf den Stack
- **Warum sind zwei Durchläufe des Assemblers notwendig?**
  - Im ersten Durchlauf werden die Adressen von Symbolen (Marken) aufgelöst. Diese können dann im zweiten Durchlauf verwendet werden

- **Was sind die Aufgaben des Linkers?**
  - Verknüpft verschiedene object files in ein ausführbares Programm
- **Was ist der Unterschied zwischen statisch und dynamisch gebundenen Bibliotheken?**
  - Dynamische Bibliotheken: Werden erst zur Laufzeit nachgeladen, sobald das Programm diese benötigt
  - Statische Bibliotheken: Werden in die entstehende ausführbare Datei kopiert
- **Was ist eine Aufrufkonvention?**
  - Menge von Regeln, wie Parameter / Rückgabewerte an eine Funktion übergeben werden
  - Welche Register überschrieben werden dürfen. Ob der aufrufende, oder der aufgerufene die Register sichern muss.
- **Was bedeutet ABI?**
  - Application Binary Interface
  - Beschreibt die Aufrufkonvention, sowie die Spezifikation, wie Funktionen des Betriebssystems aufgerufen werden
- **Was ist der Unterschied zwischen einem Compiler und einem Interpreter?**
  - Compiler: Generiert aus einem in einer Hochsprache geschriebenen Programm eine ausführbare Datei.
  - Interpreter: 1) Ist selber ein Maschinenprogramm, welches ein, in einer Hochsprache geschriebenes Programm ausführen kann; 2) Bekommt als Eingabe das Programm in der Hochsprache. Dieses wird nie übersetzt, sondern jede Instruktion nacheinander ausgeführt
- **Was ist das Frontend / Backend eines Compilers?**
  - Frontend: Der hochsprachenabhängige Teil des Compilers
  - Backend: Maschinenabhängige Teil des Compilers
- **Was ist cross compiling? Wie funktioniert cross compiling?**
  - Cross compiling ist die Möglichkeit Maschinencode zu generieren, welcher auf einer anderen Architektur, als der im Rechner verbauten, ausgeführt werden kann
  - Es muss für den Compiler die Architektur, das Betriebssystem, sowie die ABI spezifiziert werden
- **Was ist ein Basisblock?**
  - Ein Basisblock ist ein Stück Code, mit genau einem Einstiegs- und Ausstiegspunkt
  - Es kann nicht in einen Teil des Blocks gesprungen werden. Ebenfalls ist es nicht möglich an einer Zwischenstelle aus dem Code heraus zu springen
- **Wie viele Basisblöcke hat eine Schleife?**
  - Zwei. 1) Ein Block für die Initialisierung der Schleifenvariable.
    2) Ein weiterer wird benötigt für den auszuführenden Block der Schleife

Technische Realisierung von Rechnern

- **Was ist der Vorteil einer Schichtung bei der Realisierung eines Rechners?**
  - Es muss nur die Schnittstelle der jeweils unteren Schicht bekannt sein. Alle darunter liegenden Schichten sollten unsichtbar sein
  - Somit können untere Schichten beliebig ausgetauscht werden, falls diese die gleiche Schnittstelle haben
- **Was ist eine Mikrooperation?**
  - Eine Mikrooperation ist die kleinstmögliche Einheit, welche auf der Mikroinstruktionsebene verwendet werden kann
- **Was ist das Verhältnis zwischen Mikroinstruktionen und Mikrooperationen?**
  - Eine Mikroinstruktion besteht aus mehreren Mikrooperationen
  - Es können mehrere Mikrooperationen gleichzeitig ausgeführt werden
- **Was ist der Tri-State-Bus?**
  - Es können Komponenten mit Komponenten kommunizieren, wobei nur eine Leitung benötigt wird
  - Es kann einer der drei folgenden Werte auf dem Bus liegen: 1, 0, keine Information
- **Aus welchen Klassen von Elementen besteht die Mikroarchitektur?**
  - Verknüpfende Elemente: Nehmen binäre Signale als Eingang und berechnen daraus einen Ausgang
  - Verbindungen: Diese Elemente verbinden mehrere anderer Elemente (z.B. Tri-State-Bus)
  - Speicher: Können Zustände über einen Zeitraum halten (z.B. Flip Flop)
- **Was ist mit AT und ST im Zyklus einer Mikroinstruktion gemeint?**
  - AT: Adressteil, steuert die Fortschaltung der Befehle innerhalb des Mikroprogramms
  - ST: Steuerteil, steuert die Komponenten der Maschine

- **Was sind die wichtigsten Befehle des Am2910?**
  - CONT (Continue)
  - JZ (Jump Zero) -> Sprung zu IFETCH
  - CJMP (Conditional Jump) -> Bedingter Sprung
  - CALL (Unterprogrammaufrufe)
  - LOOP (Schleifenbefehle)
- **Warum existieren solche komplizierten Möglichkeiten im Am2910?**
  - Da der Baustein Am2910 so kompliziert ist, kann die Anzahl der benötigten Mikroinstruktionen deutlich reduziert werden
  - Dies erlaubt eine schnellere Ausführung von Mikroprogrammen und benötigt weniger Platz im Mikroprogrammspeicher. Somit kann dieser kleiner und näher am Am2910 verbaut werden
- **Warum werden die Bausteine der MI-Maschine bit slice Prozessoren genannt?**
  - Jeder der Bausteine des Am2904 (ALU) kann 4 bit parallel verarbeiten
  - Somit können beliebige Wortbreiten in einer Architektur unterstützt werden
- **Wann können mehrere bit slice Bausteine aneinander geschaltet werden?**
  - Wenn eine Operation in kleinere Teile aufgeteilt werden kann. Bei der Addition ist dies durch einen Carry Lookahead Baustein möglich
- **Was ist ein Beispiel für Parallelität in z.B. der ALU auf Mikroebene?**
  - Es können in einem Takt Daten aus einem Register gelesen, verarbeitet und wieder in ein Register gespeichert werden
- **Sind die zwei Statusregister (Mikrostatusregister und Maschinenstatusregister) eine Verletzung der Abstraktionsschichten in der Von Neumann Architektur?**
  - Nein, da für den Maschinenprogrammierer immer nur das Maschinenstatusregister sichtbar ist. Dieser kann nicht auf das Mikrostatusregister zugreifen
- **Was ist ein Ruhe- / Arbeitskontakt? Wo liegt der Unterschied zwischen den Beiden?**
  - Ruhekontakt: Ist geöffnet, falls eine Steuerspannung anliegt
  - Arbeitskontakt: Ist geschlossen, falls eine Steuerspannung anliegt

Schaltnetze

- **Wie sind das Null- und Einselement in der Boolschen Algebra definiert?**
  - Nullelement: Wird durch Verundung von einem Element mit seinem Komplementär erzeugt ($a\bar{a} = 0$)
  - Einselement: Wird durch Veroderung von einem Element mit seinem Komplementär erzeugt ($a + \bar{a} = 1$)
- **Was ist die Einsmenge eines Ausdrucks?**
  - Bei einer Schaltfunktion mit Eingängen, gibt die Einsmenge alle Belegungen n an, sodass die Funktion wahr wird (1 ausgibt)
- **Was sind Optimierungsziele für Schaltnetze?**
  - Es sollten möglichst wenig Komponenten verwendet werden und das Schaltnetz sollte möglichst schnell sein (geringe Tiefe der Schaltkomponenten)
  - Verwendung von möglichst einfachen Gatten (NAND, NOR). Bestehen aus nur 4 Schaltkomponenten
  - Es sollten möglichst wenige Elemente kaskadiert werden (länger andauernde Hazards)
- **Was ist der Unterschied zwischen einem Minterm und einem Monom?**
  - Minterm: 1) Ein Term, welcher nur aus Konjunktionen besteht
    2) Ein Minterm muss immer alle Eingänge der Schaltung als Signale verwenden
    3) Spezialisierung eines Monoms
  - Monom: Ein Monom muss nicht alle Eingänge einer Schaltung auch als Signale verwenden
- **Was ist der Unterschied zwischen einer disjunktiven Normalform und einer konjunktiven Normalform?**
  - In jeder der Normalformen existieren immer genau 2 Ebenen
    - DNF -> Disjunktion von Konjunktionen
    - KNF -> Konjunktion von Disjunktionen
- **Was ist der Vorteil einer Schaltung in DNF / KNF?**
  - Diese Normalformen stellen eine einfache Möglichkeit dar, eine beliebige Funktion als ein Schaltnetz zu realisieren

Schaltwerke

- **Warum können Schaltwerke als endliche Automaten modelliert werden?**
  - Der gespeicherte interne Zustand kann durch die Zustände eines endlichen Automaten dargestellt werden
  - Es kann exakt beschrieben werden, wie ein Schaltwerk funktionieren soll
- **Was ist der Unterschied zwischen einem Mealy und einem Moore Automat?**
  - Moore Automat: Ausgabe hängt nur vom Zustand ab
  - Mealy Automat: Ausgabe hängt von Zustand und Eingabe ab
  - Beide sind gleich mächtig
- **Wie werden Verzögerungen in Schaltwerken idealisiert dargestellt?**
  - Es werden keine Verzögerungen für die verwendeten Schaltnetze modelliert
  - Alle Verzögerungen werden in einem einzelnen Baustein zusammengefasst
  - Verzögerung wird in der Rückkopplungsschleife dargestellt
  - Es muss eine Verzögerung gewählt werden, die größer ist, als die Summe aller tatsächlichen Verzögerungen
- **Was ist ein synchroner Prozess? Wofür wird dieser benötigt?**
  - Ein synchroner Prozess ist eine Abfolge von Befehlen, welche voneinander abhängig sind
  - Dieser wird benötigt, da durch einen synchronen Prozess ein deterministisches System gebaut werden kann. Somit ist der Prozess funktional
- **Was ist die Verbindung zwischen einem Hazard und einem Race?**
  - Hazard: Klingt nach einem bestimmten Zeitpunkt ab
  - Race: 1) Race entsteht durch einen Hazard auf dem Rückkopplungsweg des Schaltwerks
    2) Klingt nicht ab, da dieses innerhalb der Rückkopplung entsteht
- **Ist ein D-Flip-Flop ein synchrones, oder ein asynchrones Schaltwerk?**
  - Es ist asynchron, da keine Taktung innerhalb der Rückkopplung existiert
- **Was ist der Unterschied einer Flankensteuerung und einer Taktpegelsteuerung?**
  - Flankensteuerung: 1) Es wird nur reagiert, falls eine Taktflanke vorliegt; 2) Um die Flankensteuerung zu realisieren wird der Hazard eines verknüpften NOT und AND Gatters verwendet, welches bei einer sich ändernden Taktflanke ein kurzes Signal ausgibt
  - Taktpegelsteuerung: Es wird nur reagiert, falls ein Taktpegel anliegt

Speicherhierarchie

- **Warum wird in modernen Prozessoren ein Cache verwendet?**
  - Es wird ein Cache eingebaut, um auf kleine Teile des Speichers schneller zugreifen zu können
  - Ein kleiner Speicher erlaubt hohe Zugriffsgeschwindigkeiten
  - Desto größer eine baumartige Speicherstruktur ist, desto länger dauert es, in dieser die richtige Adresse zu finden
- **Warum funktionieren Caches?**
  - Ein Programm besitzt meist zeit- und örtlich lokale Speicherzugriffe. Diese können in einem Cache zwischengespeichert werden
- **Wie wird die örtliche Lokalität ausgenutzt?**
  - Es werden ganze Blöcke aus dem Speicher geladen, anstatt nur einzelne Zeilen
- **Wie wird die zeitliche Lokalität ausgenutzt?**
  - Es werden immer die ältesten Daten im Cache von neueren überschrieben
  - LRU: least recently used
  - LFU: least frequently used
- **Warum werden die Caches in eine Hierarchie aufgeteilt?**
  - Da somit verschieden große und schnelle Caches kombiniert werden können, um mehr Daten in der Summe aller Cache Level halten zu können
  - Eine mehrstufige Hierarchie macht Speicherzugriffe langsamer, da jeder der Caches überprüft werden muss, ob dieser die gesuchten Daten enthält
- **Warum wird der L1 Cache in Daten und Code aufgeteilt?**
  - Damit sich die verschiedene Arten von Speicherzugriffen nicht gegenseitig ausbremsen (Von Neumann Flaschenhals)
  - Caches können Speziell für die unterschiedlichen Anforderungen der Datentypen optimiert werden

- **Was ist der Von Neumann Flaschenhals?**
  - Da Code und Daten im Gleichen Speicher stehen, kommt es zu vielen Zugriffen auf den Hauptspeicher
  - Durch Caching lässt sich die Anzahl der benötigten Zugriffe verringern
- **Was ist eine Cache Optimierung?**
  - Veränderung des Datenlayouts im Speicher, um effizienter darauf zugreifen zu können
  - Jede Art von Veränderung im Ablauf eines Programms, um die Ausnutzung von Caches zu verbessern
  - Es wird versucht die zeitliche und örtliche Lokalität zu verbessern
    - Daten zusammen speichern
    - Zugriffe auf die gleichen Daten zeitlich zusammenfassen
- **Warum wird eine Speichermatrix verwendet, anstatt einer langen Zeile?**
  - Es wird die Anzahl der benötigten Speicherleitungen minimiert
- **Wofür steht TLB?**
  - Translation Lookaside Buffer
- **Wofür wird der TLB benötigt?**
  - Er wird zur Übersetzung von virtuellen auf physikalische Adressen verwendet
  - Es kann mehr Speicher, als physikalisch verfügbar einem Programm zugewiesen werden. Überlaufender Speicher wird dann auf die Festplatte ausgelagert
- **Was steht in einem TLB?**
  - Der TLB cached für jeden virtuellen Block wie dieser auf den physikalischen Speicher abgebildet ist
- **Was passiert, wenn für einen Block kein Eintrag im TLB existiert?**
  - Es muss zu einem Seitenfehler kommen. Der Prozessor sendet eine Ausnahme an das Betriebssystem. Dieses findet dann den gesuchten Speicherblock und legt einen Eintrag im TLB an
- **Welche Möglichkeiten gibt es für einen Zugriff auf den TLB?**
  - Der Eintrag existiert im TLB
    - Falls das Benutzerprogramm Zugriff auf den Speicherblock hat, ist dieser schnell.
    - Falls das Benutzerprogramm keinen Zugriff auf den Speicherblock hat, kommt es zu einer Ausnahme
  - Der Eintrag existiert nicht im TLB
    - Falls ein Eintrag in den Pagetabellen des Prozessors existiert, kann dieser selbständig den Speicherzugriff übersetzen. Somit muss keine Ausnahme an das Betriebssystem geworfen werden. Es wird ein “page walk” durch die Hardware durchgeführt
    - Es kommt zu einer Ausnahme des Prozessors. Dieser informiert das Betriebssystem

Moderne Mikroprozessoren

- **Was ist der Vorteil von Pipelining?**
  - Die Schritte einer Instruktion können parallel ausgeführt werden
  - Es werden bereits Teile folgenden Instruktionen ausgeführt
- **Welcher Art von Parallelismus kann Pipelining zugeordnet werden?**
  - Parallelismus auf der Instruktionsebene (Instruction level parallelism)
- **Welche Arten von Parallelismus gibt es?**
  - Bitebene: Wortbreite, mit der Signale in Hardware verdrahtet werden
  - Datenebene: Ein Befehl wird auf mehrere Daten gleichzeitig angewendet (Vektorbefehle)
  - Thread Ebene: 1) Es werden mehrere Instruktionsstränge gleichzeitig ausgeführt
    2) Muss vom Programmierer explizit verwendet werden
- **Welche Arten von Pipeline Konflikten gibt es?**
  - Datenkonflikte: Ein Prozess greift auf Daten zu, dessen Berechnung noch nicht abgeschlossen ist
  - Sprungkonflikte: Das Ziel eines Sprungs muss bekannt sein. Sonst kann dieser nicht ausgeführt werden
  - Ressourcenkonflikte: Falls mehrere Zugriffe auf die selben Daten benötigt werden, muss die Abarbeitung serialisiert werden

Vorzeichenbit: Ein extra Bit nur fürs Vorzeichen "0" ✓, "1" ✗
- ⊕ Analog zu Dezimalsystem
- ⊖ Doppelte Null (+0 = 0000, -0 = 1000)
- ⊖ neue Arithmetik nötig

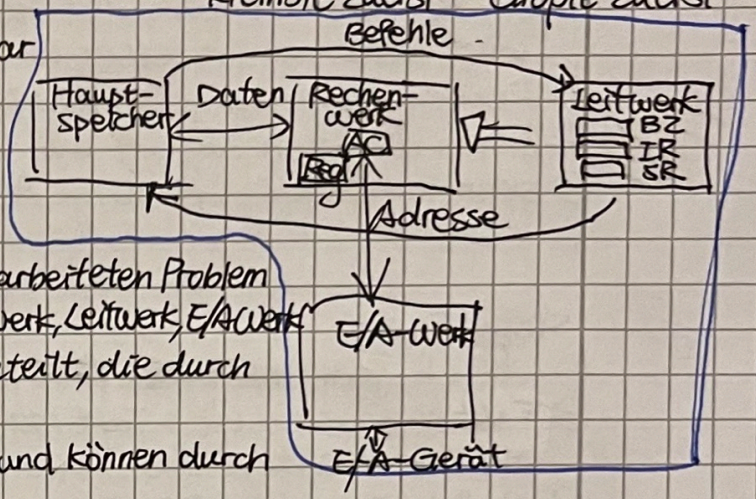
Einerkomplement: alle Bits invertieren
- ⊕ intuitiv
- ⊖ Doppelte Null (+0 = 0000, -0 = 1111)
- ⊖ neue Arithmetik nötig

Zweierkomplement: Einerkomplement + 1
- ⊕ Volle Ausnutzung des Zahlenraums
- ⊕ Eindeutigkeit der Null
- ⊖ Zahl nicht direkt ~~les~~ablesbar

0x76543210

	Little Endian		Big-Endian
N+3	76	N+3	10
N+2	54	N+2	32
N+1	32	N+1	54
N	10	N	76

"kleinste zuerst" "größte zuerst"



Von-Neumann-Architekturkonzept
1. Die Struktur des Rechners ist unabhängig von bearbeiteten Problem
2. Rechner besteht aus 4 Werken: Speicher, Rechenwerk, Leitwerk, E/A-Werk
3. Der Hauptspeicher ist in Zeilen gleicher Größe geteilt, die durch fortlaufenden Adressen bezeichnet werden
4. Programme und Daten stehen im selben Speichern und können durch die Maschiene verändert werden
5. Die Maschiene benützt Binärcode
6. Das Programm besteht aus einer Folge von Befehlen (nacheinander gespeichert and ausgeführt)
7. Von der Folge der Befehle können mit (un)bedingten Sprüngen abgewichen werden

CISC (Complex Instruction Set Computer) ⇒ Intel X86
Definition: viele komplexe Instruktionen
Vorteil: kompakte Maschienencode-Darstellung; geringer Speicherbedarf des Quellcode
Nachteil: höhere Komplexität um dies auf Prozessor umzusetzen; viele ungenutzte Funktion

RISC (Reduced Instruktion Set Computer) ⇒ M1 Chip ARM
Definition: weniger und sehr einfache Instruktionen
Vorteil: wenige Befehle → einfacher zu implementieren
aufgrund des reduzierten Befehlssatz wird weniger ~~Platz~~ Chipplatz benötigt
Nachteil: schwieriger zu programmieren, wird meist ein Compiler verwendet

Adressierungsformen

Nulladdressformen: Kellermaschiene (Java VM), Operand auf dem Stack

Einadressformen: Akkumulatormaschiene, Operand im Akkumulator
- ⊖ Unflexibel, häufiges Laden u. Speichern des Akkumulators nötig

zweiadressformen: Register-Speicher-Modell, einer im Register, andere im H-Speicher
- ⊖ Ergebnis überschreibt immer den ersten Operand

Dreiaddressformen: Speicher-Speicher-Modell (frühere CISC-Prozessoren), im Hauptspeicher
- ⊕ Volle Flexibilität für Programmierer
- ⊖ viele Speicherzugriffe, lange Befehle, langsam

Register-Register-Modell (fast alle RISC-V, außer load&store), im Register
- ⊕ kurze Befehle, schneller
- ⊖ weniger Auswahlmöglichkeiten, weil Operand & Ergebnis nur im Register stehen dürfen

einheitliches Befehlsformat
→ alle Befehle haben eine feste Länge
⊕ schnell & einfach zu dekodieren
⊖ unflexibel, kein Platz für große Konstante im Befehl

Variables Befehlsformat
→ Befehle können unterschiedlich lang sein
⊕ extrem flexibel, alle Addressierungsart möglich
⊖ Dekodierung sehr zeitaufwändig, Problem mit Speicherverwaltung

Assembly
→ eine Hardware nahe Programmiersprache
ein Assemblierer übersetzt ein Programm in Maschienensprache, somit wird dieses vom Prozessor ausführbar
⊕ Rechengeschwindigkeit schnell: Programme können sehr effizient geschrieben
⊕ bieten direkten Zugriff auf Hardware
⊖ sehr aufwändig (niedrige Abstraktion) und fehleranfällig

singelCycle
⊕ einfach, jede Instruktion wird in einem Takt ausgeführt
⊖ Performanceverlust: Takt muss lang genug für langsamste Instruktion (lw) sein, auch wenn andere Instruktion viel schneller arbeiten
⊖ 2 Addierer für PC-Rechnung notwendig
⊖ getrennte Speicher für Instruktionen und Daten

MultiCycle
⊕ Höhere Taktgeschwindigkeit, da kürzere kritische Pfad
~~⊕ einfache Instruktionen sind schneller fertig~~
⊕ Wiederverwendung von HW → nur ein Addierer erforderlich
⊖ Ausführung einer Instruktion benötigt mehrere Takte
⊖ etwas komplizierteres Steuerwerk
⊕ Befehle beliebiger Länge lassen sich effizienter arbeiten (kein Warten für unnötige Schritte)

SRAM (Statisches Random Access Memory)
→ Speicherinhalt bleibt bei anliegender Spannung erhalten
⊕ CMOS Technologie (einfach integrierbar)
schneller Zugriff; behält Wert solange Strom anliegt
⊖ relativ groß und teuer, großer Energiebedarf wenn aktiv
nur kleine Kapazität möglich

DRAM (Dynamisches Random Access Memory)
→ Speicherzustand muss immer wieder periodisch aktualisiert werden
⊕ weniger komplexe Herstellung, deshalb Kostengünstigere Erzeugung
große Kapazitäten leicht möglich
⊖ braucht Kondensator (schwer integrierbar) und refresch
große Energie Bedarf, langsamer Zugriff

Reservieren ohne Initialisieren

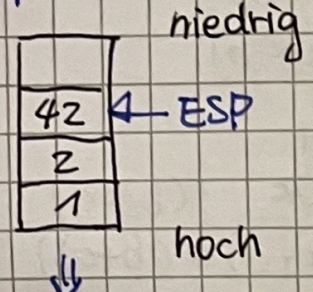
```
section .bss
    RESB 1   ; 1 Byte
    RESW 1   ; 2 Byte
    RESD 1   ; 4 Byte
    RESQ 1   ; 8 Byte
```

Reservieren mit Initialisieren "Wert"

```
section .data
DB Wert   ; 1 Byte
DW Wert   ; 2 Byte
DD Wert   ; 4 Byte
DQ Wert   ; 8 Byte
```

Immidiate
Byte 1 Byte
Word 2 Byte
DWord 4 Byte
@Word 8 Byte

push 42 pop eax ⇒ eax=42



Allokation
sub esp, Größe

Freigabe
add esp, Größe

Links-Rechtsschieben nach n Bits

vorzeichenlos vorzeichenbehaftet

SHL n } ⇒ Multiplikation mit 2^n
SAL n } fehlende Bits mit 0 gefüllt
CF = MSB(dest)
OF = MSB(dest) xor CF

SHR n } ⇒ Division durch 2^n
SAR n }

SHR { fehlende Bits mit 0 gefüllt
CF = LSB(dest) OF = MSB(dest)

SAR { falls MSB=1 ⇒ mit 1 gefüllt
falls MSB=0 ⇒ mit 0 gefüllt
CF = LSB(dest) OF = 0

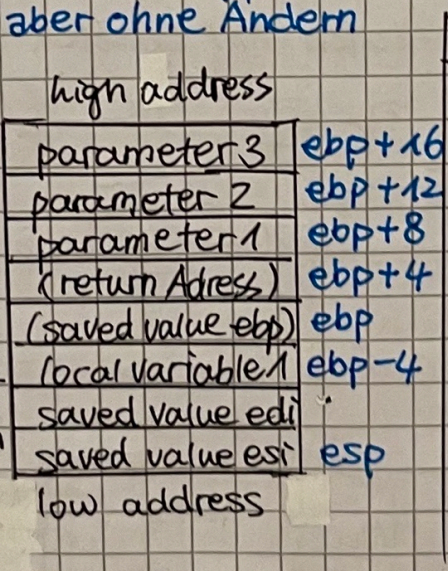
ROL CF = MSB(dest)
OF = MSB(dest) xor CF (falls 1)
ROR CF = LSB(dest)

Flags

Overflow Flag → signed Overflow $[-2^{N-1}, 2^{N-1}-1]$
Carry Flag → unsigned Overflow $[0, 2^{N-1}]$

mov, push, pop setzen keine Flags
für and, or, test, xor: CF=0 OF=0
wie and, aber ohne Ändern

Neg Zweierkomplement
Not Bitweise Negation

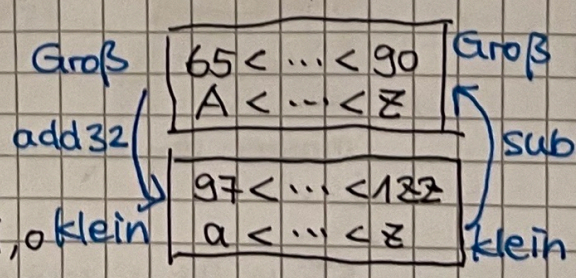


function(parameter 1, parameter 2, parameter 3)

```
push parameter 3
push parameter 2
push parameter 1
sub esp, 4
call function
add esp, 4
```

```
function:
    push ebp
    mov ebp, esp
    ....
    pop ebp
    ret
```

$a \bmod 2^n$
= and a, 2^n-1




```
length_loop:
    cmp byte [eax+ecx], 0
    je got_length
    inc ecx
    jmp length_loop
```

AND &

oder ≥1

Anti-Valenz XOR ⊕ =1

NAND &

NOR ≥1

Negation 1

Addition $a+b \equiv a \vee b$ Oder

Multiplikation $a*b \equiv a \wedge b$ Und

komplementär	Idempotenz	neutrale Element
$a+\neg a=1$	$a+a=a$	$0+a=a$
$a*\neg a=0$	$a*a=a$	$1*a=a$

Absorption
$(a+b)*a=a$
$(a*b)+a=a$

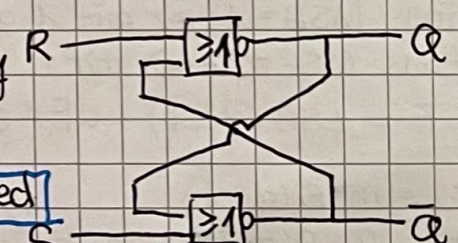
De Morgansche Regel
$(\neg a*\neg b)=\neg(a+b)$
$(\neg a+\neg b)=\neg(a*b)$

Kommutativität
$a+b=b+a$
$a*b=b*a$

Distributivität
$(a+b)*c=(a*c)+(b*c)$
$(a*b)+c=(a+c)*(b+c)$

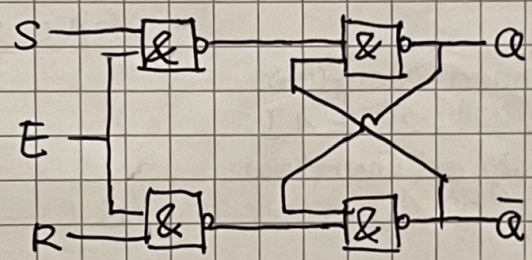
RS-FlipFlop

S R	Q $\bar{Q}$
0 0	memory
0 1	0 1
1 0	1 0
1 1	not used

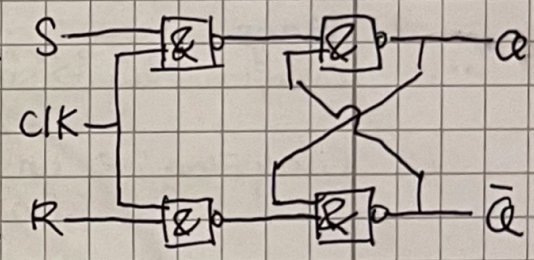


{NOR-Gatter $Q=\bar{Q}=0$
NAND-Gatter $Q=\bar{Q}=1$

RS-Latch (mit Enable)



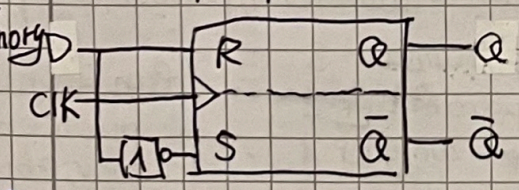
RS-FlipFlop (mit clk)



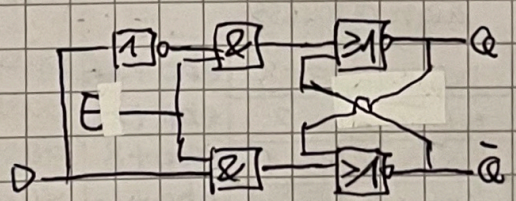
D-FlipFlop (mit clk)

E D	Q
0 X	memory
1 0	0
1 1	1

Q copys D



D-Latch (mit Enable)

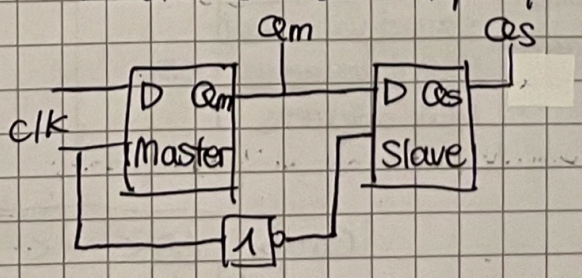


JK-FlipFlop

clk	J	K	Q
0	X	X	memory
1	0	1	0
1	1	0	1
1	1	1	$\bar{Q}$

Q = 0, 1, 0, 1 ...
immer wechselnd

Master Slave D-FlipFlop



Master Qm übernimmt bei steigender Flanke den Ausgang des Schaltnetzes
(→ z.B. follow D bei D-FlipFlop)

Slave Qs übernimmt bei fallender Flanke den Ausgang des Masters

RegWrite ⇒ nur bei sw und Branch 0, sonst 1 (Resultat abspeichern)

ALUSrc ⇒ nur bei sw, lw, I-Typ wird 1 (Speicheradresse berechnen)

MemWrite ⇒ nur bei sw wird 1 (Daten in Speicher ablegen)

ALUOp ⇒ bei { lw, sw 00
{ beq 01
{ sonst 10

ResultSrc ⇒ { I-Typ, R-Typ 00
{ lw 01 (gelesene Daten speichern)
{ jal 10 (PC im Register speichern)
{ B-Typ und sw --

ImmSrc ⇒ { sw 01 Adresse berechnen
{ B-Typ 10 } Sprungadresse berechnen
{ jal 11 }

lw 5 Takte
sw 4 Takte
I-Typ 4 Takte
R-Typ 4 Takte
JAL 4 Takte
Branch 3 Takte

Takt-Anzahl
SingleCycle: Takt-Anzahl = Anzahl der ausgeführten Instruktionen
Multicycle: Takt-Anzahl = Summe der jeweils benötigten Takten
Pipeline: Takt-Anzahl = Anzahl der ausgeführten Instruktionen (inklusiv NOPs) + 4 Takte (bis alles fertig)

hat mehr 1
Spaltendominanz wird die dominierende Spalten gestrichen
Zeilendominanz wird die dominierte Zeile gestrichen
hat weniger 1

2-fach assoziativ, 4 Cachemenge (Sets) ⇒ 8 Cachezeile
jede Cachezeile 8 Byte ⇒ 8×8 = 64 Byte (Cache Gesamt-Größe)
offset = $\log_2 8 = 3$
index = $\log_2 4 = 2$
tag = 32 − 3 − 2 = 27

Frequenz f = Frequenz des Prozessors
Instruction Count (IC) = Anzahl der auszuführenden Instruktion
Cycles per Instruction (CPI) = ø Anzahl an Zyklen pro Instruction
Memory Access Rate: ø Anzahl an Speicherzugriffe pro Instruktion

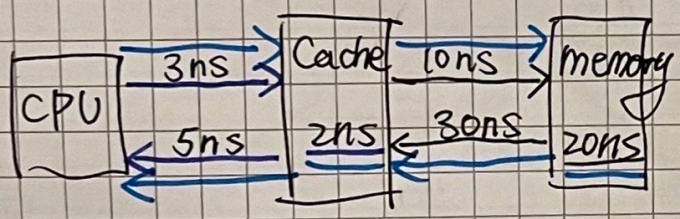
$$\text{Datenverarbeitungszeit} = IC \times \frac{CPI}{f}$$

$$\text{Speicherzugriffszeit} = IC \times \frac{\text{Memory Accessen}}{\text{Instruction}} \times \text{Average Memory Access Time}$$

$$\text{CPU Time} = \text{Datenverarbeitungszeit} + \text{Speicherzugriffszeit}$$

$$= IC \times \left(\frac{CPI}{f} + \frac{\text{Memory Accesses}}{\text{Instruction}} \times \text{Average Memory Access Time}\right)$$

Average memory access time = Hit Rate × Hit Latency + Miss Rate × Miss Latency



Hit Latency = 3ns + 2ns + 5ns = 10ns
Miss Penalty = 10ns + 20ns + 30ns = 60ns ⇒ die zusätzlich von der Speicherhierarchie benötigten Antwortszeit, wenn ein Cache Miss auftritt
Miss Latency = Hit Latency + Miss Penalty
= 10ns + 60ns
= 70ns

MESI Protokoll
Read I → E falls mehrere ⇒ shared S
Write diese Cachezeile S/I/E ⇒ M —falls noch Read→ S (beide Prozessor)
andere Cachezeile → I (veraltet)

Amdahl's Gesetz: theoretische SpeedUp bei konstante Problemgröße

$$\text{Speedup}_{\text{Amdahl}}(n) = \frac{T}{t_s + \frac{t_p}{n}} \qquad \lim_{n\to\infty} \text{annähend zu } \frac{1}{t_s}$$

Gustafon's Gesetz: bei Problemgröße proportional zur Prozessoranzahl

$$\text{Speedup}_{\text{Gustafon}}(n) = \frac{t_s + n \cdot t_p}{T}$$

unendlich groß bei $n \to \infty$

Speicherzugriffsmuster

zeitliche Lokalität: Zugriff auf ein Datum erhöht Wahrscheinlichkeit auf Zugriff desselben Datums in der Zukunft

räumliche Lokalität: Zugriff auf ein Datum erhöht Wahrscheinlichkeit auf Zugriff nahegelegener Daten

Cache Typen

Direkt Mapped Cache: jede Adresse passt genau zu einer Cachezeile
⊕ schneller Zugriff auf Cache-Daten (einfach & billig)
⊖ unflexibel, häufiges Verdrängung von Cachezeilen

Voll-Assoziative Cache: jede Adresse kann auf jede Cache Zeile abgebildet werden
⊕ sehr flexibel, keine Berechnung für Index-Bit
⊖ sehr langsame & teuere Suche

n-fach Assoziativ: jede Adresse kann auf einen Teil der Cache Zeilen abgebildet
⊕ Kompromiss zwischen Flexibilität & Kosten
⊖ Indexberechnung notwendig

Cache Misses Typen

(Compulsory) Cold Miss: erster Zugriff auf eine Adresse
→ können nur durch Prefetching vermieden, sonst kann nicht vermeiden

Conflict misses: Daten in einer Cache wurden überschrieben
→ können nur durch höhere Assoziativität vermieden, Cache wird aber langsamer

Capacity Misses: Die benötigten Daten eines Programms ist größer als Cache
→ können durch größere Cache reduziert, Cache wird aber langsamer

Ersetzungsstrategie

Least Recently Used (LRU): immer die älteste Cachezeile ersetzt
Least Frequently Used (LFU): immer die am seltensten benutzte Cachezeile (neue Blöcke werden bevorzugt eliminiert)

Random: keine Berücksichtigung von Lokalität
FIFO: keine Berücksichtigung von Nutzungsfrequenz

Aktualisierungsstrategie

Bei Cache-Hit:
1) Durchschreibe-Strategie (Write Through): Schreibzugriffe werden im Cache und Hauptspeicher aktualisiert
2) Rückschreibe-Strategie (Write Back): Schreibzugriffe werden nur im Cache aktualisiert; Rückschreiben auf Hauptspeicher, solange Cachezeile überschrieben wird

Bei Cache MISS
Write-Allokation: Speicher wird in Cache geladen und aktualisiert
No-Write-Allokation: keine Aktion des Caches

Memory Management Unit (MMU)

→ bei jedem Speicherzugriff übersetzt die MMU die virtuelle Adresse in eine physikalische Adresse im Hauptspeicher

Virtuelly Indexed Cache → Cache zwischen MMU und Prozessor

⊕ direkter Zugang zum Cache ohne Übersetzung, gut für kleine Caches

⊖ mehrere Prozessoren verwenden die gleichen virtuellen Adressen

Physikally Indexed Cache → Cache zwischen Hauptspeicher und MMU

⊕ übersteht Prozess/Kontextwechsel, gut für große Caches

⊖ jeder Zugriff muss zuerst durch die MMU

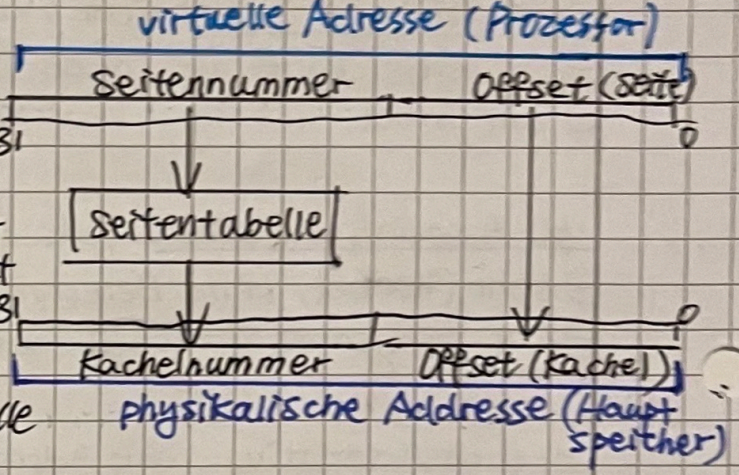
Paging: Adressumsetzung-Prinzip

→ Innerhalb der MMU werden die virtuelle Adressen über Seitentabelle des Betriebssystem in physikalische Adresse übersetzt

⊕ Das Betriebssystem kann einem Prozessor mehr Speicher zuweisen, als verfügbar ist

⊕ jeder Prozessor kann die gleichen virtuelle Adressen verwenden

⊖ jeder Speicherzugriff muss übersetzt werden



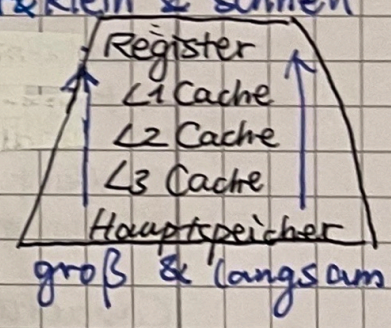
kleinere Seitengröße: ⊕ mehr Flexibilität, weniger Speicherverschnitt

größere Seitengröße: ⊕ kleinere Seitentabelle, schnellere Übersetzung, wenig TLB Misses

Cache Hierarchie

Inklusive Cache: alle in einem Cache enthaltenen Blöcke sind auch in einem darunterliegenden größeren Cache enthalten

Exclusive Cache: keine zwei Blöcke enthalten den gleichen Speicherblock. Falls ein Cache voll ist, Verdrängung in den darunterliegenden tieferen Speicher (L1 → L2 → L3)



Von-Neumann-Flaschenhals

→ 2 verschiedene Speicherzugriff-Typen können sich gegenseitig behindern und zu unnötigen Leistungseinbußen führen

↳ wird durchs Trennen von Daten und Code gelöst

Harvard Architektur

→ Code und Daten werden getrennt gespeichert

Von Neumann Architektur

→ nur L1 wird in Code und Daten getrennt

sequetielle Lesezugriffe

komplexe Lese- und Schreibzugriffe

Prefetching: um die räumliche Lokalität von Speicherzugriffe auszunutzen, werden mehr Blöcke in den Speicher geladen, als in der momentanen Anfrage benötigt wird.

Hardware Prefetching
→ Hardware untersucht alle Zugriffsadressen und entdecke regelmäßige Zugriffe
⊕ einfach zu implementieren, gut für reguläre Zugriffe
⊖ nicht hilfreich für komplexe Zugriffsmuster

Software Prefetching
→ Ein Compiler fügt Prefetching Instruktion in den Maschienecode ein, um vorliegende Zugriffe präziser zu entdecken
⊕ auch unregelmäßige Zugriffe können entdeckt werden
⊖ fehlende Transparenz, benötigt Codeänderung

Dynamisches Scheduling (Out of Order Execute)
→ (Daten-) unabhängige Instruktionen können parallel ausgeführt werden, obwohl vorherige Operationen blockiert sind.
⊕ bessere Ausnutzung von vorhandenen Ressourcen, bessere Nutzung von Parallelität im Programm

Superskalarprinzip
→ mehrere einzelne unabhängige Maschienenbefehle können gleichzeitig ausgeführt werden

Superskalarprinzip & Pipeling
Gemeinsamkeiten & beide erhöhen Leistung
↳ bessere Ausnutzung von Einheiten und Parallelität
↳ vor allem in Kombination mit Out of Order Execute
Unterschiede: Pipeling: mehrere verschiedene Stufen gleichzeitig
Superskalarprinzip: mehrere gleiche Einheiten gleichzeitig

Finn's Classifikation – explizite Parallelität

SISD (singel Instruktion Single Data) → 1 Leitwerk, 1 Rechenwerk
↳ klassische sequentielle Abarbeitung nach VNM

SIMD (Single Instruction multiple Data) → 1 Leitwerk, n Rechenwerk
↳ mehrere Daten auf einmal innerhalb des gleichen Befehls verarbeiten
⊕ höhere Rechenleistung durch Parallelität, weniger Aufwand für Programmabarbeitung

MISD (multiple Instruction Single Data) → n Leitwerk, 1 Rechenwerk (sinnlos)

MIMD (multiple Instruction multiple Data) → n Leitwerk, n Rechenwerk
↳ mehrere HardwareThreads, Ausführungseinheiten, Befehlszähler
→ jeder führt getrennte Instruktionsströme aus

mehrere Prozessoren ⊕ bessere Isolierung, getrennte Ressourcen & Binärcode

mehere Threads ⊕ gemeinsame Adressraum/Ressourcen/Netzanbindung verwalten
⊕ einfache Kommunikation zwischen einzelnen Threads