Architektur und Programmierung von Grafik- und Koprozessoren

Performanz von Computerprogrammen

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

Speicherhierarchien und Caches

Speicherhierarchien

Capacity / Access Time

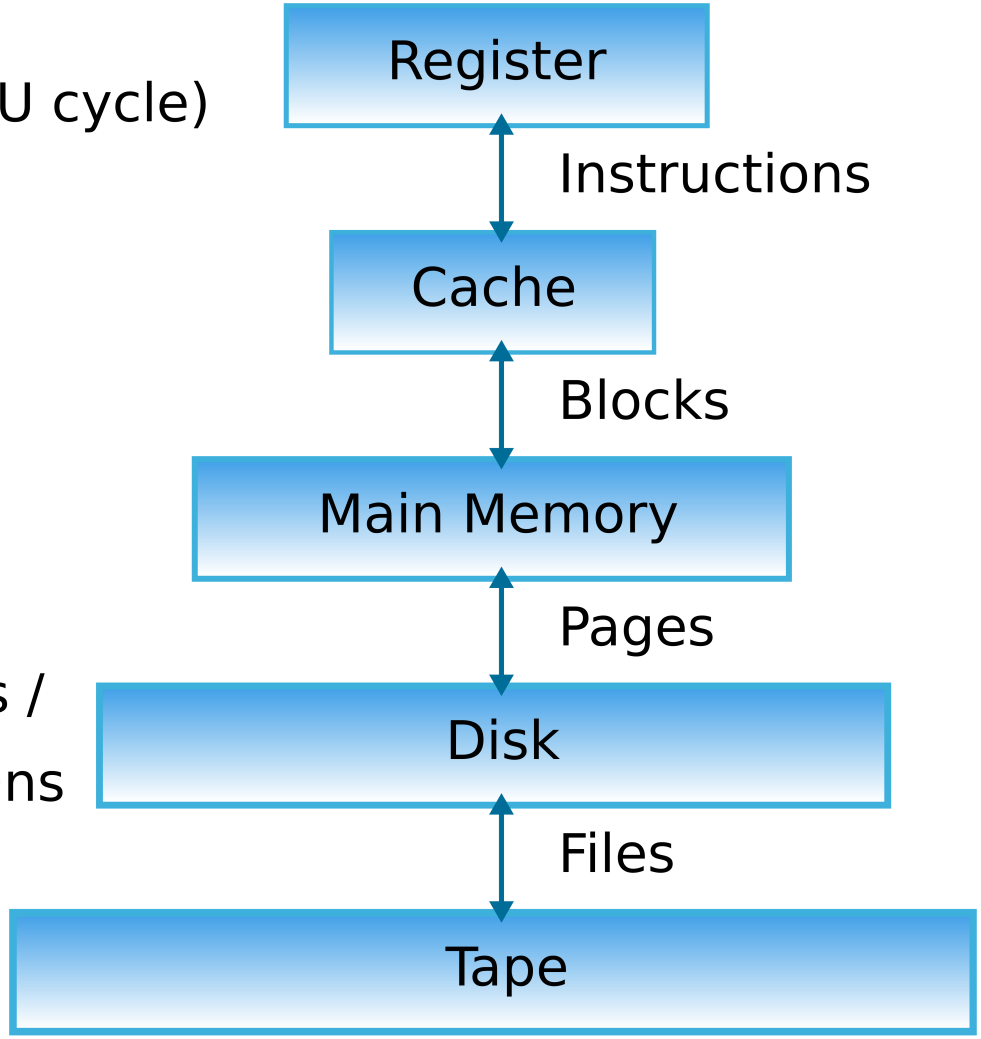
100s Bytes /
< 0.5 ns (typ. 1 CPU cycle)

10s-100s Kbytes
1-5ns

Few Gbytes /
50-150ns

100s-1000s Gbytes /
500.000-1.500.000ns

Infinite /
sec.-min.



Speicherhierarchien

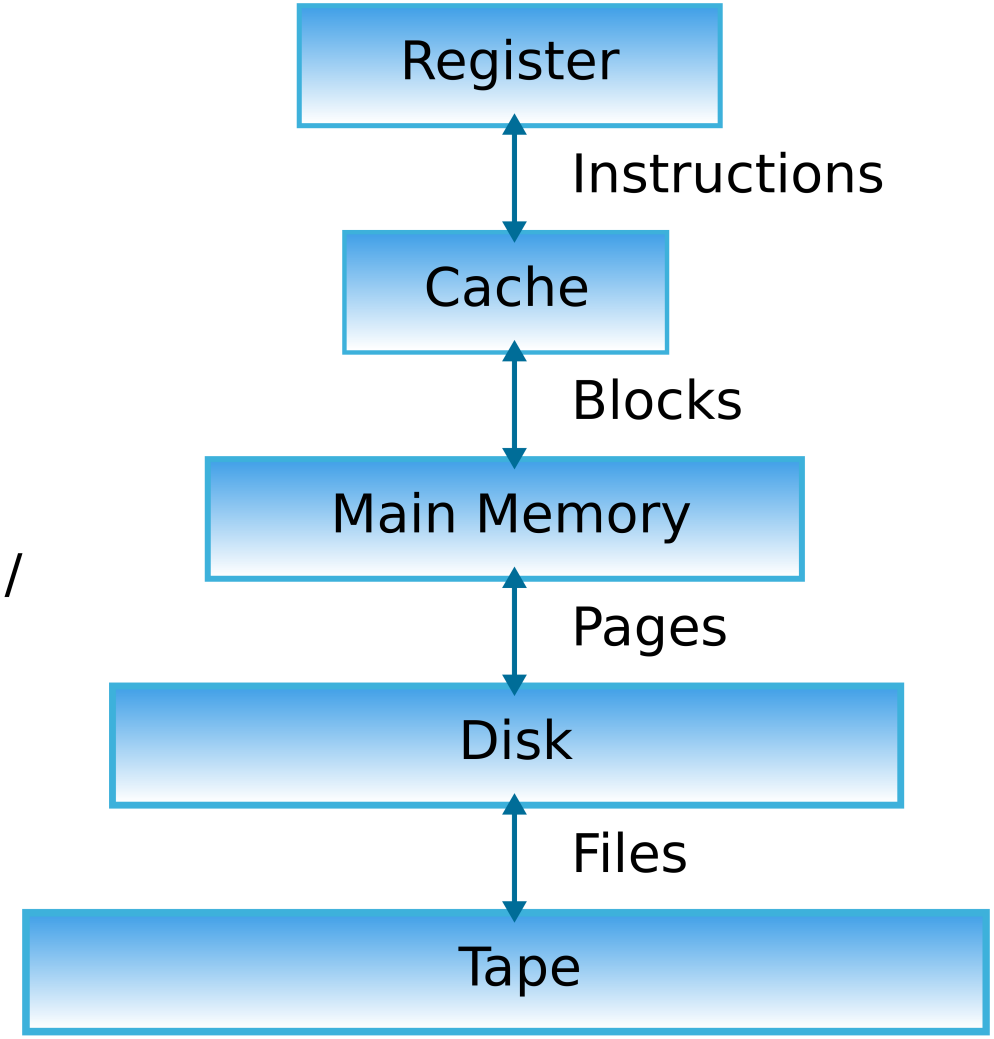
Staging / Transfer Unit

Compiler & CPU /
1-8 Bytes

Cache Ctrl. /
8-128 Bytes

Operating System /
512-4096 Bytes

User / Operator /
Mbytes



Cache Hierarchien

Lokalität

- Räumliche Lokalität: wenn ein Programm auf eine Speicherstelle zugreift, ist die Wahrscheinlichkeit hoch, dass es unmittelbar danach auf **benachbarte** Speicherstellen zugreift.
- Temporale Lokalität: wenn ein Programm auf eine Speicherstelle zugreift, ist die Wahrscheinlichkeit hoch, dass es **später** wieder auf die gleiche Speicherstelle zugreift.
- Instruktionslokalität: **Instruktionen**, die ein Programm ein Mal ausführt, werden mit hoher Wahrscheinlichkeit später wieder ausgeführt.
- Lokalität kann u. a. durch die geeignete Wahl von Algorithmen beeinflusst werden.

Cache Hierarchien

- eine Alternative zur Erhöhung der Taktfrequenz ist es, mit mehr Transistoren größere Caches in den Prozessor einzubauen.
- Herausforderungen für den Programmierer durch größere Caches werden im folgenden kurz erläutert.
- Cache Hierarchien spielen auch im Zusammenhang mit Multi-Core und Many-Core Systemen eine Rolle. Diese Aspekte werden im weiteren Verlauf der Vorlesung intensiver behandelt.

Cache Hierarchien

- in Computersystemen sind verschiedene Arten von Speicher verbaut.
- z. B. statischer Speicher sehr schnell, dafür teuer in der Herstellung. Computersysteme verbauen daher wenig statischen Speicher mit enger Anbindung an Prozessor, und viel langsameren, dynamischen Speicher als Hauptspeicher.
- Caching: halte häufig genutzte Daten in statischem Speicher vor, halte weniger häufig genutzte Daten in dynamischem Speicher (z. B. DDR) vor.
- Problem: welche Daten werden häufig benutzt, und welche nicht.

Zwei Arten von RAM

Links: Statische Speicherzelle, rechts: dynamische Speicherzelle

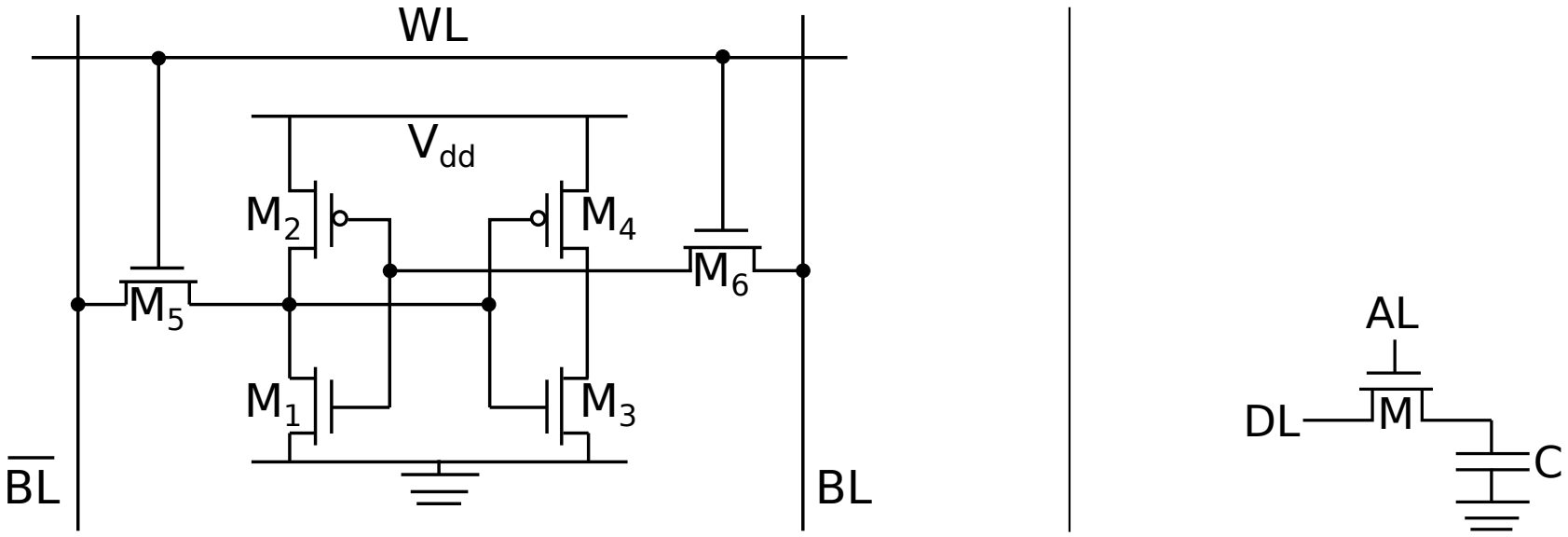
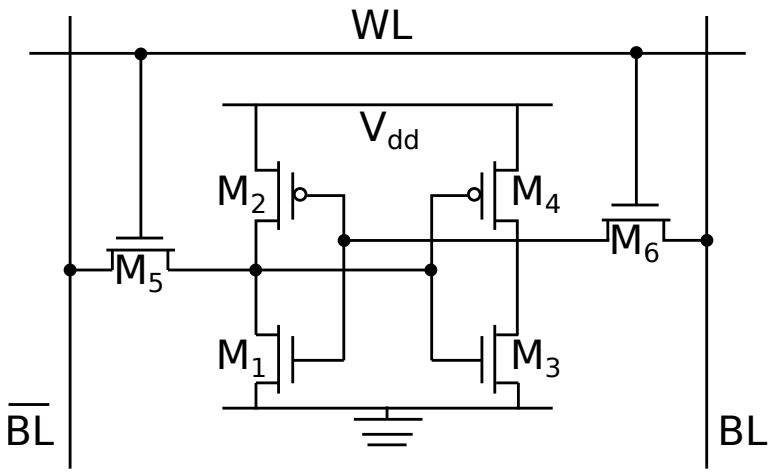


Abbildung: vgl. Drepper, What Every Programmer Should Know About Memory (2007) und auch `http://www.graphics.stanford.edu/courses/cs448a-01-fall/lectures/dram/dram.2up.pdf`

Statischer RAM

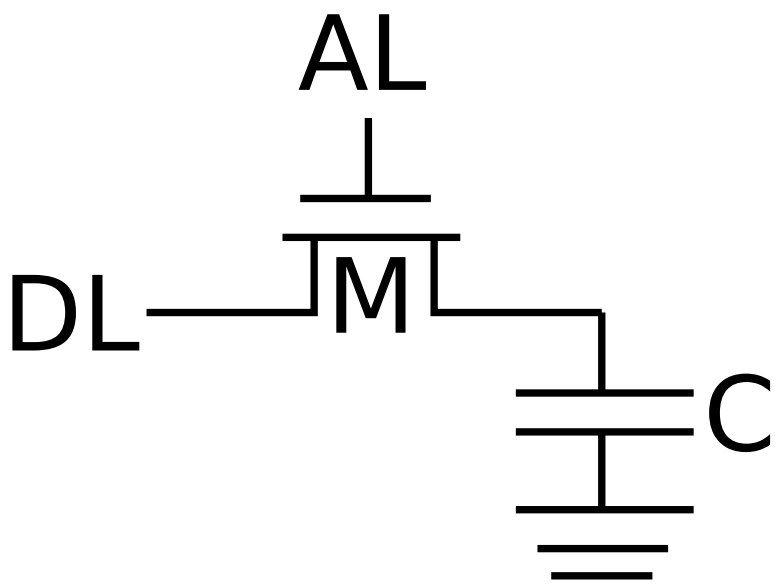


- Zwei Zustände (0 oder 1). "High-signal" auf WL ("word access line") macht Zustände auf BL und $\overline{BL}$ **unmittelbar verfügbar**.
- Benutzt "cross-coupled inverter" Logik (flip-flop) um Zustand zu halten.
- **Sechs Transistoren** ($M_1..M_6$).
- **Permanente Eingangsspannung** (V_{dd}) nötig.

Statischer RAM

Statischer RAM ist **teuer** in **Herstellung** und **Betrieb**. Daher wird er i. d. R. für CPU-nahe Caches verbaut.

Dynamischer RAM



- Besteht nur aus **einem Transistor** und **einem Kondensator**.
- Kondensator bestimmt den Zustand (geladen oder nicht geladen).
- Lege "High-Signal" an "access line" AL an. Je nach Ladung des Kondensators fließt Strom durch "data line" DL.

Dynamischer RAM

Nachteile dynamischer Speicherzellen (1/2)

- Leakage: nach gewisser Zeit muss neu geladen werden (*Refresh Cycle*).
- Während des Refresh Cycles kann nicht von der Speicherzelle gelesen werden.
- Wegen der hohen Anzahl an Speicherzellen dürfen die einzelnen Kondensatoren nur niedrig geladen sein (sie entladen sich also noch schneller).

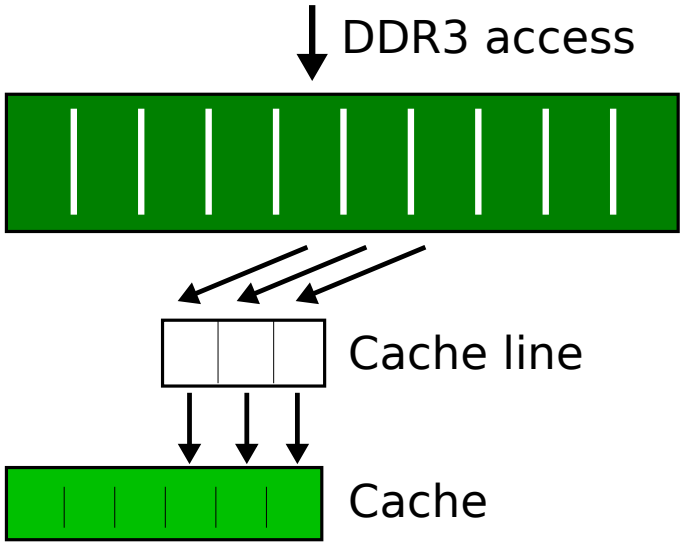
Dynamischer RAM

Nachteile dynamischer Speicherzellen (2/2)

- Kondensator verliert beim Lesevorgang seine Ladung.
- Da Spannung am Kondensator sehr niedrig, kann das Signal nicht unmittelbar gelesen werden, sondern muss erst verstärkt werden.
- Lade- und Entladezeiten von Kondensatoren nicht instantan.
  - Im Vergleich: SRAM Signal rechteckig.
  - Bei DRAM muss konservativ geschätzt werden, wann nach Ladevorgang das Signal stark genug ist.

Caches und Speicherzugriffe

Computer halten daher immer häufig genutzte Daten in Cache Speicher vor. Wird auf ein Datum zugegriffen, werden, um Latenz zu vermeiden, gleich mehrere benachbarte Daten mit in den Cache geholt (vgl. Lokalität). Dies passiert über *Cache Lines* (syn.: Cache Blöcke). Typische Cache Line Größe: 64 Bytes.



Cache Assoziativität

Cache Blöcke speichern (im Falle assoziativer Caches) das *Cache Set*, zu dem sie gehören, sowie einen Offset in das Cache Set.

Über Cache Assoziativität wird eine Zuordnung hergestellt, welche Speicheradressbereiche (Hauptspeicher) an welche Cache Adressen geschrieben werden dürfen.

Sucht man im Cache, findet man die richtige Adresse also mittels Offset und Set Index. Über das *Tag* kann man herausfinden, ob *Set* [*Offset*] sich auf die Hauptspeicheradresse bezieht, nach der man sucht.

Cache Assoziativität

Set Assoziativität

Cache ist in *Sets* unterteilt. Bestimmte Speicheradressen mappen in bestimmte Sets.

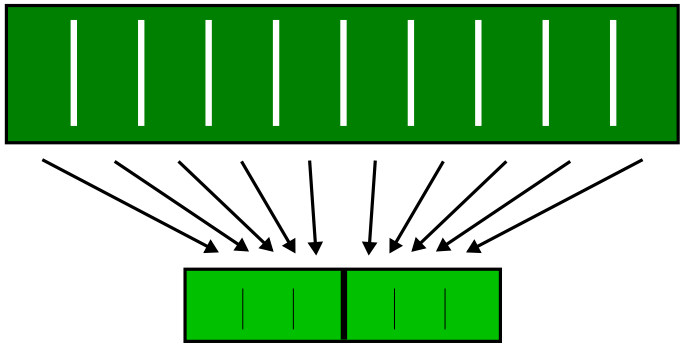


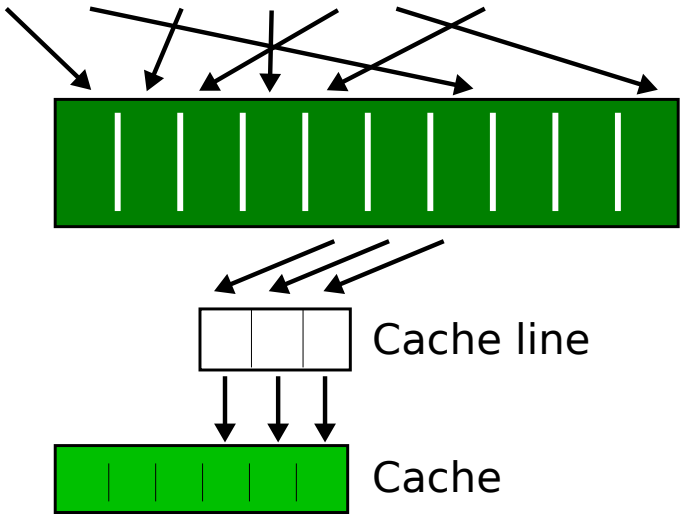
Abbildung: 3-Way Cache Associativity (in jedes Set passen drei Einträge).

Cache Assoziativität

Assoziativität wirkt sich auf Hardware Komplexität und Zugriffsgeschwindigkeit aus.

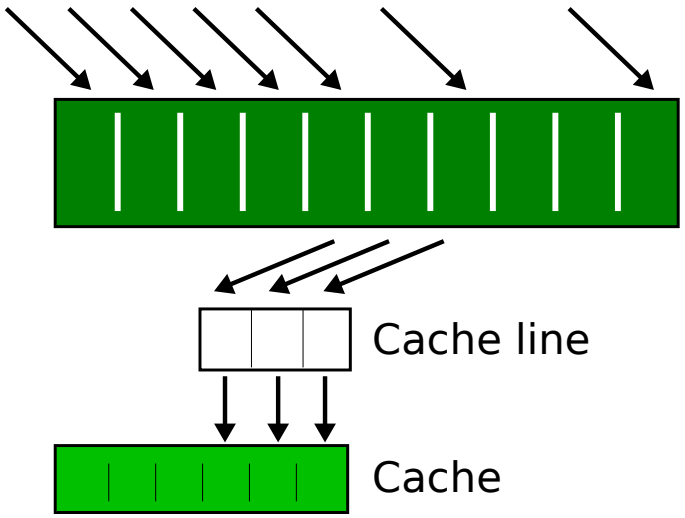
- **Fully Associative** - jeder Cache Block kann überall in den Cache geschrieben werden. Keine Cache Sets, lineare Suche über alle Einträge. Nur für sehr kleine Caches sinnvoll. (z. B. *translation lookaside buffer* (TLB) bei manchen Prozessoren.)
- **4-Way oder 2-Way Set Associative** - Sets, in die jeweils vier oder zwei Blöcke passen.
- **Direct Map** - jeder Cache Block kann an genau einer fest dafür vorgesehenen Stelle abgelegt werden $\Rightarrow$ sehr schnelles Finden. Problem: erfordert Multiplexer - desto größer der Cache, umso größer der Multiplexer. Langsam wegen Hardware Komplexität.

Kohärenz von Speicherzugriffen



Problem: Algorithmen mit inkohärenten Speicherzugriffsmustern. Cache Line wird häufig Daten enthalten, die später nicht gebraucht werden.

Kohärenz von Speicherzugriffen



Versuche Algorithmen so umzuformulieren, dass Speicherzugriffe kohärent.

Kohärenz von Speicherzugriffen

- Häufig nicht trivial, Algorithmen bzgl. kohärenter Speicherzugriffe umzuformulieren.
- Geht oft mit einem Sortierschritt und *Compaction* einher, um Indirektion aufzulösen.
  - Manchmal Sortieren teurer als Cache Probleme.
  - Individuelles Profiling nötig.

Cache Größe Historische Entwicklung

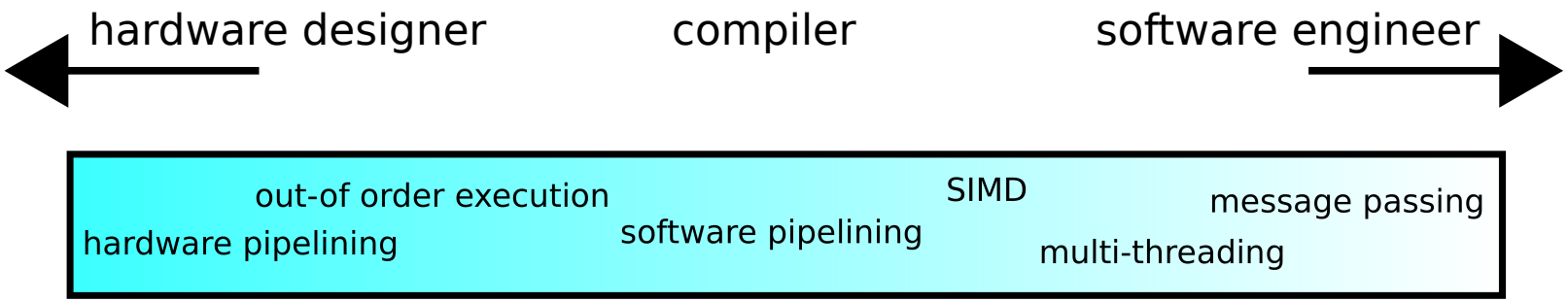
Name	Jahr	L1-Cache	L2	L3	Kerne
80286	1982	–	–	–	1
80386	1985	–	–	–	1
80486	1989	8KB	–	–	1
Pentium	1993	8 + 8KB	–	–	1
Pentium Pro	1997	8 + 8KB	1024KB	–	1
Pentium IV	2001	8KB	256KB	–	1
Pentium IV	2004	16KB	1024KB	–	1
Core 2	2007	32 +32KB	2 x 4MB	–	4
Core i5	2010	32 + 32KB	2 x 256KB	4MB	2
...					
Core i9 X	2018	18 x 512+512KB	16MB	24MB	18

Nebenläufigkeit und Parallele Architekturen

Parallele Architekturen

- Parallele Verarbeitung hat bei Prozessoren schon immer eine wichtige Rolle gespielt.
- *Instruktionslevelparallelismus* (ILP) tangiert jedoch weniger den Programmierer, sondern mehr Hardware Designer und Compiler Entwickler (der Programmierer sollte dennoch immer die Charakteristiken der Target Hardware kennen).
- SIMD Extensions und Multi-Core Architekturen erfordern explizitere Programmiermodelle.
- High-Performance Computing: traditionell Parallelprogrammierung (Vektorrechner, Shared Memory, Message Passing). Programmiermodelle für Commodity Architekturen orientieren sich teils daran.

Arten von Parallelismus



Pipelining

Pipelining

Pipelining ist eine ILP Technik, die omnipräsent bei der Implementierung von Prozessorarchitekturen ist. Mehrere Instruktionen werden dabei überlappend wie bei der Bearbeitung mit Hilfe von Fließbändern ausgeführt. Mittels Pipelining kann der *Instruktionsdurchsatz* einer CPU / eines Prozessors i. Allg. erhöht werden.

Pipelining findet sich auch auf höherem Abstraktionsniveau als im Rahmen von ILP, z. B. in der später noch zu behandelnden Grafikpipeline. Daher ist es wichtig, Pipelining konzeptionell zu verstehen.

Software Pipelining Beispiel (1)

```
void saxpy(float a, float* x, float* y, int N) {
    for (int i = 0; i < N; ++i)
        y[i] = a * x[i] + y[i];
}
```

Nehmen wir an, dass die Schleife in folgende (Pseudo-)Instruktionsabfolge übersetzt wird.

```
LD   ; y[i],y[i]
MUL  ; a * x[i]
ADD  ; (..) + y[i]
ST   ; y[i] = (..)
```

Software Pipelining Beispiel (2)

```
LD   ; y[i],y[i]
MUL  ; a * x[i]
ADD  ; (..) + y[i]
ST   ; y[i] = (..)
```

Naiv würde die Instruktionsfolge wie folgt von der CPU ausgeführt.

```
LD MUL ADD ST
               LD MUL ADD ST
                             LD MUL ADD ST
                                           LD MUL ADD ST
```

Software Pipelining Beispiel (3)

```
LD   ; y[i],y[i]
MUL  ; a * x[i]
ADD  ; (..) + y[i]
ST   ; y[i] = (..)
```

Auf einer hypothetischen Maschine, auf der arithmetische sowie Speicherzugriffsinstruktionen jeweils einen Taktzyklus benötigen, und auf der beide Instruktionstypen überlappend ausgeführt werden können, kann die Instruktionsfolge "*gepipelined*" werden:

```
LD MUL ADD ST
       LD  MUL ADD ST
               LD  MUL ADD ST
                       LD  MUL ADD ST
```

Software Pipelining Beispiel (4)

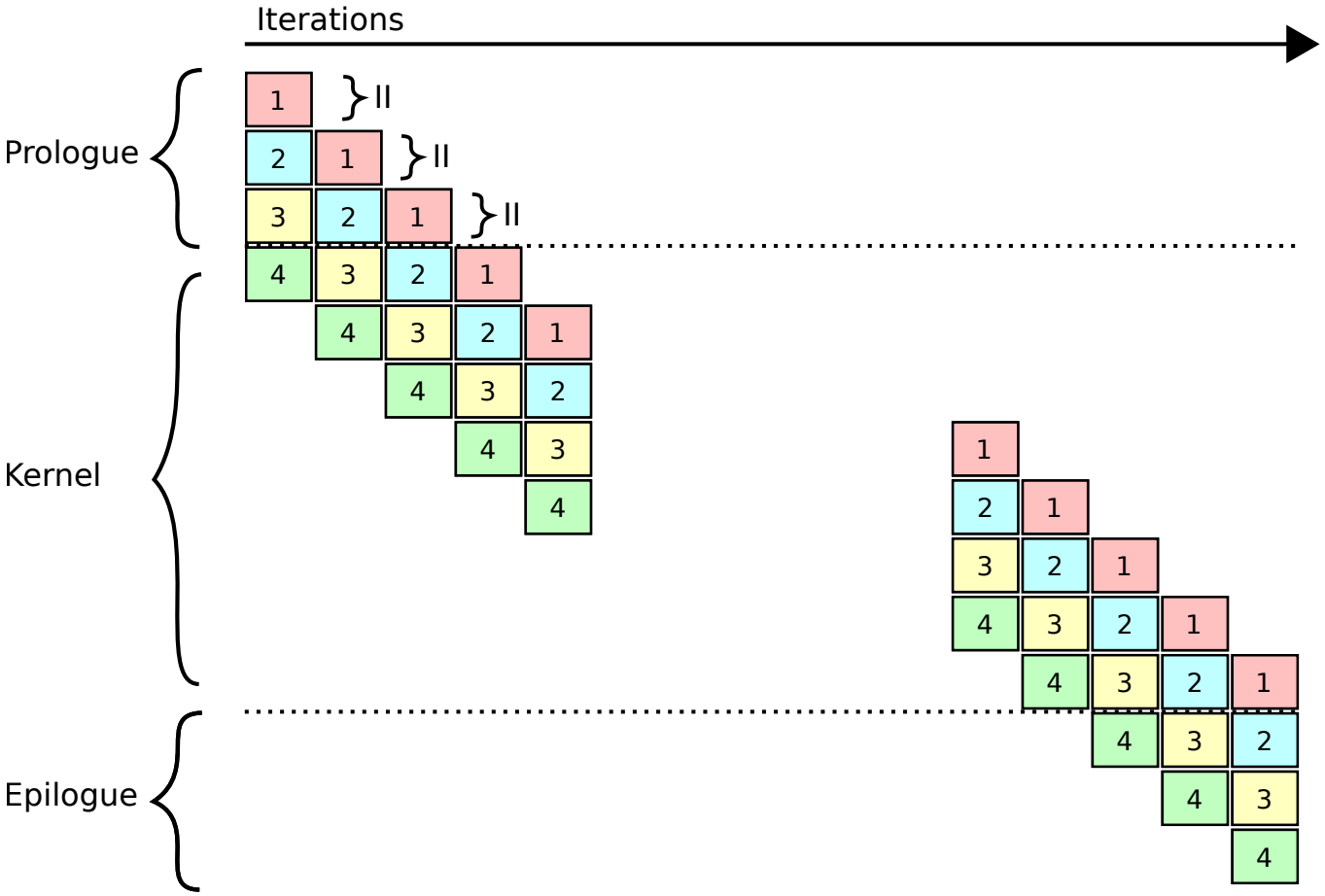
```
LD   ; y[i],y[i]
MUL  ; a * x[i]
ADD  ; (..) + y[i]
ST   ; y[i] = (..)
```

Auf einer hypothetischen Maschine, auf der arithmetische sowie Speicherzugriffsinstruktionen jeweils einen Taktzyklus benötigen, und auf der beide Instruktionstypen überlappend ausgeführt werden können, kann die Instruktionsfolge "*gepipelined*" werden:

```
LD MUL ADD ST
       LD  MUL ADD ST
               LD  MUL ADD ST
                       LD  MUL ADD ST
```

Der $i + 1$te Schleifendurchlauf kann also erst anlaufen, sobald die ersten beiden Instruktionen des $i$ten Schleifendurchlaufs ausgeführt wurden. Man spricht von einem *Initiation Interval* (II) von *zwei*.

Software Pipelining Begriffe



Software Pipelining Bemerkungen

- Die Zeit für eine Operationsabfolge (im Beispiel: $y_i = a_i \times x_i + y_i$) bleibt die gleiche.
- Es erhöht sich jedoch der *Durchsatz* für große N.
- Die Zeitkomplexität hängt vom Initiation Interval ab, dem Zeitinterval, das zwischen zwei Schleifendurchläufen durchlaufen wird.
- Software Pipelining wird als Teil der Optimierungsphase von modernen Compilern durchgeführt. Dabei wird die CPU Architektur berücksichtigt, für die kompiliert wird.
- Pipelining passiert automatisch. Aber: als Programmierer, und auf Architekturen mit tiefer Pipeline sollte man Sorge tragen, dass die Anzahl Iterationen groß genug ist, sodass Schleifendurchführung nicht durch Prolog und Epilog dominiert wird.

Software Pipelining als Optimierungsproblem

Das Software Pipelining Problem lässt sich formulieren als:

- Finde einen Instruktionsablaufplan, der das Initiation Interval minimiert,
- der außerdem gültig ist:
- bzgl. der zur Verfügung stehenden Ressourcen.
- bzgl. Datenabhängigkeiten zwischen Instruktionen.

Software Pipelining als Optimierungsproblem

Datenabhängigkeiten

Datenabhängigkeiten bestehen, wenn Instruktionen von Ergebnissen vorheriger Instruktionen abhängen. Es ergeben sich also Teilinstruktionsablaufpläne aufgrund von Datenabhängigkeiten.

Ressourcenverfügbarkeit

Instruktionsablaufpläne müssen sich auf die zur Verfügung stehenden Ressourcen beziehen - z. B. ist die Anzahl an ALUs architekturbedingt begrenzt, sodass etwa nur eine limitierte Anzahl arithmetischer Operationen gleichzeitig durchgeführt werden kann.

Datenabhängigkeitsgraphen

Wir betrachten die Datenabhängigkeit in Schleifen. In der folgenden Schleife gibt es eine *Intra Iterations*-Datenabhängigkeit zwischen y_i, a_i und x_i.

```
for (int i = 0; i < N; ++i)
    y[i] = a * x[i] + y[i];
```

Die folgende Schleife illustriert *Inter Iterations*-Datenabhängigkeiten.

```
for (int i = 2; i < N; ++i)
    y[i] = a * x[i-1] + y[i-2];
```

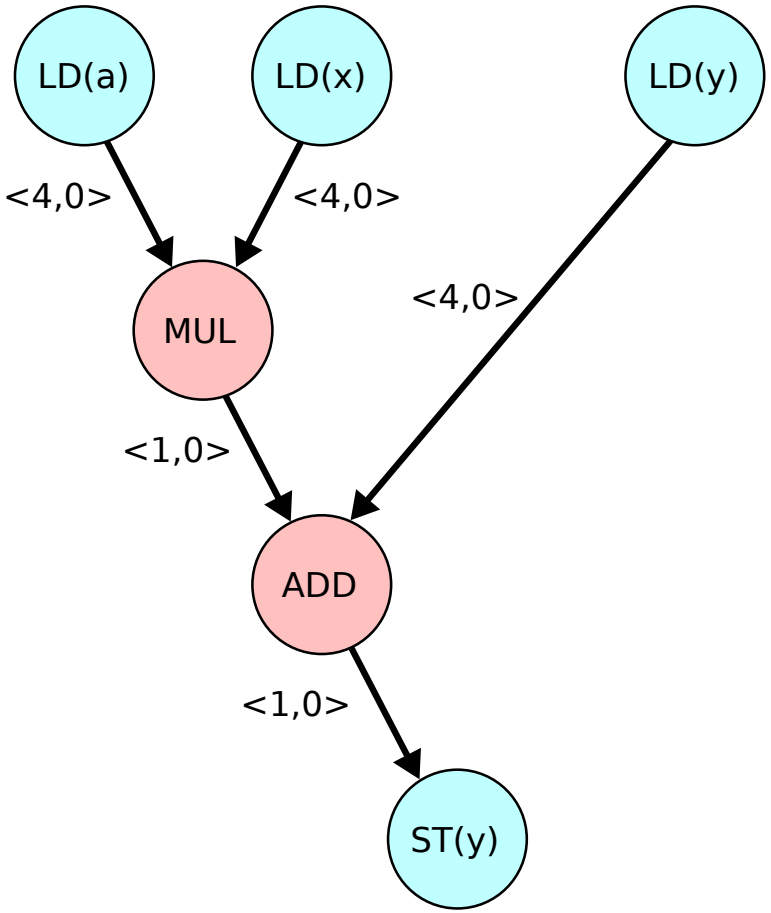
Der Wert von y_i hängt von vorherigen Schleifeniterationen ab, mit einer *Distanz* von 2.

Datenabhängigkeitsgraphen

Im Datenabhängigkeitsgraphen (DDG) werden Instruktionen als Knoten modelliert. Gerichtete Kanten modellieren Datenabhängigkeiten. Kanten werden beschriftet: <Delay,Distanz>. *Delay* bezeichnet die Ausführungslatenz (Taktzyklen) der jeweiligen Instruktion.

Datenabhängigkeitsgraphen

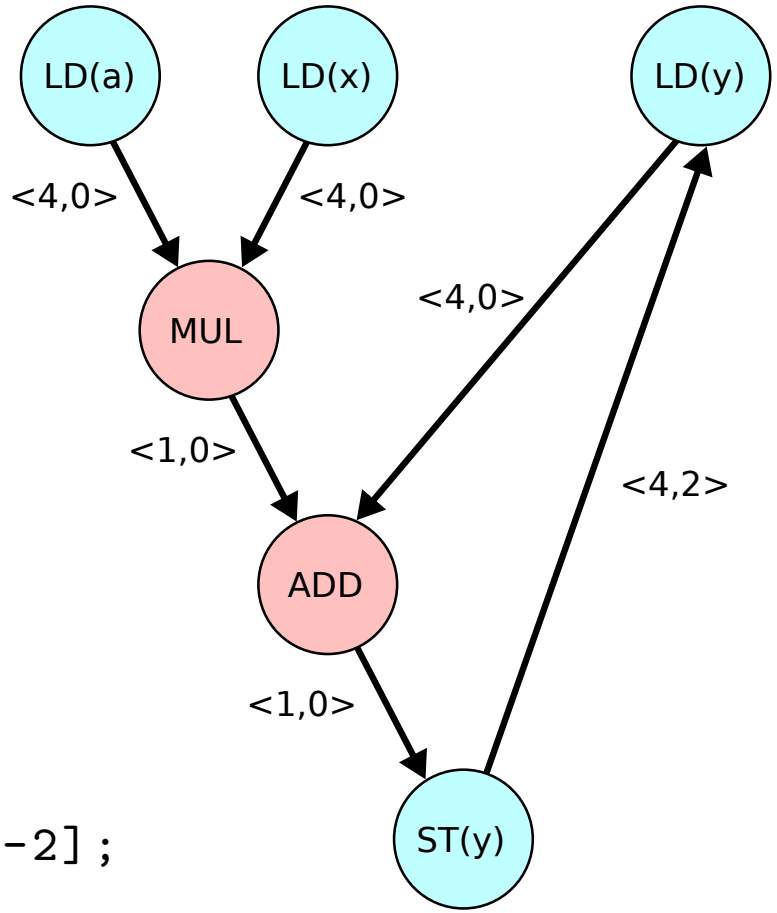
DDG für die erste Schleife für einen Prozessor, bei dem Speicherzugriffsinstruktionen eine Latenz von 4 Taktzyklen, und arithmetisch- logische Operationen eine Latenz von 1 Taktzyklus haben.




```
for (int i = 0; i < N; ++i)
    y[i] = a * x[i] + y[i];
```

Datenabhängigkeitsgraphen

DDG für die zweite Schleife für einen Prozessor, bei dem Speicherzugriffsinstruktionen eine Latenz von 4 Taktzyklen, und arithmetisch- logische Operationen eine Latenz von 1 Taktzyklus haben.




```
for (int i = 2; i < N; ++i)
    y[i] = a * x[i-1] + y[i-2];
```

Reservierungstabellen

Reservierungstabellen sind *binäre* Tabellen. Eine Dimension bezieht sich auf Taktzyklen (0,1,..), die andere Dimension bezieht sich auf die zur Verfügung stehenden Ressourcen.

Jeder zu planenden Instruktion sind eine oder mehrere alternative Reservierungstabellen zugeordnet. Jede Tabelle kodiert eine Möglichkeit, wie die Instruktion auf den von ihr benötigten Ressourcen geplant werden kann.

Reservierungstabellen Beispiel

Eine hypothetische CPU verfüge über zwei ALUs sowie über eine *Memory Management Einheit* (MMU). Die Instruktion LD benötigt auf dieser Architektur drei Taktzyklen. Im ersten Taktzyklus wird eine ALU benötigt, um den Speicheroffset zu berechnen. Die übrigen Taktzyklen belegen die MMU. Für die LD Instruktion ergeben sich die beiden alternativen Reservierungstabellen:

	ALU0	ALU1	MMU
0	x		
1			x
2			x

	ALU0	ALU1	MMU
0		x	
1			x
2			x

Reservierungstabellen

Mit Reservierungstabellen ist es relativ einfach möglich, zu bestimmen, wann der nächste Teilablaufplan initiiert werden darf. Wir nehmen eine ähnliche Maschine an wie oben: drei Taktzyklen für LD und ST, (der erste davon belegt eine ALU). ADD und MUL benötigen jeweils einen ALU Taktzyklus. Es gibt eine ALU und eine MMU. Wir betrachten wieder die Schleife:

```
for (int i = 0; i < N; ++i)
    y[i] = a * x[i] + y[i];
```

Der Einfachheit halber wollen wir annehmen, dass sich alle Operanden mit einer einzigen LD Operation aus dem Speicher laden lassen. Die Reservierungstabelle für **eine** Schleifeniteration lautet:

	0	1	2	3	4	5	6	7
MMU		x	x				x	x
ALU	x			x	x	x		

Reservierungstabellen

```
for (int i = 0; i < N; ++i)
    y[i] = a * x[i] + y[i];
```

	0	1	2	3	4	5	6	7
MMU		x	x				x	x
ALU	x			x	x	x		

Aus der Reservierungstabelle kann man die ungültigen Latenzen bestimmen:

$$UL = \bigcup_{i \in R} k - j,\ \forall j, k < C, k > j, x_{ij} \neq 0, x_{ik} \neq 0, j, k \in \mathbb{N} \quad (7)$$

R ist die Menge der Ressourcen, C ist die Anzahl Taktzyklen (im Beispiel 8). x_{ij} ist die Belegung (binär) für Ressource i in Taktzyklus j.

Reservierungstabellen

```
for (int i = 0; i < N; ++i)
    y[i] = a * x[i] + y[i];
```

	0	1	2	3	4	5	6	7
MMU		x	x				x	x
ALU	x			x	x	x		

Die ungültigen Latenzen bzgl. MMU im Beispiel lauten: $\{1, 5, 6, 4, 5, 1\}$. Für die ALU ergeben sich zusätzlich die Latenzen $\{3, 4, 5, 1, 2, 1\}$. Die nächste Iteration kann also frühestens nach 7 Taktzyklen eingeplant werden. (Achtung: es ergibt sich ein neuer Teilablaufplan!)

Reservierungstabellen

Kontrolle: nach wievielen Taktzyklen kann die zweite Iteration der Schleife anlaufen:

```
for (int i = 2; i < N; ++i)
    y[i] = a * x[i-1] + y[i-2];
```

	0	1	2	3	4	5	6	7
MMU								
ALU								

Iteratives Modulo Scheduling

- Bestimme Minimum Initiation Interval (MII).
  - Untere Schranke für II.
  - max(ResMII, RecMII).
    - ResMII: MII bzgl. Ressourcen, bestimme kleinstes II für **eine** Iteration.
    - RecMII: bzgl. Rekurrenz im DDG: Latenz des längsten **Zyklus**.
- II := MII
- **Unrolle die gesamte Schleife.** Schedule Schleife für aktuelles II. Falls kein gültiger Instruktionsablaufplan gefunden werden kann, inkrementiere II solange, bis die Schleife geplant werden kann.
- "Modulo" Scheduling: $II \ mod \ MII \stackrel{!}{=} 0$