Grundlagen der Rechnerarchitektur

SS 2015

Inhaltsverzeichnis

1 Einführung

Primär muss ein Computer drei Funktionen besitzen:

1. Daten verarbeiten
2. Daten speichern
3. Daten transportieren

1.1 Begriff der Rechnerarchitektur

In der Literatur finden sich diverse Beschreibungsversuche für den Begriff der Rechnerarchitektur. Allgemein ist man sich aber auf eine Unterscheidung zwischen Schnittstelle und Implementierung einig.

	Ungerer	**R. Brück**	**W. Stallings**
Schnittstelle	Befehlsarchitektur	Rechnerarchitektur	Computer Architecture
Implementierung	Endoarchitektur	Implementierung	Computer Organization

Tabelle 1.1: Verschiedene Versuche, den Begriff der Rechnerarchitektur zu definieren

1.2 Von Neumann'sche Universalrechenautomat

Urvater der meisten Rechner ist der klassische Universalrechenautomat. Dessen Grundprinzip findet sich auch heute noch in modernen Mikroprozessoren. Ein solcher Rechner basiert auf sieben Prinzipien:

1. Der Rechner besteht aus 4 Werken: **Leitwerk** (interpretiert Programme), **Speicherwerk** (Haupt- bzw. Arbeitsspeicher für Programme und Daten), **Rechenwerk** (führt arithmetische und logische Operationen aus) und **Ein-/Ausgabewerk** (kommuniziert mit der Umwelt; ferner: als Sekundärspeicher fungierender Langzeitspeicher).
2. Struktur des Rechners unabhängig vom Problem: Anders als in der Hardwareentwicklung ist ein Computer programmgesteuert, d.h. Befehle werden durch einen Interpreter in Kontrollsignale für All-Zweck arithmetische und logische Funktionen umgewandelt.
3. Programme und Daten stehen im demselben Speicher, sind prinzipiell durch Rechner modifizierbar.
4. Hauptspeicher ist in Zellen gleicher Größe eingeteilt, die durch fortlaufende Nummern (Adresse) benannt werden; über Adresse werden Daten und Programmbefehle angesprochen.
5. Programm besteht aus einer Folge von Befehlen, die im allgemeinen nacheinander ausgeführt werden (Prinzip der Sequentialität als Fortschaltungsregel).
6. Abweichungen von der sequentiellen Ausführung der Instruktionen durch bedingte und unbedingte Sprungbefehle. Ein bedingter Sprung ist dabei ein Sprung, der von der Auswertung gespeicherter Werte abhängig ist.
7. Der Universalrechenautomat besitzt Binärcodes, Zahlen werden im Dualsystem dargestellt.

1.2.1 Bedeutung heute

Das Prinzip der von Neumann'schen Befehlsbearbeitung kommt im Prinzip in nahezu allen kommerziellen Prozessoren zur Anwendung.

Aus Gründen der Leistungssteigerung und der Zuverlässigkeit sind heute jedoch einige Modifikationen üblich:

- Vervielfachung einer oder mehrerer Teilwerke: Mehrere E-/A-Werke, um Ein-/Ausgabe zu beschleunigen bzw. den Datendurchsatz zu erhöhen; Mehrere Leit- und Rechenwerke, um mehrere Befehle gleichzeitig zu bearbeiten.
- Anstelle der zweistufigen Speicherhierarchie (Haupt- und Hintergrundspeicher) mehrstufige Hierarchie: Besseres Preis/Leistungsverhältnis führt zu mehrstufigen Hintergrundspeichern ($\rightarrow$ Caches).
- (Logisch) Getrennte Speicher und Busse für Daten und Befehle.
- Prinzip der Selbstmodifikation aus Sicherheitsgründen aufgegeben.

Achtung! Ein Cache war beim Neumann'schen Rechner nicht vorgesehen. Zwar sind Caches heute gewöhnlicherweise auf dem Chip der CPU platziert, trotzdem gehören die Caches aber zum Speicherwerk, nicht zum Rechenwerk.

1.3 Operationszyklen

Ein Rechner arbeitet mit verschiedenen Operationszyklen. Das galt damals für den IAS-Rechner wie es heute für moderne Rechner gilt. Der IAS-Rechner operierte mit zwei Zyklen: Befehlsholzyklus (BH) und Ausführungszyklus (EX).

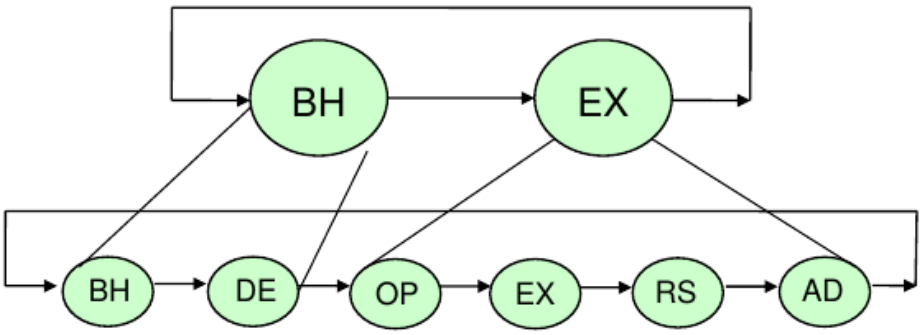


Abbildung 1.1: Die Zyklen des IAS Rechners

Die zwei Zyklen des IAS-Rechners werden wieder in mehrere Teilzyklen (Phasen) aufgeteilt:

Befehlsholphase (BH) Auf Basis des Befehlszählers wird der nächste zu bearbeitende Befehl aus dem Speicher ins Instruktionsregister eingelesen.

Dekodierungsphase (DE) Dekodiert Operationscode und generiert Steuerungssignale.

Operandenholphase (OP) Stellt der ALU die im Maschinenbefehl im Adressteil spezifizierten Operanden zur Verfügung.

Ausführungsphase (AU) Verknüpft in den Registern des Rechenwerkes die zuvor geholten Operanden.

Rückschreibphase (RS) Die während der Ausführungsphase produzierten Ergebnisse werden in die vorgesehenen Speicherstellen (Speicher, Register) zurückgeschrieben.

Adressierungsphase (AD) Adresse des nächsten Befehls wird bestimmt und im Befehlszähler abgelegt.

2 Programmierung von Rechnern

Auch wenn es heute recht unüblich ist, in Assembler zu programmieren, bietet dieser Ansatz einige in gewissen Situationen nützliche Vorteile: Evtl. ist ein besseres Ergebnis bzgl. Größe und Geschwindigkeit des Programms erzielbar. Zudem ist es deutlich vorhersagbarer, wie lange ein Programm genau laufen wird, um ein gewünschtes Ergebnis zu produzieren (Echtzeitanwendungen im Embedded-Bereich).

Typischerweise verwendet man Hochsprachen (C, C++, Haskell, ...) und übersetzt diesen Programmcode dann in von der Recheneinheit verarbeitbare Befehle. Die Vorteile einer Hochsprache sind offensichtlich: Anstelle sich Gedanken über die tatsächliche Rechnerarchitektur zu machen, kann in Hochsprachen problemorientiert ein Programm entwickelt werden.

2.1 Übersetzung

Programme, die in Hochsprachen geschrieben wurden, müssen zuerst durch den Einsatz diverser Tools für die Recheneinheit ausführbar gemacht werden.

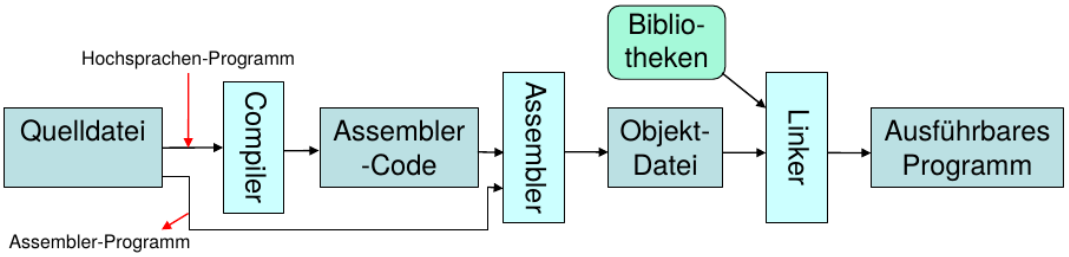


Abbildung 2.1: Der Weg eines C-Programms von Quelldatei bei zum ausführbaren Programm.

2.1.1 Format der Objektdateien

Assembler erzeugen Objektdateien, bestehen aus 6 Bereichen:

Header Beschreibt Größe der Objektdatei und Position der anderen 5 Bereiche in der Objektdatei.

Text Enthält Routinen der Quelldatei in Maschinensprachen-Code und mögliche unaufgelöste Referenzen.

Data Binäre Darstellung der im Programm initialisierten Daten. Auch hier Referenzen auf Labels in anderen Dateien möglich.

Relocation Segment, dass anzeigt wo und welche Referenzen auf Instruktionen, z.B. externe Aufrufe, oder Datenworten im Textsegment enthalten sind, die vom Linker aufzulösen sind.

Symbol Table Verbindet Adressen in der Objektdatei mit globalen Labeln.

Debug Informationen für den Debugger zur Abbildung von übersetzten Instruktionen zu den Zeilen im Quellprogramm.

2.2 Assemblerprogrammierung

In diesem Kapitel werden die Grundlagen der Programmierung in x86-Assembler beschrieben.

2.2.1 Register

Die Architektur verfügt über 8 nahezu universell verwendbare Register der Breite 32 bit: `eax`, `ebx`, `ecx`, `edx`, `edi`, `esi`, `ebp` und `esp`. Greift man auf diese Register zu, entspricht das einem Zugriff auf die gesamten 32 Bits.

Es ist allerdings auch möglich, nur auf einen kleineren Teil der Register zuzugreifen.

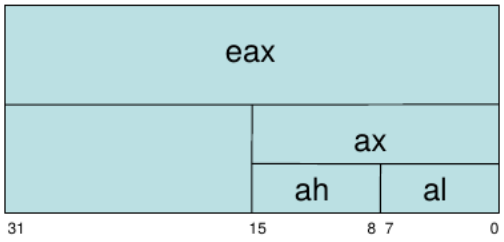


Abbildung 2.2: Darstellung des Registers `eax`. Möchte man nur auf die weniger signifikanten 16 Bit zugreifen, verwendet man den Alias `ax`. `ax` ist wieder in `ah` und `al` aufgeteilt.

Zusätzlich zu diesen Universalregistern gibt es u.A. noch den Befehlszähler `eip` und die Flags. Beide Register sind wie die anderen Register auch 32 bit breit.

2.2.2 Syntax

Es sind zwei verschiedene Syntaxarten in Verwendung: AT&T und Intel. Gewöhnlicherweise produziert `gcc -S` Code in der AT&T-Syntax.

	AT&T	**Intel**
Register:	`%eax`	`eax`
Unmittelbar:	`$123`	`123`
Direkt/Absolut:	`123`	`[123]`
Register indirekt:	`(%eax)`	`[eax]`
Displacement:	`123(%eax)`	`[eax + 123]`
Indiziert:	`(%eax, %edx, 2)`	`[eax + edx*2]`
Skaliert:	`123(%eax, %edx, 2)`	`[eax + edx*2 + 123]`

Insbesondere unterscheiden sich die beiden Syntaxen durch die Anordnung ihrer Operanden. AT&T verwendet `op src, dest` während Intel `op dest, src` verwendet.

2.2.3 Arithmetik

- Arithmetisch:
  - add – addiere
  - adc – addiere mit Carry-Flag
  - sub – subtrahiere
  - sbb – subtrahiere mit Carry-Flag („borrow“-Semantik)
  - mul – multipliziere, Quelle und Ziel stets Register %eax
  - div – dividiere, Quelle und Ziel stets Register %eax
  - inc – erhöhe um 1
  - dec – erniedrige um 1
- Logisch
  - and – bitweises Und
  - or – bitweises Oder
  - xor – bitweises exklusives Oder
  - not – Invertieren

Abbildung 2.3: Auszug typischer arithmetischer und logischer Operationen.

2.2.4 Behandlung von Variablen

Persistente (globale) Variablen erhalten eine Speicherstelle im Datensegment. Eventuell wird dabei noch zwischen nur lesbaren (immutable) und beschreibbaren (mutable) Variablen unterschieden.

Lokale Variablen sind nicht persistent. Sie werden auf dem Stack gespeichert.

2.2.5 Bedingte Sprünge

Die Ausführung von bedingten Sprüngen ist an die Auswertung von Bedingungen geknüpft. Diese werden durch die Inhalte von sog. Flags bestimmt. Flags werden bei arithmetisch/logischen Operationen durch die ALU ermittelt. Ein Register nimmt jeweils die neuen Werte auf.

jOP Ziel	Verwendete Flags	Semantik	
• je	Zero=1	=	
• jne	Zero=0	!=	
• jg	Zero=0 & Overflow=Sign	>	signed
• jge	Overflow=Sign	>=	signed
• jl	Sign != Overflow	<	signed
• jle	Sign != Overflow | Zero=0	<=	signed
• ja	Carry=0, Zero=0	>	unsigned
• jae	Carry=0	>=	unsigned
• jb	Carry=1, Zero=0	<	unsigned
• jbe	Carry=1	<=	unsigned

Abbildung 2.4: Bedingte Instruktionen.

Bessere Vergleiche

Um die Flags für den Vergleich zu berechnen, muss subtrahiert werden. Das Ergebnis wird jedoch nicht gebraucht, lediglich die Flags. Hierfür wird eine besondere Subtrahier-Anweisung `cmp` eingeführt: Subtrahiere

B von A, aber speichere das Ergebnis nicht!

Es werden also nur die Flags gesetzt, auf die das Programm dann reagieren kann.

2.2.6 Stack

Der Stack ist ein zugewiesener Teil des Speichers, der durch `push <src>` und `pop <dest>` verwendet werden kann.

Implementiert wird der Stack in x86 durch zwei Register: `esp` („Stack Pointer“) und `ebp` („Frame Pointer“ oder „Base Pointer“). `esp` stellt das Ende des Stacks da, also gerade die Stelle im Speicher, in dem sich der letzte Wert des Stacks befindet. `ebp` wird gewöhnlicherweise am Anfang einer Funktion auf den aktuellen Wert von `esp` gesetzt (Prolog). Sollen dann lokale Variablen der Funktion aufgerufen werden, subtrahiert man ein entsprechendes Offset von `ebp`, um dessen Wert zu erhalten.

Zwar ist dies im Vergleich zu Registern relativ langsam (der Stack ist ja Teil des Speichers), doch sind die Register ja nur in sehr begrenzter Anzahl zur Verfügung. Zudem ermöglicht die Implementierung eines Stacks Rekursion in Funktionen, was natürlich sehr nützlich ist.

2.2.7 Speicherbehandlung bei Unterprogrammen

Problem: Unterprogramm überschreibt Register, deren Inhalt noch von Aufrufer benötigt werden.

Lösung: Registerinhalte werden auf dem Stack gesichert. Hierbei unterscheidet man per Konvention zwischen zwei Arten von Registern:

caller save `eax`, `ecx` und `edx` dürfen überschrieben werden; der Aufrufer muss diese auf den Stack sichern, sofern Werte noch benötigt werden.

callee save `ebx`, `ebp`, `edi`, `esi`: die aufgerufene Funktion muss die alten Werte wiederherstellen, sofern diese Register benutzt werden.

2.2.8 Funktionsaufrufe

Andere Funktionen werden in ASM durch `call` aufgerufen. Effektiv handelt es sich dabei um einen Sprung zu der Stelle im Programm. Sind wir in der Funktion angekommen, wird gewöhnlicherweise ein Prolog und am Schluss ein Epilog ausgeführt.

Vor dem Sprung in die neue Funktion wird die Rücksprungadresse auf den Stack gelegt, damit später mit `ret` zurückgesprungen werden kann.

Prolog

Der Prolog besteht aus drei Instruktionen: (1) Zuerst wird der alte Base Pointer `ebp` auf den Stack gepusht, damit er später wiederhergestellt werden kann. (2) Im zweiten Schritt wird ein neuer Base Pointer definiert. Hierfür wird genau der derzeitige Stack Pointer `esp` verwendet, der auf das vorderste Element des Stacks zeigt. (3) Zuletzt wird `esp` noch dekrementiert, also der Stack vergrößert. So wird Platz für lokale Variablen geschaffen.

```
pushl %ebp      ; (1)
movl %esp, %ebp ; (2)
subl $12, %esp  ; (3) (es werden 12 Bytes reserviert)
```

Diese drei Befehle könnten auch durch die Instruktion `enter $12, 0` ausgeführt werden. Diese ist jedoch um einiges langsamer und wird deswegen kaum genutzt. `0` ist eine fixe Flag für die `enter`-Instruktion.

Epilog

Am Ende einer Funktion wird noch vor dem Aurufen von `ret` der Epilog ausgeführt. Auch der Epilog besteht meistens aus drei Schritten: (1) Das Ergebnis der Berechnung wird in das Register `eax` geschrieben. (2) Der Stack wird wieder auf den ursprünglichen Anfang zurückgesetzt. (3) Der Base Pointer `ebp` wird wieder vom Stack gelesen.

```
movl result, %eax  ; (1)
movl %ebp, %esp    ; (2)
popl %ebp          ; (3)
```

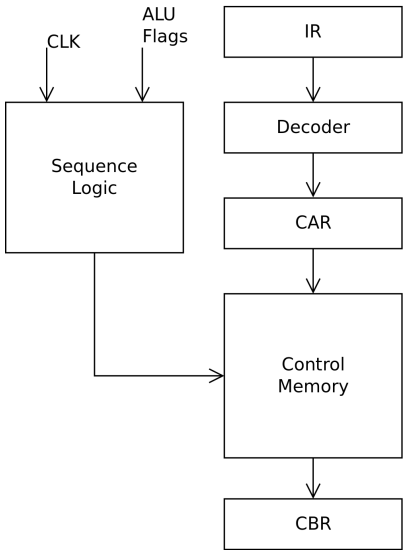
Die letzten zwei Zeilen des Epilogs, also Schritte (2) und (3), können auch mit dem einfachen Befehl `leave` (ohne Parameter) durchgeführt werden. Tatsächlich ist das üblich, da `leave` schneller ist.

2.3 Mikroprogrammierung

Ein Computer wird mittels Software durch Makrobefehle programmiert. Die ASM-Instruktionen sind solche Markobefehle, also `addl`, `movl`, etc.

Dise Makrobefehle sind selber aber als Mikrobefehle implementiert. Man spricht von Mikroprogrammierung. Diese Mikroprogrammierung ist Teil der Firmware eines Rechners.

2.3.1 Elemente eines mikroprogrammierten Leitwerks



CAR Leitwerkadressregister, enthält Adressse der nächsten Mikroinstruktion, d.h. es handelt sich um den Program Counter des Mikroprogramms.

CBR (Control Buffer Register) nimmt Inhalt aus Mikroprogrammspeicher und enthält dann den auszuführenden Mikroprogrammbefehl.

2.3.2 Horizontale und vertikale Mikroprogrammierung

Bei der horizontalen Mikroprogrammierung enthält das Register CBR bereits die tatsächlichen Steuersignale für die Hardware. Anders ist das bei der vertikalen Mikroprogrammierung, dort müssen die Inhalte von CBR noch einmal von einem Decoder auf die tatsächlichen Steuersignale übersetzt werden.

2.3.3 Vorteile

- Mikroprogrammspeicher ist veränderbar, das führt zu einer hohen Flexibilität.
- Neue Prozessorversionen verstehen Befehle der alten Version.
- Andere Befehle können emuliert werden. Ein Prozessor kann so den Befehlssatz einer anderen Architektur interpretieren.

Mikroprogrammierung hat viele Vorteile, denn die Alternative ist die Implementierung des Leitwerks als Finite State Machine. Ein solcher Automat kann natürlich nach der Auslieferung nicht mehr aktualisiert werden, da die Logik dann fest verdrahtet auf der Hardware ist.

3 Architektur

Ein Rechner ist in viele Teile unterteilt, wie Speicher, Peripherie, etc. Auf diese wird nun detailierter eingegangen.

3.1 Befehlssatz-Architekturen

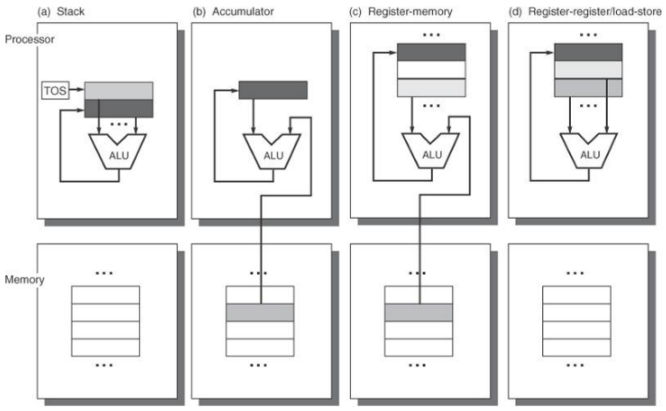


Abbildung 3.1: Die vier üblichen Befehlssatz-Architekturen.

Die verschieden Architekturen sollen jeweils an Pseudocode erklärt werden, der eine Addition der zwei Variablen `A` und `B` ausführt. `A` und `B` befinden sich auf dem Hauptspeicher.

3.1.1 Stack

`add` popt hier die letzten zwei Werte vom Stack, errechnet die Summe und pusht das Ergebnis dann wieder auf den Stack. Operanden und Ergebnis sind also alle allein auf dem Stack.

```
push A
push B
add
pop C
```

Beispiel hierfür: Taschenrecher, Java Virtual Machine (JVM).

3.1.2 Register-Register

Hier arbeitet die ALU nur auf Registern, es kann nicht direkt auf den Hauptspeicher zugegriffen werden. Zuerst werden die Werte `A` und `B` in die Register `R1` und `R2` geladen. `add` nimmt in diesem Beispiel drei Operanden, nämlich zusätzlich zu den zwei Summanden noch das Zielregister `R3`.

Auf den Speicher wird hier mit `load` und `store` zugegriffen.

```
load R1, A
load R2, B
add R3, R1, R2
store R3, C
```

Beispiel hierfür: ARM, MIPS, PowerPC. Die gesamte Familie der RISC-Prozessoren arbeitet nach diesem Prinzip.

3.1.3 Register-Memory

Hier ist es möglich, bei Operationen auf der ALU direkt auf den Speicher zuzugreifen, zumindest mindestens für einen Operand.

```
mov R1, A
add R1, B
mov C, R1
```

Beispiel hierfür: x86.

3.1.4 Akkumulator

Werte werden in und aus dem Akkumulator geladen. Operationen auf der ALU arbeiten mit einem Wert aus dem Akku und einem Wert aus dem Hauptspeicher. Das Ergebnis wird wieder in den Akku gesichert. Von dort aus kann es durch `store` wieder in den Speicher geschrieben werden, im folgenden Beispiel in die Variable `C`.

```
load A
add B
store C
```

Beispiel hierfür: Einige Mikrocontroller.

3.2 Alignment

Alignment ist die Anordnung eines Datenwortes im Speicher. Ein Objekt bestehend aus s Bytes abgelegt im Speicher unter der Adresse A ist gerade dann exakt ausgerichtet (engl.: „aligned“), wenn gilt:

$$A \mod s = 0$$

3.3 Endianness

Endianness ist die Reihenfolge der Bytes eines Multi-Byte-Datums im Speicher. Man unterscheidet zwischen little endian und big endian:

little endian least significant byte first (x86)

big endian most significant byte last (übliche Zahlendarstellung, Verwendung in Netzwerkprotokollen)

- Endianness betrachtet nur die Reihenfolge der Bytes, nicht der Bits. Die Bits sind immer gleich sortiert: von most significant zu least significant.
- Strings sind unabhängig von der Endianness, wenn der Datentyp `char` nur 1 Byte breit ist.

3.4 Cache

Der Speicher ist maßgebend für die Leistungsfähigkeit und Kosten eines Rechners. Idealerweise ist er ausreichend groß und die Zugriffszeit kann mit der Verarbeitungsgeschwindigkeit des Prozessors mithalten. Aus wirtschaftlichen Gründen ist das aber schwierig, weshalb man eine mehrstufige Speicherhierarchie verwendet. Mit steigender Kapazität werden günstigere, aber dafür langsamere Technologien verwendet. Allgemein ist deswegen der kleinste Speicher meistens auch der schnellste.

Zwischen Register und Hauptspeicher befinden sich ein oder mehrere Cache(s). Allgemein ist ein Cache ein schneller Zwischenspeicher der Ausschnitte aus den jeweils nächsten (nicht zwingend direkt nächsten) Stufen enthält.

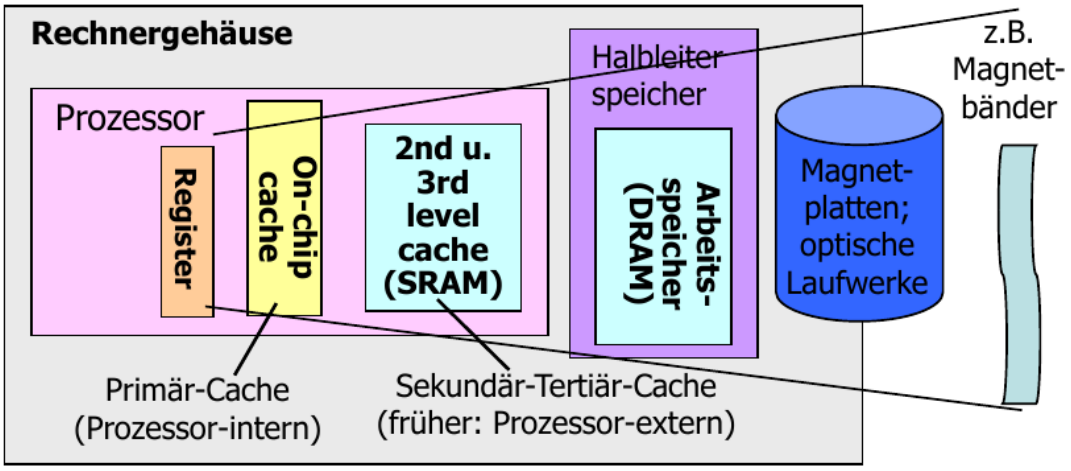


Abbildung 3.2: Zwischen den verschiedenen Stufen herscht eine Inklusionsbedingung. So muss der Inhalt des L1-Cache im L2-Cache Platz haben. Es sei bemerkt, dass das nicht bedeutet, dass der gesamte Inhalt eines Caches zwangsläufig auch in den späteren Schichten vorhanden ist.

3.4.1 Lesen aus dem Cache

Wird auf ein Datum zugegriffen, kann dies entweder zu einem Hit oder Miss führen.

Hit Datum ist im Cache, ein schneller Zugriff ist sofort möglich.

Miss Datum ist nicht im Cache. Dies führt zu einer Anfrage in der jeweils nächsten Speicherstufe. Sobald das Datum verfügbar ist, wird es auch im ursprünglichen Cache zwischengelagert.

3.4.2 Schreiben in den Cache

Wie auch beim Lesen kann es zu Hit und Miss kommen. Jeweils für Hit und Miss gibt es zwei Vorgehensweisen.

Hit Datum ist bereits im Cache zwischengespeichert. Man unterscheidet zwischen zwei Aktualisierungsstrategien:

write-back Das Datum wird nur in der aktuellen Stufe aktualisiert. Kommt es später zur Verdrängung des Datums aus dem Cache, wird der Wert erst dann in den späteren Stufen aktualisiert.

write-through Das Datum wird sofort auf allen Ebenen aktualisiert (Verhalten von `volatile` in C oder Java).

Miss Datum ist gerade nicht im Cache zwischengespeichert. Auch hier gibt es zwei Ansätze, damit umzugehen:

write-allocate Wert wird im Hauptspeicher aktualisiert. Außerdem wird der Wert vorsorglich auch im Cache zwischengespeichert.

non-write-allocate Wert wird erst gar nicht in den Cache zwischengespeichert, sondern nur an die nächste Stufe weitergeleitet.

3.4.3 Cachestrategien

Ein Cache muss vorhersagen, welche Daten wohl als nächstes verwendet werden. Nur so führt er zu einer Verbesserung der Perfomanz. Hierbei nutzt man die zeitliche und die räumliche Lokalität aus.

Zeitliche Lokalität: Nach Zugriff auf ein Datum erfolgt in naher Zukunft erneut ein Zugriff auf genau dasselbe Datum.

Beispiele: lokale Variablen, Programmcode einer Schleife.
Umsetzung: Caching des Datums selbst, Laden bei Miss.

Räumliche Lokalität: Nach Zugriff auf ein Datum erfolgt in naher Zukunft ein Zugriff auf benachbarte Daten.

Beispiele: Sequentielle Ausführung eines Programms, Objekte (C++) / Strukturen mit zusammenhängenden Membern, Arrays.
Umsetzung: Nicht nur das gesuchte Datum selber in den Cache laden, sondern gleich ganze Blöcke.

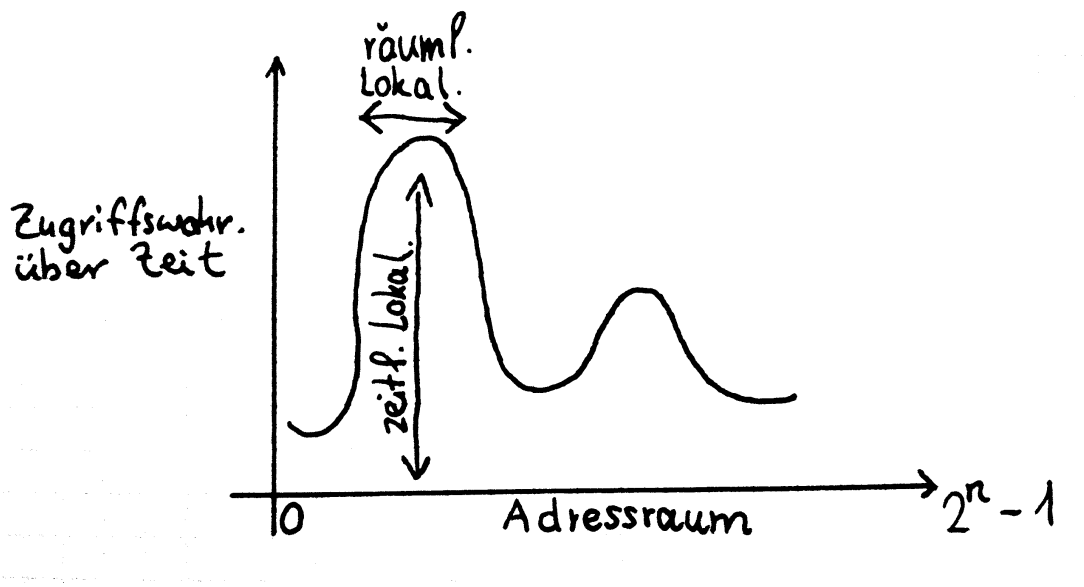


Abbildung 3.3: Zeitliche und räumliche Lokalität im Diagramm.

3.4.4 Organisationsformen von Cache

Der Cache kann unterschiedlich organisiert werden. Für jede Organisationsform gilt aber: Jede Cachezeile hat ein Validbit, welches angibt, ob die Zeile gerade belegt ist oder nicht.

Außerdem muss jedes im Cache gespeicherte Datum auch nochmal eindeutig identifiziert werden, damit richtig auf Hit/Miss geprüft werden kann.

Direktabbildend (direct mapping)

Cache-Zeile eindeutig für die gesamte Anzahl von Hauptspeicherzeilen. Die Berechnung ist trivial über eine mod-Operation abbildbar:

$$i = j \mod m$$

Wobei i die Cachezeile, j die Speicherzeile und m die Anzahl der Zeilen im Cache ist. Beim direktabbildenden Cache bedarf es nur einem einfachen Tagfeld, um eine Cachezeile einer Speicherzeile zuzuordnen. Dieser Tag ist einfach der Rest der Speicheradresse, also nur die höchstwertigen Bits ohne i.

Vollassoziativ (full associative)

Ein gesuchter Block kann überall im Cache liegen. Dadurch verhindet man verschwendeten Speicherplatz im Cache, nimmt dafür aber eine Suche in $\mathcal{O}(n)$ in Kauf.

Wie beim direktabbildenden Cache ist das Tagfeld einfach der „Rest“ der Speicheradresse.

n-fach Assoziativ (n-way associative oder set-associative)

Unterteilung der Cache-Zeilen in Mengen. Die Abbildung von Block zu Menge ist eindeutig, innerhalb der Menge erfolgt die Verwaltung aber vollassoziativ. n ist gerade die Anzahl von Cache-Zeilen pro Menge.

Zusätzlich zum Tag gibt hier der Index die verwendete Menge an.

n-fach assoziativer Cache ist die am häufigsten verwendete Organistationsform.

3.4.5 Eindeutige Identifikation

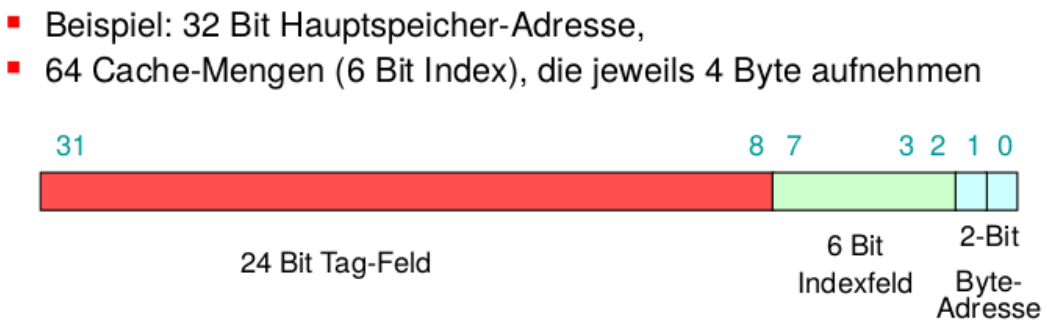


Abbildung 3.4: Cacheadresse. Ein Teil der Adresse ergibt sich aus der Adresse des Cacheblockes. Als Kennung oder Taginformation wird gewöhnlicherweise der restliche Teil der Hauptspeicheradresse verwendet. Der Index ist beim n-fach assoziativen Cache die gewählte Menge. Durch die Byteadresse (auch „Offset“) kann genau ein Byte adressiert werden.

3.4.6 Klassifikation von Fehlzugriffen

Compulsory Der erste Zugriff auf einen Block trifft nicht den Cache, Block muss erstmals geladen werden. Auch als Kaltstart-Miss bezeichnet.

Capacity Der Cache hat nicht genug Platz, um alle Blöcke der aktuell zu bearbeitenden Befehlsfolge zu enthalten. **Tritt nur beim vollassoziativen Cache auf.**

Conflict In nicht voll-assoziativen Caches werden Blöcke aufgrund von Adresskonflikten überschrieben und ggf. später zurückgeladen. **Tritt bei direktabbildendenden oder n-fach assoziativen Caches auf.**

3.5 Speichertechnologien

In Computern finden sich hauptsächlich zwei Arten von Speicher: SRAM und DRAM. Diese unterscheiden sich in Geschwindigkeit, Kapazität und Kosten.

3.5.1 SRAM (static random access memory)

- Speicherung in Transistorschaltungen (Flip-Flops).
- Schnelle Zugriffszeiten.
- Vergleichsweise teuer.
- Verwendung als Cache.

3.5.2 DRAM (dynamic random access memory)

- Speicherung erfolgt durch Kondensatoren.
- Nach dem Lesen und auch regelmäßig in relativ kurzen Zeitintervallen (32 ms) muss der Inhalt refreshed werden, da sich die Kondensatoren entladen und so die Informationen verloren gehen.
- Günstig, aber im Vergleich zum SRAM eher langsam.
- Verwendung als Hauptspeicher.

3.5.3 DDR-Technologien im Vergleich

Heute sind DDR-(Double Data Rate)-Technologien üblich. Auslesen dieser Speicher erfolgt sowohl bei steigender als auch bei fallender Flanke.

SDRAM 1 Bit-Zugriff. Vollständig synchron zum Bustakt.

DDR1 2 Bit-Zugriff (Prefetch). Mit steigender und fallender Taktflanke werden zwei aufeinanderfolgende Adressen auf einmal gelesen.

DDR2 Anstatt zwei werden gleich vier Bit ausgelesen.

DDR3 Anstatt vier werden acht Bit ausgelesen.

GDDR3, bzw. GDDR5 (Graphics Data Double Rate) Entworfen für Hochleistungsanwendungen, die hohe Bandbreiten verlangen, z.B. Anwendungen auf Grafikkarten; Basieren auf DDR2 bzw. DDR3 aber mit mehr Bänken, die gleichzeitig ausgelesen werden.

3.5.4 Burst-Zugriff

Beim Burst-Zugriff werden größere zusammenhängende Datenblöcke als ununterbrochenes Bündel kleiner Dateneinheiten übertragen. Dies ergibt eine verbesserte Performance.

Man nutzt dabei die Aufteilung des Speichers in Zeilen und Spalten auf: Anstelle jedes Bit mit seiner Adresse von Zeile und Spalte zu adressieren genügt es, einmal die Zeile anzugeben und dann nur die angelegte Spalte zu inkrementieren.

3.5.5 Speicherverschränkung (memory interleaving)

Problem: Vor Zugriff muss bei DRAM-Technologie darauf gewartet werden, bis die alte Zeile zurückgeschrieben wurde. Im schlimmsten Fall führt das dazu, dass die Zugriffszeit immer mindestens der relativ langen Refresh-Zeit entspricht.

Lösung: Interleaving: Speicher wird logisch in Bänke aufgeteilt. Während eine Bank zurückschreibt, kann auf eine andere Bank zugegriffen werden. Bei der Wahl der Bänke nutzt man die räumliche Lokalität aus, benachbarte Daten werden unterschiedlichen Bänken zugeordnet.

3.6 Interrupts

Interrupts sind ein wichtiger Baustein heutiger Computer. Sie ermöglichen es erst, dass eine CPU mit asynchronen Events, wie gedrückten Tasten auf einer Tastatur, umgehen kann.

3.6.1 Arten von Interrupts

Prinzipiell kennt die CPU nur eine Art von Interrupt, das heißt alle im folgenden genannten Arten von Interrupts werden von der CPU gleich behandelt.

Interrupt (auch: Hardwareinterrupt) ist eine asynchrone Programmunterbrechung durch ein externes Gerät.

Trap (auch: Softwareinterrupt) ist ein synchroner Interrupt, der z.B. auf x86 durch den Befehl `int` erzeugt wird.

Exception Fehler, die eine Trap triggern, z.B. eine Division durch 0.

Syscall Ein in Software verursachter Interrupt, der das Betriebssystem zum Ausführen einer gewünschten Funktion.

Auf einem Linuxsystem würde ein Syscall etwa so aussehen:

```
movl $4, %eax
...
int $0x80
```

3.6.2 Interrupt Handler

Interrupts werden direkt nach dem aktuell auf der CPU laufenden Befehl behandelt (Interrupt Handling).

Pointer auf diese Interrupt Handlers finden sich bei x86 in der sogenannten Interrupt Vector Table (IVT). Einige hiervon sind bereits architekturabhängig fest belegt (z.B. Teilen durch 0), die restlichen sind vom Betriebssystem programmierbar.

3.6.3 Sicherung

Da das gerade laufende Programm nichts vom Interrupt wissen kann, muss extra für den Interrupt Handler der Zustand der Register gesichert werden, so dass das Programm nach dem Handling weiterlaufen kann.

- Multi-Purpose-Register werden je nach Bedarf vom Interrupt-Handler selber gesichert.
- Ein minimialer Zustand (Flags, Programmcounter) wird von der Hardware gesichert.

Interrupt-Handler auf x86 verwenden nicht `ret`, sondern das besondere `reti` als Return-Statement.

3.7 Busse

Zur Kommunikation zwischen den verschiedenen Bauelementen im Rechner wird ein Bussystem verwendet. Allgemein besteht ein solches Bussystem aus verschiedenen Leitern unterschiedlicher Kapazität:

- Interrupt
- Memory Read
- Memory Write
- I/O Read
- I/O Write
- Adresse
- Daten

Je nach Anwedungsfall werden nur gewisse Leitungen benötigt.

3.7.1 PCI Express

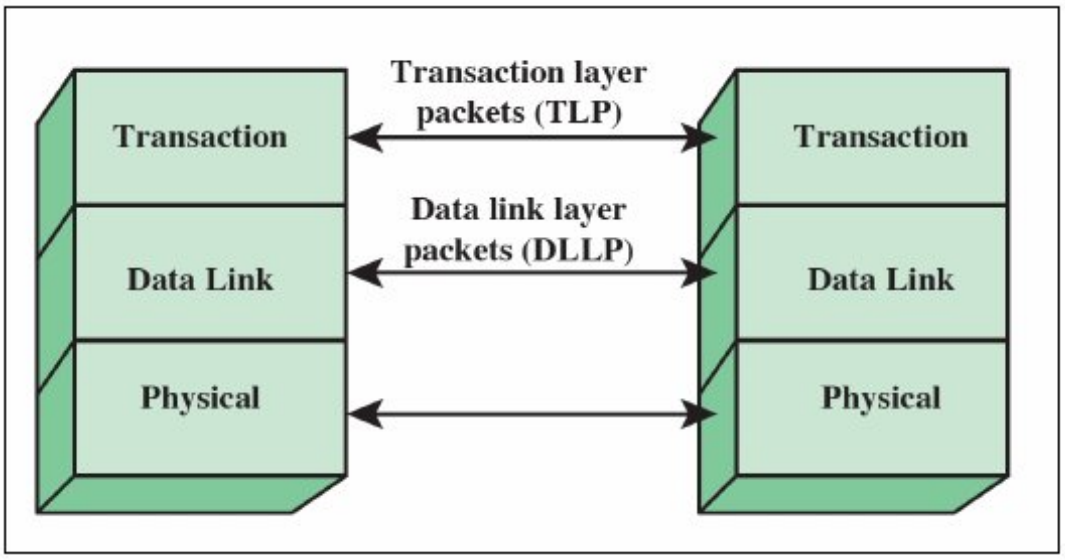


Abbildung 3.5: PCIe besteht aus drei Protokollschichten: Physical, Data Link und Transaction.

Physikalische Schicht Legt elektrische Spezifikation der Signale transportierenden Leitungen fest. Byteströme von verschiedenen Qullen werden gebündelt bzw. augetrennt.

Datenverbindungsschicht Regelt zuverlässige Übertragung zwischen zwei direkten Nachbarn einer physikalischen Verbindung (link)

Transaktionsschicht Erzeugt und verarbeitet Datenpakete fester Länge aus den darüber liegenden Softwareprogrammen um Lade-/Speicheroperationen umzusetzen. Regelt auch den Datenfluss dieser Pakete zwischen Ende-zu-Endepunkt. Gibt Datenpaketen eindeutige Bezeichner.

3.7.2 Unterschied PCI und PCI Express

PCI

- Von Intel als adäquates Bussystem für Pentium entwickelt.
- Bustakt synchron zum CPU-Takt und maximal 33/66 MHz.
- Multiplexing für Adressen und Daten.
- Ausgeklügeltes Bus-Master- und Slave-Prinzip. Ein PCI-Master kann Daten in den Arbeitsspeicher schreiben oder aus ihm lesen, ohne die CPU dafür in Anspruch zu nehmen (DMA-(direct memory access)-Prinzip). Ein Slave dagegen kann nur als Empfänger fungieren (z.B. eine Grafikkarte).

PCIe (PCI Express)

- Heutiger Stand der Technik.
- Höherer Takt.
- Schnellerer Speicher und langsamerer I/O-Controller ersetzt in einem einzigen Bridge genannten Datenpuffer.
- Wenigere, dafür schnell gemultiplexte Leitungen („lanes in links“).

3.7.3 Einbindung von Geräten

Memory-Mapped I/O

Gerät teilt sich mit dem Hauptspeicher einen gemeinsamen Adressraum. Der Zugriff auf das Gerät erfolgt dann einfach durch die normalen Speicheroperationen (z.B. `movl` bei x86).

Wichtig ist hier, dass die Daten des Geräts nicht im Cache zwischengespeichert werden, sondern wieder sofort an das Gerät mitgeteilt werden. Konkret bedeutet das:

Beim Schreiben write-through verwenden.

Beim Lesen Cache vermeiden, non-write-allocate verwenden.

Verwendung bei Grafikkartenspeicher und Festplattenbuffer.

Isolated I/O

Getrennte Adressräume für Gerät und Hauptspeicher. Zugriff erfolgt über spezielle I/O-Befehle (z.B. `in` und `out` bei x86).

Verwendung etwa bei der Konfiguration der Tastatur (APIC, advanced programmable interrupt controller).

3.7.4 Datentransfer

Verschiedene Geräte transferieren Daten, damit die CPU dann damit arbeiten kann. Spricht man vom Datentransfer, so meint man damit die Einheit aus *Bereitschaft prüfen* und dem *tatsächlichen Transfer* der Daten.

Programmierte I/O

- Es werden explizite Befehle (`load`, `store`) von der CPU ausgeführt.
- So wird sowohl auf Bereitschaft geprüft, als auch der Datentransfer ausgeführt.

Unterbrechnungsgesteuerte I/O

- Interrupt benachrichtigt das Betriebssystem von der Bereitschaft eines Gerätes, z.B. nach dem Auslösen einer Taste auf der Tastatur.
- So kann nur die Bereitschaft signalisiert werden! Der eigentliche Transfer muss anders erfolgen.

DMA: direct memory access

- CPU programmiert den DMA-Controller.
- DMA kopiert dann als alleinstehende Hardware Daten in den Speicher, kümmert sich also um die Übertragung. Währendessen kann sich die CPU um andere Dinge kümmern, da auch der Umweg HDD → CPU (Register) → RAM entfällt.
- DMA signalisiert dann Fertigstellung durch einen Interrupt.

Beispiele

Tastatur auslesen Interrupt → P I/O

Festplatte P I/O → Interrupt → DMA

Netzwerkkarte Interrupt → DMA

3.8 Prozessoren

Heute finden sich größtenteils zwei Arten von Prozessoren in den meisten Rechnern: CISC und RISC.

CISC (complex instruction set computing)

- Typsiche Beispiele: x86, Motrolla 68000
- Befehle sind unterschiedlich lang
- Mikroprogrammiert
- Können sehr komplexe Befehle verarbeiten (AES-Erweiterung bei x86)
- Wenige Register
- Register-Memory
- Komplexe Hardware
- Pipelining nur über Umwege möglich

RISC (reduced instruction set computing)

- Typsiche Beispiele: MIPS, ARM
- Alle Befehle sind gleich lang
- Die Logik der Chips ist „fest verdrahtet“
- Unterstützt nur elementare Befehle
- Viele Register (viel Platz auf dem Chip)
- Register-Register
- Komplexe Compiler
- Pipelining ist „direkt möglich“

3.9 Pipelining

Pipelining: pseudoparallele Verarbeitung eines sequentiellen Instruktionsstroms.

Fallstrick: die langsamste Stufe der Pipelie (meisten die ALU) bestimmt den Takt der ganzen Pipeline.

3.9.1 Asymptotischer Speedup

Sei n die Anzahl der Befehle, k die Anzahl der Pipelinestufen und τ die Taktdauer. So ergibt sich der asymptotische Speedup S_k wie folgt:

$$S_k = \lim_{n\to\infty} \frac{n \cdot k \cdot \tau}{(n + (k-1)) \cdot \tau} = k$$

$(k-1)$ entspricht der Einschwingphase der Pipeline. Eben erst dann erreichen die ersten Befehle das Ende der Pipeline.

3.9.2 Warum keine unendlich lange Pipeline?

- Hardwarekosten
- Latenz der Hardware
- Data- u. Control-Hazards (räumliches Lokalitätsproblem)
- Einschwingphase

3.9.3 Hazards

Datenhazards (Datenabhängigkeiten) Ein Befehl benötigt das Ergebnis eines vorherigen Befehls. Dessen Ergebnis wurde aber noch nicht in die Register zurückgeschrieben.

Controlhazards (Steuerungshazards, Verzweigungen/Sprünge) Nächster Befehl in Pipeline ist nicht der Richtige.

3.9.4 Forwarding

Beim Forwarding werden Ergebnisse der ALU direkt wieder an den Eingang der ALU gelegt. So kann man Data-Hazards entgegenwirken.

3.9.5 Instruction Level Parallelism (ILP)

ILP Maß dafür, wie viele Instruktionen eines sequentiellen Instruktionsstroms ausgeführt werden können.

Beispiel

Gegeben sei folgender Pseudocode mit insgesamt drei Instruktionen.

```
a = x + y // independent instruction
b = f * g // independent instruction
c = a + b // depends on the previous two instructions
```

Die letzte Instruktion ist von den vorherigen beiden Instruktionen abhängig. Der ILP errechnet sich aus der Anzahl der gesamten Instruktionen geteilt durch die Anzahl der parallel ausführbaren Instruktionen. Konkret im Beispiel ist das Ergebnis also $ILP = 3/2$.

3.10 Superskalare Ausführung

- Superskalar steht für eine echt-gleichzeitige Ausführung von mehreren Befehlen eines einzelen sequentiellen Instruktionsstroms.
- Es gibt hier noch keine unterschiedlichen Threads!
- Benötigt mehrere Rechenwerke.
- Mehrere Befehle werden gleichzeitig geholt/decodiert.

3.10.1 Statisch und Dynamisch

statisch Superskalare Eigenschaften werden durch den Compiler erzeugt. Beispiele: VLIW.

dynamisch Superskalare Eigenschaften werden durch die Hardware realisiert. Beispiele hierfür ist Pipelining, out-of-order execution und allgemein Superskalare Architekturen.

3.10.2 Superskalare Architekturen

Superskalare Architekturen sind Architekturen, die Maschinencode automatisch superskalar ausführen ($\rightarrow$ Dynamisch).

- Insturktionssatz ändert sich nicht.
- Programmierer muss keine Änderungen am Code vornehmen und nicht explizit synchronisieren o. Ä.

3.10.3 Very Long Instruction Word (VLIW)

- Mehrere Rechenwerke.
- Ein Befehlswort mit merheren Teilbefehlen für die verschiedenen Rechenwerke.
- Compiler muss natürlich entsprechend optimieren.

Programmierung

Die Programmierung für VLIW-Architekturen ist für den Hochsprachenprogrammierer gleich. Nur die vom Compiler erzeugten Maschinenprogramme müssen entsprechend angepasst sein.

Insbesondere muss ein Compiler für VLIW-Architekturen einen Datenabhängigkeitsanalyse („Gruppierung“) durchführen. Ist es nicht möglich, ein ganzes langes Instruktionswort zu füllen, wird mit `NOP` aufgefüllt.

3.11 Multithreading

Multithreading (MT) „Gleichzeitige“ Ausführung von Befehlen aus mehreren Instruktionsströmen.

3.11.1 Arten von Multithreading

Zeitscheiben-MT Hardware schaltet zyklisch zwischen den Threads um (analog zum preämtiven Scheduling, aber weniger Overhead).

Es läuft in einem Moment tatsächlich nur genau ein Thread, also ein sequentieller Instruktionsstrom.

Ereignisgesteuertes MT Umschaltung, wenn aktueller Thread blockiert, z.B. beim Zugriff auf Eingabegeräte (allgemein: I/O). Dadurch sollen Latenzen verborgen werden und so eine allgemein bessere Perfomanz erreicht werden.

Es läuft in einem Moment tatsächlich nur genau ein Thread, also ein sequentieller Instruktionsstrom.

Simulatanes MT (Intel: *Hyperthreading*) Instruktionen unterschiedlicher Threads werden gleichzeitig ausgeführt.

- Benötigt eine superskalare CPU.
- Bessere Auslastung der verschiedenen Ports einer CPU (Haswell von Intel)
- Funktioniert allgemein sehr gut, da die verschiedenen Threads voneinander unabhängig sind.
- Mehrere Threads sind also gleichzeitig aktiv. Sind n Threads aktiv bedarf es deswegen auch n mal alle direkt adressierbare Register.

3.11.2 OpenMP

OpenMP ist eine Anwender-Programmierschnittstelle (API) für Speicher-gekoppelte Multiprozessoren.

- Eignet sich sehr gut, um einfach CPUs mit simultanem Multithreading (SMT) auszunutzen.
- Folgt dem Paradigma Single Instruction, Multiple Data (SIMD).

3.12 Zugriffsschutz

Absichtliches und unabsichtliches Fehlverhalten eines Prozesses soll nicht das komplette System beeinträchtigen. Deswegen verfügen heutige CPUs über Zugriffsschutz.

3.12.1 Arten von Zugriffsschutz

horizontaler Zugriffsschutz Abschottung der verschiedenen Prozesse voneinander.

vertikaler Zugriffsschutz Prozess wird gegenüber dem Betriebssystem abgeschottet. → ermöglicht erst den horizontalen Zugriffsschutz.

Rechteverwaltung Manche Teile des Speichers dürfen z.B. nur gelesen werden; analog zur Rechteverwaltung in Unix (`rwx`).

3.12.2 Privilegierungsstufen

Prozesse laufen in verschiedenen Privilegierungsstufen (engl. „Ring“). Entsprechend nach Ring ist der Zugriff auf Speicher, I/O und besondere Insturktionen evtl. eingeschränkt. Insbesondere ist es nur priviligierten Prozessen erlaubt, Befehle wie `cli` (clear interrupts) oder `sie` (set interrupts enabled) zu verwenden.

→ Notwendig, um Speicherschutz (und mehr) zu realisieren.

Der Wechsel zwischen den Ringen erfolgt immer bei Interrupt, Exception, Syscall.

Obwohl x86 insgesamt vier Ringe vorsieht (Ring 0, Ring 1, Ring 2, Ring 3) verwendet Linux nur Ring 0 für den Kernel und Ring 3 für Prozesse im Userspace.

3.13 Paging

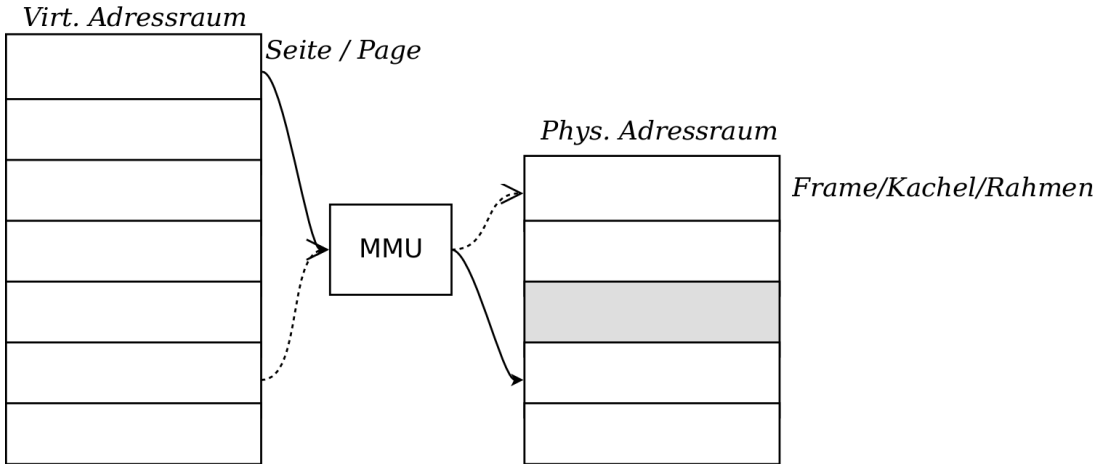


Abbildung 3.6: Die hier grau dargestellte Kachel ist nicht tatsächlich mit Hauptspeicher belegt. Stattdessen ist sie z.B. für memory mapped IO reserviert.

Beim Paging wird zuerst zwischen virtuellen und physikalischen Adressraum unterschieden. Alle Programme arbeiten mit virtuellen Adressen, die dann von der Hardware, speziell der MMU, in physikalische Adressen übersetzt werden.

- Alle Pages und Kacheln sind gleich groß, gewöhnlicherweise 4 KiB.
- Virtueller und physikalischer Adressraum müssen nicht gleich groß sein. Auf heutigen 64-Bit-Maschinen ist der virtuelle Adressraum deutlich größer als der physikalische.

3.13.1 Tabellen

Das Mapping von virtuellen auf physikalischen Adressen wird durch eine Hierarchie von Tabellen, speziell für jeden Prozess eine eigene, realisiert. So wird Speicherschutz durch Paging realisiert: Anderen Prozessen ist es gar nicht möglich, auf die physikalische Adresse zuzugreifen, auf der sich unser Prozess befindet.

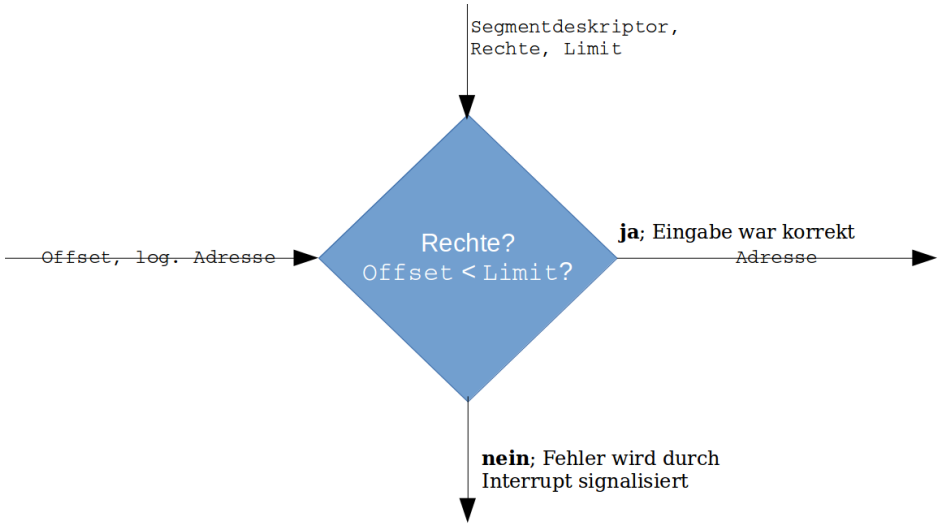
Problem: Sehr viele Speicherzugriffe auf die Tabelle, im schlimmsten Fall mit jedem einzelen Speicherzugriff. Gelöst wird das Problem durch einen zusätzlichen Cache, dem TLB (Tranlsation Lookaside Buffer).

3.13.2 Segementierung

Segment Zusammenhänder Speicher variabler Größe.

- Adressierung erfolgt über „Segmentname“ (ein Bitstring, explizit oder implizit) und einem Offset.

3.13.3 Speicherzugriff bei der Segmentierung



1. Check der Zugriffsrechte des Segments.
2. Check ob Adresse wirklich im Segment liegt. Konkret wird hier das Offset auf Richtigkeit überprüft.

3.13.4 Caches im Zusammenhang mit Paging und Segmentierung

Physikalischer Cache

- Übersetzung erfolgt vor dem Cache-Zugriff.
- Zu einem Zeitpunkt können Blöcke verschiedener Prozess im Cache gespeichert sein.
- Rechte sind „direkt überprüfbar“.
- Für jeden Zugriff muss die MMU bemüht werden, was einige Zeit in Anspruch nehmen kann.

Virtueller Cache

- Übersetzung erfolgt nach dem Cache-Zugriff.
- Schnell, da die MMU weniger ausgelastet wird.
- Virtuelle Adressen überlappen sich! Aus diesem Grund muss beim Prozesswechsel jedes mal der Cache geflusht werden.