



V1

	Anzahl Bits	Werte	Alternativ-Name
Bit	1	0-1	
Nibble	4	0-15	
Byte	8	0-255	
Word	16	0-65.535	
Double Word	32	0-4.294.967.296	Word
Quad Word	64	0-1,8 * 10 ¹⁹	Double Word

Gängige Wortlängen



	Bits	Unsigned	Signed
Bit	1	0-1	n/a
Nibble	4	0-15	-8 - 7
Byte	8	0-255	-128 - 127
Word	16	0-65.535	-32768 - 32767
Double Word	32	0-4.294.967.295	-2147483648 - 2147483647
Quad Word	64	0-1,8 * 10 ¹⁹	-9,2 * 10 ¹⁸ - 9,2 * 10 ¹⁸

Darstellungen von Realen Zahlen

Binäre Darstellungen sind prinzipiell auf ganze Zahlen beschränkt

- War schon für negative Zahlen ein Problem
- Benötigen eine spezielle Darstellung von realen Zahlen

Option: Fest-Komma Darstellung

- Tupel aus zwei Zahlen
 - Zahl vor dem Komma
 - Zahl nach dem Komma
- Repräsentierung durch Verknüpfung der zwei Zahlen

42

31

= 42.31

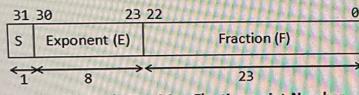
Kann numerisch problematisch sein

- Begrenzte Reichweite
- Unnötige Genauigkeit, die festgeschrieben ist

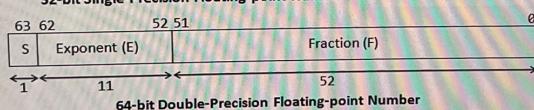
Gleitkommazahlen (Floating Point Numbers)

Für erhöhte Flexibilität: Aufteilung in Mantisse (auch Fraction) und Exponent (M*basis^E)

- Die meisten Systeme halten sich an IEEE-754
- Vorteil: breitere Reichweite
- Nachteil: ungleichmäßige Verteilung



Format: $(-1)^S * (1 + F) * 2^{E-BIAS}$



Code: "Hochsprache", kodiert als Text, nicht direkt ausführbar

Assembler-Sprache: Kompliziert hängt von Design ab, direkte Verarbeitung von Speicher und Argumenten (Ressourcen)

Instruction Set Architecture (ISA)

Datentypen

- Kodierung/Repräsentation

Speichermodell

- Was kann ich wo/wie speichern?

Sichtbarer Zustand des Prozessors

- Zugriff auf interne Daten
- Externe Schnittstellen

Verfügbare Befehle (Assembler)

- Ausführungssemantik

1.8 Conversion between Two Number Systems with Fractional Part

1. Separate the integral and the fractional parts.
2. For the integral part, divide by the target radix repeatedly, and collect the remainder in reverse order.
3. For the fractional part, multiply the fractional part by the target radix repeatedly, and collect the integer part.

Example 1: Decimal to Binary

Convert 18.6875D to binary

Integral Part = 18D

18/2 => quotient=9 remainder=0
9/2 => quotient=4 remainder=1
4/2 => quotient=2 remainder=0
2/2 => quotient=1 remainder=0
1/2 => quotient=0 remainder=1 (quotient=0 stop)

Hence, 18D = 10010B

Fractional Part = .6875D

.6875*2=1.375 => whole number is 1
.375*2=0.75 => whole number is 0
.75*2=1.5 => whole number is 1
.5*2=1.0 => whole number is 1

Hence .6875D = .1011B

Combine, 18.6875D = 10010.1011B

Example 2: Decimal to Hexadecimal

Convert 18.6875D to hexadecimal

Integral Part = 18D

18/16 => quotient=1 remainder=2
1/16 => quotient=0 remainder=1 (quotient=0 stop)
Hence, 18D = 12H

Fractional Part = .6875D

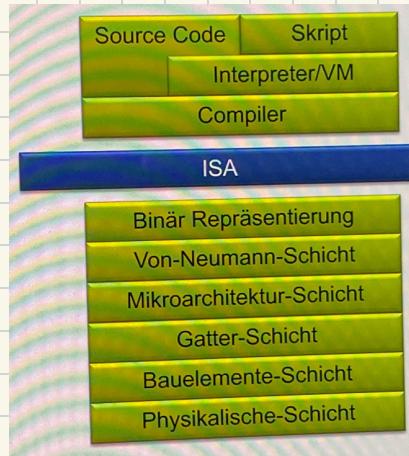
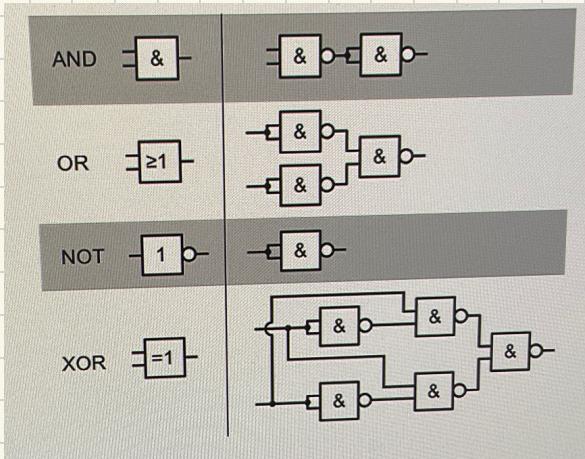
.6875*16=11.0 => whole number is 11D (BH)
Hence .6875D = .BH

Combine, 18.6875D = 12.BH

n	Minimum	Maximum
8	0	$(2^8)-1 (=255)$
16	0	$(2^{16})-1 (=65,535)$
32	0	$(2^{32})-1 (=4,294,967,295) (9+ digits)$
64	0	$(2^{64})-1 (=18,446,744,073,709,551,615) (19+ digits)$

n	minimum	maximum
8	$-(2^7) (= -128)$	$+(2^7)-1 (= +127)$
16	$-(2^{15}) (= -32,768)$	$+(2^{15})-1 (= +32,767)$
32	$-(2^{31}) (= -2,147,483,648)$	$+(2^{31})-1 (= +2,147,483,647) (9+ digits)$
64	$-(2^{63}) (= -9,223,372,036,854,775,808)$	$+(2^{63})-1 (= +9,223,372,036,854,775,807) (18+ digits)$

2's Complement Numbers



Anmerkungen zu Abstraktionsebenen

Abstraktionen erlauben den Austausch von Ebenen

- Das gleiche C Programm kann auf verschiedenen ISAs laufen
- Neue ISAs haben eine Chance (Beispiel: RISC-V)
- Für x86-64 bestehen zwei Implementierungen (AMD und Intel)
- Neue physikalische Schichten beeinflussen obere Schichten nicht

Unterschiedliche Ziele können zu verschiedenen Design führen

- Numerische Simulationen brauchen hohe Gleitkomma Leistung
- Transaktionen (Datenbanken) brauchen Integer und I/O Leistung
- Prozessoren für Handys müssen Energieeffizient sein

Abstraktionen sind nicht "in Stein gemeißelt"

- Datenrepräsentationen können je Architektur verschieden sein

Binär Dateien

- Reihen von 0/1, nicht (ohne weiteres) lesbar

```
>> hexdump count
00000000 457f 464c 0102 0001 0000 0000 0000 0000
00000010 0003 00f3 0101 0000 05b0 0000 0000 0000
00000020 0040 0000 0000 0000 1a70 0000 0000 0000
00000030 0005 0000 0040 0038 000a 0040 001c 001b
...
```

Was ist darin kodiert?

- Programm
- Startwerte für Daten
- Metadaten

Format und Inhalte hängen ab von

- Betriebssystem
- Hardware Plattform

Assemblersprache

Maschinen-spezifische Programmiersprache

- Niedrigste Abstraktionsbene
- Primitive Befehle
 - Elementare Berechnungsschritte
 - Keine komplexen Datenstrukturen
- 1:1 Übersetzung in Binärkode
- Manuelle Verwaltung von Ressourcen
 - Speicherelemente
 - Datenübertragungen

Oft nahe an der Implementierung

- Einzelne Befehle direkt implementiert
- Kodierung passend zur internen Architektur
- Festgelegte "Breite"
 - Daten und/oder Instruktionen
- Limitierungen möglich
- Oft spezifische Erweiterungen

ISAs = Instruction Set Architectures

Assembler-Sprache

- Daten-Breite
- Vorhandene Befehle und deren Kodierung
 - Verschiedene Operationen
 - Verschiedene Varianten
- Erweiterungen

Unterstützte Datenkodierungen

Systemeigenschaften

- Speicher
- Power/Energy Kontrolle

Boot-Up Prozess

- Konfigurationen und Konfigurationsabfrage

Komplexität der ISA: RISC vs. CISC

CISC: Complex Instruction Set Computer

- Komfortabel und relativ mächtig
- Vorteil: einfache Programmierbarkeit,
- Realisiert durch Mikroprogramme
- Nachteil: komplexe (langsame) Implementierung insbesondere Dekodierung
 - Evtl. viele Funktionen ungenutzt

RISC: Reduced Instruction Set Computer

- Minimalistisch, auf das absolut Notwendige beschränkt
- Vorteil: einfache, effiziente, schnelle Implementierung
- Realisiert durch feste Verdrahtung
- Nachteil: schwierigere Programmierbarkeit
 - Aber: das erledigt der Compiler (?)

Heute: Oft keine saubere Trennung mehr möglich

- CISC Architekturen haben einen RISC Kern
- Erweiterungen in RISC Architekturen oft CISC-like

Drei verschiedene Gesichtspunkte

- Compiler → Binärdatei → Disassembly
 - Nur minimale extra Information

Assemblersprache sind Plattform spezifisch

Oft gruppiert in Familien

- Intel und AMD's IA32 und x86_64
 - Gewachsen von 8bit bis 64bit
 - Intel vs. AMD: fast gleiche Sprache, aber komplett unabhängige Implementierung
 - Dominant in PCs
- ARM
 - Keine eigene Herstellung
 - Lizenzierung der Sprache
 - Von embedded bis HPC

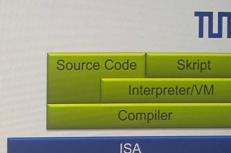
Beispiel: x86 Familie

- 8086 (16 bit) → 80286 → ... Pentium (32 bit)
 - ... Pentium/Netburst (64 bit)
 - ... „Core“-Architektur
- Neue Wortbreiten durch neue Ausführungsmodi
- Alle alten Ausführungsmodi weiterhin unterstützt
- ABER: jede Wortbreite definiert eigene ISA
- Lizenziert an AMD in den 80ern

IA-32: 32bit Variante



VL 2



Einige
Programmiersprache,
die ein Prozessor
verstehen kann
=
Zentrale Nutzer-
Schnittstelle

Befehlsformat: Codierung der Befehle

Ein Befehl besteht grundsätzlich immer aus

- der Spezifikation der Operation (**Opcode**)
- der **Spezifikation der Operanden**

Zwei mögliche Varianten:

- Einheitliches Befehlsformat
 - Alle Befehle haben eine **feste Länge**
 - Klassische Lösung für RISC
- Variables Befehlsformat
 - Befehle können verschiedene viele und verschieden-formatige Adressen enthalten
 - Typisch für **CISC**
 - Extremfall: DEC VAX: 1..53 Bytes pro Befehl
- In RISC-V sind (normalerweise) alle Befehle 32bit
 - Es gibt aber auch 16bit Instruktionen
 - Längere Instruktionen können definiert werden

Drei Hauptklassen von Befehlen:

- **Arithmetische und logische Operationen**
- **Datentransfer**
- **Steuerung des Programmablaufs**

Erzeugung von 32-BIT Konstanten

Größere Konstanten müssen in zwei Schritten erzeugt werden

- load upper immediate (**lui**) und **addi**
- **lui**: lädt eine Konstante in die oberen 20 Bits des Zielregisters und 0 in die unteren 12 Bits
- **addi**: addiert die unteren 12 Bits dazu

C Code

```
int a = 0xFEDC8765;
```

RISC-V Assembler

```
# s0 = a  
lui s0, 0xFEDC8    # s0 = 0xFEDC8000  
addi s0, s0, 0x765
```

Wichtig:

Bei **addi** 12-Bit-Immediate Vorzeichenerweiterung (sign extension) verwenden

- **Zusätzliche Bits mit Vorzeichenbit füllen**
- Ersetzt die unteren 12 Bits und die müssen zu den oberen 20 Bits passen

mul rd, rs1, rs2	Multiply	$rd := (rs1 * rs2)_{31:0}$
mulh rd, rs1, rs2	Multiply high signed * signed	$rd := (rs1 * rs2)_{63:32}$
mulhsu rd, rs1, rs2	Multiply high signed * unsigned	$rd := (rs1 * rs2)_{63:32}$
mulhu rd, rs1, rs2	Multiply high unsigned * unsigned	$rd := (rs1 * rs2)_{63:32}$

Bei RISC-V liegen alle Operanden in Registern

Register

- Kleine Anzahl von Speicherzellen
- Speichern je ein Wort in **"ISA Breite"**
- Innerhalb des Prozessors
- Schneller Zugriff

Eigenschaften

- Gibt es nur für bestimmte Befehle
 - Z.B. kein „**subi**“ Befehl
 - Nutzung negativer Konstanten
- Konstanten auf 12bit beschränkt
- Begrenzte Länge einer Instruktion

Zweckkomplement

T
logische
Befehle:
AND, OR, XOR...

Division und REST

Nur verfügbar mit der M-Erweiterung

- Ermitteln des Divisionsergebnisses

div rd, rs1, rs2	divide signed	$rd := rs1 / rs2$
divu rd, rs1, rs2	divide unsigned	$rd := rs1 / rs2$

- REM zum ermitteln des Rests der Division

rem rd, rs1, rs2	remainder signed	$rd := rs1 \% rs2$
remu rd, rs1, rs2	remainder unsigned	$rd := rs1 \% rs2$

sll rd, rs1, rs2	shift left logical	$rd := rs1 << rs2_{4:0}$
slli rd, rs1, uimm	shift left logical immediate	$rd := rs1 << uimm$
srl rd, rs1, rs2	shift right logical	$rd := rs1 >> rs2_{4:0}$
srlti rd, rs1, uimm	shift right logical immediate	$rd := rs1 >> uimm$
sra rd, rs1, rs2	shift right arithmetic	$rd := rs1 >>> rs2_{4:0}$
sraui rd, rs1, uimm	shift right arithmetic immediate	$rd := rs1 >>> uimm$

Datenspeicherung

Operanden nur aus Registern und Konstanten

- Register sind schnell, aber auch sehr begrenzt
- Nur als Zwischenspeicher gedacht

Zusätzlich: Hauptspeicher

- Große Speicherkapazität
- Ablage aller zu bearbeitenden Daten
- Nicht zum Verwechseln mit „Hintergrundspeicher“
 - Hauptspeicher meist flüchtig
 - Daten müssen in den Hauptspeicher geladen werden
- Meist als „Random Access Memory“ oder RAM
 - Man kann auf jedes gespeicherte Element zugreifen
 - Kein „vorspulen“ / „zurückspulen“

Konsequenz

- Daten müssen übertragen werden
- „Load“ und „Store“ Befehle → RISC-V ist eine „Load and Store Architecture“
- Andere ISAs erlauben Operanden direkt aus dem Hauptspeicher

Hauptspeicher Konzept

Hauptspeicher	
Adresse	Wert
000	
001	
010	
011	
100	
101	
110	
111	

Menge von Speicherzellen

- Logisch üblicherweise passend zur ISA Breite
- Jede Zelle speichert einen Wert

Jede Zelle hat eine Adresse

- Üblicherweise ab Adresse 0

Feste Adresslänge

- Im Beispiel: 3bit
- Legt maximale Speichergröße fest

„Random Access Memory“ (RAM)

- Immer Zugriff auf jede Zelle möglich

Speichergrößen

Dezimal			Binär		
Wert	Abk.	Metrik	Wert	Abk.	Metrik
1	B	byte	1	B	byte
$1000 = 10^3$	KB	kilobyte	$1024 = 2^{10}$	KiB	kibibyte
1000^2	MB	megabyte	1024^2	MiB	mebibyte
1000^3	GB	gigabyte	1024^3	GiB	gibibyte
1000^4	TB	terabyte	1024^4	TiB	tebibyte
1000^5	PB	petabyte	1024^5	PiB	pebibyte
1000^6	EB	exabyte	1024^6	EiB	extibyte
1000^7	ZB	zettabyte	1024^7	ZiB	zebibyte

Ein Adressraum mit folgender Adresslänge hat folgende Größe

- 16 bit: $2^{16} = 64$ KiB
- 32 bit: $2^{32} = 4$ GiB
- 64 bit: $2^{64} = 16$ EiB

Speicher/Zellenbreite

Speicherzellen sind üblicherweise mehrere Byte breit

Entspricht der Verarbeitungsgröße

- 16bit vs. 32bit vs. 64bit vs. 128bit Rechner
- Anzahl der Datenleitungen
- Warum? Effizienterer Transfer, Einfachere Logik

In der Praxis sind Speicher **byte-adressierbar**

- Adresse eines Datums bezieht sich immer auf **Bytegrenzen**
- **Unabhängig vom Datenformat**
- **Mehrere Adressen zeigen in eine Zelle**

Aber: **Unterschiede bei der Behandlung größerer Datenformate**

- Ausrichtung der Daten im Speicher
- Byte-Reihenfolge

Ausrichtung der Daten im Speicher

Definition: Ausrichtung von Daten („data alignment“)

- Ein Datum ist auf eine n-Byte-Grenze ausgerichtet, wenn seine Adresse A ein ganzzahliges Vielfaches von n ist:
 $A \bmod n = 0$

- Bei Ausrichtung auf „Datenbreite“ der ISA → immer nur ganze „Wort-Zellen“

Häufige Forderung

- Eine solche Ausrichtung ist oft gewünscht oder gefordert
 - Gewünscht: **schnellere Zugriffe**, aber **nicht ausgerichtete Zugriffe funktionieren** (x86)
 - Gefordert: nicht ausgerichtete Zugriffe **führen zu einem Fehler** (ARM bei manchen Typen)
 - Option: **getrennte Befehle für ausgerichtete und nicht ausgerichtete Zugriffe** (MIPS, RISC-V)
- Hintergrund: Speicher ist in Wörtern organisiert
 - Je nach Datenbusbreite liefert ein Zugriff z.B. 32 Bit oder 64 Bit
 - Bei nicht ausgerichteten Daten: Zugriff über Zellgrenzen hinweg
 - Mehrere Zugriffe
 - Bei ausgerichteten Operanden: minimale Anzahl von Speicherzugriffen

Byte-Reihenfolge

	24	16	8	0	
Beispiel: Ablage des Worts im Speicher	76	54	32	10	=0x76543210

Little Endian

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

Beispiel: Intel IA-32, typisch für PC

Big Endian

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

Beispiel: SPARC, Motorola, MIPS64

- Heute: Little Endian dominiert (x86 / x86_64)
- Die meisten anderen Architekturen bieten Wechselmöglichkeit (auch RISC-V)

Verallgemeinerung: Adressierungsarten

Speicher (HS/Regs)	
Adresse	Wert
000	
001	
010	
011	
100	
101	
110	
111	

Jeder Maschinenbefehl muss die **zu bearbeitenden Daten** (**Operanden**) spezifizieren

Operanden können Konstante oder Werte in Registern bzw. im Speicher sein (je nach ISA)

Adressierung

- Spezifikation der Operanden eines Befehls

Adressierungsart

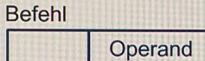
- Konkrete Methode zur Spezifikation eines Operanden

Welche Adressierungsarten vorhanden sind ist auch ein wesentliches Merkmal einer ISA

Adressierungsarten

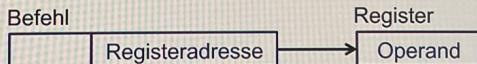
Unmittelbare Adressierung

- Der Operand steht als Konstante im Befehl
- RISC-V: mit bis zu 12bit möglich



Register-Adressierung

- Der Operand steht in einem Register
- Die Nummer des Registers (Registeradresse) steht im Befehl
- RISC-V: für alle Operanden



Dynamische Adressrechnung



Neben der direkten Angabe der Adresse erlauben Prozessoren verschiedene **Adressmodifikationen**

Effektive Adresse

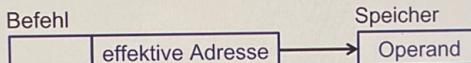
- Diese kann aus mehreren Teilen berechnet werden
 - Adresse in Registern oder im Hauptspeicher stehen
- Die Berechnung der effektiven Adresse erfolgt zur Laufzeit
 - Während der Befehlausführung

Warum? Flexibilität

- Ein Rechenprozess muss die Adressen von Operanden, Ergebnissen und Sprungzielen zur Laufzeit ändern können
 - Elemente mit berechnetem Index in Feldern
 - Dynamische Datenstrukturen (Listen, Bäume, dynamische Felder, ...)
- Erleichtert die Erzeugung von verschiebbaren Code- und Datenbereichen

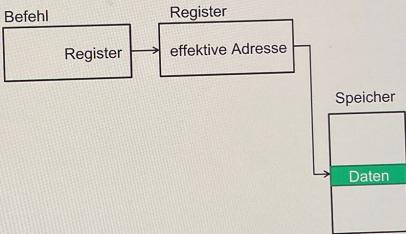
Speicheradressierung: Direkte Adressierung

- Der Operand steht im Speicher
- Die effektive Adresse steht (als Konstante) im Befehl
- RISC-V: nicht möglich



Registerindirekte Adressierung

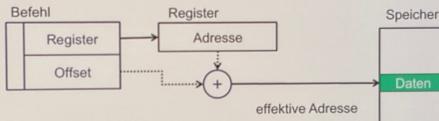
- Die Adresse steht in einem Register
- RISC-V: Möglich mit Offset 0



Registerindirekt mit Displacement

Zu der im Register stehenden Adresse wird ein konstanter Wert (Offset) addiert

- Offset kann positiv oder negativ sein
- Auch als **Basisrelative Adressierung** bezeichnet.
- RISC-V: Möglich, mit 12bit Offset



Verwendungszweck

- Zugriff auf Datenbereiche mit zur Übersetzungszeit bekannter Struktur
- Zugriff auf Elemente eines Feldes mit festem Index**
- Bezeichnung: **indizierte Adressierung**

Indizierte Adressierung mit Skalierungsfaktor

- RISC-V: nicht vorgesehen (standardmäßig)

Diskussion Adressierungsarten

Viele Adressierungsarten möglich

- Jede hat mögliche Verwendungszwecke
- Wäre als Teil einer ISA sinnvoll

ABER: Jede erzeugt Komplexität

- Mehrere Zugriffe
- Zusätzliche Arithmetik
- Mehr Befehle

Oft nur zur Programmierbarkeit

- Einfachere Programme
- Kann aber durch Programmsegmente implementiert werden

Steuerung des Programmablaufs: Sprungbefehle

Unbedingter Sprung

- Programm wird an anderer Stelle fortgesetzt
 - Implementierung: Neusetzen des PC
- j imm: Jump: Programm wird an PC+imm fortgesetzt (kurzer relativer Sprung)
j r reg,imm: Jump Register: Programm wird an Register Wert + imm fortgesetzt



Bedingter Sprung

- Sprung nur wenn eine Bedingung erfüllt ist
- Bedingung muss kodiert werden
- Notwendig für Schleifen (Sprung zurück wenn Schleifenende noch nicht erreicht)

Format: bxx r1,r2,imm

Falls Bedingung xx erfüllt ist, wird das Programm an PC+imm fortgeführt
Ansonsten beim nächsten Befehl

Bedingungen: branch if equal (beq)

r1 = r2

branch if not equal (bne)

r1 ≠ r2

branch if less than (blt)

r1 < r2

branch if greater than or equal (bge)

r1 ≥ r2

ZÜ 1

- Strings werden in Zahlen umgewandelt: Oft mit NULL terminiert (C-Strings)
- aber auch explizite Größenangabe zu Beginn möglich (Pascal-Strings)

Variante: Das Sign-Bit

- Verwende höchstwertiges Bit um Vorzeichen darzustellen
- Beispiel:
 - Darstellung einer Zahl mit 8 Bits
 - $+7_{10} = 0000_0111_2$
 - $-7_{10} = 1000_0111_2$
- Vorteile:
 - trivial zu realisieren
 - Vorzeichen durch erstes Bit repräsentiert → direkt ablesbar
- Nachteile:
 - Spezialbehandlung für Addition und Subtraktion
 - Reduktion der Anzahl an darstellbaren Werten (zwei Darstellungen für die Zahl 0)

Variante: Das Zweierkomplement

- Schritt 1: Einerkomplement
 - Bitkomplement: Invertierung jedes einzelnen Bits
 - Beispiel: Negation von Zahl 7_{10}
 $0000\ 0111_2 \rightarrow 1111\ 1000_2$
- Schritt 2: Addition von „1“
 - $-7 = 1111\ 1001_2$
- Vorteile:
 - siehe folgende Folien
 - Vorzeichen durch erstes Bit repräsentiert → direkt ablesbar, aber Zahlenwert nicht
- Nachteil:
 - Zahl nicht direkt „ablesbar“ (außerdem: Breite muss bekannt sein)
 - kein Gegenstück für kleinste negative Zahl im positiven Zahlenbereich

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

Instruction Set Architecture (ISA)

LU 2

- Abstraktes Modell eines Computers
- Definiert viele Dinge, unter anderem:
 - Befehlssatz
 - Bitbreite
 - Registeranzahl und deren Initialisierung
 - Menge der Instruktionen
 - Startadresse der Befehlsabarbeitung
 - Datentypen: Kodierung/Repräsentation
 - Und vieles mehr

Komplexität der ISA: RISC vs. CISC

- CISC:
 - Komfortabel und relativ mächtige Instruktionsmenge
 - Vorteil: einfacher programmierbar
 - Realisiert durch Mikroprogramme (mehrere Taktzyklen pro Instruktion)
 - Wenige Instruktionen pro Programm
- RISC:
 - Minimalistisch
 - Vorteil: einfache, effiziente, schnelle Implementierung
 - Realisiert durch feste Verdrahtung (1 Instruktion pro Taktzyklus)
 - Viele Instruktionen pro Programm

• Nützliche „Hacks“:

- Negation über `sub rd, x0, rs`
- Register kopieren: `addi rd, rs, 0`

- Das 0. Bit in t0 auf 1 setzen: `ori t0, t0, 1`
- Das 5. Bit in s2 invertieren: `xori s2, s2, 32`
- Invertierung aller Bits in a7: `xori a7, a7, -1`
- a0 auf 1 setzen wenn 10. Bit in a0 Eins ist, sonst Nullen:
`srl a0, a0, 10; andi a0, a0, 1`

VL 3

Aufrufkonvention

Soviel wie möglich via Register

- Eingabedaten

- Rückgabewert

Gezielte Aufgaben

- a0-a7: Eingaben
- a0-a1: Ergebnisse
- ra: Rücksprungadresse

Caller-saved

- Aufrufer/Caller muss Werte selbst sichern, dürfen von Callee verändert werden

Callee-saved

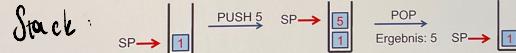
- Aufgerufene Funktion darf Werte nicht verändern oder muss sie wieder herstellen

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-7	a0-a2	Temporaries	Caller
x8	a0/a2p	Save register/frame pointer	Callee
x9	—	Saved registers	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	—
f0-7	ft0-7	FP temporaries	Caller
f8-9	fa0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments/return values	Caller
f12-17	ft2-7	FP temporaries	Caller
f18-27	ft2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	—

From: <https://scv.org/technical/specifications/>

14

LIFO Prinzip: Last In First Out



Gegenteil:

FIFO Prinzip: First In First Out



Warteschlange/Queue

Designentscheidung: Wo stehen die Operanden bzw. das Ergebnis?

- Hauptspeicher (mit verschiedenen Adressierungsarten) oder Register?
- Hauptspeicher: evtl. lange Adresse, hohe Zugriffszeit

Designentscheidung: Welche (wie viele) Operanden werden explizit spezifiziert?

- Explizit: Operand (bzw. Zielort des Ergebnisses) wird im Befehl frei festgelegt
- Implizit: Adresse ist vom Befehl fest vorgegeben
- Überdeckte Adressierung
 - Zieladresse ist identisch mit einer der Quelladressen
 - D.h. ein Operand wird mit dem Ergebnis überschrieben

Zweiadressform

Das Ergebnis überschreibt ersten Operanden

- D.h. Befehle sind von der Art $x = y + z$
- Reduziert benötigte Bits im Instruktionsformat

Meist: Register-Speicher-Modell

- Nur einer der Operanden kann im Hauptspeicher stehen
- Ergebnis kann Register- oder Speicheroperanden überschreiben
 - Reg. = Reg. + z
 - $x = x + \text{Reg.}$

Beispiel: Intel IA-32

Ausführungsmodell: Registermaschine

Übersicht Adressformen

Operationen benötigen meist 3 Operanden: $x = y + z$

Frage: wie viele sind in einem Befehl kodiert?

- Null-Adressform: alle Operanden implizit, typischerweise im Keller
Kellermaschine
- Ein-Adressform: eine Eingabe und Ausgabe im Akkumulator
Akkumulatormaschine
- Zwei-Adressform: erster Operand wird überschreiben
Registermaschine
- Drei-Adressform: alle Operanden separat adressierbar
Speicher-Speicher-Modell oder **Register-Register-Modell**

Dreiadressform

Operanden und Ergebnis dürfen nur in Registern stehen

- Ausnahme: spezielle Befehle zum Laden und Speichern von Registern (Load/Store-Architektur)
- Beispiele: fast alle RISC-Prozessoren
- RISC-V: addw a5, a5, a4 (benötigt Kodierung von 3x5bit)

Register-Register-Modell

Operanden und Ergebnis werden explizit adressiert

- Operanden und Ergebnis dürfen (auch) im Hauptspeicher stehen
- Beispiele: Frühe CISC-Prozessoren (VAX)
 - Nachteile
 - Befehle benötigen sehr viel Speicherplatz (VAX: 1..53 Bytes pro Befehl)
 - Viele Speicherzugriffe
 - Komplexe Dekodierung und Ausführung

Speicher-Speicher-Modell

Einadressform

Ein Operand ist immer ein ausgezeichnetes Register (Akkumulator)

- Ergebnis wird wieder im Akkumulator (AC) abgelegt
- D.h. Befehle sind von der Art
 - $AC = AC + z$
 - $AC = y$
 - $x = AC$
- Ketten von arithmetischen Operationen
- Geringe Komplexität des Instruktionsformats

Beispiele:

- Erste Systeme mit Programmsteuerung
- Frühe 8-Bit Prozessoren

Nachteile:

- Unflexibel
- Häufiges Laden / Speichern des Akkumulators erforderlich

Nulladressform

Keine explizite Operandenspezifikation

Operanden: oberste Kellerelemente

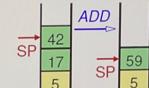
- Werden aus dem Keller entfernt
- Ergebnis wird wieder auf den Keller gelegt
- Gut geeignet zur Auswertung von Ausdrücken

Beispiele:

- Java Virtual Machine
- IA-32 Gleitkommabefehle

- Diskussion
- Pro: Einfaches Modell Instruktionsformat
 - Con: Viele extra Operationen (mit Speicherzugriff)

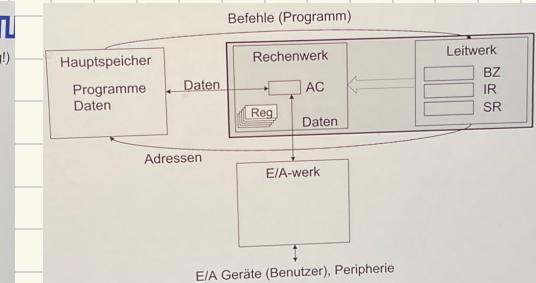
Ausführungsmodell: Kellermaschine



Von-Neumann Architekturkonzept

- Die Struktur des Rechners ist unabhängig vom bearbeiteten Problem (Programmsteuerung)
- Rechner besteht aus vier Werken
 - Haupt- bzw. Arbeitsspeicher (speichert Programme und Daten)
 - Leitwerk (arbeitet Befehlszyklus ab)
 - Rechenwerk (führt arithmetische Operationen aus)
 - Ein/Ausgabewerk inklusive Sekundärspeicher (kommuniziert mit Umgebung, inkl. „Langfrist“-Speicher)

- Der Hauptspeicher ist in Zellen gleicher Größe geteilt, die durch fortlaufende Nummern (Adressen) bezeichnet werden
- Programm und Daten stehen in demselben Speicher (Hauptspeicher) und können durch die Maschine verändert werden
- Die Maschine benutzt BinärCodes; Zahlen werden dual dargestellt
- Das Programm besteht aus einer Folge von Befehlen (Instruktionen)
 - i.a. nacheinander gespeichert (aufsteigende Adressen)
 - i.a. nacheinander ausgeführt (sequentiell)
- Von der Folge kann durch bedingte oder unbedingte Sprungbefehle abweichen werden (Programmfortsetzung aus einer anderen Zelle)



Zusammenfassung / Unterschiede von ISAs

Alle ISAs haben sehr ähnliche Prinzipien

- Basierend auf von Neumann Prinzipien
- Arithmetische, Speicher und Sprungbefehle
- Stack zur Speicherung von lokalen Daten

Unterschiede

- Anzahl und Zweck von Registern
- Addressierungsarten und Ausführungsmodelle
- Aufrufkonventionen
- Bedingungen / Existenz von Flags

(Unter-) Programm Struktur hängt stark vom Compiler ab

- Bei gleichem Compiler oft „gleiche Tricks“ → z.B. Abkürzungen
- Stack- und Registerverwaltung
- Ohne Optimierung oft viele unnötige Transferbefehle

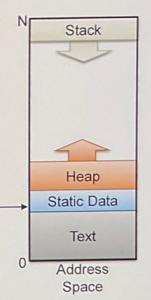
wichtige Unterschiede

- U-Typ Befehle (z.B. lui) benutzen 20-Bit lange immediates
- I-Typ Befehle (z.B. addi) benutzen 12-Bit lange immediates
- jeweils erkennbar an ui bzw. nur i im Namen

Speicherlayout

wichtige sog. Sections (Speicherbereiche):

- .text
 - beinhaltet ausführbaren Code
- .data
 - beinhaltet globale, schreibbare, initialisierte Daten
- .rodata
 - beinhaltet globale, nicht-schreibbare, initialisierte Daten
- .bss (=block starting symbol)
 - beinhaltet globale, schreibbare, nicht-initialisierte Daten
 - effektiv nur ein Marker für „hier kommen n Bytes“



.data-Section in QtRvSim

```
.org 0x400
.data
    .word 1, 2, 3, 4          // boilerplate
                            // Start der .data Section
    .word 5, 6, 7, 8          // Allotiert 4 Wörter (= 32 Bit) nacheinander (als Array)
                            // ob und initialisiert diese mit 1,2,3,4
    .asciz "I <3 ERA!"      // asciiz = ASCII-Zero -> fügt automatisch NULL-Byte an
                            // Allotiert 10 Bytes nacheinander und initialisiert
                            // diese mit "I <3 ERA!" zzgl. NULL-Byte
    .text_1: .asciz "\0"       // Erinnerung: Das Ende von Strings wird mit einem
                            // NULL-Byte markiert
```

YU3

VL 4

Time Sharing

- Möglichkeit Programmausführung zu unterbrechen
- Transparentes Umschalten auf ein anderes Programm
- Spätere Wiederaufnahme der Ausführung

Herausforderungen in Mehrbenutzersystemen

Time Sharing wird im Betriebssystem implementiert

- Automatische Unterbrechungen → „Time Slices“
- Transparentes Umschalten

Scheduler

- Welches Programm läuft als nächstes?
- Ziel: Fairness

Voraussetzung für Mehrbenutzersysteme

- Mögliche Unterbrechung von Programmen
- **Scheduling unabhängig von Benutzerprogrammen**
- Schutz der Daten zwischen Programmen
- Kontrolle von Ressourcen (z.B. Stacks and Heaps)

Benötigt Unterstützung aus der Hardware bzw. Architektur

- Unterbrechungen / Fehlerindikationen
- Privilegierter Modus, in dem nur das Betriebssystem arbeitet
- Speicherverwaltung

Unterbrechungen („Interrupts“)

Erlauben dem Prozessor auf **externe, asynchrone Ereignisse** zu reagieren

- Spezielles Eingangssignal: Unterbrechungsanforderung
- Möglichkeit der Maskierung oder nicht (Ignorierung)

Unterschiede: Interrupts vs. Exceptions/Traps

Beide führen zu Unterbrechungen

- Abbruch des gegenwärtigen Programmes
- Abspeichern des Systemzustandes (v.a. BZ) **Befehlszähler**
- Betriebssystem übernimmt Kontrolle
- Nach Abschluss Rückgabe ans Programm

Interrupts sind **asynchron**

- Ausgelöst von **externen Ereignissen**
- Können nach beliebigen Befehlen auftreten
- Interrupts sind im Regelfall "normaler Betrieb"

Ausnahmen/Exceptions/Traps sind **synchron**

- Ausgelöst durch Abarbeitung eines Befehles
- Treten **nach diesem Befehl auf**
- Signalisieren im Regelfall **einen Fehler in der Ausführung**

Einfluß darauf was die Behandlungsroutine machen kann/darf

Systemmodus

- Voller (=privilegierter) Zugriff auf alle Rechnerkomponenten
- Kontrolle von Unterbrechungen
- Für das Betriebssystem

Benutzermodus

- Eingeschränkter Zugriff
- Keine privilegierten Befehle wie z.B. Ein-/Auszug
- Kein Zugriff auf Konfigurationsregister

Interrupts, Exceptions und Traps oft gemeinsam oder ähnlich implementiert

- Ein Mechanismus zum Eintritt in den privilegierten Modus
- Gleicher Einsprung oder einheitliche Einsprungtabelle

RISC-V Umsetzung von "Traps" / "System Calls"

ecall Befehl (Environment Call)
ecall (ohne Operanden)

Ausnahmen („Exceptions“)

Die Ausführung einer unzulässigen Operation führt zu einer Ausnahme

- Englisch: „Exception“ oder „SW Interrupt“
- Sprung an vordefinierte Adresse im Betriebssystem („Exception Handler“)

Dynamische Speicherverwaltung

- Auch "Heap" genannt
- Kann angefragt oder freigegeben werden
- Wächst nach oben

Keller/Stack wächst nach unten

- Optimale Platzausnutzung
- Wenn sie sich treffen -> Fehler

Einführung Mehrere Adressräume

Beliebig viele "Virtuelle" Adressräume

- Jeder Prozess bekommt seinen eigenen
- Jeder Prozess sieht nur seinen eigenen
- Jeder Prozess läuft isoliert

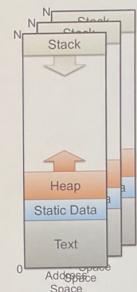
Betriebssystem schaltet Adressräume um

- Während des Schedulings
- Adressraum wird Teil des Prozesses
- Neben BZ und Register

ABER: weiterhin nur ein physikalischer Speicher

Speicherzugriffe in einem Prozess

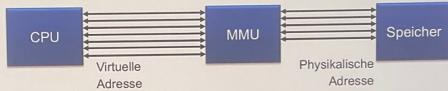
- Effektive Adressen sind virtuelle Adressen
- Prozessor rechnet diese um
- Zugriff im Speicher über physikalische Adressen



Übersetzung Virtueller Adressen

Adressen müssen übersetzt werden

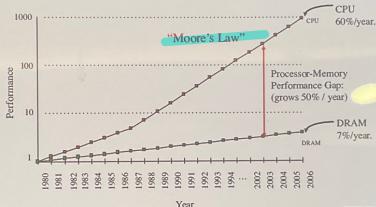
- Bei jedem Zugriff
- In Hardware



"Memory Management Unit" (MMU)

- Übersetzt jede Adresse, die vom Speicher angefordert wird
- Zugriff auf physikalische Adresse im Hauptspeicher
 - U.U. verschiedene Adresslängen
- (Benutzer)Programme haben keinen direkten Speicherzugriff mehr
- Übersetzung vom Betriebssystem kontrolliert

Trend: Speicher vs. CPU Leistung



Lokalitätsprinzip

Zeitliche Lokalität von Speicherzugriffen (temporal locality)

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

Räumliche Lokalität von Speicherzugriffen (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

Aufteilung einer Adresse:

Tag	Index	Offset
-----	-------	--------

Offset: untere Bits der Adresse als Index in die Zeile

Index: bestimmt die Cachezeile

Tag: beschreibt die Daten die abgelegt werden

Arten von Caches:

- Direct mapped: Index beschreibt eindeutige Zeile
- Fully associative: Keine Index-bits, jede Cachezeile möglich
- N-way associative: Index beschreibt Menge aus N Zeilen
Konsequenz: $\log_2(N)$ Index Bits

Assoziative Caches

Direct Mapped Caches

- Jede Adresse kann auf genau eine Cache Zeile abgebildet werden
- Große Flexibilität für Hardware, Kleinstes für ProgrammiererIn, aber auch geringste Komplexität

Voll-assoziativer Cache („Fully Associative Cache“)

- Jede Adresse kann auf jede Cache Zeile abgebildet werden
- Nur ein „Cache Set“, keine Index bits
- Große Flexibilität für ProgrammiererIn, aber auch größte Komplexität (z.B. Anzahl von Vergleichen)

Kompromiss: Mengenassoziativer Cache („Set-Associative Cache“)

- Jede Adresse kann auf einen Teil der Cache Zeilen abgebildet werden
- Mehrere Cache-Sets / Reduzierte Zahl von Index bits
- Anzahl der Zeilen pro Cache Set = Assoziativität des Caches
 - Beispiel: 4-way associative = 4 Zeilen pro Cache Set
- Mehr Flexibilität
- Handhabbare Komplexität (z.B. Anzahl von Vergleichen)

Klassifikation von „Cache Misses“

Compulsory (cold) Misses

- Erster Zugriff auf eine Adresse
- Treten auch in unendlich großem Cache auf
- Nicht zu vermeiden

Conflict Misses

- Speicherblock wurde verdrängt, da ein anderer Block auf das gleiche Cache Set abgebildet wurde
- Treten nur in mengenassoziativem oder direkt abbildendem Cache auf
- Kann (manchmal) durch Programmiertricks umgangen werden

Capacity Misses

- Der Speicherblock wäre auch verdrängt worden, wenn der Cache vollassoziativ wäre
- Treten auch in voll-assoziativem Cache auf
- Nur zu umgehen durch andere Zugriffsmuster

Oft: LRU (Least Recently Used) oder LFU (Least Frequently Used)

- Gut für temporale Lokalität
- Praxis: oft kombiniert für vereinfachter Hardware

Inklusive / exklusive Cache-Hierarchie

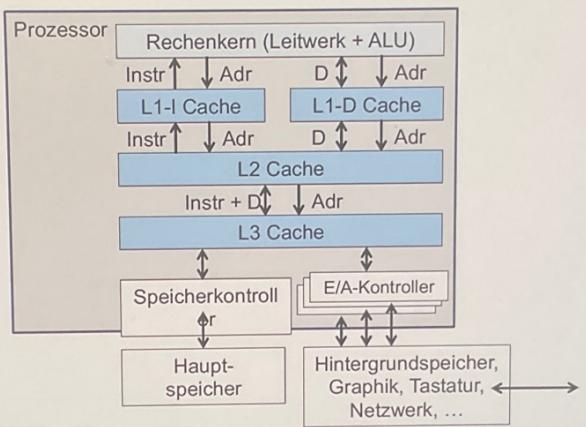
IU11

Inklusiv

- Alle in einem **kleineren Cache** enthaltenen Speicherblöcke sind **auch im darunterliegenden größeren Cache** enthalten
- Bei **Zugriff auf Hauptspeicher** werden **Kopien in allen Ebenen** abgelegt
- Anzahl der Speicherblöcke, die in der gesamten Cache-Hierarchie vorhanden sein können, hängt von der **Größe des größten Caches** ab

Exklusiv

- Jeder Speicherblock ist **höchstens einmal als Kopie in der Cachehierarchie** enthalten
- Hauptspeicherzugriff lädt in L1, von wo Kopien in L2 verdrängt werden, falls L1 voll. Ebenfalls Verdrängung von L2 in L3
- Anzahl der Speicherblöcke in der Cache-Hierarchie hängt von der **Summe der Cachegrößen in der Hierarchie** ab



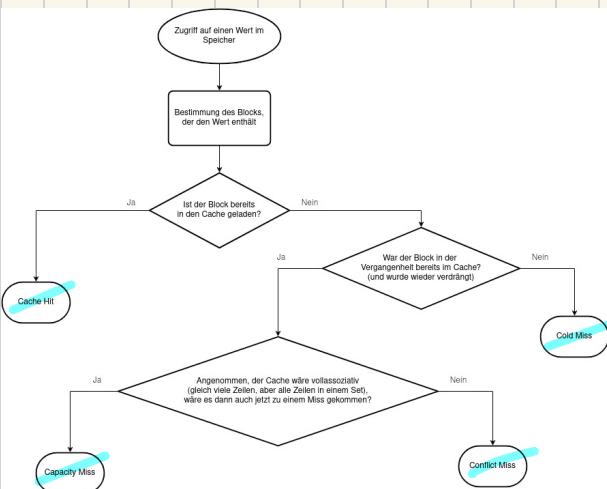
L1 Instruktionscache vs. Datencache

Trennung von Instruktionen und Daten

- Transparent für BenutzerInnen
- Prozessor kümmert sich um Kohärenz** (z.B. schreiben von Opcodes in den Speicher)
- Trotzdem: klare Abwendung von "von Neumann"

Gründe:

- Andere Zugriffsmuster**
 - Instruktionen (dank von Neumann) mehr sequentiell
- Verbesserte Cache-Strukturen**
 - Oft Einsatz von sog. "Trace-Caches" als L1 Cache
- Vorverarbeitung**
 - Cache kann u.U. dekodierte Instruktionen abspeichern
- Lässt mehr Platz für Daten**
 - Vermeidet einen einzelnen größeren Cache



Z04

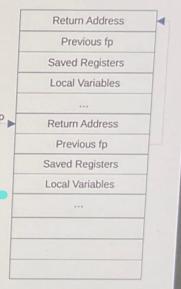
- Caller:
 - Argumente in a0-a7 übergeben
 - Alle t- und a-Register sichern
 - ra-Register sichern
 - Unterprogrammaufruf: `jal ra, callee`
 - Register wiederherstellen
 - Ergebnis ist in a0/a1

Calling Convention - Register

- Unterteilung in Caller und Callee saved Register
- Caller saved: **t-Register, a-Register, Rücksprungadresse**
- Callee saved: **s-Register, Stack Pointer**
- **Vorteile:**
 - Zu alle Callee saved: **Nicht alle Register müssen vor der Verwendung gesichert werden**
 - Zu alle Caller saved: **Nicht alle Register müssen vor Unterprogrammaufruf gesichert werden**

Frame Pointer

- s0 hat auch den Namen **fp (Frame Pointer)**
- markiert **Anfang des Stack Frames des Programms**
- Gibt fixe Basisadresse auf Stack z.B. falls Stackpointer sich ändert
- Nachteil: **alter Frame Pointer muss gesichert und wieder hergestellt werden**
- Kann für Backtracing verwendet werden (Linked List)



- Callee
 - **s-Register sichern** falls sie verändert werden
 - Ergebnis nach a0/a1 schreiben
 - **Register herstellen**
 - Return: jr ra

- Datentypen **kleiner als 32-Bit** werden auf 32-Bit sign-extended
- 32-Bit Werte (oder kleiner) in **a-Registern, aufsteigend sortiert**
- 64-Bit Werte in **zwei a-Registern pro Wert, aufsteigend sortiert, untere Bit in niedrigerem Register**
- Größere Datentypen (>64 Bit) werden **als Pointer** übergeben
- Weitere Parameter werden **über den Stack übergeben**

- Vorteile:
 - Viele Programme haben rekursive Struktur, dadurch häufig **Intuitiv**
- Nachteile:
 - Größerer **Speicherbedarf**
 - **Mehr Sprünge**, dadurch häufig weniger performant

VL 5

XNOR \rightarrow opposite of XOR		
Aquivalenz-Verknüpfung		
□ Gesamtaussage ist dann wahr, wenn beide Aussagen den gleichen Wahrheitswert haben	$x \neq x$	XNOR
□ Symbol: \equiv	$\begin{array}{ c c } \hline x & y \\ \hline 0 & 0 \\ 1 & 1 \\ \hline \end{array}$	XNOR
□ Beispiel: $a \equiv b$	$\begin{array}{ c c } \hline x & y \\ \hline 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ \hline \end{array}$	XNOR

Antivalenz		
□ Gesamtaussage ist wahr, wenn genau eine Teilaussage wahr ist		
□ Symbol: \neq	$\begin{array}{ c c } \hline x & y \\ \hline 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ \hline \end{array}$	XOR
□ Beispiel: $a \neq b$	$\begin{array}{ c c } \hline x & y \\ \hline 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ \hline \end{array}$	XOR

Implikation		
□ Gesamtaussage ist nur dann falsch, wenn erste Teilaussage wahr und zweite Teilaussage falsch ist		
□ Entspricht in etwa der umgangssprachlichen Wenn-dann Formulierung (ist aber exakter, z.B. wenn die erste Teilaussage falsch ist)	$\begin{array}{ c c } \hline x & y \\ \hline 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ \hline \end{array}$	XOR
□ Symbol: \Rightarrow	$\begin{array}{ c c } \hline x & y \\ \hline 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ \hline \end{array}$	XOR

Tautologie		
□ Aussage, die immer wahr ist		
□ Beispiele:	$a \vee \neg a$	
○	$(a \Rightarrow b) \vee (b \Rightarrow a)$	
○	$(a \vee b) \vee (\neg a \vee b)$	
Kontradiktion		
□ Aussage, die immer falsch ist		
□ Beispiel:	$a \wedge \neg a$	

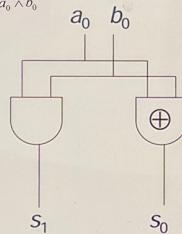
Gängige Kostenmaße		
□ Anzahl der Gatter (Größe)		
□ Tiefe, d.h. Zahl der Gatter auf dem längsten Pfad von einem primären Eingang zu einem primären Ausgang (Geschwindigkeit)		

Schaltkreis eines Halbaddierers

Folglich:

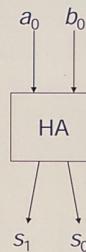
$$ha_0 = a_0 \oplus b_0$$

$$ha_1 = a_0 \wedge b_0$$

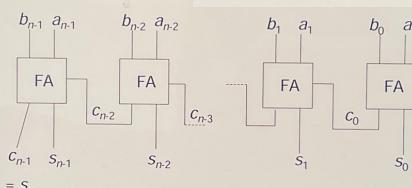


Kosten und Tiefe eines HA:

$$C(HA) = 2, \quad \text{depth}(HA) = 1$$



Aufbau eines Carry Ripple Addierers



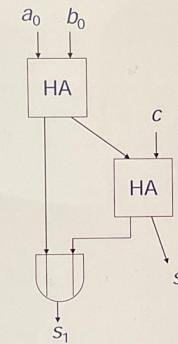
Kosten eines CR_n:

$$C(CR_n) = n \cdot C(HA) = 5n$$

Tiefe eines CR_n:

$$\text{depth}(CR_n) = 3 + 2(n-1)$$

Schaltkreis eines Volladdierers

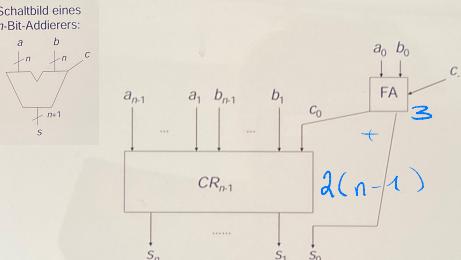


$$\begin{aligned} fa_0 &= a_0 \oplus b_0 \oplus c = ha_0(c, ha_0(a_0, b_0)) \\ fa_1 &= a_0 \wedge b_0 \vee c \wedge (a_0 \oplus b_0) \\ &= ha_1(c, ha_0(a_0, b_0)) \end{aligned}$$

Kosten und Tiefe eines FA:

$$C(FA) = 5, \quad \text{depth}(FA) = 3$$

Schaltbild des n-Carry Ripple Addierers (CR_n)



Idempotenz

$$a + a = a$$

$$a * a = a$$

Absorption

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

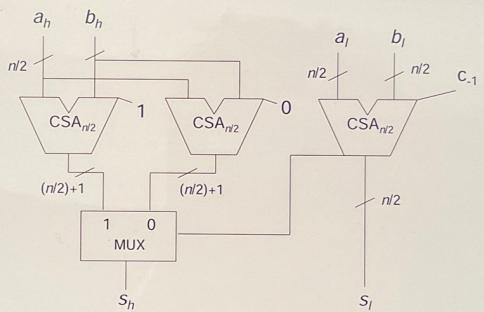
$$(a * b) + a = a$$

De Morgansche Regeln

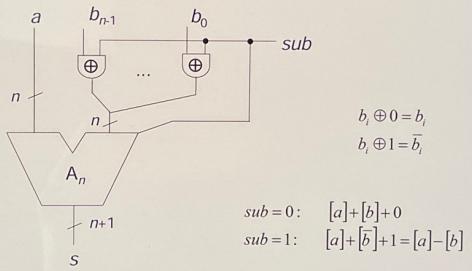
$$(\neg a * \neg b) = \neg (a + b)$$

$$(\neg a + \neg b) = \neg (a * b)$$

Schaltbild des CSA_n



Schaltbild für einen Addierer/Subtrahierer



$$sub = 0 : [a] + [b] + 0$$

$$sub = 1 : [a] + [\bar{b}] + 1 = [a] - [b]$$

VL 6

Name	Funktion	Symbol in Schaltplan		
		IEC 6017-12 : 1997 & ANSI/IEEE Std 91/91a-1991	DIN 40700 (vor 1976)	
Und-Gatter (AND)	$Y = A \wedge B$ $Y = A \cdot B$			
Oder-Gatter (OR)	$Y = A \vee B$ $Y = A + B$			
Nicht-Gatter (NOT)	$Y = \bar{A}$ $Y = \neg A$			
NAND-Gatter (NICHT UND) (NOT AND)	$Y = \bar{A} \wedge \bar{B}$ $Y = \bar{A} \wedge \bar{B}$			
NOR-Gatter (NICHT ODER) (NOT OR)	$Y = \bar{A} \vee \bar{B}$ $Y = \bar{A} \vee \bar{B}$			
XOR-Gatter (Exklusiv-ODER, Antivalenz) (EXCLUSIVE OR)	$Y = A \vee B$ $Y = \bar{A} \oplus B$			
XNOR-Gatter (Exklusiv-Nicht-ODER, Äquivalenz) (EXCLUSIVE NOT OR)	$Y = \bar{A} \vee \bar{B}$ $Y = \bar{A} \oplus \bar{B}$			

Paralleler Multiplizierer #1

■ Multiplikation zweier Bits

$$B_0 * A_0 = B_0 A_0$$

→ Und-Verknüpfung

■ Multiplikation einer mehrstelligen Zahl mit einem Bit

$$(B_1, B_0) * A_0 = (B_1 A_0, B_0 A_0)$$

→ Und-Verknüpfung der Stellen mit dem Bit

■ Multiplikation zweier mehrstelliger Zahlen

$$(B_1, B_0) * (A_1, A_0)$$

$$= (B_1, B_0) * A_0 + (B_1, B_0) * A_1 * 2$$

cause of the shift left

Beispiel: 1011 * 0101



Schnelle Addition von n Partialprodukten der Länge $2n$.

Mit CLAs lösbar mit Kosten $O(n^2)$,

Tiefe $O(n \log(n))$ bei linearem Aufsummieren der Partialprodukte $((\dots(pp_0+pp_1)+pp_2)+\dots)+pp_{n-1})$.

Tiefe $O(\log(n))$ bei baumartigem Zusammenfassen der Partialprodukte.

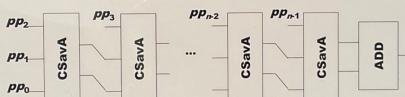
1. Serielle Lösung:

- Hintereinanderschalten von $n-2$ CSava-Addierern der Länge $2n$
→ Fasse n Partialprodukte zu zwei $2n$ -Bit-Worten zusammen
- Addiere die $2n$ -Bit-Worte mit CLA
- siehe Abb. der Addierstufe
- Kosten $O(n^2)$, Tiefe $O(n)$

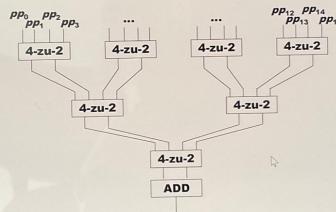
2. Baumartige Lösung:

- Neue Grundzelle zur Reduktion von vier $2n$ -Bit Eingabeworten zu zwei Ausgabeworten, bestehend aus zwei CSavAs (siehe Abb. zur Reduktionszelle)
- Baumartiges Zusammenfassen der Partialprodukte mit 4-zu-2-Bausteinen zu zwei $2n$ -Bit-Worten
- Addiere die $2n$ -Bit-Worte mit CLA
- siehe Abb. der Addierstufe mit log. Zeit
- Kosten $O(n)$, Tiefe $O(\log(n))$

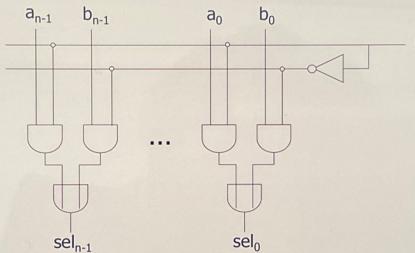
Addierstufe im Multiplizierer



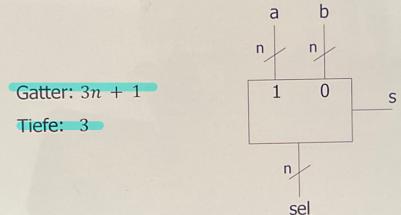
Addierstufe des log-Zeit-Multiplizierers für 16 Bit



Realisierung eines n-Bit Multiplexer



Kosten eines n-Bit MUX



Rückkopplungen

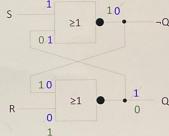
- Was passiert, wenn Ausgang einer Schaltung auf Eingang zurückwirkt?



- Annahme: a und y am Anfang 0
 - Solange a auf 0 bleibt, keine Änderung an y
 - a wird 1, dann wird y auch 1 und bleibt so
 - Schaltung hat „Gedächtnis“ *memory*

RS-Flipflop

- Schaltung zum Speichern von Information
- Eingänge zum Setzen (Set) und Löschen (Reset)



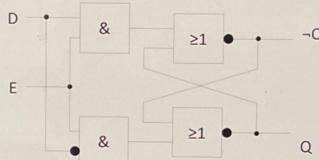
R	S	$Q(t+1)$	$\neg Q(t+1)$	Verhalten
0	0	$Q(t)$	$\neg Q(t)$	Speichern
1	0	0	1	Rücksetzen
0	1	1	0	Setzen
1	1	?	?	?

nicht erlaubt, da beide Ausgänge Q und $\neg Q$ 0 werden

Nachteil des RS-Flipflop

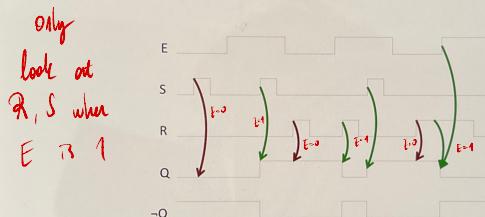
- Beim Speichern eines Wertes 0 oder 1 muss man den Wert kennen!
- Ziel: Speichern unbekannter Werte

D-Latch



- S und R werden aus D berechnet
 $S = D, R = \neg D$
- Enable steuert die Datenübernahme

RS-Latch: Zeitdiagramm

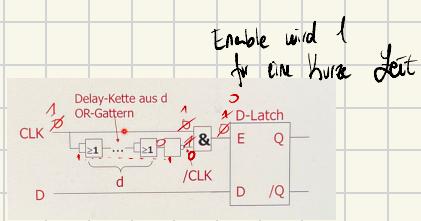
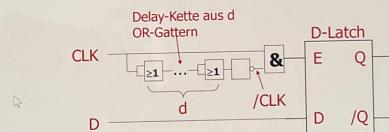


Eigenschaften eines D-Latches

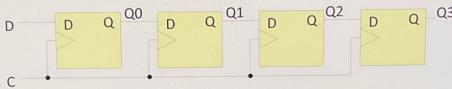
- Das D-Latch heißt **transparent**, wenn das **Schreibsignal aktiv ist**.
- E muss lange genug aktiv sein, damit sich der neue Zustand im RS-FF einstellen kann.
- Das D-Latch ist **pulsgesteuert** (**Schreibpuls E**).
- Einfache 1-Bit Speicherzelle

Taktflankengesteuerte D-Flipflops

- Taktflankengesteuerte Flip-Flops wie das D-Flip-Flop übernehmen Daten zu einem bestimmten Zeitpunkt (kein transparenter Modus), nämlich bei der steigenden Flanke des sog. Clocksignals
- Vorteil:** Daten müssen lediglich bei der steigenden Taktflanke stabil sein (zzgl. Setup- und Holdzeit)

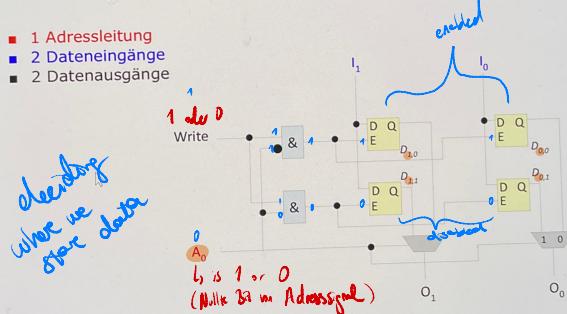


Schieberegister #1



- Alle D-FF übernehmen bei steigender Flanke
- Änderungen geschehen gleichzeitig
- $Q_0 = D$, $Q_1 = Q_0$, $Q_2 = Q_1$, $Q_3 = Q_2$
- Information wird um eine Bitposition nach rechts geschoben

Beispiel: 2x2 Bit Speicher

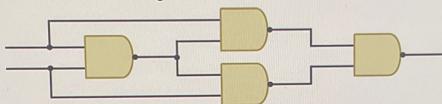


- Übertragbar auf mehrere Adressleitungen
- n Adressleitungen $\Rightarrow 2^n$ -zu-1 Multiplexer
 - Manchmal auch 2ⁿ Speicherplätze genannt

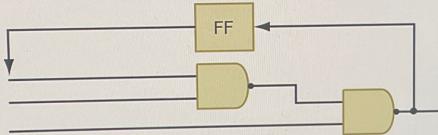
Reale Speicher #1

- Speicherung in D-Latch **zu aufwändig**
- In realen Speichern wird jeweils ein Bit gespeichert:
 - **In einem Kondensator**
 - Dynamic Random Access Memory (**DRAM**)
 - Verliert seine Ladung, muss daher **regelmäßig aufgefrischt** werden
 - **In einer 6-Transistor-Zelle**
 - Static Random Access Memory (**SRAM**)
 - Hält die Information **beliebig lang**
 - Spezielle Varianten für Festwertspeicher (ROM, EPROM, ...)

Kombinatorische Schaltungen



Sequentielle Schaltungen

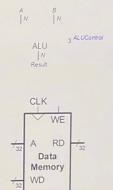


VL 7

RISC-V Registersatz

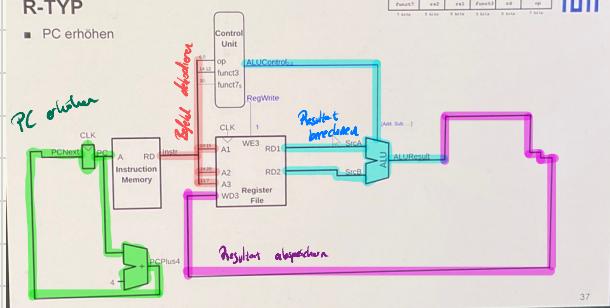
■ Konventionen legen fest, wie Register heißen und wie sie verwendet werden sollen

- Bei RISC-V sind alle Instruktionen 32 Bit lang!
- Da die verschiedenen Instruktionen unterschiedlich viele Operanden haben, werden 6 Instruktionsformate verwendet:
 - 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



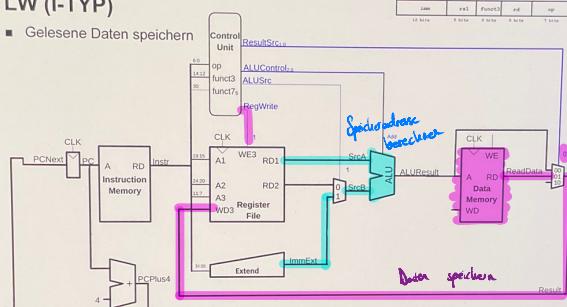
R-TYP

- PC erhöhen



LW (I-TYP)

- Gelesene Daten speichern



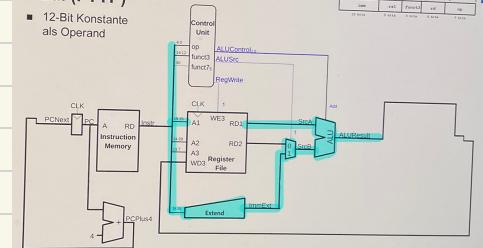
Komponenten (2)

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

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

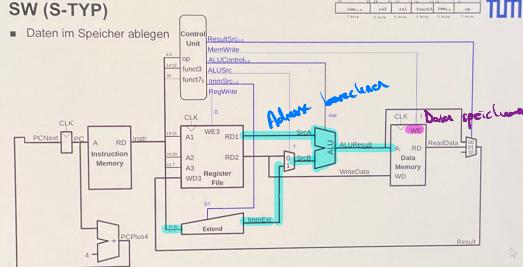
Addi (I-TYP)

- 12-Bit Konstante als Operand



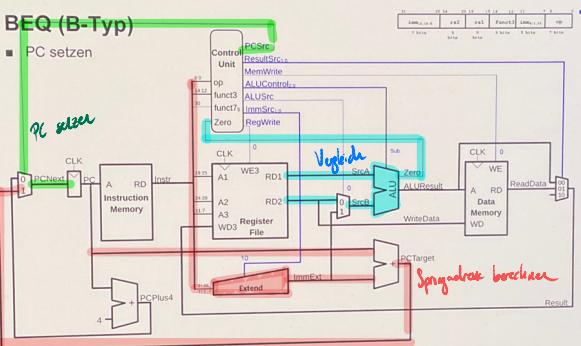
SW (S-TYP)

- Daten im Speicher ablegen



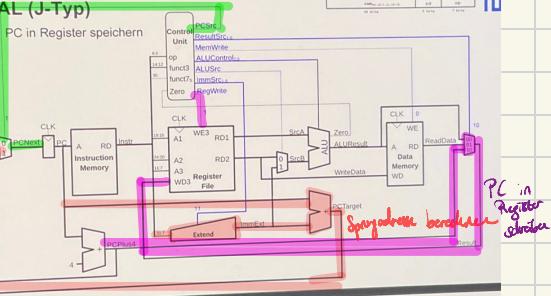
BEQ (B-Typ)

- PC setzen



JAL (J-Typ)

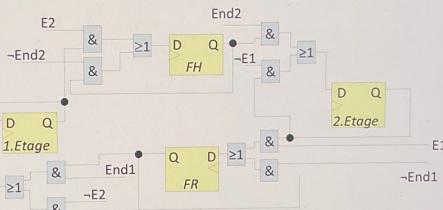
- PC in Register speichern



Endlicher Automat

- Repräsentiert die Funktion einer sequentiellen Schaltung
- Mathematische Darstellung: $A(I, S, s_0, d, O)$
- I ... Menge der Eingaben (**inputs**)
- S ... Menge der Zustände (**states**)
- s_0 ... Initialzustand (**initial state**)
- d ... Zustandsübergangsfunktion (**transition relation**)
- O ... Menge der Ausgaben (**outputs**)

One-Hot Kodierung



Binäre Kodierung #2

$Z0$ und $Z1$ bilden zusammen den aktuellen Zustand
 $Z0'$ und $Z1'$ bilden zusammen den Folgezustand

$$Z0' = \neg Z1 \cdot Z0 \cdot E2 + \neg Z1 \cdot Z0 \cdot \neg End2 + Z1 \cdot Z0 \cdot E1 + Z1 \cdot Z0 \cdot \neg End1$$

$$Z1' = \neg Z1 \cdot Z0 \cdot End2 + Z1 \cdot \neg Z0 \cdot E1 + Z1 \cdot \neg Z0 \cdot \neg End1$$

Zusammenfassen der beiden mittleren Produkte für $Z1'$
 $Z1' = \neg Z1 \cdot Z0 \cdot End2 + Z1 \cdot \neg Z0 + Z1 \cdot Z0 \cdot \neg End1$

Z1	Z0	E1	E2	End1	End2	Z1'	Z0'
0	0	X	0	X	X	0	0
0	0	X	1	X	X	0	1
0	1	X	X	X	0	0	1
0	1	X	X	X	1	1	0
1	0	0	X	X	X	1	0
1	0	1	X	X	X	1	1
1	1	X	X	0	X	1	1
1	1	X	X	1	X	0	0

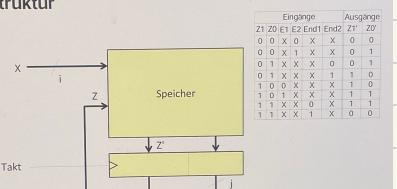
Binäre Kodierung - Implementierung



Nachteile der Gatterrealisierung

- Implementierung ist **aufwändig**
 - Ermitteln der Zustandsübergangsgleichungen
 - Minimierung der resultierenden Schaltung erforderlich
- **Struktur der Schaltung korrespondiert nicht mit Automat**
 - Verifikation nicht einfach möglich
 - Kleine Änderungen am Automaten können große Änderungen an der Schaltung bewirken
- Lösung:
Strukturierte Implementierung des Automaten (Mikroprogramm-STW)

Grundstruktur



Mikroprogrammierte Steuerwerke

- Strukturierte Implementierung von Automaten
- Realisierung der Zustandsübergangsfunktion durch einen Speicher
- Adresse wird gebildet aus
 - **Zustand Z (n Bit)**
 - **Eingangssignale X (i Bit)**
- Ausgang liefert
 - **Folgezustand Z' (n Bit)**
 - **Ausgangssignale Y (j Bit)**
- Größe des Speichers: $2^{(n+i)} \cdot (n+j)$ Bit

Motivation

■ Single-cycle Prozessor

▪ Einfach (jede Instruktion wird **in einem Takt ausgeführt**)

▪ Getrennte Speicher für Instruktionen und Daten

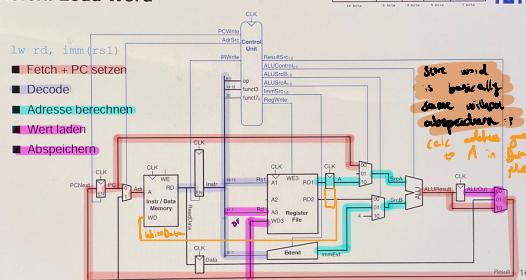
▪ Takt muss lang genug für **langsamste Instruktion (lw)** sein, auch wenn andere Instruktionen viel schneller arbeiten \Rightarrow Performanceverlust

▪ Drei Addierer notwendig (ALU, 2x für PC Berechnung)

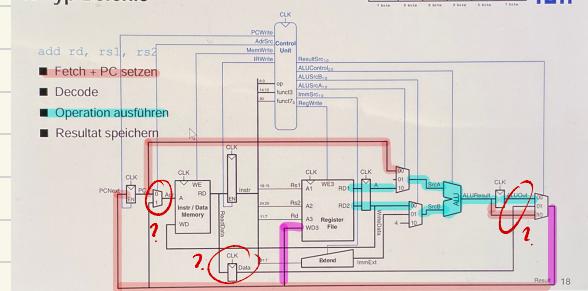
■ Lösungsansatz: Multicycle Prozessor

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

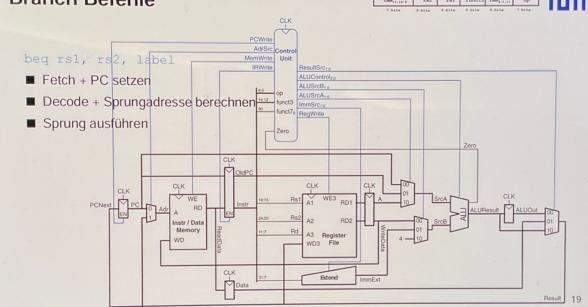
Befehl Load Word



R-Typ Befehle



Branch Befehle



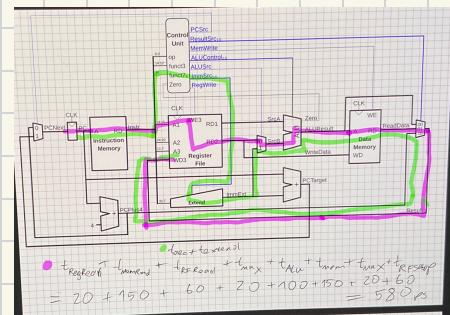
Multicycle RISC-V CPU: Diskussion

■ Vorteile:

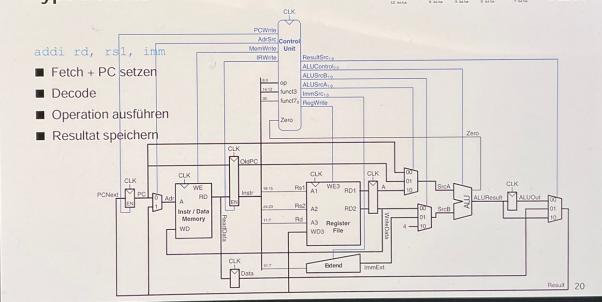
- Befehle beliebiger Länge lassen sich effizienter abarbeiten
 - kein Warten für nicht benötigte Schritte
- Weniger redundante Hardware (z.B. alle Additionen werden in der ALU durchgeführt)
- Gemeinsamer Speicher für Daten und Instruktionen

■ Nachteile:

- Pipelining nicht/nur schwierig möglich
- Etwas komplizierteres Steuerwerk
- ...

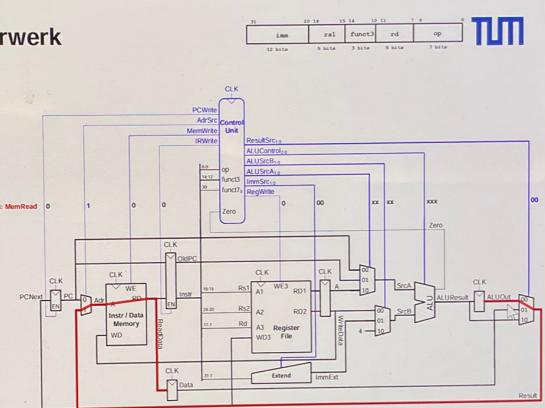
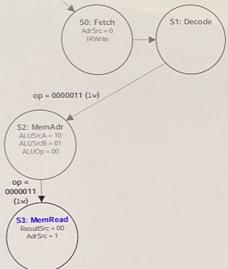


I-Type Befehle



Sequentielles Steuerwerk

- Lade Wert aus Speicher



28

V
U
g

- Single-Cycle RISC-V Prozessor für eine Teilmenge der RV32I Befehle
- Multicycle RISC-V
 - benötigt mehrere Taktzyklen zur Abarbeitung eines Befehls
 - erreicht eine höhere Taktfrequenz als Single-Cycle RISC-V, aber ist insgesamt langsamer
 - Auslastung der Hardware?
 - Keine permanente Auslastung aller HW-Ressourcen, oft nur teilweise Verwendung
 - Ziel: Effizienzsteigerung durch Nutzung möglichst vieler Ressourcen zu jedem Zeitpunkt
- Lösungsansatz: Parallelie Bearbeitung mehrerer Instruktionen durch Fließbandverarbeitung (Pipelining)

Aufteilung der Befehlsabarbeitung in Phasen

- Beispiel: Abarbeitung in 5 Schritten:
 1. Befehls-Holphase (**Fetch**)
 2. Dekodierphase/Lesen von Operanden aus Registern (**Decode**)
 3. Ausführung/Adressberechnung (**Execute**)
 4. Speicherzugriff (**Memory**)
 5. Abspeicherphase (**Writeback**)
- Pipelining schwierig bei CISC:
 - Dauer der Dekodier- und Ausführungsphase (evtl. mehrere Mikroprogrammbefehle) bei den verschiedenen Maschinenbefehlen sehr unterschiedlich
- Umsetzung von Pipelining in Hardware?
 - Hinzufügen von Pipeline-Registern zwischen den Phasen
 - Wir betrachten 5-stufige Pipeline ausgehend von Single-Cycle RISC-V

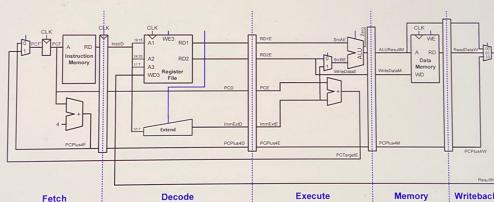


TUM

TUM

Pipelined RISC-V: Datenpfad

- Signale im Pipelined Prozessor sind mit dem ersten Buchstaben der Stufe (z.B., PCF, PCD, PCE) gekennzeichnet
- Keine Pipeline-Register auf Feedbackpfaden



Data Hazards Lösen

- Einfügen von NOPs (NOP = no operation); Stalling

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	NOP	addi			
3	NOP	NOP	addi		
4	NOP	NOP	NOP	addi	
5	add	NOP	NOP	NOP	addi
6		NOP	NOP	NOP	NOP

- Zwischen zwei direkt aufeinander folgenden Befehlen mit RAW-Abhängigkeit müssen sich mindestens 3 andere Befehle befinden

Alternative: Sprungrichtungsvorhersage (Branch Prediction)

- „Raten“ der nächsten Instruktion (z.B. durch Analysen der Häufigkeit des einen oder anderen Ausgangs der Abfrage) und spekulativer Sprung
- Spulen („Flushen“) der Pipeline-Register bei falscher Vorhersage
- Kosten eines solchen falsch vorhergesagten Sprunges: Anzahl von zu entfernenden Instruktionen falls Sprung genommen

- Einfügen von NOPs (NOP = no operation); Stalling

- 2 bei NOPs für unsere Pipelined RISC-V CPU

→ Overhead falls nicht gesprung wird

Sprungrichtungsvorhersage

TUM

Statisch

- ein Bit im Opcode des Sprungbefehls definiert, ob der Sprung spekulativ ausgeführt werden soll oder nicht
- Schleifen: Wahrscheinlichkeit hoch, dass gesprungen wird
- if-then-else: 2/3 der Ausführungen gehen in den ELSE-Teil

Dynamisch

- Eine oder mehrere prozessorinterne Tabellen werden ständig aktualisiert und zur Erzeugung einer Vorhersage ausgewertet
- Tabellen mit Hilfe der Befehlsadresse des Sprungbefehls adressiert
- Hohe Vorhersagegenauigkeit erreichbar

Forwarding: Abkürzungen einbauen

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
```

Neue 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
```

```
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
```



- Prüfe ob Quellreg. in Execute mit dem Zielregister des Befehls in Memory oder Writeback übereinstimmt; falls ja forwarding

- Bsp.: Wert von s2 wurde in Schritt 3 berechnet (Exec) und in Exec in Schritt 4 von add benötigt ⇒ Forwarding ohne Stalling hier möglich

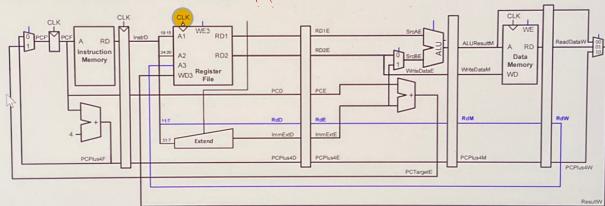
Achtung:

- Funktioniert nur bei direkt aufeinanderfolgenden Compute Befehlen
- Problembsp: lw s1, s2(40) and s3,s2,s1 # Wert aus Datensp. nicht in Ex verfügbar.

52

Pipelined RISC-V: Korrigierter Datenpfad

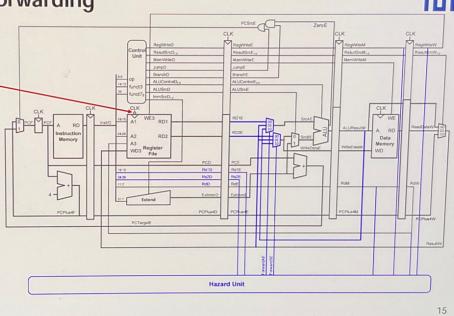
- Optimierung: Registerbank wird zur fallenden Flanke von CLK geschrieben (damit Schreiben in erster Hälfte des Taks und Lesen des geschriebenen Wertes in zweiter Hälfte des Taks möglich).
- Dadurch müssen sich zwischen zwei direkt aufeinander folgenden Befehlen mit RAW-Abhängigkeit nur noch mindestens 2 andere Befehle befinden



Hazard Unit: Forwarding

S1 addi \$8, \$4, 5
S2 sub \$2, \$8, \$3
S3 or \$9, \$6, \$8
S4 and \$7, \$8, \$2

- Hazard Unit prüft ob Quellregister des Befehls in Execute mit dem Zielregister in Memory oder Writeback übereinstimmt
- Muss für beide Register geprüft werden
- Kein Forwarding von Memory zu Decode nötig

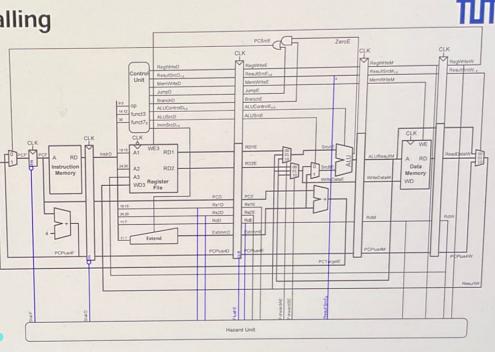


15

Hazard Unit: Stalling

S1 lw \$8, \$4, 5
S2 sub \$2, \$8, \$3
S3 or \$9, \$6, \$8
S4 and \$7, \$8, \$2

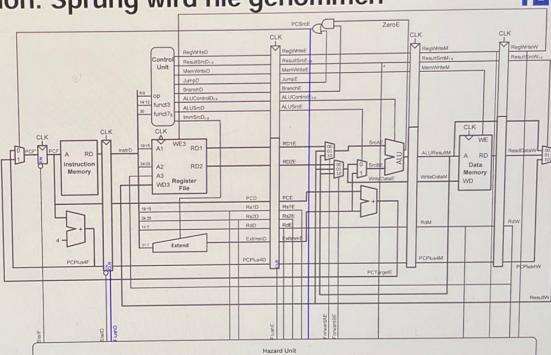
- Überprüfe ob sich lw in Execute befindet
- Überprüfe ob der nächste Befehl aus dem Zielregister des lw liest
- Verhindere Laden des nächsten Befehls
- Wenn lw in Execute ist wird Decode geflushed
 - Im nächsten Zyklus wird somit ein NOP in Execute geladen



22

Branch Prediction: Sprung wird nie genommen

- Der Prozessor lädt die nächste Instruktion unabhängig davon ob gesprungen wird oder nicht
- Hazard Unit prüft ob gesprungen wird oder nicht
- Falls gesprungen wird, müssen die bereits geladenen Instruktionen geflushed werden



One-Bit Dynamic Branch Predictor
↳ Branch Misprediction beim Ein- und Aussteuern der Schleife

2-Bit Dynamic Predictor
↳ Branch Misprediction nur beim Aussteuern aus der Schleife

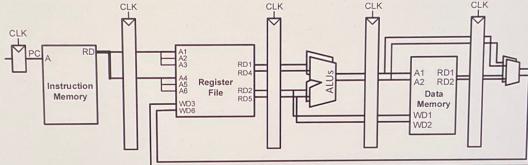
28

Deep Pipelines

- Einfachste Möglichkeit Speed-Up in Prozessoren zu erreichen ist es die Anzahl der Pipelinestufen zu erhöhen
- 8-20 Stufen in modernen Prozessoren üblich
- Aber:** zusätzliche Pipelinestufen erhöhen die Abhängigkeiten zwischen Stufen
 - Manche Abhängigkeiten können durch Forwarding gelöst werden
 - Andere Abhängigkeiten können Stalls verursachen

Superskalare Prozessoren

- Idee: Unabhängige Prozesse können parallel abgearbeitet werden



Restlicher Datenpfad wird entsprechend verdoppelt

Out of Order Prozessoren: Beispiel

- Idee: Eliminiere WAR Abhängigkeiten durch dynamisches Ändern des Zielregisters
- Out of Order Prozessoren lösen Abhängigkeiten durch dynamisches Umordnen der Instruktionen

- Befehlsfolge:

S1	lw	s8, 40(s0)
S2	add	s9, s8, t1
S3	sub	s8, t2, t3
S4	and	s10, s4, s8
S5	or	s11, t6, t6
S6	sw	s7, 80(s11)

- Out of Order Befehlsfolge:

S1	lw	s8, 40(s0)
S2	or	s11, t6, t6
S3	sw	s7, 80(s11)
S4	add	s9, s8, t1
S5	sub	s8, t2, t3
S6	and	s10, s4, s8

- 6 Befehle werden innerhalb von 4 Taktzyklen in den Prozessor geladen.

Zeitschritt	F	D	E	M	W
1	lw or				
2	sw or	lw			
3	add sub	sw or	lw		
4	and sub	sw	lw or		

Superskalare Prozessoren: Beispiel

- Idee: Unabhängige Prozesse können parallel abgearbeitet werden

- Befehlsfolge:

S1	lw	s8, 40(s0)
S2	add	s9, s8, t1
S3	sub	s8, t2, t3
S4	and	s10, s4, s8
S5	or	s11, t6, t6
S6	sw	s7, 80(s11)

Zeitschritt	F	D	E	M	W
1	lw				
2	Stall	lw			
3	add	sub	Stall	lw	
4	and	sub	Stall	lw	
5	sw	and	sub	Stall	lw

- 6 Befehle werden innerhalb von 5 Taktzyklen in den Prozessor geladen

Register Renaming: Beispiel

- Idee: Eliminiere WAR Abhängigkeiten durch dynamisches Ändern des Zielregisters

- Befehlsfolge:

S1	lw	s8, 40(s0)
S2	add	s9, s8, t1
S3	sub	s8, t2, t3
S4	and	s10, s4, s8
S5	or	s11, t6, t6
S6	sw	s7, 80(s11)

Zeitschritt	F	D	E	M	W
1	lw				
2	add	s9, s8, t1			
3	sub	t10, t2, t3			
4	and	s10, s4, s8			
5	or	s11, t6, t6			
6	sw	s7, 80(t11)			

- WAR Abhängigkeit zwischen S4 und S5 kann durch Renaming gelöst werden
- S4 hatte eine RAW Abhängigkeit von S3 bezüglich Register s8
- Folgende Lesezugriffe auf s8 müssen ebenfalls von r0 lesen

VL 10

■ Zwei Bauelemente-Typen



if a then 'leitend'
else 'nichtleitend'

Arbeitskontakt



if a then 'nichtleitend'
else 'leitend'

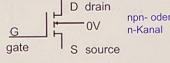
Ruhkontaktekt

Feldeffekttransistoren

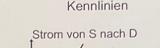
Die Schalter lassen sich besonders leicht mit Feldeffekttransistoren realisieren
→ schnell, klein, billig



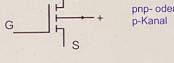
Arbeitskontakt



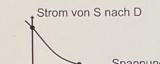
npn- oder
n-Kanal



Ruhkontaktekt



pnp- oder
p-Kanal



Komplettierung der Schichtung

- Schichtung von Assembler ...
 - Programmierung
 - Konventionen
 - ISA Differenzen und Design
- ... über gängige Architekturen ...
 - Von-Neumann
- ... über Implementierungsoptionen ...
 - Mikroarchitekturkonzepte
 - Komponenten Design
- ... bis zur Auflösung in
□ Schaltungen
- Einzelne Transistoren



Halbleiterspeicher: Arten und Techniken

Speicher

- Random Access Memory (**RAM**) als Speicher für von Neumann
- **Jederzeit Zugriff auf jede Speicherzelle**
- Flüchtiger vs. Nichtflüchtiger Speicher

Optionen für **flüchtigen Speicher** (hält Information nur im Betrieb)

- Statisches RAM (SRAM): **Flip-Flops**
- Dynamisches RAM (DRAM): **Ladung in Kondensatoren**

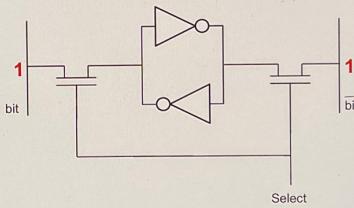
Aus technologischen und ökonomischen Gründen:

- **Große Speicher sind langsam (DRAM)**
- **Schnelle Speicher sind klein (SRAM)**

SRAM Lesen

"Bit" und "nicht bit" werden auf "1" gestzt

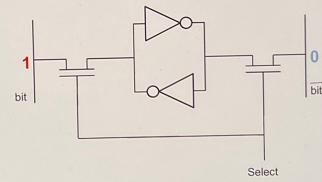
- Inverter versuchen einen Ausgang auf 0 zu ziehen
- Spannungsabfall kann registriert werden
- Verstärkerschaltung nötig



SRAM Schreiben

Daten werden über "bit" und "nicht bit" angelegt

- Schreiben erfolgt wenn Select auf 1
- Inverter schwingen sich ein



Diskussion: SRAM

Eine SRAM Zelle

- Typischerweise aus 6 Transistoren aufgebaut
- Einfach in CMOS Schaltungen zu integrieren
- Mögliche Überlappung von Dekodierung, Zugriff, Ausgabe

Vorteil

- **Behält Wert solange Strom anliegt**
- **Sehr schnell**: Zugriffszeiten im Bereich von 1-30 ns
- **Unempfindlicher hardy**

Nachteil

- **Relativ groß und teuer**
- **Großer Energiebedarf wenn aktiv**

Wird für Register und Caches verwendet

DRAM Speicher

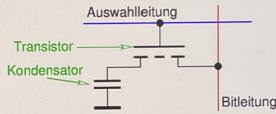
Dynamic Random Access Memory

Aufbau einer DRAM-Zelle

- Kondensator, der Ladung (Information) halten kann
- Auswahlleitung & Bitleitung

Schreiben

- Auswahlsignal setzen
- 0 oder 1 an Bitleitung
- Lädt oder entlädt Kondensator



Lesen

- Auswahlsignal setzen
- Wert zwischen 0 und 1 an Bitleitung
- Bei 0, lädt sich der Kondensator, Bit wird 0
- Bei 1, entlädt sich der Kondensator, Bit wird 1

Diskussion: DRAM

Eine DRAM Zelle

- Typischerweise aus einem Transistor und einem Kondensator aufgebaut
- **Schwerer in CMOS Logik zu integrieren**

Vorteil

- **Einfacher Aufbau**
- **Kann sehr groß und „preiswert“ gebaut werden**

Nachteil

- **Großer Energiebedarf wenn aktiv und passiv**
- **Zugriff ist langsamer**

Wird hauptsächlich für **Hauptspeicher** verwendet

- Benötigt Speichercontroller
- Speichercontroller oft/meistens in den Prozessoren integriert
- **Parallelität wichtig (mehrere Kontroller)**

Diskussion: DRAM

DRAM Matrizen / Array

- Sehr breit angelegt um Kapazität zu erhöhen
 - Bis zu mehreren tausend Bits
- Mehr als Wort pro Zeile
- Bei Lesezugriffen werden Daten in einem Zeilenpuffer („Row Buffer“) zwischengespeichert

Konsequenz:

Nacheinander folgender Zugriff auf Werte innerhalb einer Zeile schneller als mit Zeilenwechsel

Zugriffszeiten

- ~20ns falls Daten im Zeilenpuffer liegen
- ~40ns falls Daten nicht im Zeilenpuffer liegen
- ~60ns falls Zeilenpuffer im Moment belegt („row buffer conflict“)

Parallelität wichtig

- Mehrere Zugriffe für mehrere Matrizen
- Parallelle Kanäle

DRAM Speicher

Dynamic Random Access Memory

- Problem 1: **Lesen zerstört die Information**
- Problem 2: **Kondensator verliert Ladung**

Lösung:

- **Nach Lesen werden Daten sofort wieder geschrieben**
- Regelmäßiges „Auffrischen“ des Zustands
- DRAM Refresh

Refresh im Baustein integriert

Motivation Caches

- Caches: Klein, aber fein (=schnell)

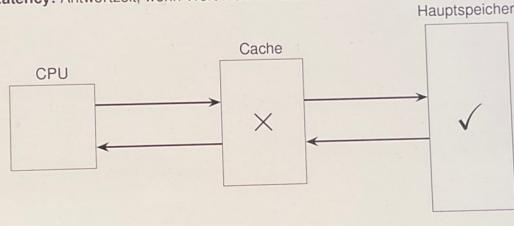
Lokalitätsprinzip

Problem #1: Was soll im Cache liegen?

- zeitliche Lokalität: Ein Zugriff auf eine Zelle erhöht die Wahrscheinlichkeit, dass bald auf diese Zelle erneut zugegriffen wird.
- räumliche Lokalität: Ein Zugriff auf eine Zelle erhöht die Wahrscheinlichkeit, dass auf nahegelegene Zellen zugegriffen wird.

Cache-Terminologie

Miss Latency: Antwortzeit, wenn Wert nicht im Cache



Frequenz (f): Frequenz des Prozessors¹ *cycles/s*

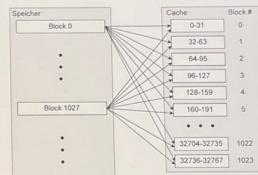
Instruction Count (IC): Anzahl an auszuführenden Instruktionen

Cycles per Instruction (CPI): Durchschnitts-Anzahl an Zyklen pro Instruktion

Memory Access Rate: Durchschnitts-Anzahl an Speicherzugriffen pro Instruktion

Voll-assoziativer Cache

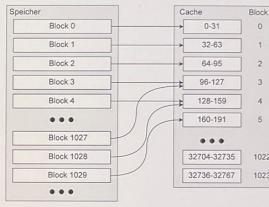
- Jede Adresse kann auf jede Cache Zeile abgebildet werden
- Blocknummer = $\frac{\text{Speicheradresse}}{\text{Cachezeilen-Größe}}$
- #Conflict Misses = 0
- größte Flexibilität
- größte Komplexität (dadurch auch höchster Preis)



Voll-assoziativer Cache am Beispiel eines 32 KiB Caches

Direct Mapped Cache

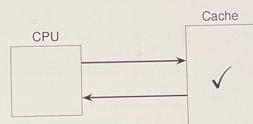
- Ein Block kann an genau einer Stelle im Cache abgelegt werden
- unflexibel
- häufige Verdrängung → mehr Conflict Misses → höhere Miss Rate
- Cachezeile = $\text{Blocknummer} \bmod \# \text{Cachezeilen}$
- mit Blocknummer = $\frac{\text{Speicheradresse}}{\text{Cachezeilen-Größe}}$
- Warum? mod „extrahiert“ untere Tag-Bits



Direct Mapped Cache am Beispiel eines 32 KiB Caches

Cache-Terminologie

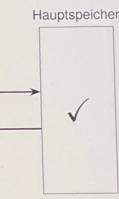
Hit Latency: Antwortzeit, wenn Wert im Cache



Dominic Prinz | Jakob Schäffeler | ERA – Zentralübung #10: Caches

Cache-Terminologie

Miss Penalty: Miss Latency – Hit Latency



Dominic Prinz | Jakob Schäffeler | ERA – Zentralübung #10: Caches

Hit Rate: $\frac{\text{Hits}}{\#\text{Requests}}$

Miss Rate: $\frac{\text{Misses}}{\#\text{Requests}}$

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

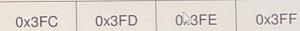
mit:

$$\text{Average Memory Access Time} = \text{Hit Rate} \cdot \text{Hit Latency} + \text{Miss Rate} \cdot \text{Miss Latency}$$

Cachezeilen

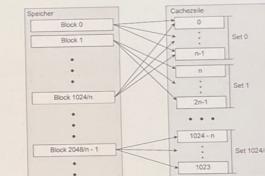
1022/32=31 1023/32=31

- Warum sind 0x03FE und 0x03FF in einer Cache-Zeile?
- Beide haben dieselbe Blocknummer: 0xFF
- Die zugehörige Cache-Zeile:



Mengenassoziativer Cache

- Jede Adresse kann auf einen Teil der Cache Zeilen abgebildet werden
- Anzahl der Zeilen pro Cache Set = Assoziativität des Caches
- Cache-Set = Blocknummer \bmod #Cache-Sets
- mit Blocknummer = $\frac{\text{Speicheradresse}}{\text{Cachezeilen-Größe}}$
- Warum? mod „extrahiert“ Index-Bits



n-Fach-assoziativer Cache am Beispiel eines 32 KiB Caches mit Cachelinigröße von 4 Bytes

Ersetzungsstrategien

- ▶ Random
 - selbsterklärend
- ▶ FIFO (First in, first out)
 - auch selbsterklärend
- ▶ LRU (Least Recently Used)
 - zur Verringerung der Wahrscheinlichkeit des Löschens von Daten, die möglicherweise bald angefordert werden.
- ▶ LFU (Least Frequently Used)
 - verfolgt, wie oft die Blöcke benutzt werden. Der am wenigsten genutzte Block wird ersetzt.
- ▶ Zusätzliche Metriken (Anzahl Zugriffe, letzter Zugriff) nötig!

Berechnungen der Bits:

Vollassoziativ:

Anzahl der Cachezeilen: Cachesize / Zeilenlänge

Index = 0

Offset = $\log_2(\text{Cachezeilenlänge})$

Tag = Adresse - Offset

Mengenassoziativ:

Anzahl der Cachezeilen: Cachesize / Zeilenlänge

Index = $\log_2(\text{Anzahl der Cachesets})$

Offset = $\log_2(\text{Cachezeilenlänge})$

Tag = Adresse - Index - Offset

Direct Mapped:

Anzahl der Cachezeilen: Cachesize / Zeilenlänge

Index = $\log_2(\text{Anzahl der Cachezeilen})$

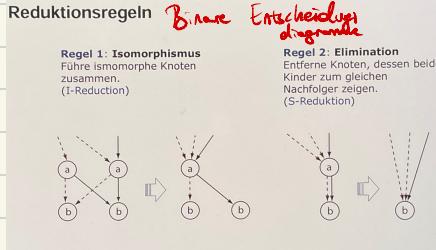
Offset = $\log_2(\text{Cachezeilenlänge})$

Tag = Adresse - Index - Offset

VL 11

- Unterscheidung zwischen
 - kombinatorischer Synthese
(Realisierung einer Booleschen Funktion)
 - sequentieller Synthese
(Realisierung eines endlichen Automaten; kann als Boolesche Funktion repräsentiert werden)

- Die Booleschen Ausdrücke x und \bar{x} heißen Literale, wobei x als positives Literal und \bar{x} als negatives Literal bezeichnet wird.
- Eine Konjunktion von Literalen wird mit **Monom** bezeichnet, wenn zusätzlich folgendes gilt:
 - jedes Literal kommt höchstens einmal vor
 - es kommt nicht sowohl das positive als auch das negative Literal einer Variable vor
- Ein Monom heißt **vollständig** oder **Minterm**, wenn jede Variable entweder als positives oder als negatives Literal vorkommt.
- Eine Disjunktion von paarweise verschiedenen Monomen heißt **Polynom**. Sind alle Monome des Polynoms vollständig, so heißt das Polynom **vollständig**.
- Eine Boolesche Funktion ist von ihrer Darstellung als Formel zu unterscheiden.
- Die Interpretationsfunktion ψ weist einer Formel die entsprechende Funktion zu.
- $f = (a + b) \cdot c$ und $g = a \cdot c + b \cdot c$ sind unterschiedliche Formeln aber $\psi(f) = \psi(g)$

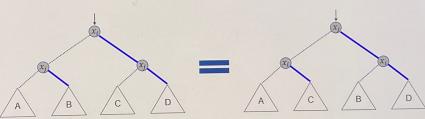


Geordnete Entscheidungsdiagramme

- Ein Entscheidungsdiagramm heißt frei, wenn auf jedem Pfad von der Wurzel zu einem Blatt jede Variable höchstens einmal als Markierung vorkommt.
 - Ein Entscheidungsdiagramm heißt geordnet, wenn auf jedem Pfad von der Wurzel zu einem Blatt die Variablen in der gleichen Reihenfolge abgefragt werden.
 - Ein Entscheidungsdiagramm heißt reduziert, wenn sich keine Reduktionsregeln mehr anwenden lassen.
- Reduced Ordered Binary Decision Diagrams (ROBDDs)



Vertauschen von Variablen in BDDs



Beobachtung

Alle binären Booleschen Operatoren können auf den ternären Operator

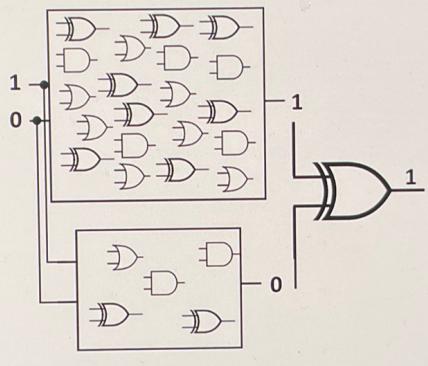
$$\text{ITE}(F, G, H) := FG + F'H$$

zurückgeführt werden.

Beispiel

$$\begin{aligned} \text{AND}(F, G) &= \text{ITE}(F, G, 0) \\ \text{OR}(F, G) &= \text{ITE}(F, 1, G) \\ \text{NOT}(F) &= \text{ITE}(F, 0, 1) \end{aligned}$$

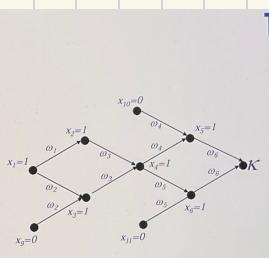
Motivation: Verifikation



Use XOR
to determine
if the
two Schaltungen
give the
same
results

Konfliktanalyse

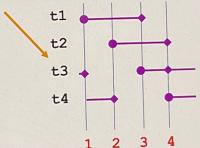
- $\omega_1 = (\neg x_1 \vee x_2)$
- $\omega_2 = (\neg x_1 \vee x_2 \vee \neg x_3)$
- $\omega_3 = (\neg x_1 \vee \neg x_2 \vee \neg x_4)$
- $\omega_4 = (\neg x_1 \vee x_5 \vee \neg x_6)$
- $\omega_5 = (\neg x_1 \vee x_6 \vee \neg x_7)$
- $\omega_6 = (\neg x_6 \vee \neg x_8)$
- $\omega_7 = (x_1 \vee x_7 \vee \neg x_{12})$
- $\omega_8 = (x_1 \vee x_9)$
- $\omega_9 = (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$



→ Implikationsgraph
→ ermöglicht Lernen einer Konfliktklausel

Registerallokation

- (1) $t1 := t3 * 10$
- (2) $t2 := t4 * 20$
- (3) $t3 := t1 + 5$
- (4) $t4 := t2 + t3$



FO 12

For CNF, do the Tauton Transformation

$$d \leftrightarrow (a \wedge b)$$



Resolution

$$\frac{(a_1 \vee a_2 \vee \dots \vee a_n \vee x) \wedge (b_1 \vee \dots \vee b_m \vee \neg x)}{a_1 \vee a_2 \vee \dots \vee b_n \vee b_m}$$

$$\frac{| \quad x \wedge (b_1 \vee b_2 \vee \dots \vee b_m \vee \neg x)}{b_1 \vee b_2 \vee \dots \vee b_m}$$

Subsumption

$$\frac{| \quad x \wedge (a_1 \vee \dots \vee a_n \vee x)}{x}$$

VL 13

Die Zahl der Transistoren verdoppelt sich alle 2 Jahre (1.5 Jahre)

Optimierungs Optionen

Pipelining

- Aufteilung von Befehlen in mehrere Schritte
- Fließband-Bearbeitung aller Schritte
- Nötig: Hazard-Erkennung und Sprungvorhersage

OoO Ausführung durch dynamisches Scheduling

- Unabhängige Instruktionen können ausgeführt werden, obwohl vorherige Operationen blockiert sind
- Ausführung wird „out of order“
- ABER: weiterhin von außen transparent

Superskalarität

- Mehrere Maschinenbefehle gleichzeitig
- Keine unabhängigen Pipelines, sondern flexible Zuordnung von Befehlen zu freien Ausführungseinheiten

Diskussion / Pipelining & Superskalare Ausführung

Beides erhöht die Leistung

- Bessere Ausnutzung von Einheiten
- Nutzung von Parallelität
- Vor allem in Kombination mit OoO Execution

Transparent in der ISA

- Sequentielle Abarbeitung nach von Neumann
- Standard in fast allen modernen Prozessoren
- Vorteil
 - Einfach zu programmieren
- Nachteil
 - Komplexität in Hardware
 - Compilerinformation fehlt
- Wie effizient sind Programme am Ende wirklich?

Kann man das auch mit Software lösen?

SIMD in Heutigen Architekturen

Vorteil

- Höhere Rechenleistung durch Parallelität
- Weniger „Aufwand“ für die Programmabarbeitung

Spekulation

- Branch Prediction ist eine Art von Spekulation
- Kann ausgedehnt werden
 - Speicheradressen
 - Abhängigkeiten
 - Zu ladende Werte
- Problem: Zustand muss zurückrollbar sein

Probleme

- Speicherschutz nicht geprüft bei spekulativer Ausführung
- Tabelle für Sprungvorhersage wird nicht gelöscht bei Prozesswechsel („Trainieren“ falscher Sprünge möglich)

Transactional Memory

- Kette von spekulierten Befehlen
- Durch Benutzer ausgedrückt / eingegrenzt
- Konflikte werden in Hardware erkannt und zurückgerollt
- Unterstützung im Speicher (oft im Cache) um Speicher zurückzurollen

VLIW Prozessoren

VLIW = Very Long Instruction Word

- Mehrere Instruktionen in einem Befehl integriert

Compiler entdeckt Parallelität und kodiert sie in Instruktionen

- Bündel von Instruktionen

Vorteile

- Explizite Abhängigkeiten → Einfachere Hardware
- Explizites Scheduling → Einfachere Hardware

Nachteile

- Komplexe (und nicht kompatible) ISA
- Beruht auf den Fähigkeiten des Compilers

Konzepte in einer GPU

Basis: Vector- / SIMD-Rechnen

- Massive Parallelität auf jedem Chip

Vorteile

- Reduziert die Komplexität des Prozessors
- Spezialisierung
- Weniger Dekodierungsaufwand
- Erhöht Energieeffizienz

Nachteile

- Verringerte Softwareflexibilität (SIMD muss genutzt werden)
- Zusätzlicher Aufwand bei der Programmierung
- Umsetzung auf SIMD
- Neue Programmiersprachen / Bibliotheken
- Komplexe Datenumgebung / Host/Beschleuniger Umgebung

GPU sind typischerweise hierarchisch

Multithreading

Mehrere Hardware Threads in einem Prozessor(kern)

- Einzelne Befehlszähler und Register
- (Fast) alle andere Ressourcen geteilt

Einfaches Multithreading

- Umschalten der Threads pro Takt
- Verschiedene Granularität
- Extremfall: jeder Takt

Simultaneous Multithreading (SMT)

- Gleichzeitige Nutzung von Ressourcen
- Passt zu OoO und Superskalarität
- Erhöhte Buchhaltungskosten

Üblich in vielen modernen Architekturen

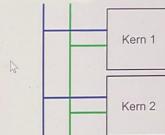
- Vgl. Intels Hyperthreading
- In modernen Systemen unabdingbar
- In anderen nicht immer sinnvoll

Quelle: [1]

Mehrkern-Architekturen

Mehrere HW-Threads in einem System

- Getrennte Ressourcen im Prozessor
 - **Getrennte Kerne**
 - Integriert in einen Prozessor/Sockel
- Replikation von Leit- und Rechenwerk
- **Gemeinsamer Speicher**
 - Teile der Cachehierarchie werden geteilt
 - Üblicherweise auch gemeinsames E/A Werk



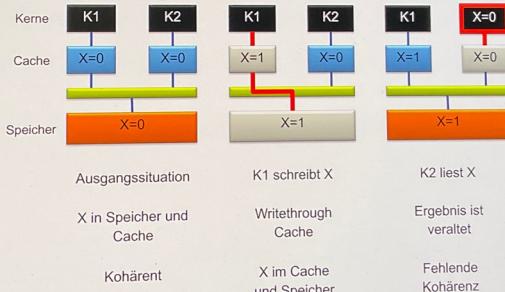
Parallele Ausführung

- **Voller Speed-Up möglich** (im Gegensatz zu SMT)
- **Jeder Core hat gleiche Komplexität** (vgl. mit SMT kleinere Komplexität)

Meist: alle Kerne identisch

- SMP: Symmetric Multiprocessing
- Ausnahme, z.B., ARMs BigLittle Architektur

Cacheprobleme mit Mehreren Kernen (1)



Lösungsmöglichkeiten

Nur ein Cache

- Nähe zu Kernen geht verloren
- Caches werden zu langsam

Manuelle Organisation

- Sehr schwer und fehleranfällig
- Kombination mit anderen Optimierungen (z.B. Prefetcher) problematisch

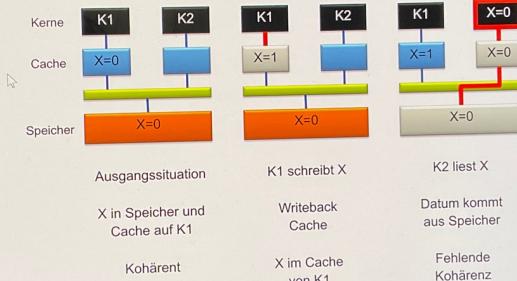
Explizite Aufteilung in Cacheable und Nicht Cacheable Daten

- Sehr schwer und fehleranfällig

Automatische Cachekohärenz

- In Hardware implementiert
- Standard in fast allen Systemen

Cacheprobleme mit Mehreren Kernen (2)



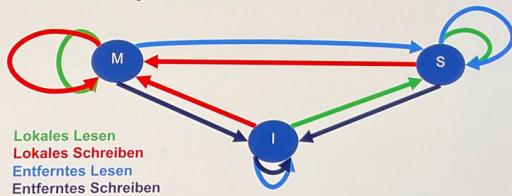
MSI Protokoll

Jede Cachezeile muss ihren Status laufend mitverfolgen

Automat mit drei Zuständen

- Modified: Cachezeile lokal geändert, Speicher nicht kohärent
- Shared: Wert in mindestens einem Cache, kohärent
- Invalid: Cacheblock nicht gültig oder veraltet

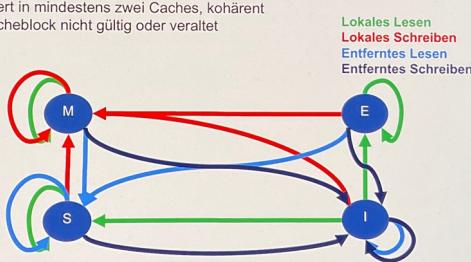
Automat wird nach jedem Zugriff aktualisiert



Erweiterung: MESI Protokoll

Automat mit vier Zuständen

- 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



Konsequenz 3 – Einfluss auf ISA

Verschiedene HW-Threads müssen kommunizieren

- Daten austausch
- Synchronisation

Daten austausch via Speicher

- Lese- und Schreibbefehle
- Problem: Races, falls zwei Zugriffe nötig sind
 - Beispiel: „test and set“
 - Wichtig zur Synchronisation

Neue Assemblerbefehle nötig

- Atomare Operationen
- „Memory Fences“
- Konsistenzmodell
 - Vertrag zwischen Maschine und Programmierer „wie, wann und wo Daten sichtbar werden“
 - Gewollt: Sequentielle Konsistenz
 - In der Praxis oft leicht abgemildert



► Kodieren 3 Zustände:

- Modified: Datenblock in dieser Cachezeile wurde modifiziert. Im Hauptspeicher ist noch ein veralteter Wert.
 - Shared: Mehrere Caches enthalten den in dieser Cachezeile gespeicherten Datenblock. Datenblock im Hauptspeicher ist noch aktuell
 - Invalid: In dieser Cachezeile ist ein veralteter Wert.
- Exclusive: Diese Cachezeile ist die einzige, die diesen Datenblock enthält. Datenblock in Hauptspeicher ist noch aktuell.

► MSI:

- normaler Write-Hit zu shared Block
- Kommunikation über Bus zu Invalidierung in anderen Caches benötigt

► MESI

- Information über Exklusivität im Zustand gespeichert
- keine Bus-Interaktion notwendig
- großer Vorteil wenn jeder Prozessor auf unterschiedlichen Datenblöcken arbeitet

Diskussion: Snooping Protokolle

Einfach zu implementieren

- Voraussetzung: gemeinsame Busse

Problem: Busse skalieren schlecht

- Überlastung bei zu viel Verkehr
- Viele Invalidierungsnachrichten

Alternative: Verzeichnisbasierte (Directory) Protokolle

- Statusbits in einem zentralen Verzeichnis
- Verzeichnis/Directory merkt sich, wo Kopien liegen
 - Gezielte Invalidierung statt Broadcast
 - Verringriger Verkehr
 - Kein zentraler Bus mehr nötig
 - Erhöhter Hardwareaufwand

Heute: meist Directory Protokolle, da keine Shared Busse mehr

Multithreading

Allgemeines Modell: Fork/Join

- Masterthread läuft sequentiell
- Fork startet neuen Thread
- Mehrere Threads laufen parallel
 - Bei mehreren Kernen: echte Parallelität (!)
 - Bei Join wartet der Master Thread bis Worker/Child Thread beendet ist
 - Anschließend weitere Ausführung

Üblichster Ansatz: Locks

Datenstruktur, die sich zwei Threads teilen

- Lock kann von einem Thread angefordert werden
- Nur ein Thread kann das Lock haben
- Der andere Thread muss warten, bis das Lock wieder frei ist
- Oft durch ein „Flag“ in einem gemeinsamen Speicher

Implementierung 1: Spin-Locks

- Falls Lock vergeben, warte in einer aktiven Schleife
- Vorteil: Schnelle Reaktionszeit
- Nachteil: blockiert HW-Threads, kostet Ressourcen

Implementierung 2: Yielding Locks

- Falls Lock vergeben, führe anderen HW-Thread
- Vorteil: Weniger Ressourcenverbrauch
- Nachteil: Längere Reaktionszeit

10 18

- ▶ Data-level parallelism (DLP):
Unabhängige Daten auf denen Operationen ausgeführt werden können.
- ▶ Task-level parallelism (TLP):
Unabhängige Aufgaben, die möglichst zeitgleich ausgeführt werden können.

- ▶ Instruction-Level Parallelism: Low-Level Instruktionenstrom wird genutzt um durch Pipelining und speulative Ausführung Parallelisierung zu nutzen.
- ▶ Vektor-Instruktionen/Architekturen: Vektoreinheiten, Grafikprozessoren und Multimedia Instruktionen werden genutzt um einzelne Instruktionen parallel auf einen Datenstrom anzuwenden (Low-Level Umsetzung von DLP).
- ▶ Thread-Level Parallelism: Effizientes Arbeiten durch eng gekoppelte Synchronisation von parallelen (Hardware-)Threads.
- ▶ Request-Level Parallelism: Parallelisierung von Prozessen, welche durch expliziten Informationsaustausch gemeinsam Anwendungen berechnen.

- ▶ "Paralleler Speedup": Leistungsgewinn durch Parallelisierung.

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

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

$$t = t_{\text{sequential}} + t_{\text{parallel}} \quad (2)$$

- ▶ Maximaler Leistungszuwachs

■ Amdahl's law: bei konstanter Problemgröße

■ Gustafson's law: bei Problemgröße proportional zur Prozessoranzahl

Speedup-Formeln

Amdahl:

$$S_{\text{Amdahl}} = \frac{\tau}{t_s + \frac{t_p}{n_p}}$$

Gustavson:

$$S_{\text{Gustavson}} = \frac{t_s + t_p * n_p}{T}$$

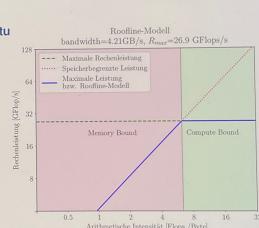
- Berechnet, wieviel schneller das Programm auf n_p Kernen läuft

- Berechnet, wie viel mehr Workload n_p Kerne in derselben Zeit erledigen könnten

Flops = Floating Point Operators per Second

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
- ▶ Achtung: log-log Achsen

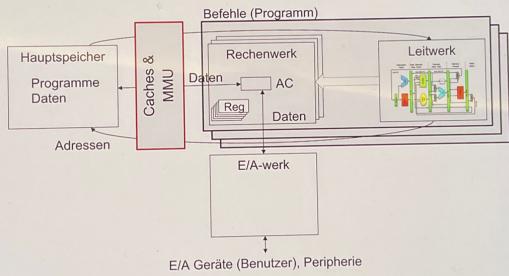


Roofline-Modell Limitierungen Identifizieren

- ▶ Durch Berechnung Limitiert:
 - Bessere Ausnutzung der Architektur / des Systems
 - z.B.: TLP, ILP, SIMD, (Upgrade!)
- ▶ Durch Speicher Limitiert:
 - Bessere Nutzung der Speicherhierarchie!
 - Caches, (L1\$, L2\$, L3\$), Prefetching, Latency Hiding (Upgrade?)

VL 16

Speicherhierarchie



Ethernet

Vorteile

- Niedriger Preis (durch hohe Stückzahlen)
- Gute Leistung
- Weit verbreitet und verfügbar
- Einfach zu nutzen (plug and play)
- Skalierbar
- Über verschiedene Größenordnungen einsetzbar

Unterschiede in (*Netzwerke → WAN, LAN, SAN*)

- Geschwindigkeit (Latenz und Bandbreite)
- Sicherheitsanforderungen
- Interoperabilitätsanforderungen