Parallelrechner 2019

Prof. Dr. Konrad Froitzheim

Das Feld

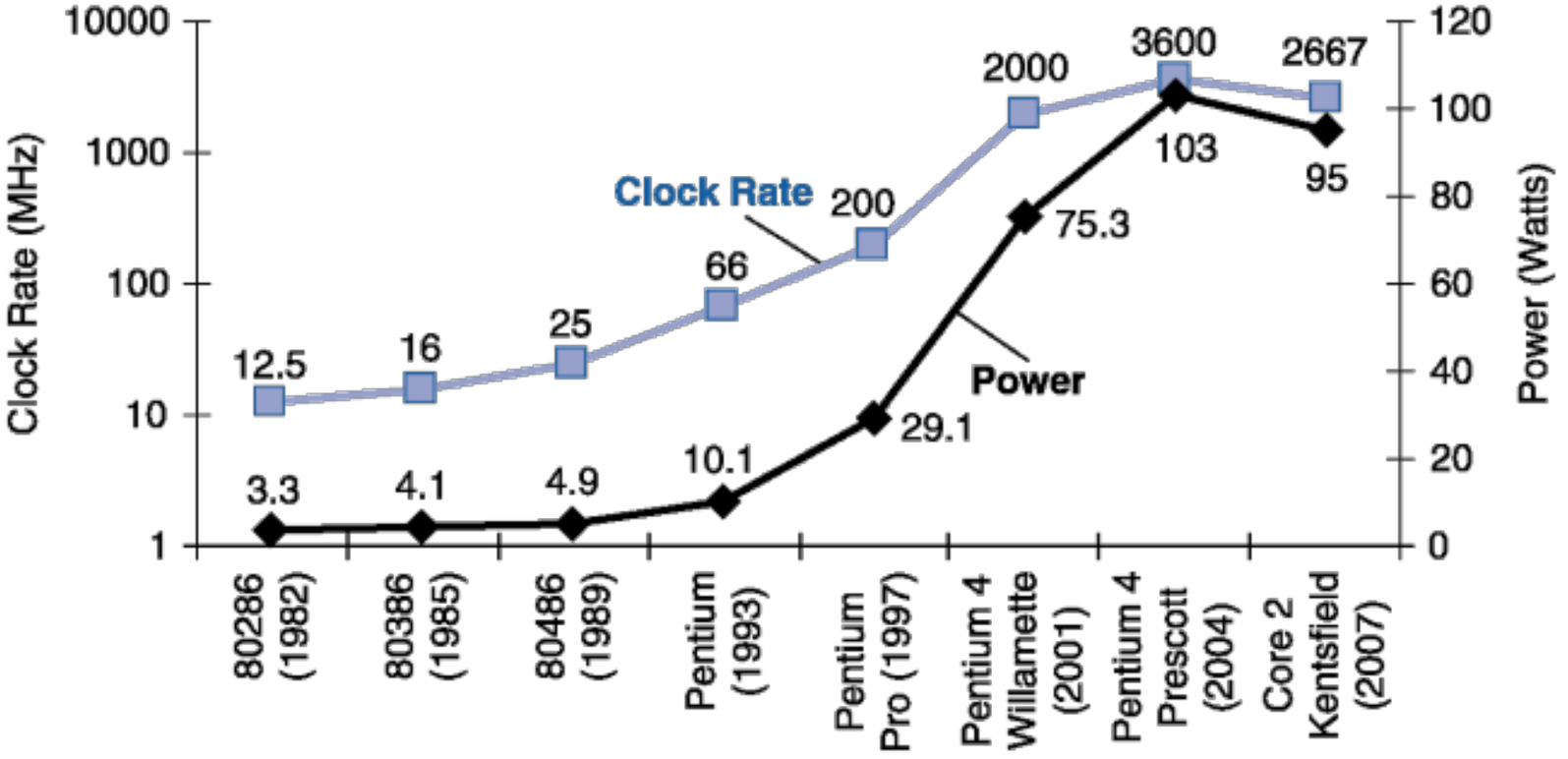
- Die Hoffnung
  - 1 Computerl Prozessor berechnet die Aufgabe in n Sekunden
  - m Computerl Prozessoren brauchen x Sekunden.
  - $x \geq n/m$ unbekannt
- Beispielaufgaben
  - Simulationen (Wetterbericht, Materialwissenschaften, …)
  - Optimierung (TSP, …)
  - Code brechen
  - Börsenspekulation
  - Videokompression
  - 3D-Grafik
  - …

- CPU-Leistung

$$CPU - Zeit = Anzahl\_Inst * \frac{Zeit}{Zyklus} * \frac{Zyklen}{Befehl}$$

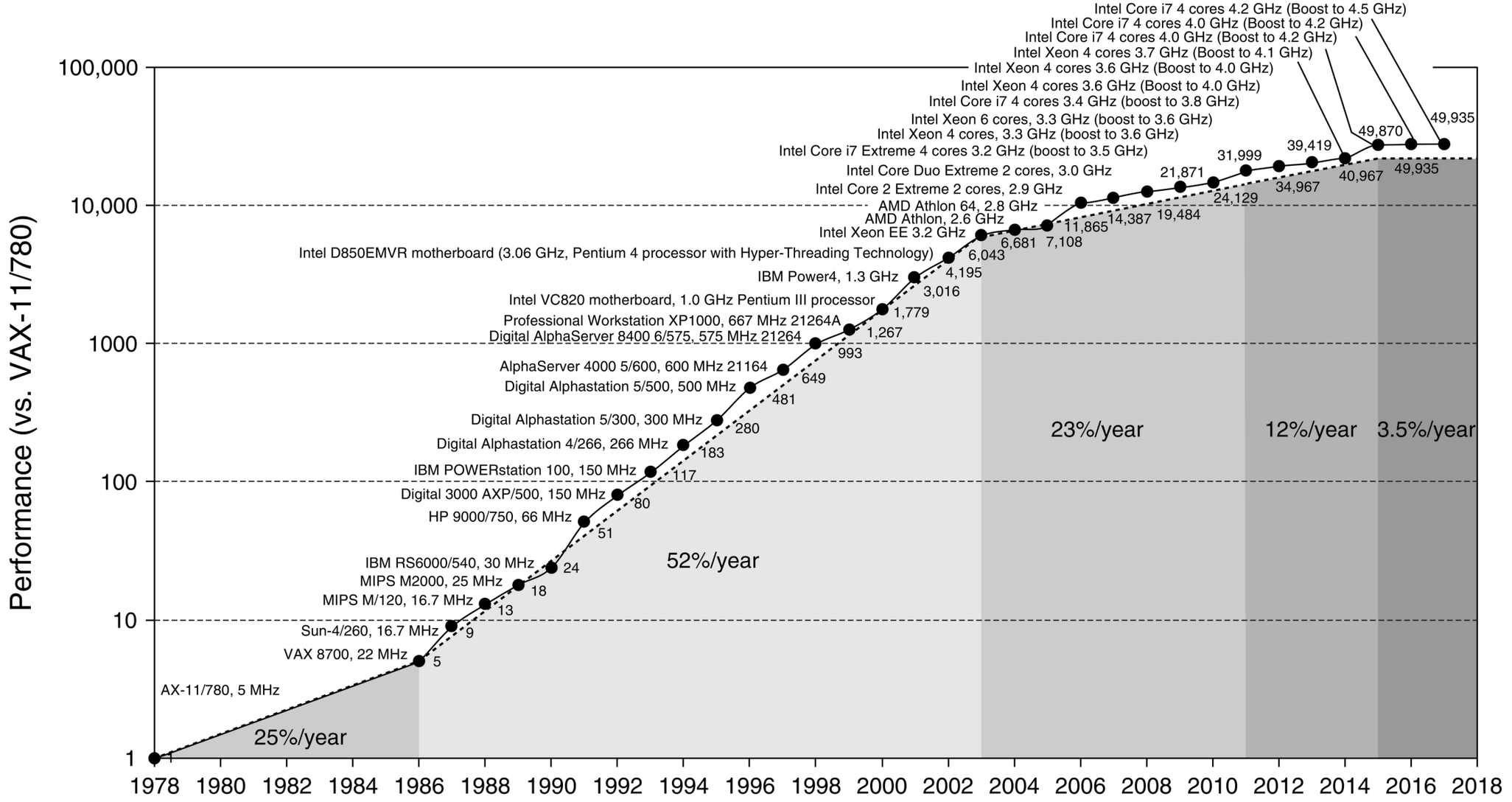
$$Stromverbrauch = Ladung * Spannung^2 * Frequenz$$

- CMOS "Power Wall"
  - Spannung kann kaum noch reduziert werden: Leckströme



[Patterson, Hennessey: Computer Organization …]

- Massnahmen
  - Kühlung
  - Energiekosten für Funktionen identifizieren
  - Paralleliät



[Hennessy, Patterson, Computer Architecture …]

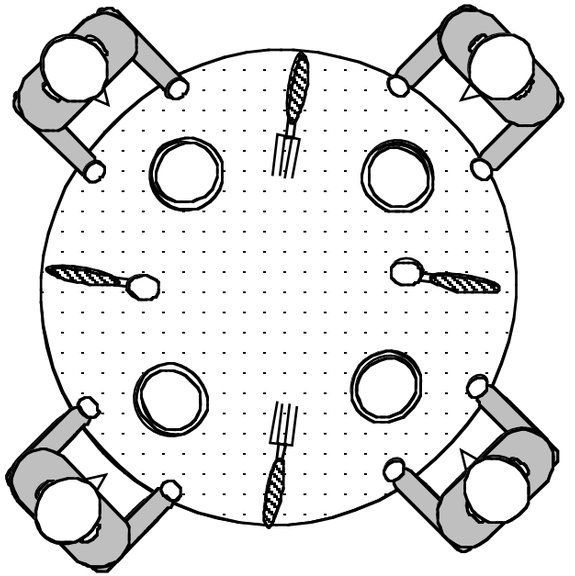
- Parallelausführung von Instruktionen in verschiedener Granularität
  - einzelne Operationen (+, -, *, Laden Speichern)
  - einzelne Instruktionen
  - Threads
  - Prozesse
  - Programme
- Serielle und parallele Codeteile (nach Bill Dally)

	Konfiguration	1 CPU Core	500 GPU Cores	10 CPU Cores	1 CPU Core + 450 GPU Cores
Hochparallel	seriell	1 sec	5 sec	1 sec	1 sec
	parallelisierbar	199 sec	0,4 sec	19,9 sec	0,44 sec
	Gesamt	200 sec	5,40 sec	20,9 sec	1,44 sec
Hochseriell	seriell	150 sec	750 sec	150 sec	150 sec
	parallelisierbar	50 sec	0,1 sec	5 sec	0,11 sec
	Gesamt	200 sec	750 sec	155 sec	150 sec

- Das zentrale Problem aller verteilten Systeme: Nebenläufigkeit
  - Concurrency

> Concurrency occurs when two or more execution flows are able to run simultaneously [Dijkstra]

  - Ereignisse beeinflussen sich *nicht* gegenseitig
  - d.h. nicht kausal abhängig
  - oft nicht gegeben
- Gleichzeitige Ausführung
  - viele Ausführungspfade
  - unterschiedlicher oder gleicher Code
  - konkurrieren um Ressourcen
  - Hardware
- 'Synchronisation'
  - Koordination abhängiger Ereignisse
  - mutual Exclusion
  - neue Probleme: Deadlocks



- Beispiel: Sequentielles Programm A:
  - ergibt 30 als Resultat in der Variablen "c"

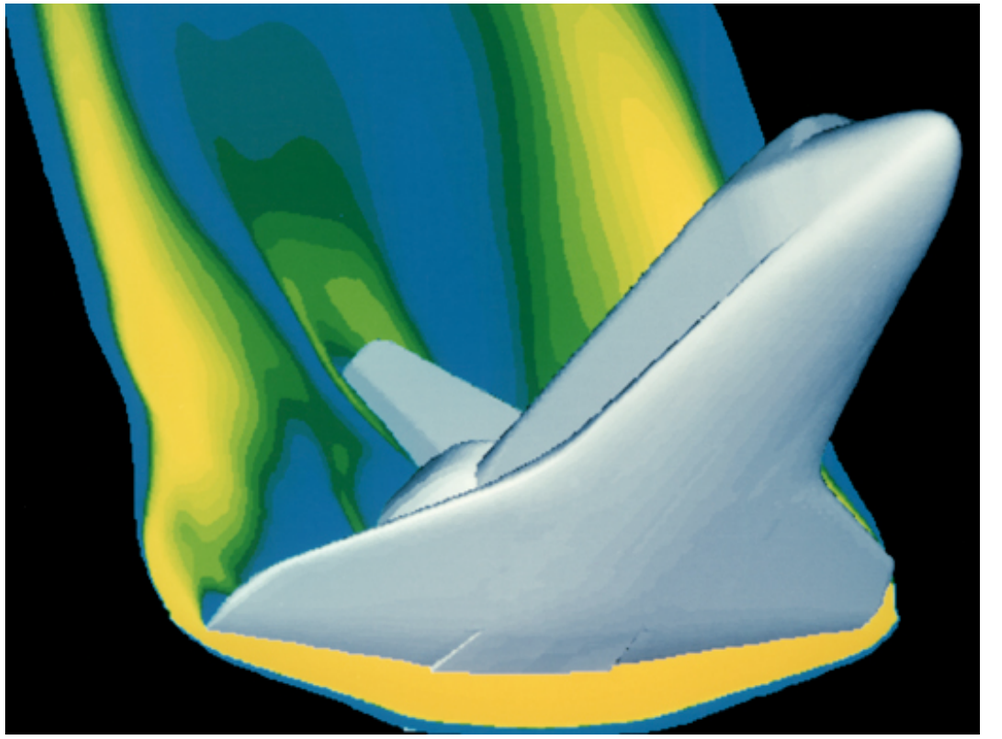
Statements

`a:= 0;`
`b:= 0;`
`a:= 10;`
`b:= 20;`
`c:= a+ b;`

- Nebenläufige Ausführung ohne Synchronisierung
  - verschiedene Resultate möglich für c
  - 120 mögliche Permutationen (5*4*3*2)
  - zum Beispiel für drei Prozessoren sei c = { 0, 10, 20, 30 ... }

CPU #1	CPU #2	CPU #3
a:=0;	*b:=0;*	*a:=10;*
b:=20;	*c:=a+b;*	

- Raytracer
  - Filme in 2D und 3D (Avatar)
  - Computerspiele
  - Architektur, CAD, …
  - Raytracing erlaubt richtige Schatten
- CFD: Computational Fluid Dynamics
  - Navier-Stokes
  - nichtlineare partielle Diff.-Gl.
  - modelliert Strömungen
  - Wetter, Aerodynamik, Akustik, …
  - Diskretisierung
  - Finite Volumen/Elemente Methode
  - z.B. Panel beschreiben Oberfläche



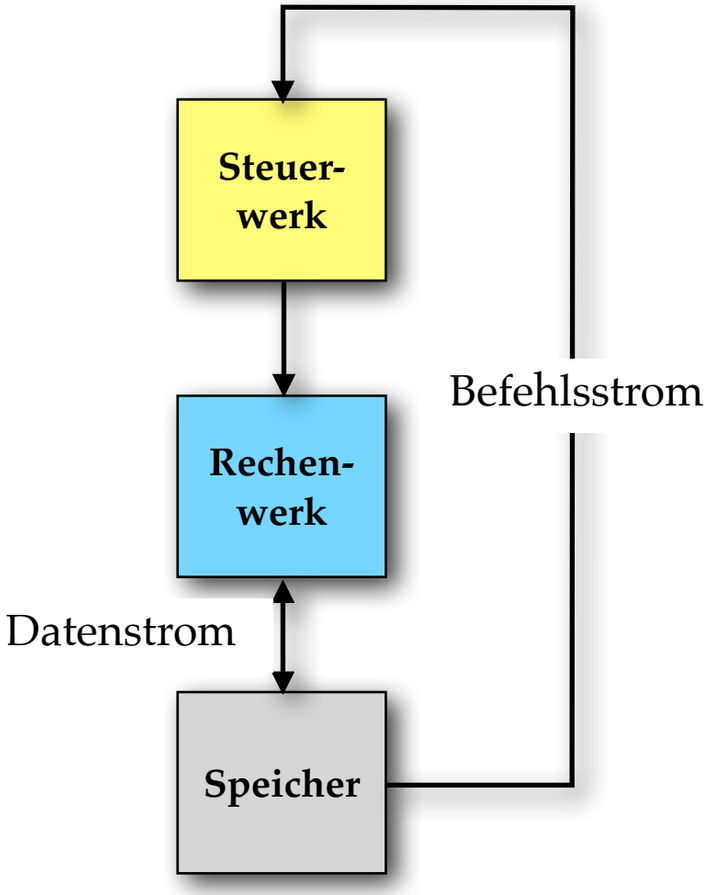
• TOP 500 (November 2018, http://www.top500.org/list/2016/11/)

Organisation, Land	Name und Beschreibung	Jahr	Hersteller	KKerne	PFlops	MW
DOE/SC/Oak Ridge National Laboratory, USA	Summit, Power9 22C 3.07 GHz, Volta GV100, Dual-rail EDR Infiniband	2018	IBM	2.398	143,5	9,4
DOE/NNSA/LLNL, USA	Sierra - POWER9 22C 3.1GHz, Volta GV100, Dual-rail EDR Infiniband	2018	IBM	1.573	94,6	7,4
National Supercomputing Center in Wuxi, China	Sunway TaiHuLight- SW26010 260C, 1.45 GHz, Sunway	2016	NRCPC	10.650	93	15,4
National Supercomputing Center in Guangzho, China	Tianhe-2A, Xeon E5-2692 12C 2.2 GHz, Phi 31S1P, TH Express-2, Matrix 2000	2013	NUDT	4.982	61,4	18,5
Swiss NSC, CH	Piz Daint - XC50 XEON E5-2690 12C 2.6 GHz, NVidia P100, Aries interconnect	2013	Cray Inc.	388	21,2	2,4
DOE/NNSA/LLNL, USA	Trinity - Xeon E5-2698 16C 2.3 GHz, Phi 7250 1.4 GHz, Aries interconnect	2015	Cray Inc.	979	20	7,6
AIST Japan	AI Bridging Cloud Infrastructure, Xeon 6148 2.4 GHZ, Tesla V100, Infiniband EDR	2018	Fujitsu	622	14	3,9
LRZ München, D	SuperMUC-NG, Xeon 8174 3.1 GHz, Intel Omni-Path	2018	Lenovo	306	19,5	
DOE/SC/Oak Ridge National Laboratory, USA	Titan - XK7 Opteron 6274 2.2 GHz, NVidia K20x, Gemini interconnect	2012	Cray Inc.	560	17,6	8,2
DOE/NNSA/LLNL, USA	Sequoia - BlueGene/Q, Power BQC 16C 1.6 GHz, Custom interconnect	2012	IBM	1.573	17,2	7,9

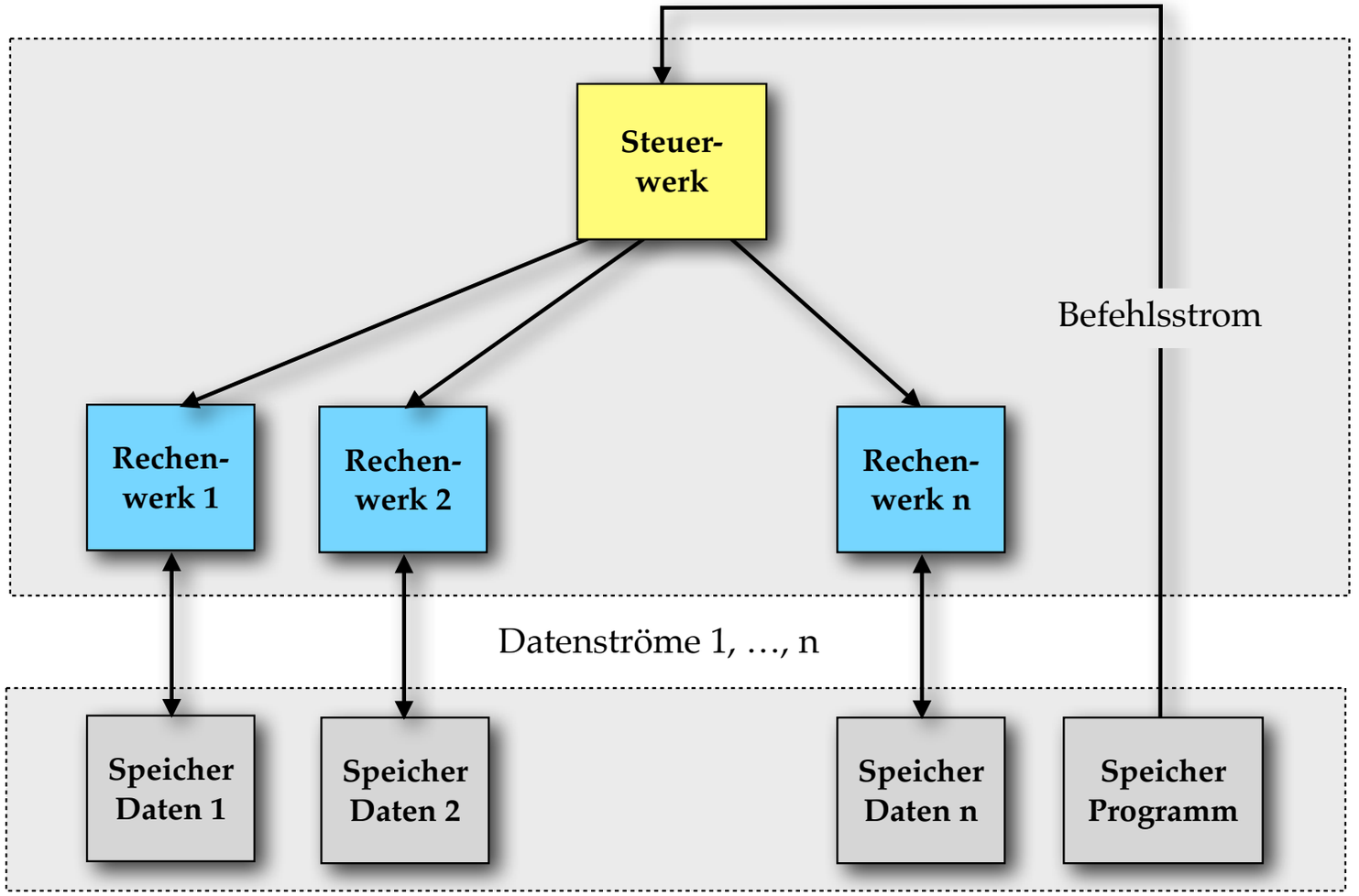
- ViginiaTech Supercomputer [2003]
  - 1100 G5 Macs, Infiniband
  - 2200 Kerne, 10 TFlops
  - Platz 3 auf der TOP500 11/2003
- DIY Supercomputer [Yann Le Du, ParisTech]
  - HPU4Science ~ $40.000
  - Ziel: Leistung eines $400.000 Komplettsystemes
  - CPU-Kern kostet $50-$250
  - GPU-Kern kostet $1 (GTX580, 512 shader, ≤ $500)
  - Gesamtpreis $1,80 pro GFlop
  - 1 Master, 6 Worker: 34.5 Tflops theoretisch
- Master: PC-Gehäuse …
  - 2 Xeon 5000: 16 threads
  - 24 GB RAM, SSD, GTX295
- Worker (Bsp): PC-Gehäuse …
  - Core i7, 4 cores, 8 threads
  - GTX 285 - 590
  - 4 GPUs

- Cloud Supercomputing [Q1/2017]
  - IBM, Amazon, Google, Nimbix
  - stundenweise mieten
  - Speicher, Rechenzeit, Traffic
- IBM Softlayer
  - Rack mieten
  - 2 Xeon C8, 128 GB RAM, 2* 800GB SSD, 2 K80
  - $5.30/Stunde
- Google GPU Cloud Service
  - NVidia K80: zwei GK210, 24 GB GDDR5
  - 2.9 TFlopDP, 8.73 TFlopSP
  - 0,5 - 4 K80 mieten
- Nimbix
  - 4 Kerne, 32 GB RAM, Tesla P100: $5/Stunde
  - 4 Kerne, 32 GB RAM, K80: $1.06/Stunde

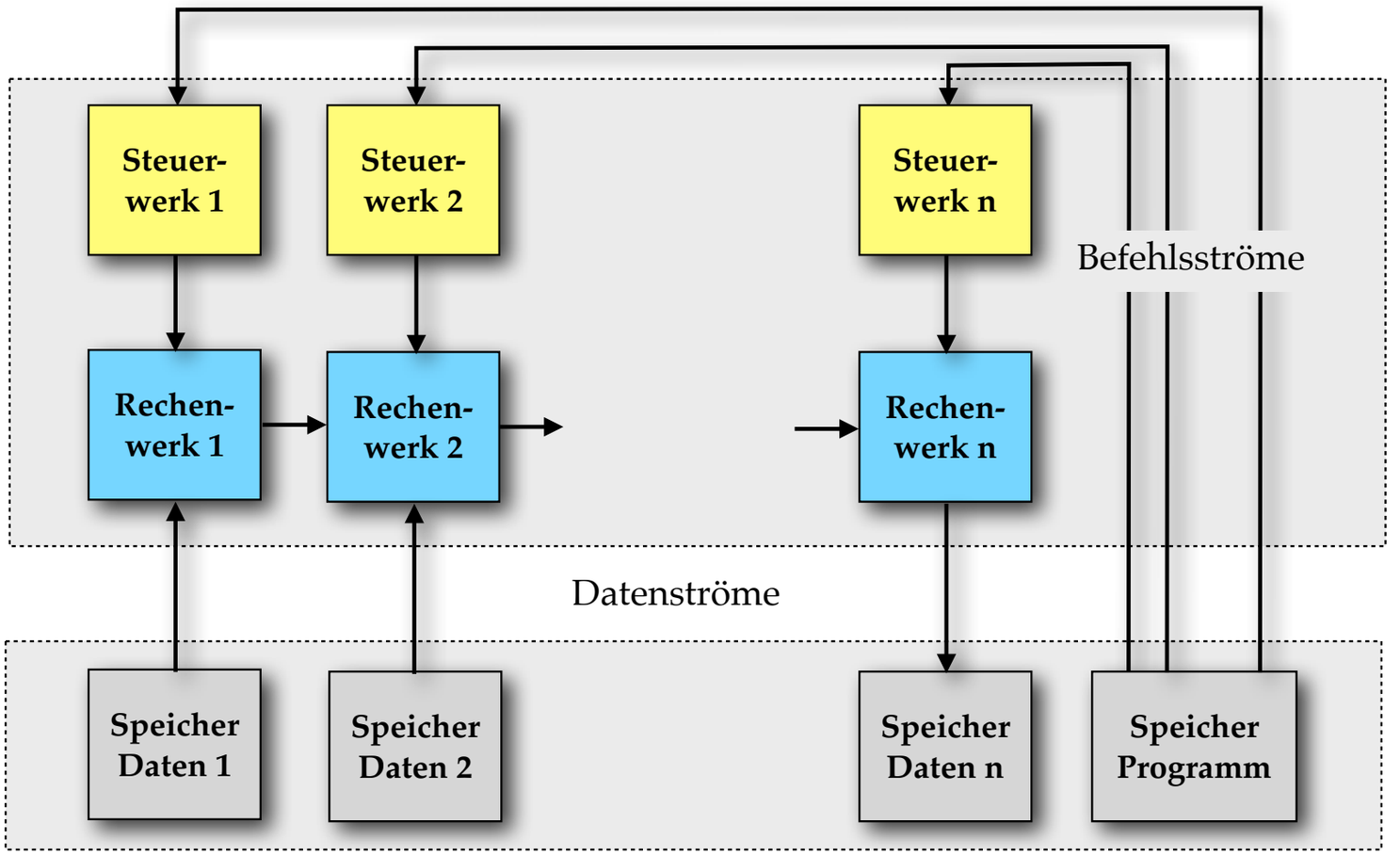
- Klassifikation nach Flynn
- SISD
  - single Instruction
  - single Data



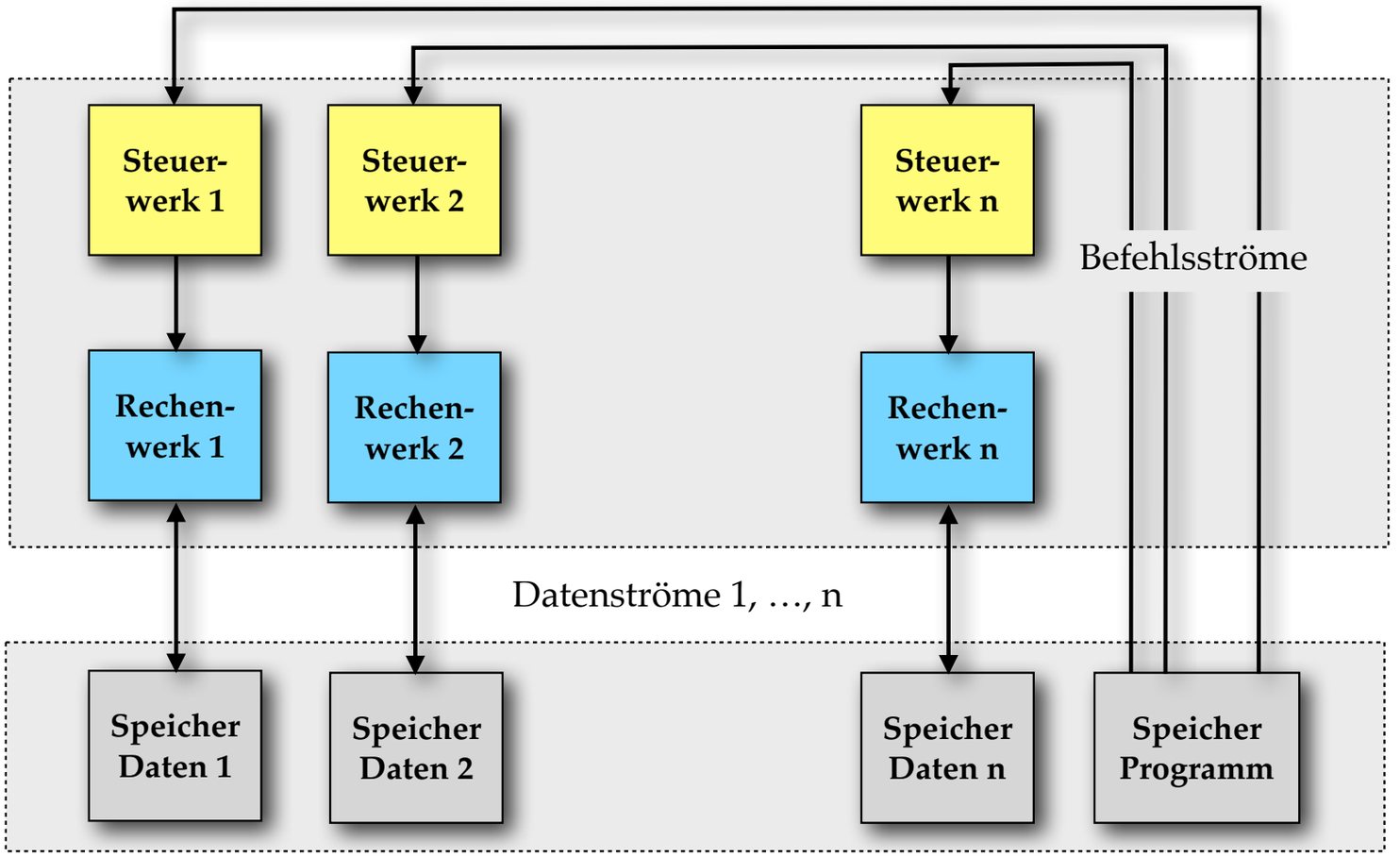
- SIMD
  - single Instruction
  - multiple Data



- MISD
  - multiple Instruction
  - single Data



- MIMD
  - multiple Instruction
  - multiple Data



- Arbeitsfelder
  - Rechnerarchitektur
  - mathematische Algorithmen
  - Algorithmen der Informatik
  - Softwarearchitektur
  - Programmiersprachen
  - Frameworks und Toolboxes
  - Patterns
- Parallele Architekturen
  - Instruktion level Parallelism (ILP)
  - Thread Level Parallelism (SMT, Hyperthreading)
  - Symmetric Multiprocessing (SMP, Multicore)
  - GPU Computing
  - Vektorrechner SIMD
  - Speichergekopplete Architekturen
  - netzwerkbasierte Parallelrechner
  - lose gekoppelt (Client-Server)

Inhaltsübersicht

Literatur

Bader, D. [Hrsg]: Petascale Computing; 2008.

Garland, M., Kirk, D.: Understanding Throughput-Oriented Architectures, 2010.

Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, 2008.

Kirk, D., Hwu, W.: Programming Massively Parallel Processors, 2010.

Lindholm, E., et al: NVIDIA Tesla: a unified …, IEEE, 2008.

Nicholls, J., Kirk, D.: Graphics and Computing GPUs, 2009.

Patterson, D., Hennessy, J.: Computer Organization and Design, 2009.

Seiler, L., et al: Larrabee: A Many-Core x86 …, ACM ToG, 2008.

Schmidt, D., Huston, S.: C++ Network Programming; 2002.

…

Formales

Termine:

Vorlesung: Mittwoch 11:00 - 12:30, 3409

Dramatis Personae:

Konrad Froitzheim

Professur Betriebsysteme und Kommunikationstechnologien

mailto:frz@tu-freiberg.de

Ben Lorenz

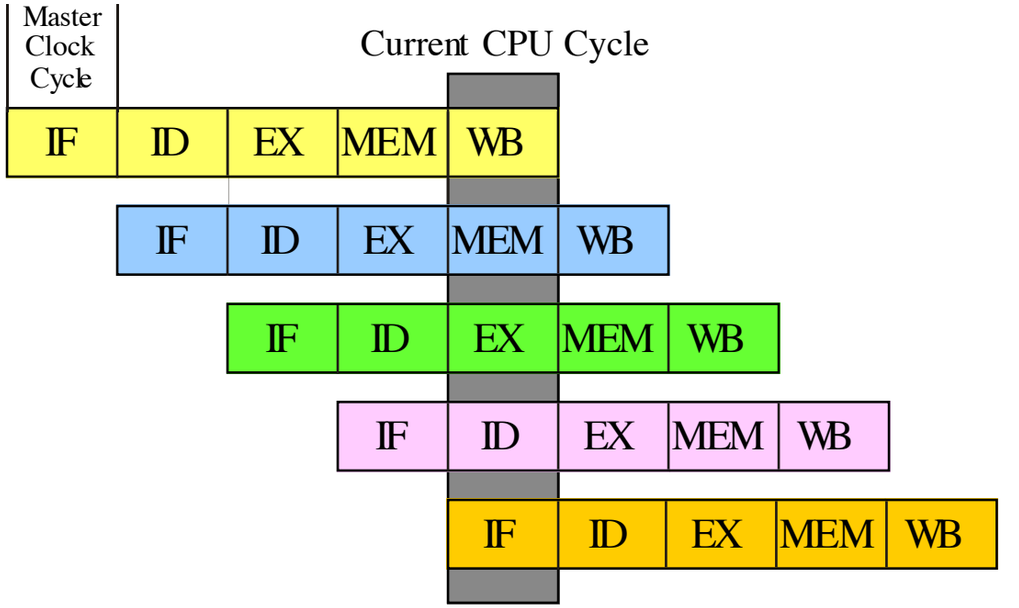
Unterlagen zur Vorlesung:

http://ara.informatik.tu-freiberg.de/vorlesungen/ParComp2017.pdf

1. Rechnerarchitektur

1.0 Wiederholung

- Fließbandarbeit
  - Instruktionen haben Teilschritte
  - parallele Ausführung der Teilschritte verschiedener Instruktionen
  - Weitergabe der Zwischenergebniss an nächste Stufe (≠MIMA)



- 1 Instruktion
  - n Maschinen-Zyklen insgesamt
  - pro Maschinen-Zyklus 1 Instruktion fertig
  - m Takte = 1 Maschinen-Zyklen; m<=4

- Unabhängigkeit der Einzelschritte in der Pipeline
  - Entkopplung der Stufen
  - genug Verarbeitungseinheiten (ALU + 1 Addierer für Adressen)
- Pipelinetakt
  - bestimmt durch längsten Teilschritt
  - z.B. Speicherzugriff oder Alu-Operation
  - Länge der Einzelschritte ausbalancieren
- Pipeline-Register entkoppeln Stufen
  - Pipeline-Latches
  - Stufe n-1 schreibt Resultat spät im Zyklus
  - Stufe n nimmt Input früh im Zyklus
- Pipeline-Risiken
  - Verzweigung, Sprung
  - Operandenbezüge
  - Resource-Mangel (z.B. ALU, Registerbank)

  => Pipeline-Stall

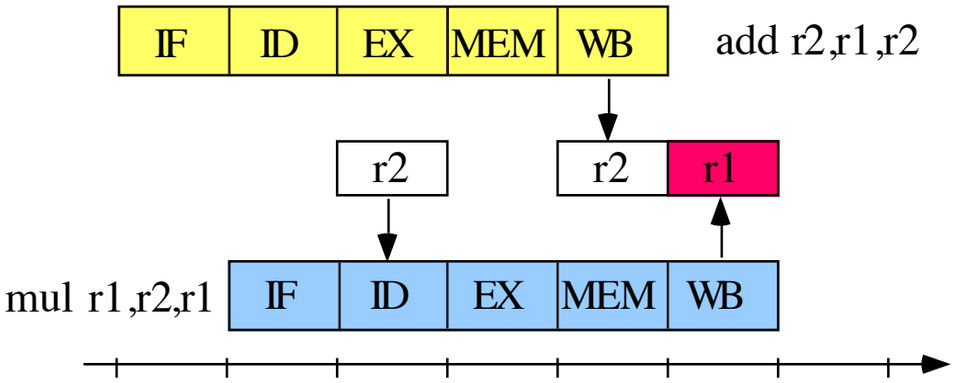
- Konflikte in Architektur oder Abhängigkeiten
- Cache-Miss
  - Pipeline anhalten
  - Warten bis Daten angekommen sind
- Hazard
  - konkurrierender Zugriff auf Ressourcen oder Daten
  - keine neuen Instruktionen starten
- Pipeline-Bubble (oder Stall)
  - Beispiel nur ein Instruktions/Datencache

Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				
Instr i+2			IF	ID	EX	MEM	WB			
Instr i+3				stall	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

- Structural Hazard
  - Nur ein Schreibeingang in das Register-file
  - 2 Schreibzugriffe in einem Zyklus
  - Nur ein Cache (Bus) für Daten und Befehle
  - Load.MEM kollidiert mit Inst3.IF

Instr	1	2	3	4	5	6	7	8
Load	IF	ID	EX	MEM	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				IF	ID	EX	MEM	WB

- Data Hazards
- Echte Abhängigkeit
  - $Inst_1$ schreibt in das Register, das $Inst_2$ benötigt
  - auch im Speicher!



- Read After Write (RAW)
  - $Instr_2$ liest Operand, bevor $Instr_1$ ihn schreibt
- Write After Read (WAR)
  - $Instr_2$ schreibt Operand, bevor $Inst_1$ ihn liest
  - nur bei superskalaren Architekturen
- Write After Write (WAW)
  - $Instr_2$ schreibt Operand, bevor $Inst_1$ ihn schreibt
  - in komplexeren Pipelines als DLX
- Namens-Abhängigkeit
  - anti-abhängig: $Inst_1$ liest Register, das $Inst_2$ später überschreibt
  - Ausgabe-abhängig: $Inst_1$ schreibt Reg., das $Inst_2$ danach überschreibt
  - keine Daten von $Inst_1$ and $Inst_2$ übergeben
  - Implementierungsproblem

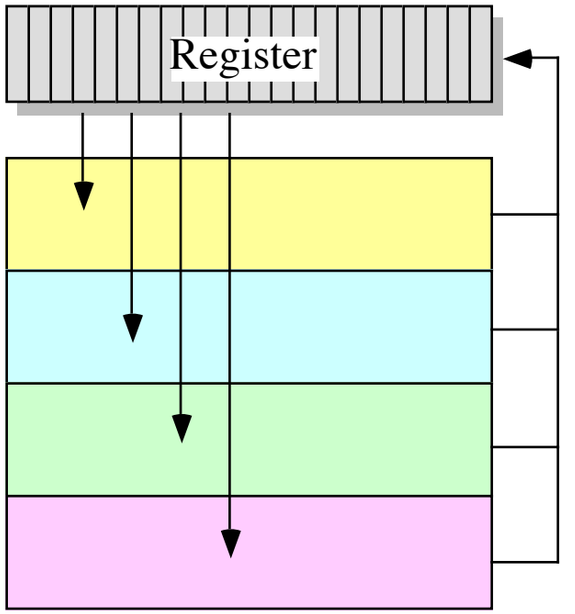
- Control Hazards
- Sprung-Instruktionen (bedingt und fest)
  - Spungziel muss berechnet werden
  - Bubbles einfügen bis Sprung MEM ausgeführt hat
  - Sprung erst in ID festgestellt
  - IF während Sprung-ID oft wertlos

IF	ID:Sprung	EX	MEM	WB					
	IF->stall	stall	stall	IF	ID	EX	MEM	WB	
					ID	EX	EX	MEM	WB

- 3 Zyklen pro Spung verloren
  - Verzweigungen ca 20%

  => 1,6 CPI (cycles per instruction)
- Abhilfe
  - schon in IF feststellen ob gesprungen wird
  - neuen PC früher berechnen
  - Branch Prediction

- **Instruction Level Parallelism**
- Instruction-Decode entscheidet über Ausführung
  - Warteschlange mit Befehlen
  - structural Hazards auswerten
  - nächsten möglichen Befehl starten
  - auf Functional-Units verteilen
  - evtl. dynamisch umordnen
- Instruktions-Resultat-Reihenfolge geändert
  - WAR, WAW können entstehen
  - Unprecise Exceptions?
  - Planung abhängig von Registern
  - Instuktionen können lange in FU hängen
- Beispiel DLX-Pipeline
  - DLX-ID prüft structural H. und data H.
  - Neu: ID aufteilen in *Issue* und *Read Operands*
  - Warteschlange zwischen Fetch und Issue
  - DLX: Nur bei Floating Point Unit
  - 1 Integer, 2 FP-MULs, 1 FP-DIV, 1 FP-ADD



- Beispielproblem

```
DIVD   F0,F2,F4      ; viele Zyklen
ADDD   F10,F0,F8
SUBD   F12,F8,F14    ; WAW: SUBD  F10,F8,F14
                     ; WAR: SUBD  F8,F8,F14
```

  - SUBD vorziehen
- Registerverwaltung
  - Auflösung der data hazards
  - Warten auf Daten
  - Warten beim Speichern in Register
- Scoreboard
  - startet Instruktionen (issue) in EX[i]
  - gibt Register als Operanden frei (RAW)
  - stellt fest, dass EX[i] ist fertig
  - überprüft ob WB Register schreiben kann (WAR, WAW)

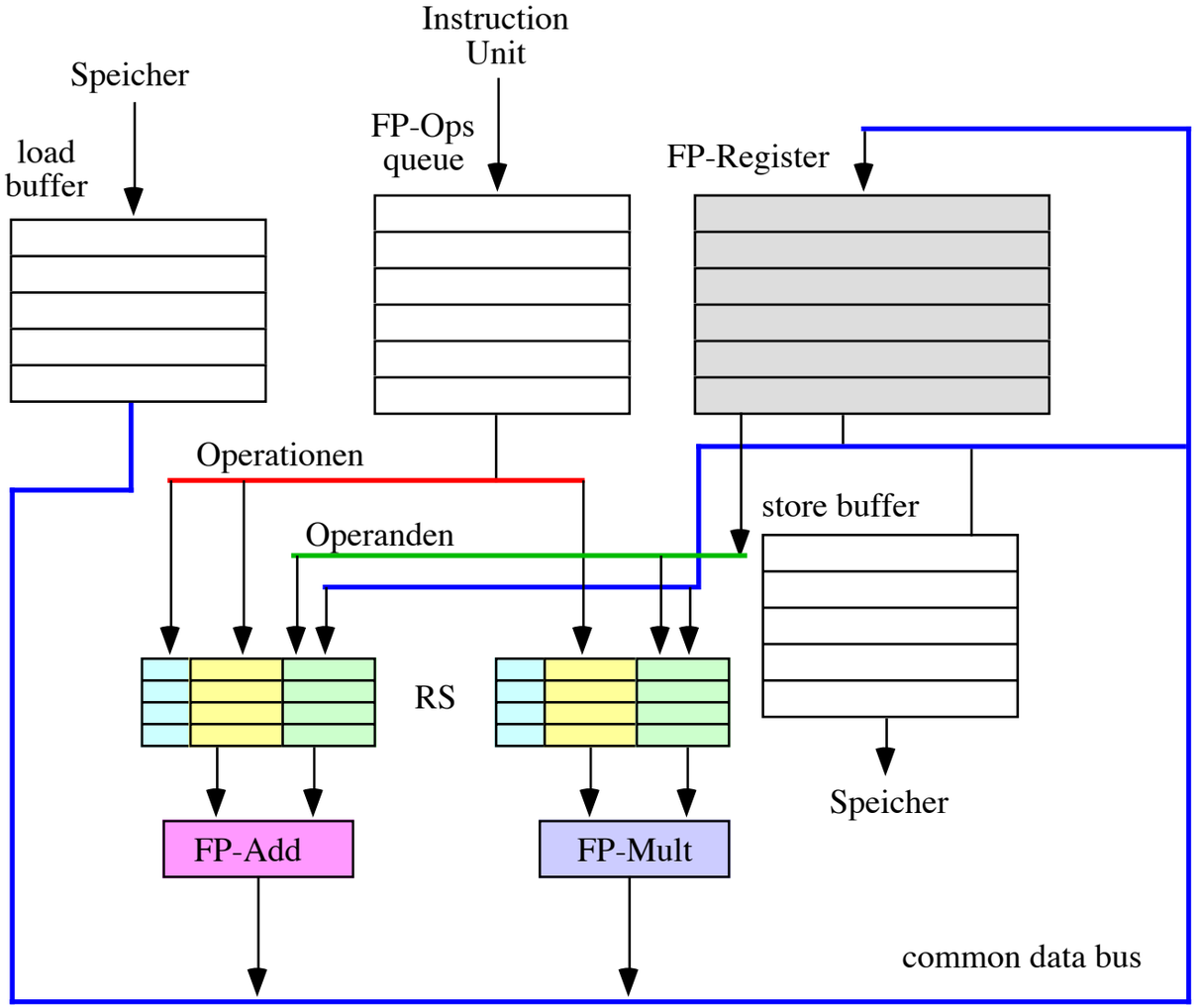
- Scoreboard-Tabellen [CDC 6600, 1964]
  - Schnappschuss für DLX-Pipeline

Instruktion		issued	read ops	exec. comp.	write back
`LD`	`F6,32(R2)`	true	true	true	true
`LD`	`F2,48(R3)`	true	true	true	
`MULTD`	`F0,F2,F4`	true			
`SUBD`	`F8,F6,F2`	true			
`DIVD`	`F10,F0,F6`	true			
`ADDD`	`F6,F8,F2`				

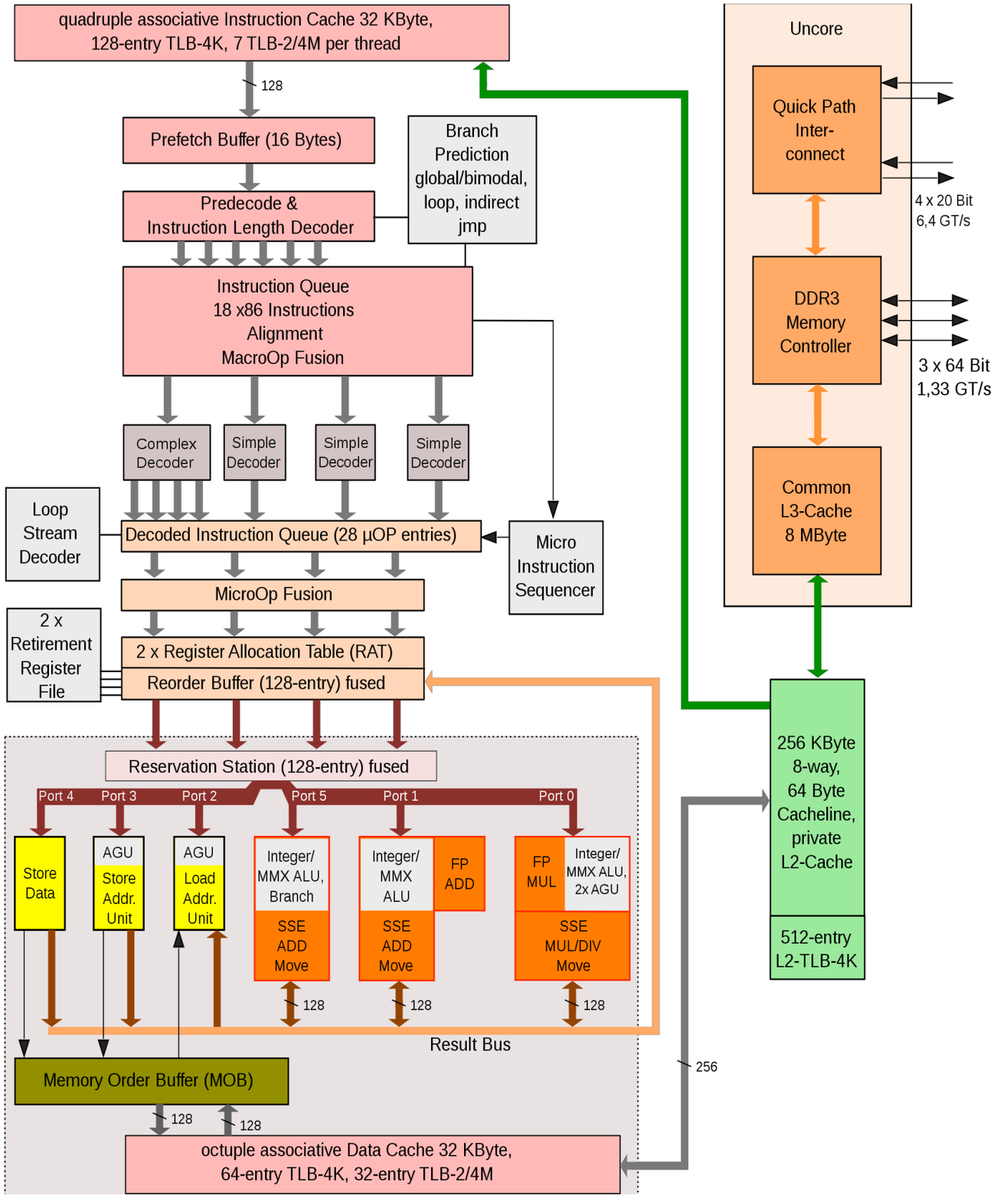
Funktionseinheit	busy	Op	RegD	RegS1	RegS2	FU1	FU2	rdy1	rdy2
Integer	true	Load (2)	F2	R3				false	
FMult1	true	Mult	F0	F2	F4	Integer		false	true
FMult2	false								
FAdd	true	Sub	F8	F6	F2		Integer	true	false
FDiv	true	Div	F10	F0	F6	FMult1		false	true

Register	F0	F2	F4	F6	F8	F10	F12	…	F30
Funktionseinheit	FMult1	Integer			FAdd	FDiv			

• Tomasulo


Instruction
Unit
Speicher
load
buffer
FP-Ops
queue
FP-Register
Operationen
Operanden
store buffer
RS
Speicher
FP-Add
FP-Mult
common data bus

- Nehalem Architektur [Wikipedia]



- Pipeline steigert Durchsatz
  - Phasen der Verarbeitung durch Puffer entkoppelt
- Probleme: Hazards
  - echte Datenabhängigkeit und Namensabhängigkeit
  - Ressource-Engpässe
  - Control-Hazards
- Statische Techniken zur Optimierung (Compiler)
  - Delay-Slot füllen
  - Schleifen ausrollen
  -> Instruktionsplanung
- Leistungssteigerung: Instruction Level Parallelism (ILP)
  - parallele Funktionseinheiten
  - Scoreboard, Tomasulu
  - completion unit bewertet spekulative Ausführung
  - VLIW oder multiple Issue

1.1 Vektorrechner und SIMD

- Idee: Pro Instruktion mehrere Datenelemente bearbeiten
  - Vektoren
  - Zeichenketten
- Beispielproblem YUV-Konvertierung
- UpSampling: 4:1:1 -> 4:4:4
- Konvertierungsmatrix YIQ - > RGB

$$\frac{1}{4096}\begin{pmatrix} 4788 & 0 & 6563 \\ 4788 & -1611 & -3343 \\ 4788 & 8295 & 0 \end{pmatrix}\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

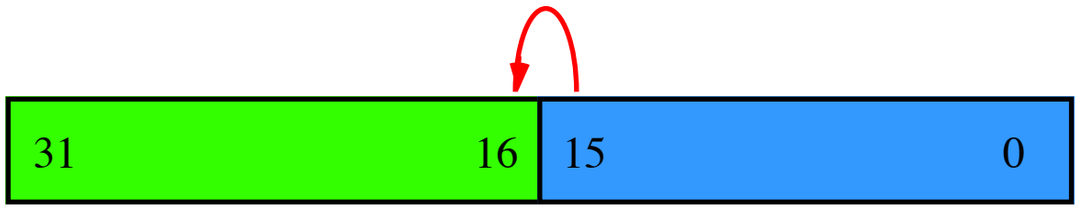
- Full HD: 1080*1920*25*30 = 1.555.200.000 Instruktionen/sec
  - 7 (5) Multiplikationen, 4 Additionen
  - 8 Laden, 3 Speichern, 3 Shifts
  - pro Pixel 23 Instruktionen plus Adressmanipulation ~ 30 Inst.
- Tabelle 8 * 3 MB groß

```
display^[i] := rgb[BSL(y,16)+BSL(u,8)+v];
```

- Programmbeispiel

```
FOR i := 0 TO LineLength-1 DO BEGIN                          (* 2 *)
    tmpy := 4788 * y^[i];                                    (* 3 *)
    r:=BSR((tmpy + 6563 * v^[vi]), 12);                      (* 4 *)
    IF r < 0 THEN r := 0 ELSE IF r > 255 THEN r := 255;      (* 6 *)
       (* Sättigungs-Clipping *)

    g:=BSR((tmpy -1611*u^[ui] -3343*v^[vi]),12);             (* 7 *)
    IF g < 0 THEN g := 0 ELSE IF g > 255 THEN g := 255;      (* 6 *)

    b:=BSR((tmpy +8295*u^[ui]),12);                          (* 4 *)
    IF b < 0 THEN b := 0 ELSE IF b > 255 THEN b := 255;      (* 6 *)

    display^[i] := BSL(r, 16) + BSL(g, 8) + b;               (* 5 *)
    IF i mod 4 = 3 THEN BEGIN ui:=ui+1; vi:=vi+1 END;        (* 5 *)
END;
```

- 48 Instruktionen
- Zahlen nur 16 bit groß
  - 2 Pixel parallel addieren/multiplizieren <-> Carry



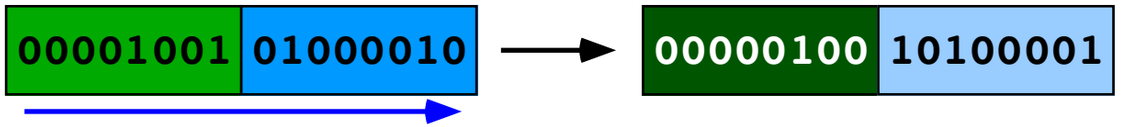
- Mittelwertbildung
- Farbtabellenberechnung mit Octree

```
theClut[i]:= Average(theClut[i],theClut[j]);
```

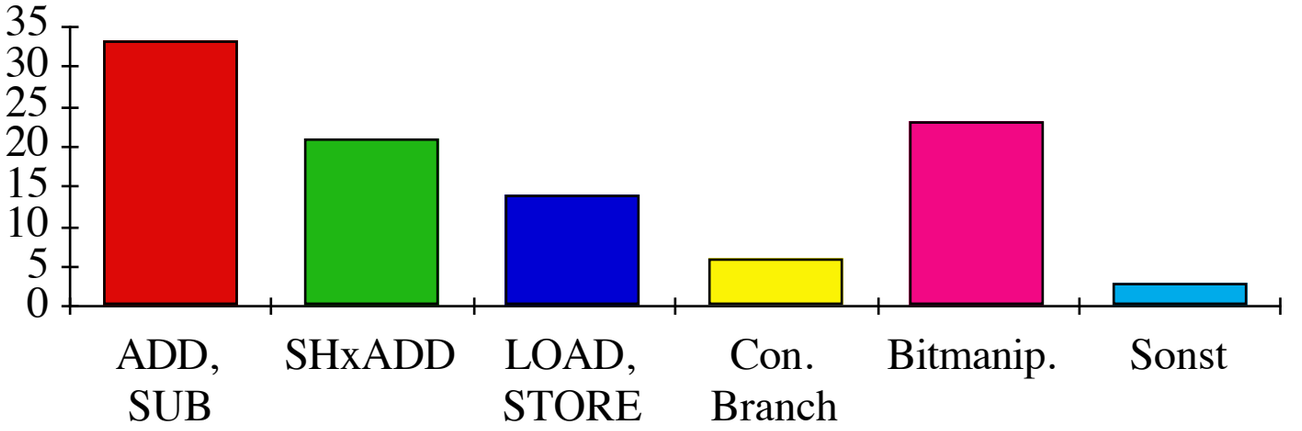
  - Elementweise Addition und Division / 2

```
theClut[i].r:= (theClut[i].r + theClut[j].r) SHR 1; (*5*)
theClut[i].g:= (theClut[i].g + theClut[j].g) SHR 1; (*5*)
theClut[i].b:= (theClut[i].b + theClut[j].b) SHR 1; (*5*)
```

- Problem bei Parallelausführung
  - ungerade + gerade => ungerade
  - ungerade SHR 1 => niedrigerer Wert + 128 (bzw. 32768)



- Codeanalyse IDCT



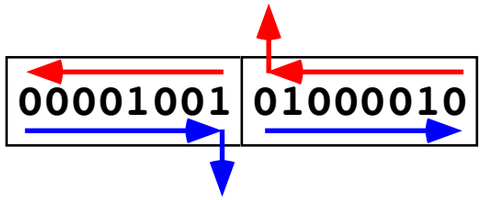
- SIMD-Instruktionen
  - Werte im Speicher nebeneinander => 1 Bustransfer für 2/4 Werte
  - kleine Zahlen, große Register (32, 64 bit)

```
  00010010 01001001          00010010 11001001
+ 01100010 00100000        + 01100010 01100000
  01110100 01101001          01110101 00101001
```

- Konfigurierbare ALU
  - Paralleles Addieren/Subtrahieren

```
    00010010  11001001
+   01100010  01100000
    01110100  00101001
```

  - Carry Unterbrechung konfigurieren
  - Paralleles Shiften



  - Paralleles Multiplizieren 16 * 16
- Operation ShiftxAdd
  - RISC ohne Integermultiplikation
  - besonders für Multiplikation mit (kleinen) Konstanten

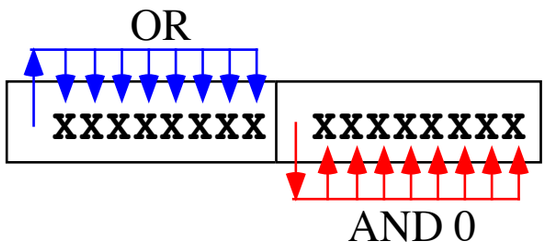
- Sättigungsarthmetik

```
    00010010 11001001          00010010 11001001
+   01100010 01100000      +   01100010 01100000
    01110100 00101001          01110100 11111111
```

```
erg := a + b;
IF erg < 0 THEN erg:= 0 ELSE IF erg > max THEN erg:= max;
```

- besondere Carry/Borrow Behandlung

```
IF borrow THEN regPart := 0;
IF carry THEN regPart := maxPartInt;
```



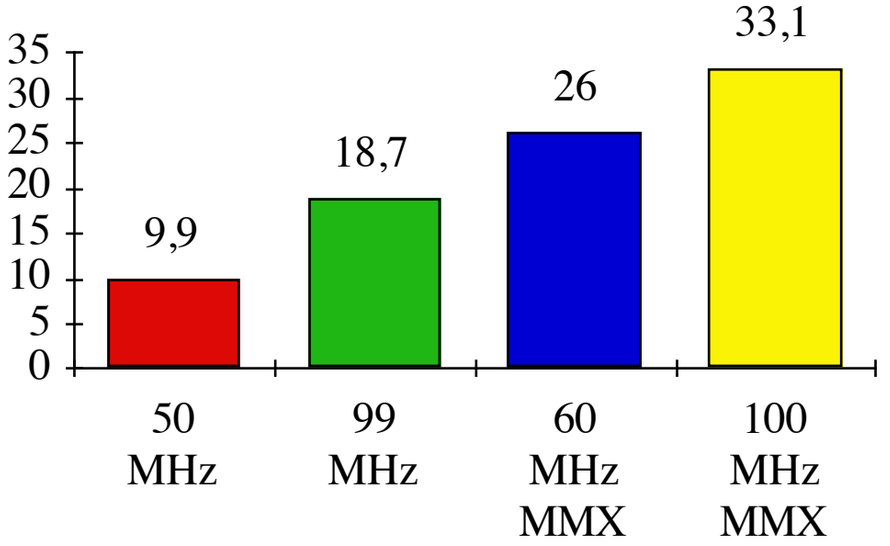
- Auch für 2-seitigen Begrenzer (Clipping)

• YUV mit Parallel-Add/Mult und Sättigungsarithmetik

```
YRGB = BSL(299,16)+299; VR = BSL(410,16)+410; UG,VG,UB …
i:=0; ui:=0; vi:=0;

REPEAT
   thisV := LoadHigh(v^[vi])+v^[vi];                    (* 2 *)
   thisU := LoadHigh(u^[ui])+u^[ui];                    (* 2 *)

   tmpy := YRGB * Load32(y^[i]);                        (* 2 *)
   r:=PBSR(tmpy + VR * thisV,8);                        (* 3 *)
   g:=PBSR(tmpy + UG * thisU + VG * thisV,8);       (* 5 *)
   b:=PBSR(tmpy + UB * thisU,8);                        (* 3 *)

   display^[i]:=      BSL(High(r),16)                   (* 2 *)
                  +   BSL(High(g),8)   + High(b);       (* 5 *)
   i := i+1;                                            (* 1 *)
   display^[i]:=      BSL(Low(r),16)                    (* 2 *)
                  +   BSL(Low(g),8)+ Low(b)         (* 5 *)
   i := i+1;                                            (* 1 *)
   …                                                    (* 29 *)
   ui:=ui+1; vi:=vi+1;                                  (* 2 *)
UNTIL i>=LineLength;                                    (* 2 *)
```

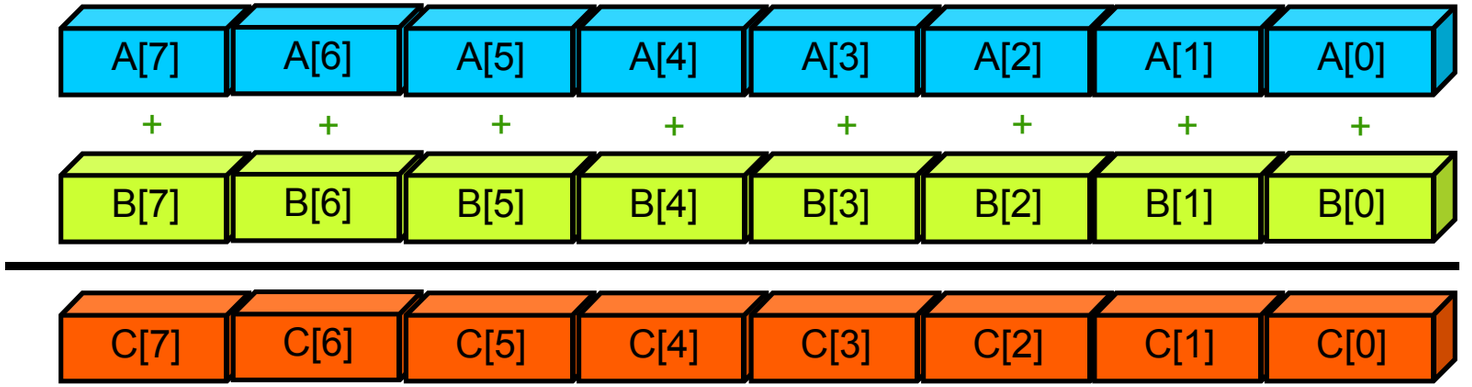
• 66 Instruktionen für 4 Pixel: 16,5 Inst/Pixel

- HP PA-RISC 7100LC, 8000
  - HADD,ss ra, rb, rt
  - HADD,us ra, rb, rt
  - HSUB, HAVE (Mittelwert)
  - HSHLADD, HSHRADD als Teil der Multiplikation
- Bsp: MPEG Video Dekompression (352*240 Pixel)



- Single Instruction Multiple Data
  - Schleifen ausrollen

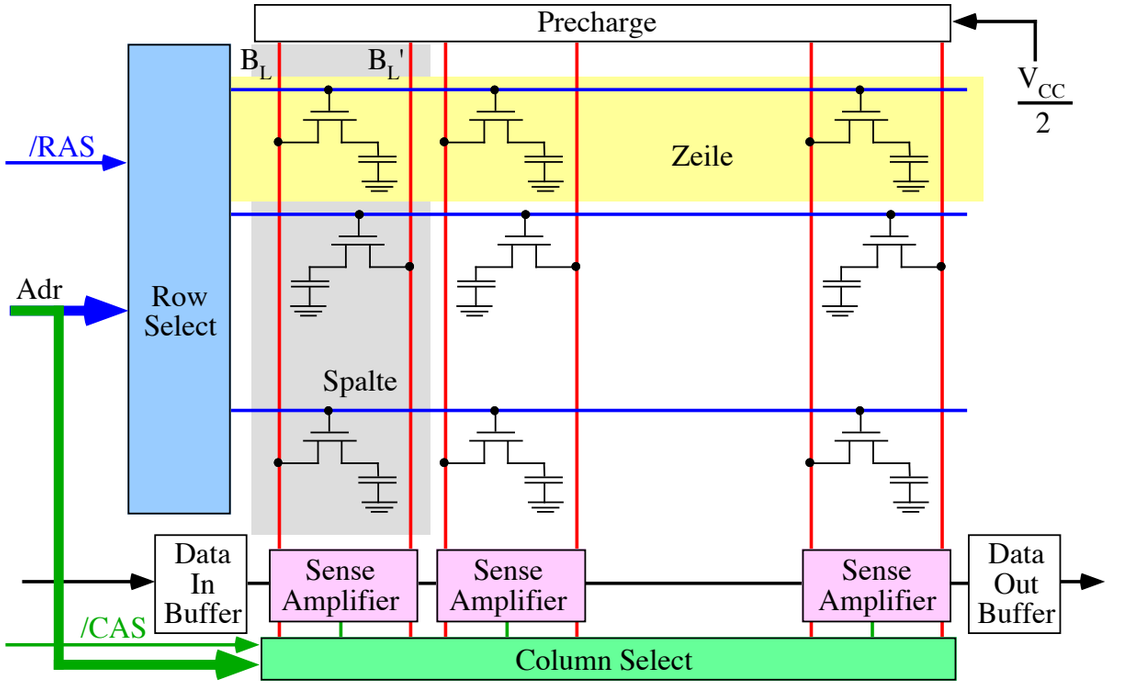
```
for (i=0;i<=MAX;i++)
  c[i]=a[i]+b[i];
```

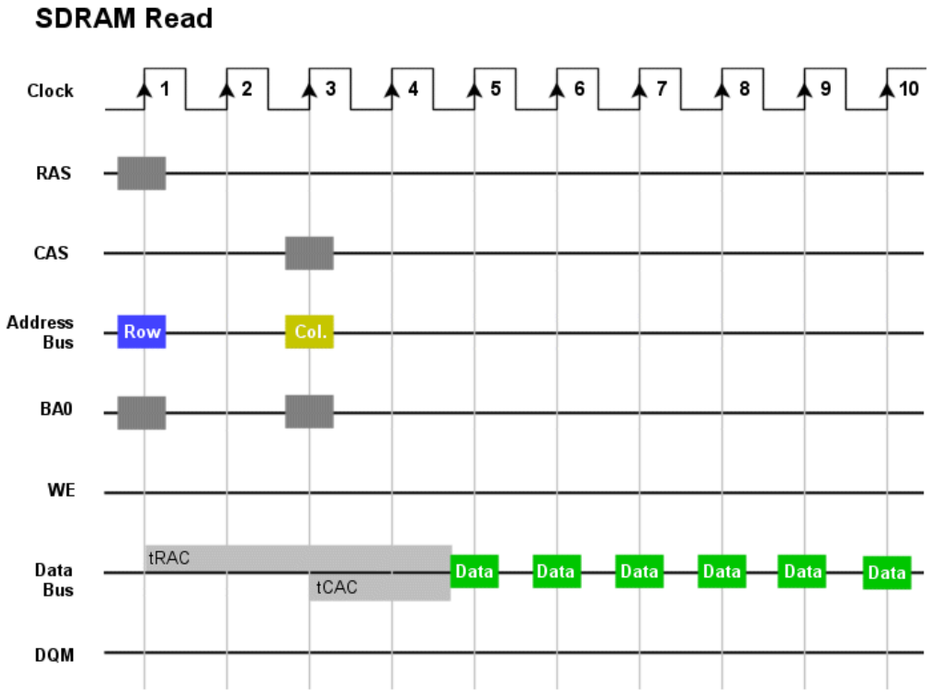


[Wells, Hewitt; Intel, 2011]

- Weniger Instruktionen
  - parallele Ausführung in einem Takt
  - Daten effizient bereitlegen?
  - breite Register, aber wie kommen die Daten zum Prozessor?

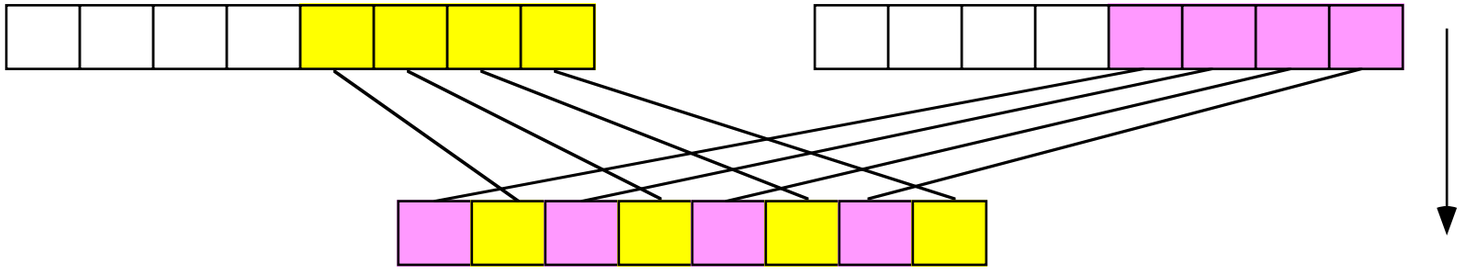
- DRAM zeilenweise strukturiert
  - RAM-Zeilen z.B. 4096 bit
  - z.B. 64 Bit Bus
  - Burst mit vier Worten: 256 Bit
  - Vektorrregister 256 Bit
  - passt auch zur Cache-Zeile
  - nicht alignierten Daten?



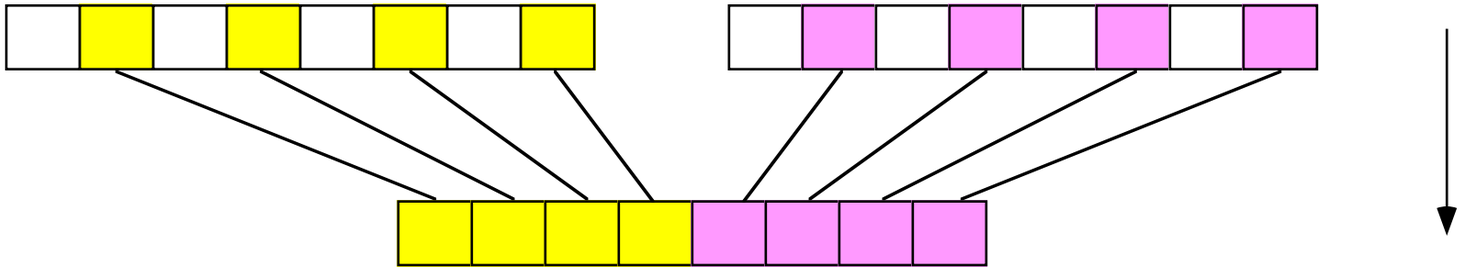


- Intel Pentium MMX
  - 8 64-bit-Register, mm0 .. mm7
  - entweder die FP's nach Umschaltung oder zusätzlich
  - 64 = 2*32 = 4*16 = 8*8
  - 57 zusätzliche Instruktionen
- Parallele Operationen
  - paddw dest, source ; add words
  - paddusb dest, source ; add bytes saturated
  - psrlw dest, count ; shift word right
- Parallele Multiplikation und Multiply-Add
  - pmulhw dest, source ; multiply word: 16*16=>32
    ; 4 high-words -> dest (oder low words)
  - 3 Prozessorzyklen, aber Pipeline
  - pmaddwd dest, source ; multiply and accumulate word
- Testen umständlich
  - pcmpeq[b,w,d] reg, reg/mm ; auch gt
  - Ergebnisse als 00/11 gepackt in Register dest

- Laden und Packen
  - movq dest, source ; 8 byte
  - punpcklbw dest, source ; unpack low bytes to word

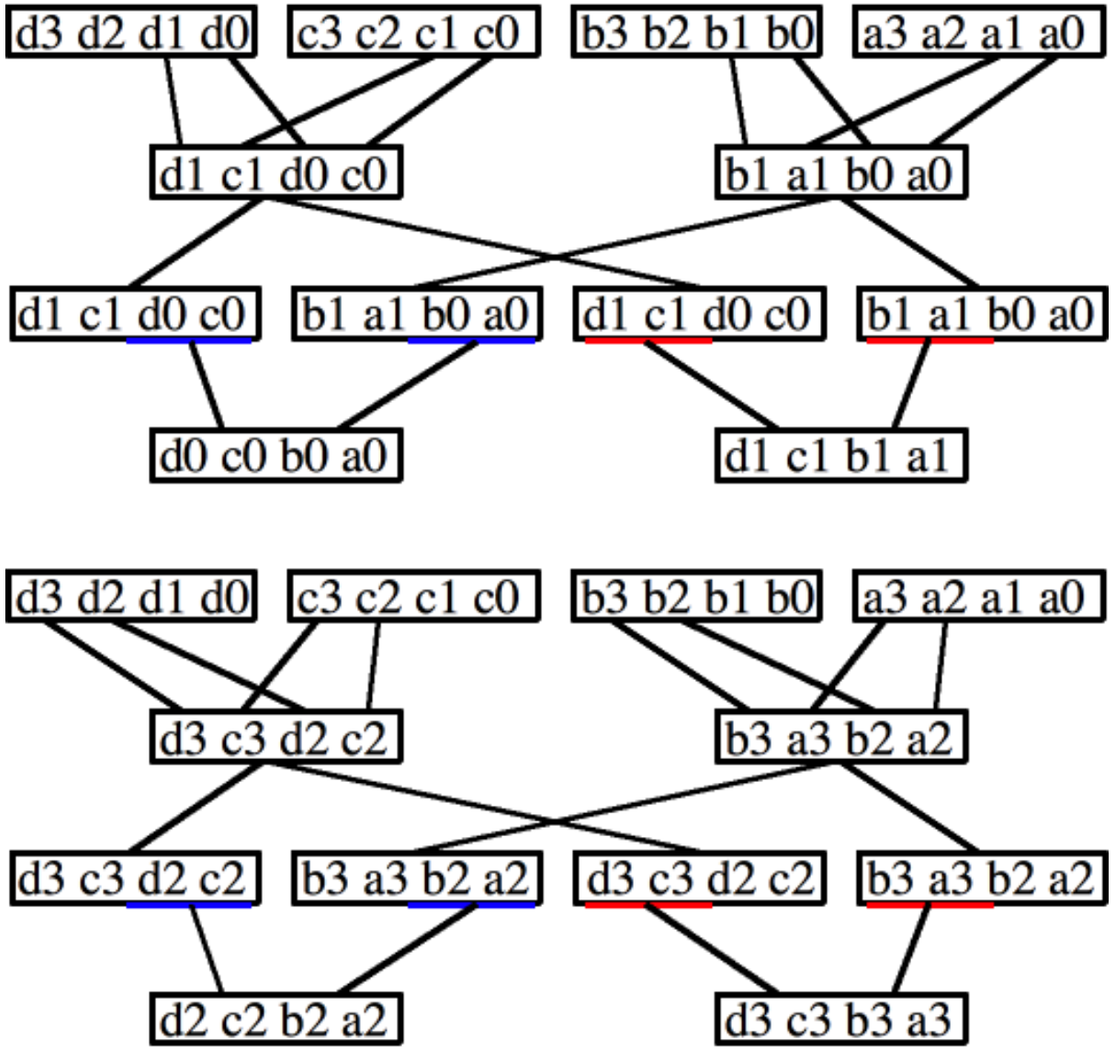


  - packuswb dest, source ; dest:= d6 d4 d2 d0 s6 s4 s2 s0
    ; unsigned saturation



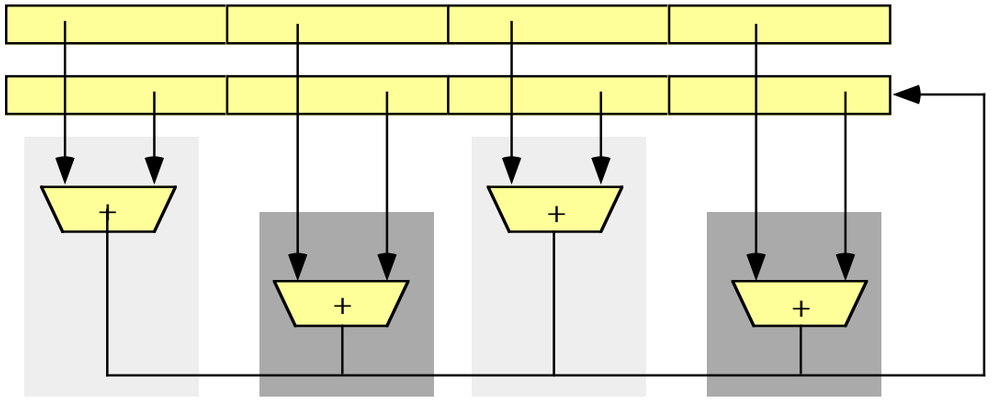
- Matrix-Transposition: 4*4

```
moveq       mm1,row1
moveq       mm2,row2
moveq       mm3,row3
moveq       mm4,row4
punpcklwd   mm1,mm2
punpcklwd   mm3,mm4
moveq       mm6,mm1
punpckldq   mm1,mm3   ;1. Zeile
punpckhdq   mm6,mm3   ;2. Zeile
moveq       mm5,mm1
moveq       mm1,row1
moveq       mm3,row3
punpckhwd   mm1,mm2
punpckhwd   mm3,mm4
moveq       mm7,mm1
punpckldq   mm1,mm3   ;3. Zeile
punpckhdq   mm7,mm3   ;4. Zeile
```

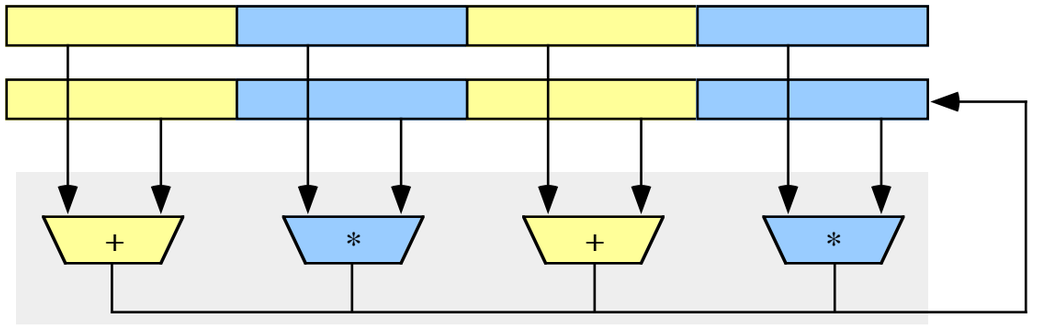


- SIMD in Mikroprozessoren
- Motorola Altivec
  - G4-Kern, FPU
  - AltiVec-Erweiterung
  - 32 Vektor-Register je 128 Bit
  - Vector ALU: Integer und Floating Point SIMD:
    - 16*8 Integer
    - 8*16 Integer
    - 4*32 Integer und Float
  - Vector Permute Unit: pack, unpack, load, store
- Intel MMX
  - Pentium-Erweiterung
  - Integer SIMD
  - Wiederverwendung von Pentium-Einheiten
  - 8 Vektor-Register je 64 Bit (= 8 FPU-Register)
  - Umschaltung FPU-MMX

- 3DNow! und SSE (Streaming SIMD Extensions)
  - MMX und Floating Point SIMD (32 bit Float)
  - 8 Vektor-Register je 128 Bit
  - Mischverwendung möglich …
  - Ausführung von 2 * 2-SIMD
  - 4-MAC pipelined
  - Produkt (4x4)*(4) aufwendig
  - (a*b)+(c*d)+(e*f)+(g*h) in 5 Zyklen

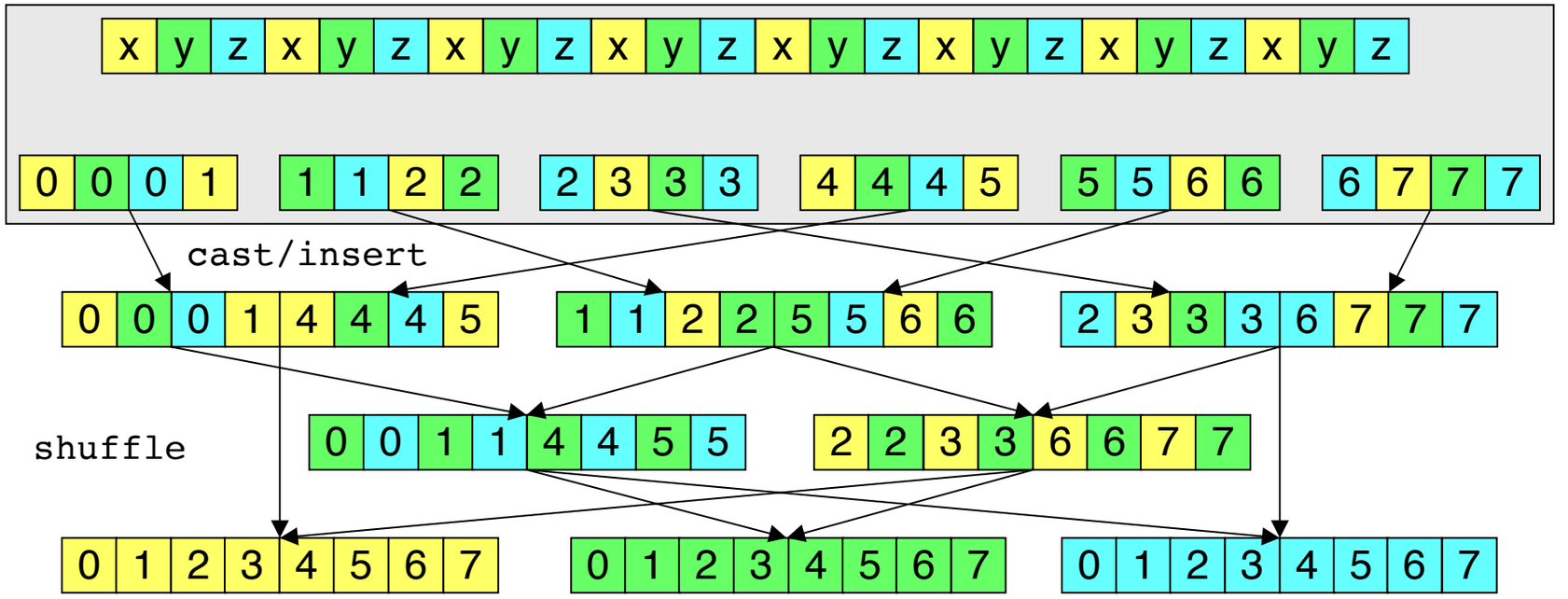


- SSE2
  - Vektor-Reg <> FP-Reg
  - double Precision Float



- SSE3: DSP-artige Befehle
- SSE4
  - Skalarprodukt: DPPS
  - MPSADBW: Differenz von 8x8 Blöcken in 7 Zyklen
- SSE5 (AMD): Fused MAC
- ARM Neon: 64 und 128 Bit Vektoren
  - MP3-Decode @ 10Mhz, GSM Adaptive Multi Rate @ 13 MHz
- ARM SVE (Scalable Vector Extension): Instruktionsformat für 2048 bit Vektoren

- AVX (Intel): Advanced Vector Instructions
  - Basis 64 Bit ISA
  - teilweise 256 Bit Vektoren
  - komplexeres Instruktionsformat
  - auch 3 und 4 Operanden
  - Fused Multiply Add
  - Instruktionsformat für bis zu 1024-Vektoren
- Speicherstrukturen
  - AOS: Array of Structures
  - SOA: Structure of Array
  - Bsp: 256 AOS -> SOA

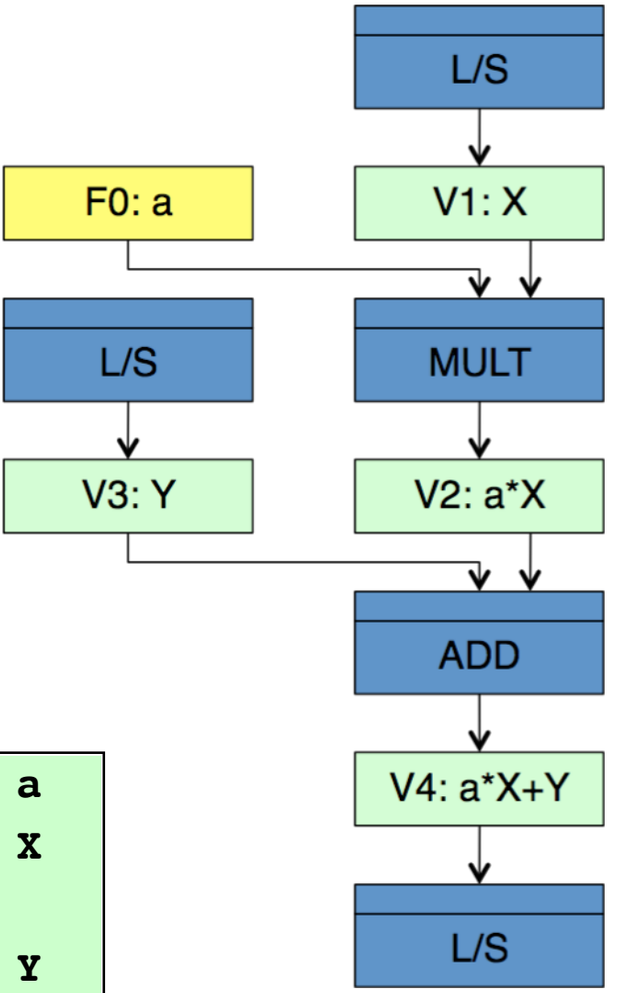


• C-Code AOS -> SOA [Melax, S.: 3D Vector Normalization 2011]

```
 float *p;  // address of first vector
__m128 *m = (__m128*) p;
__m256 m03;
__m256 m14;
__m256 m25;
m03  = _mm256_castps128_ps256(m[0]); // load lower halves
m14  = _mm256_castps128_ps256(m[1]);
m25  = _mm256_castps128_ps256(m[2]);
m03  = _mm256_insertf128_ps(m03 ,m[3],1);  // load upper halves
m14  = _mm256_insertf128_ps(m14 ,m[4],1);
m25  = _mm256_insertf128_ps(m25 ,m[5],1);

__m256 xy = _mm256_shuffle_ps(m14, m25, _MM_SHUFFLE( 2,1,3,2));
__m256 yz = _mm256_shuffle_ps(m03, m14, _MM_SHUFFLE( 1,0,2,1));
__m256 x  = _mm256_shuffle_ps(m03, xy , _MM_SHUFFLE( 2,0,3,0));
__m256 y  = _mm256_shuffle_ps(yz , xy , _MM_SHUFFLE( 3,1,2,0));
__m256 z  = _mm256_shuffle_ps(yz , m25, _MM_SHUFFLE( 3,0,3,1));
```

- Klassische Vektorrechner
  - z.B. Cray
  - 64 Elemente im Vektor
  - besonders Register enthält Vektorlänge
  - kein Schleifenoverhead
- Vektorrechner: Instruktion Load/Store Vector
  - Load über mehrere Speicherzyklen
  - linear im Speicher
  - gespreizter Speicherzugriff (stride)
  - maskengesteuert
  - Listenvektor (gather und scatter)
- Chaining
  - nur ein Stall am Anfang

- DAXPY
  - double a*X+Y
  - Linpack




```
L.D     F0,a          ;load scalar a
LV      V1,X          ;load vector X
MULVS   V2,V1,F0   ; mult
LV      V3,Y          ;load vector Y
ADDV.D  V4,V2,V3   ;add
SV      Res,V4     ;store the result
```

- Vektorisierende Compiler
  - Schleifen mit Vektoren finden
  - Ausrollen
  - Peeling
  - Re-Rolling

```
void add(char *a, char *b, int n) {
      for (int i = 0; i < n; i++)
         a[i] += b[i];              }
```

```
int p = 10;
for (int i=0; i<10; ++i)
{ y[i] = x[i] + x[p];
  p = i;                    }
```

```
y[0] = x[0] + x[10]; /*peeled off */
for (int i=1; i<10; ++i)
{ y[i] = x[i] + x[i-1]; }
```

- Bsp: Fallunterscheidung parallelisieren
  - Listenvektoren M_i und L_i

```
for (i=1;i<=n;i++) {
    a[i] = sqrt(b[i]*c[i]);
    if (a[i] > ALPHA)
       x = (a[i]+1.0)*S;
    else x = 0.0;
    d[i] = COF*x + e[i]; }
```

A_i = SQRT(B_i * C_i) i=1,2,...,N
M_i = a_i > ALPHA i=1,2,...,N
x_i = (a_i + 1.0) * Smask M_i, i=1,2,...,N
L_i = .NOT. M_i i=1,2,...,N
x_i = 0.0 mask L_i, i=1,2,...,N
D_i = COF * x_i + e_i i=1,2,...,N

- Probleme
  - Prozeduraufrufe in der Schleife
  - Schleifen irregulär verlassen
  - Datenabhängigkeiten
  - Pointer Aliasing
  - Daten nicht richtig aligniert im Memory (16, 32 Bit)
- Lösungen
  - Array of Structures
  - Compiler versucht 'strided access'
  - Lade-Instruktionen zum Auspacken

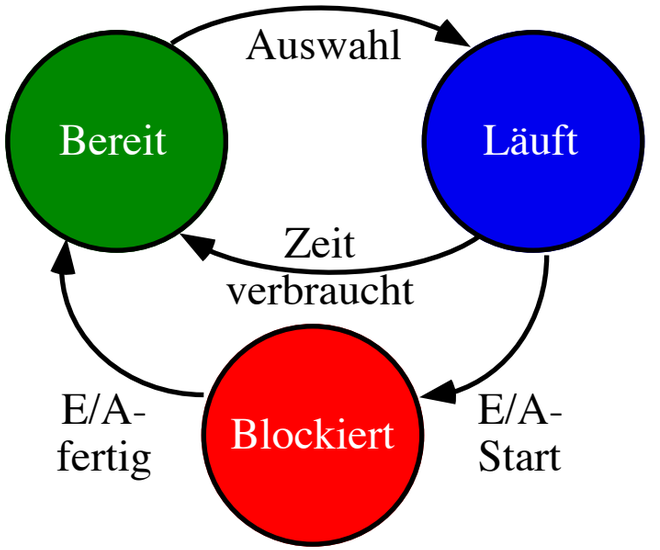
```
const size_t Width = 4;
typedef struct
{    float a[Width];
     float b[Width];
```

```
float *A = &in[0].a;
float *B = &in[0].b;
const unsigned int Stride=2;
for (int i = 0; i < n; i++)
   C[i] = A[i*Stride] + B[i*Stride];
```

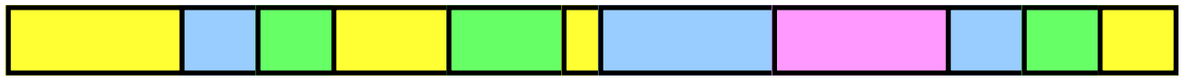
- Abhängigkeiten finden und evtl. beseitigen (WAR, WAW)
- Standard-Transformationen Schleifen
- Normalisierung: Anfangswert und Inkrement auf 1
- Dead-Code eliminieren
- Subscript Normalization
  - Scalar Forward Substitution: Ausdrücke statt Zwischenvariablen
  - Beispiel: Indexberechnungen (<-> Redundant Expression Elimination)
  - Induction Variable Substitution: Index in der Schleife inkrementiert (dek.)
  - Wrap-Around Variable Substitution
- Loop Fission (Loop Distribution)
- Scalar Expansion: Skalare Werte durch Array ersetzen
- Variable Copying
- Index Set Splitting
- Node Splitting im Abhängigkeitsgraphen

1.2.0 Einschub: Prozesse und Threads

- Prozesse
  - selbständige Codeeinheiten
  - Object-Code
  - Speicher
  - Attribute



- Multi-Tasking
  - mehrere Prozesse laufen 'verschachtelt'
  - "Zeitmultiplex"
  - für den Benutzer gleichzeitig
  - verschiedene Strategien des Wechsels



- Prozesswechsel
  - Anhalten eines Prozesses
  - Speichern der Status-Information (PC, Register etc.)
  - Laden der Status-Information des neuen Prozesses
  - Wiederaufnahme der Ausführung bei der nächsten Instruktion
  - z.B. nach Zeit t, wenn gewartet werden muß, …

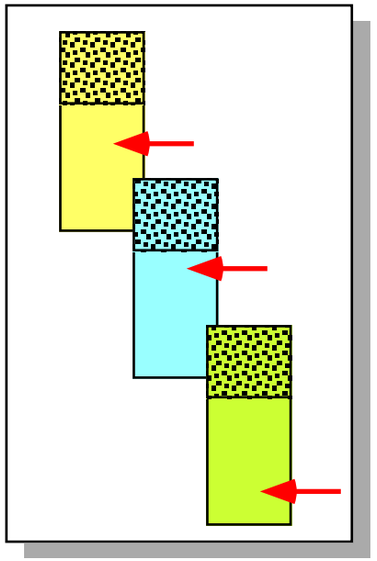
- Einplanung ≠ Scheduling
- Scheduler
  - gibt Ausführungsrecht befristet ab an Benutzerprozesse
  - verteilt Ausführungsrecht 'gerecht'
  - hat besondere Rechte
  - kennt alle anderen Prozesse
- Rückwechsel vom Benutzerprozeß zum Scheduler
  - freiwilliger Sprung in den Scheduler
  - erzwungen nach Fristablauf (=> Unterbrechung)
- Prozeßeigenschaften
  - Wichtigkeit (Priorität)
  - benutzte Ressourcen (Festplatte, Drucker, …)
  - verbrauchte Zeit und Deadlines
  - Adressraum
  - Zustandsinformation

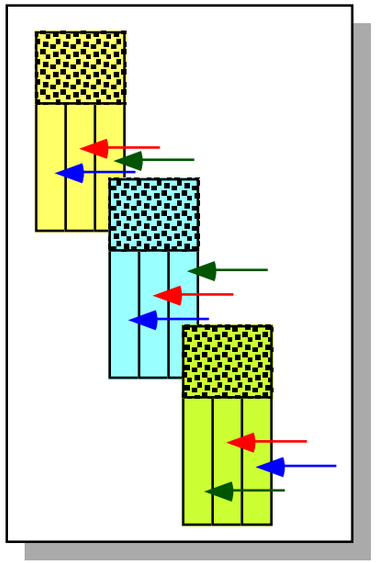
- Non-Preemptive Scheduling
  - Prozesse nicht unterbrechbar

  - Kritische Prozesse können nicht unterbrochen werden
- Preemptive Scheduling
  - Prozesse durch andere Prozesse mit höherer Priorität unterbrechbar

  - Oft in Betriebssystemen vorhanden für CPU
  - Prozesswechsel häufig und teuer
- Prioritätsfestlegung
  - Relevanz für Gesamtaufgabe
  - nahe 'Abgabe'-Termine
  - lange gelaufen => Priorität sinkt
- Problem Prioritäts-Umkehr
  - Prozess mit höherer Priorität wird gestartet
  - Priorität anderer Prozesse steigt während der Bearbeitungszeit
  - Unterbrechung des Ersten, Thrashing

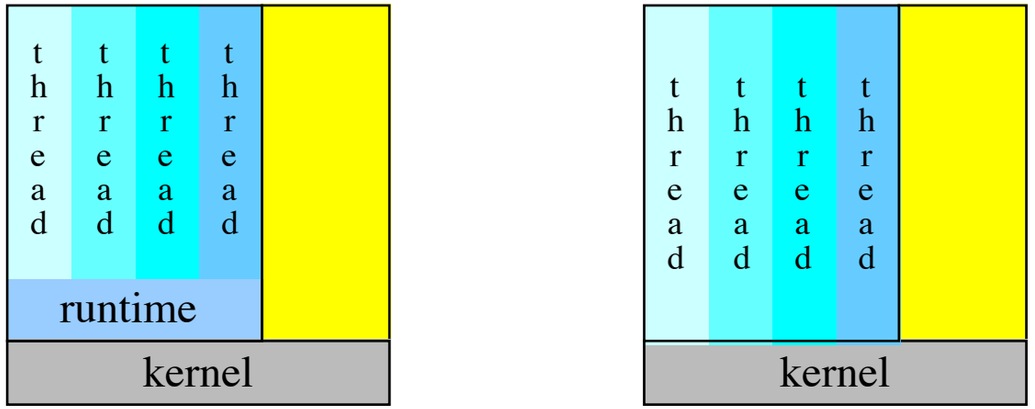
- Threads
  - lightweight process
  - Zustand
  - kein eigener Adressraum





Prozess	Thread
Instruktionszähler Register und Flags Stack Kind-Prozesse	Instruktionszähler Register und Flags Stack Kind-Threads
Adressraum globale Variablen offene Dateien Timer Signale, Semaphore Verwaltung	

- Thread blockiert, Prozess nicht
  - einfaches Programmier-Paradigma
- Implementierung von Threads
- Scheduler im Prozess (runtime)
  - blockierende Calls in der runtime umsetzen?
  - Rechte?
  - OS unverändert, portabel
  - Scheduling kontrollierbar

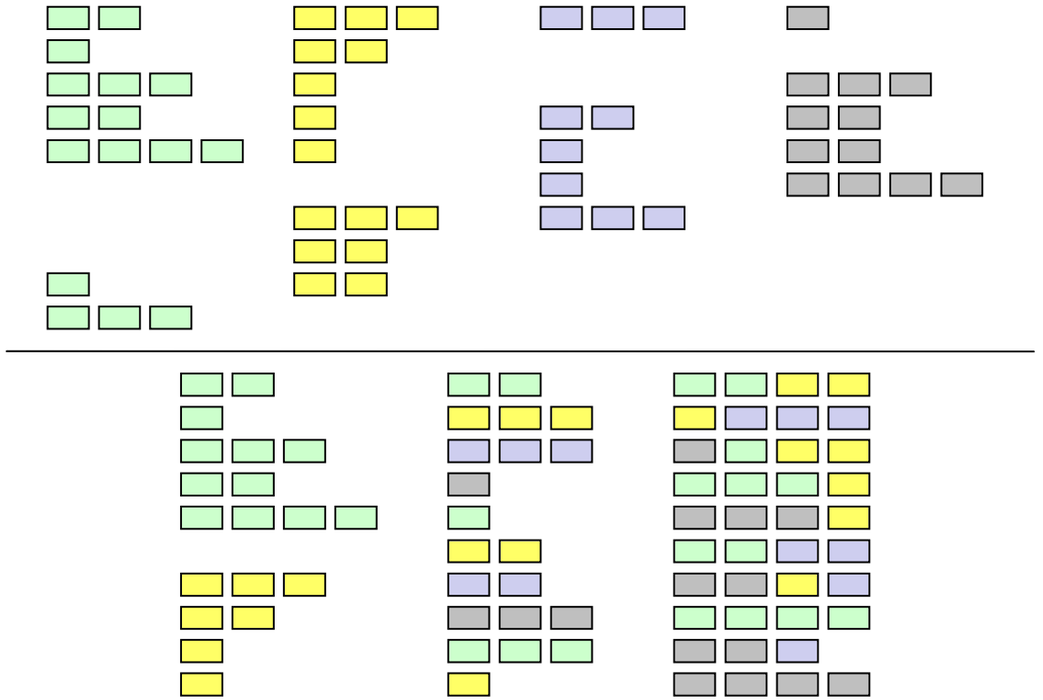


- Scheduler im Kern
  - Adressraumwechsel teuer
  - neues Betriebssystem

1.2.1 Simultaneous Multi Threading

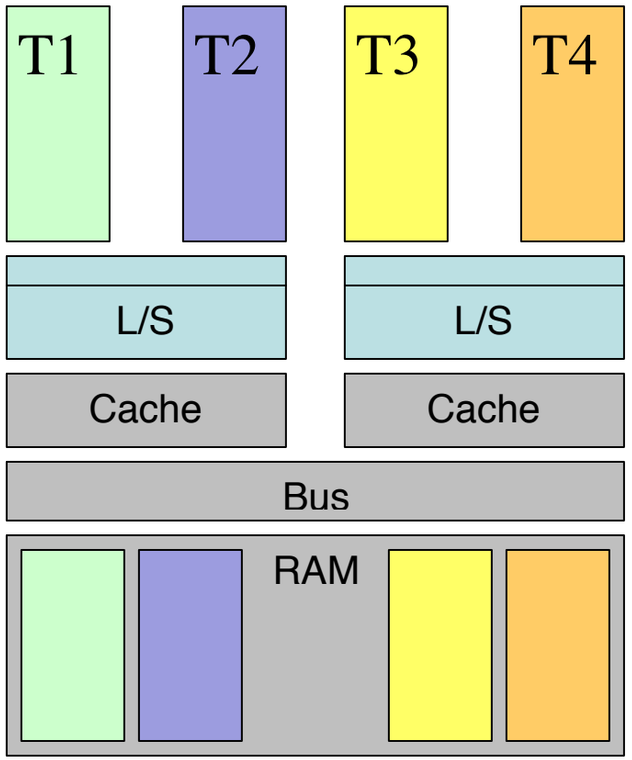
- Instruction Level Parallelism siehe VL TI und WH oben
  - Hazards erkennen, Stalls vermeiden: Scheduling
  - mehrere Ausführungseinheiten
  - mehrere Instruktionen gleichzeitig in den Funktionseinheiten
- Probleme ILP
  - echte Abhängigkeiten
  - Speicherzugriff dauert $O(10^2)$ Zyklen
  - alle anderen Instruktionen warten auf die blockierte
  - möglichst viele Instruktionen zur Auswahl?
- Thread im Sinne von SMT
  - Instruktionssequenz
  - geordnet, d.h. Abhängigkeiten, Hazards etc.
  - Betriebssystemkonzept thread hat Attribute

- Thread Level Parallelism (Hyperthreading)
  - n Threads (n=2 bei Intel Nehalem)
  - Instruktionspool zur Auswahl vergrößern
  - Abhängigkeiten zwischen Threads 'seltener'
- Granularität des Multithreading
  - coarse grain: Thread-Wechsel nur bei Cache-Miss
  - fine grain: reihum bei jeder Instruktion
  - simultaneous: multiple-issue Prozessoren



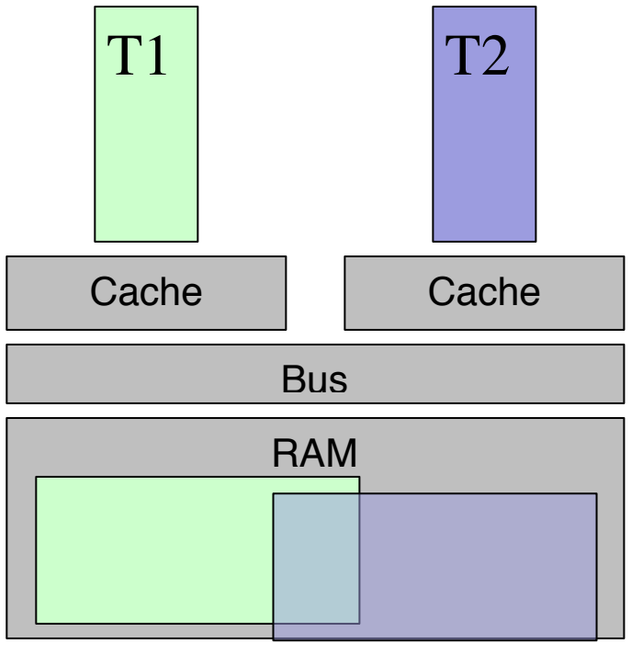
nach Patterson/Hennessey

- Front End
  - holt Instuktionsblöcke von n Threads abwechselnd
  - Dispatch wählt Instruktion aus beiden Pools
  - Functional Units unverändert
  - Reservation stations müssen Thread-Kontext kennen
- Retirement Unit
  - ordnet Ergebnisse Registern zu
  - Registersatz zweifach
- Cache SMT-unaware
  - Unterschiedliche Speicher-Zugriffsmuster
  - Thrashing möglich
- Threads mit gemeinsamen Adressraum
  - konkurrierender Zugriff
  - Synchronisation?

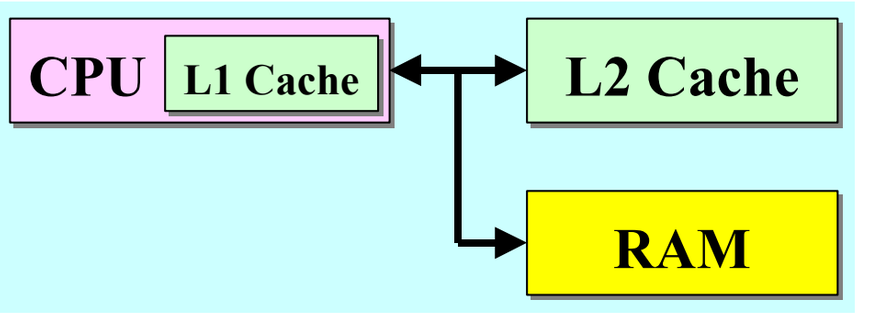


1.2.2 Symmetric Multiprocessing SMP

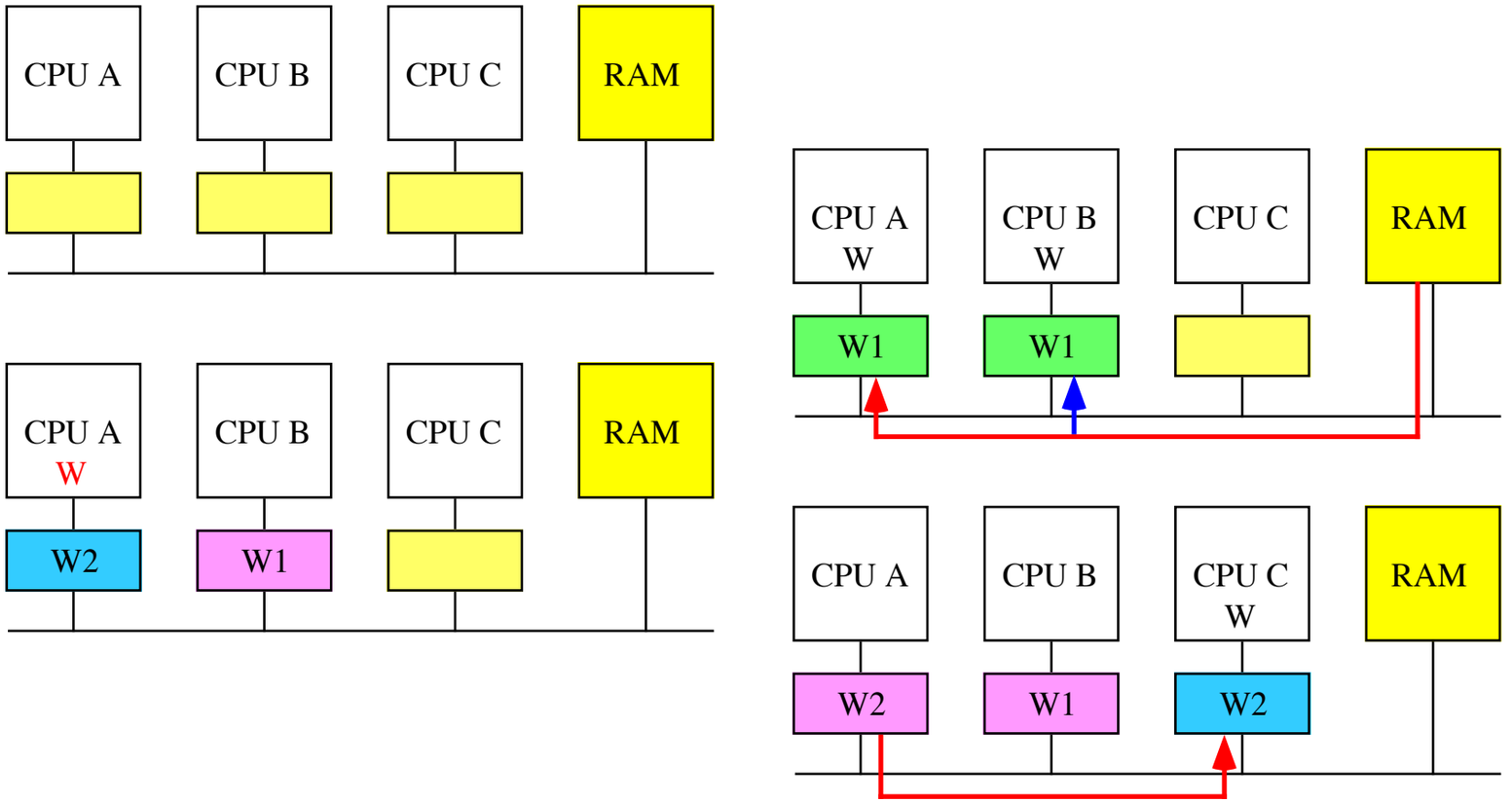
- Mehrere Cores in einem Chip
  - 2, 3, 4, 6, 8 …
  - Multicore: 2-12
  - Manycore: 80 ...
- Typische Prozessoren
  - Core Duo, Core2 Duo, i7, …
  - Athlon, Phenom
  - ARM Cortex-A9 MPCore
- Bottleneck Speicherbus
  - ein Bus vom Chip zum Speicher
  - mehrere Busse (pincount?)
  - UMA und NUMA
  - Caches um Problem zu entschärfen
  - separate Datencaches pro Core



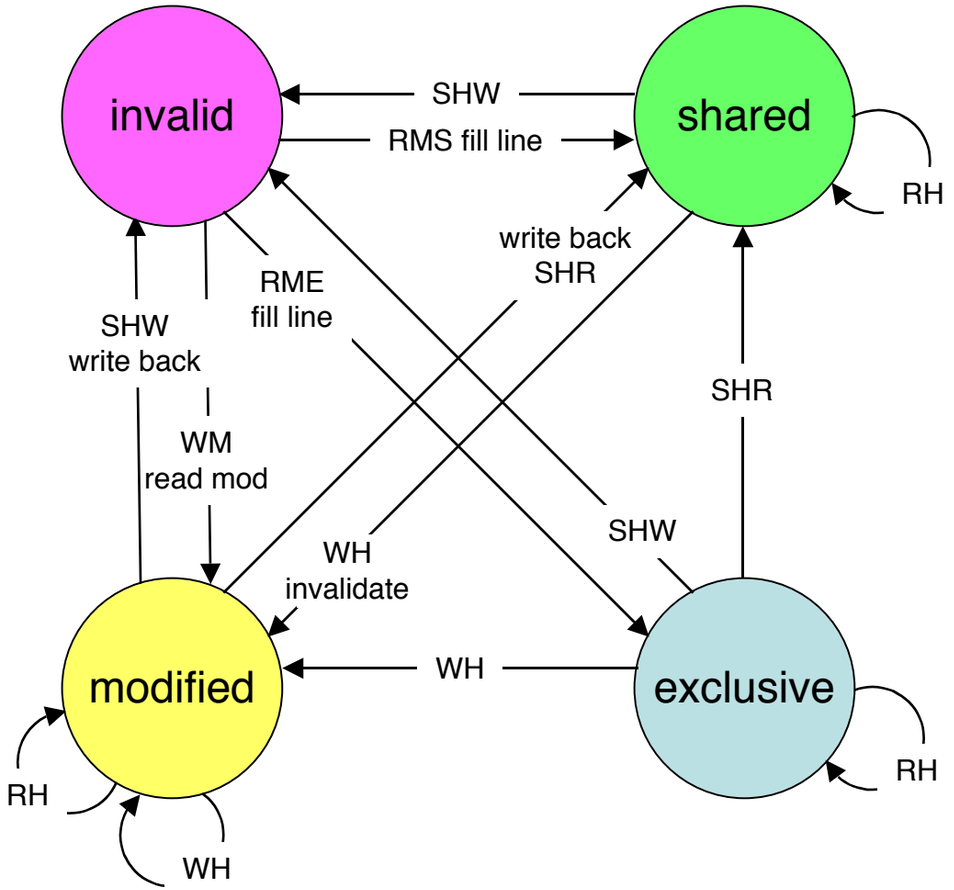
- Cache
  - zwischen Pipeline und Speicher
  - hält bereits benutzte Speicherinhalte
- Beschleunigung?
  - beim *ersten* Lesen einer Adresse Speicherwort puffern (langsam)
  - bei weiteren Lesevorgängen von Adresse Wort sofort liefern
  - beim Schreiben puffern und evtl. asynchron schreiben
- Blockstrukturiert (Cachelines)
  - Cache speichert größere Speicherstücke
- Write-Through
  - Cache-Inhalt schreiben
  - Verzögerungen durch Speicher
- Write-Back
  - Modifikationen im Cache durchführen
  - Cache-Inhalt erst beim Ersetzen der Zeile zurückschreiben
  - Dirty-Bit markiert beschriebene Cache-Zeile
  - Konsistenzproblem zwischen Cache & Hauptspeicher



- Caching Multiprocessor und MSI
  - bus snooping
  - write-back oder write-through Strategie
  - sharead (clean), invalid, modified (dirty)
  - Protokoll in einem Buszyklus

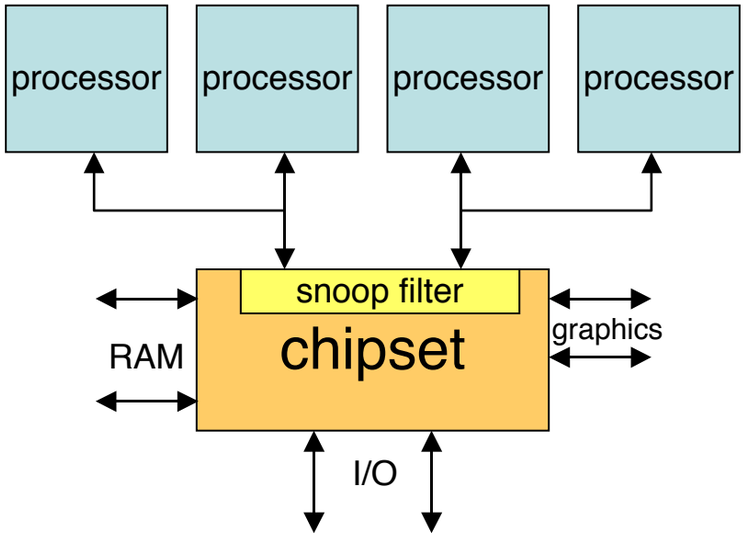


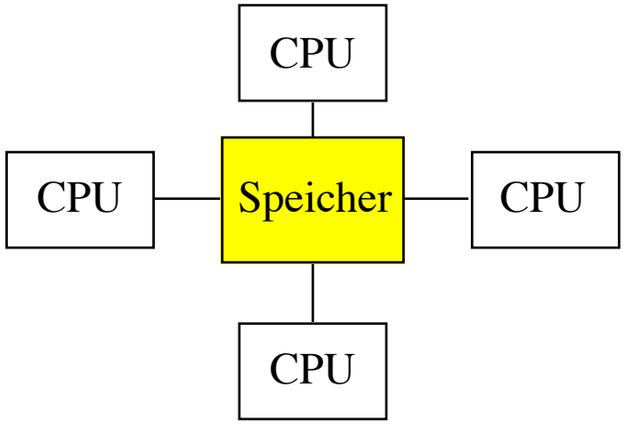
- MESI
  - modified: nur in diesem Cache, dirty
  - exclusive: zum Lesen, nicht geändert, kann shared werden
  - shared: in mehreren Caches, nicht geändert
  - invalid



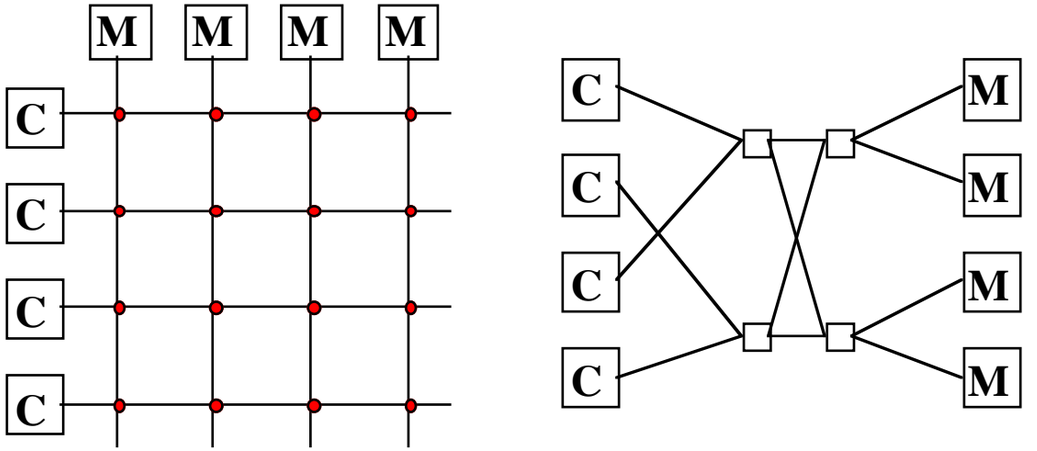
RH	Read Hit
RMS	Read Miss Shared
RME	Read Miss Exclusive
WH	Write Hit
WM	Write Miss
SHR	Snoop Hit Read
SHW	Snoop Hit Write

- UMA: Uniform Memory Access
  - mehrere Prozessoren, ein Speicher
  - Zugriffszeit für alle Prozessoren gleich
  - busbasiert
  - Speichervermittlung
  - Netzwerk
- Beispiel Intel Core2 Duo, Quad
  - busbasiert
  - chipset enhält switch
  - snoop filter für MESI
- Multiport Memory (Dirmu)
  - teure Spezialhardware
  - evtl. Speichervermittlung etc.



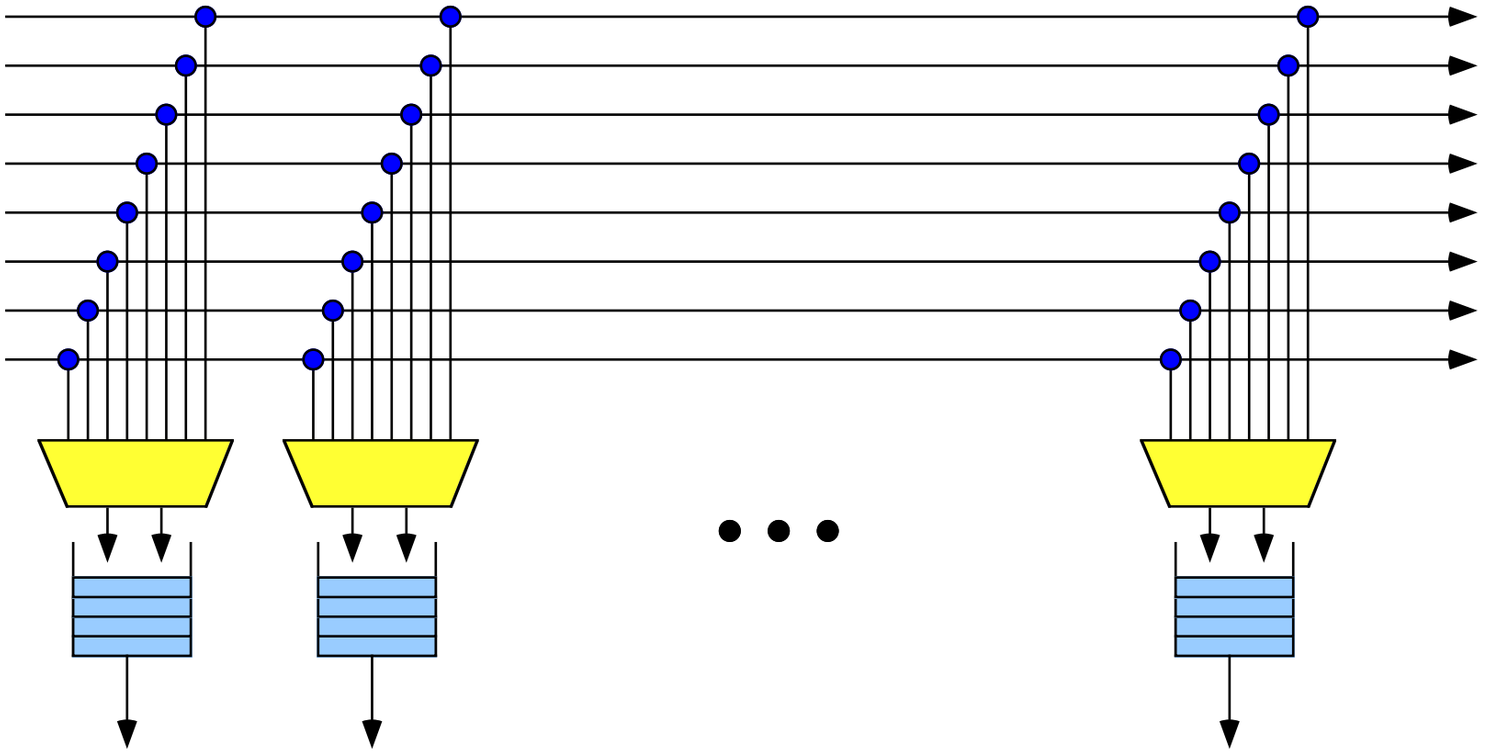


- Verbindungsnetzwerk
  - Crossbar (Kreuzschienenverteiler)
  - n^2 Schalter, Zugriff O(1)
  - Spezialnetze: Butterfly, Banyan, Batcher, Omega
  - n log(n) Schalter, Zugriff O(n) - teure Spezialhardware
  - evtl. Speichervermittlung etc.

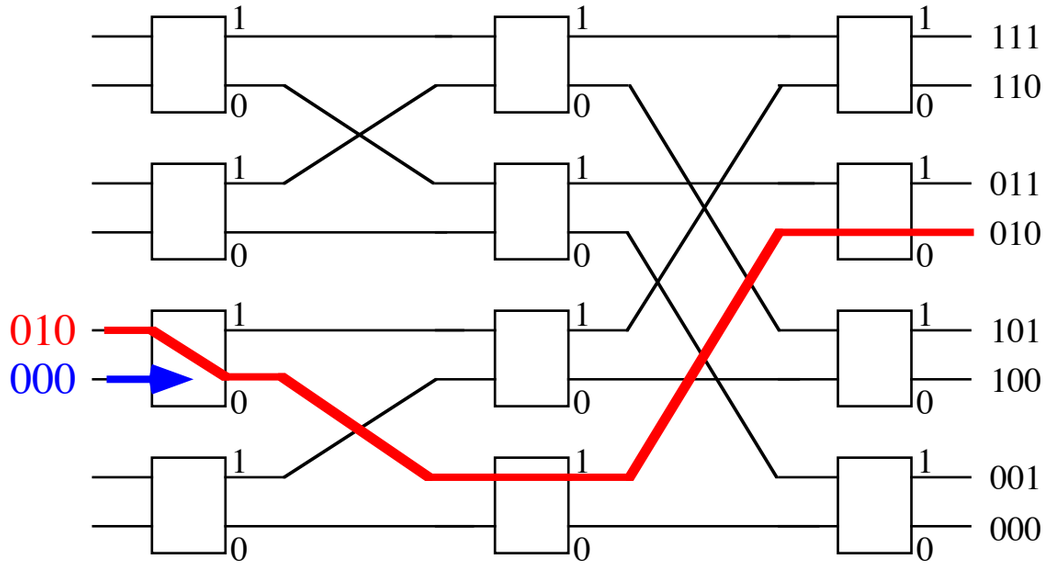


Kreuzschienenverteiler Omega Netz

- Crossbar mit Knockout
  - Arbiter entscheiden über Annahme der R/W Operation
  - steuern Output-Fifo
  - eventuell mehrstufig

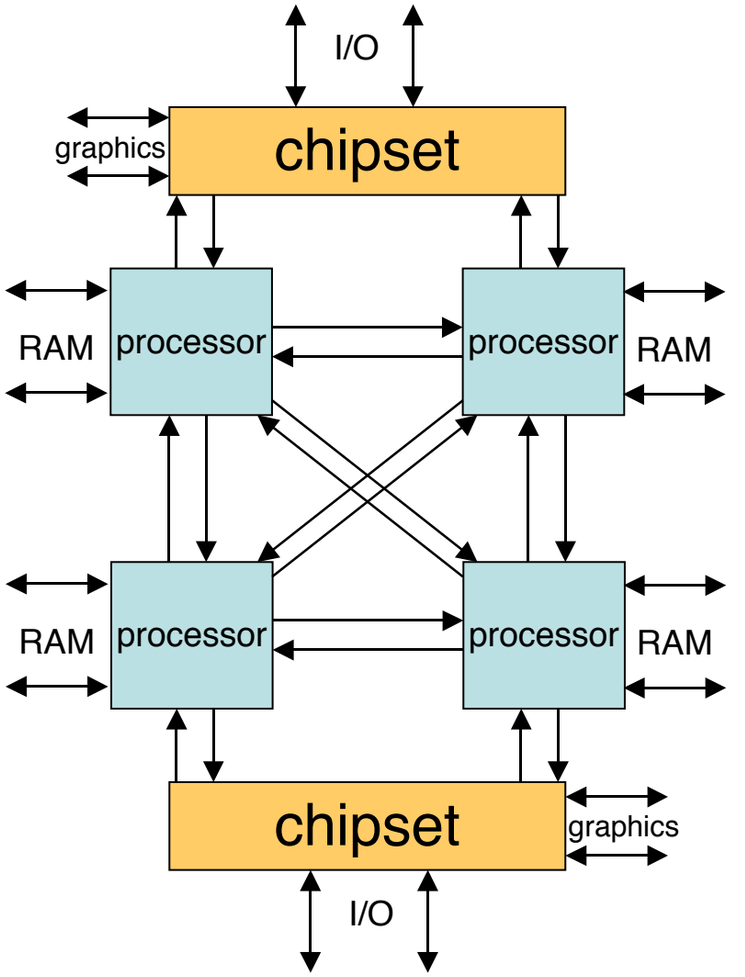


- Selbststeuernde Vermittlungsnetzwerke
  - Banyan-Delta
  - Puffer an Ein- und Ausgängen



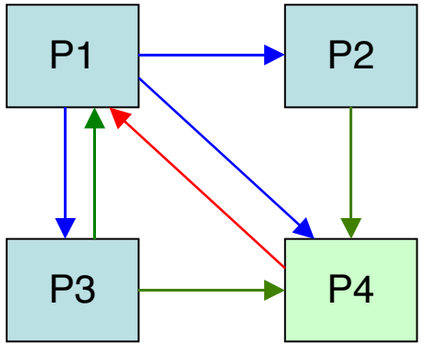
- Alternativen: Butterfly, Batcher, Omega, Perfect Shuffle
- Kollisionen an Schaltern
- Multicast ?

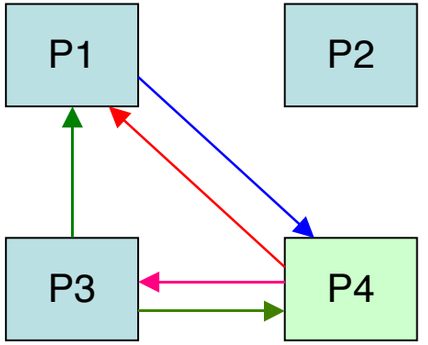
- NUMA: Non Uniform Memory Access
  - Zugriffszeiten unterschiedlich
  - Speicher z.B. Prozessoren zugeordnet
  - direkter Zugriff
  - vermittelter (langsamer) Zugriff
- Wieder Caches zur Beschleunigung
- ccNUMA
  - alle Caches kohärent
  - Zugriff transparent
  - Adress-Bewertung lokal/remote
  - z.B. bei Address Translation
- Cache Only Memory Architecture (COMA)
  - Speicher am Prozessor als Caches
  - remote-Zugriff -> Migration
- nccNUMA
  - nur lokale Zugriffe cachen
  - meist explizite Lade-Anweisung



- ccNUMA Beispiele
  - Intel Nehalem, Sandy Bridge, Itanium
  - AMD Opteron
  - DEC Alpha 2164
  - …
- Intel Quickpath Interconnect
  - verbindet Prozessoren und I/O Hubs
  - Prozessor mit Memory Controller
- QPI Physical
  - 84 pin = 42 xmit + 42 recv
  - vgl. 150 pin FSB
  - 2 clock Adern
  - full link 20 Lanes, vollduplex, 2 Adern
  - half link, quater link
  - phit: 5/10/20 lanes, 1 Bit pro Taktflanke und Lane

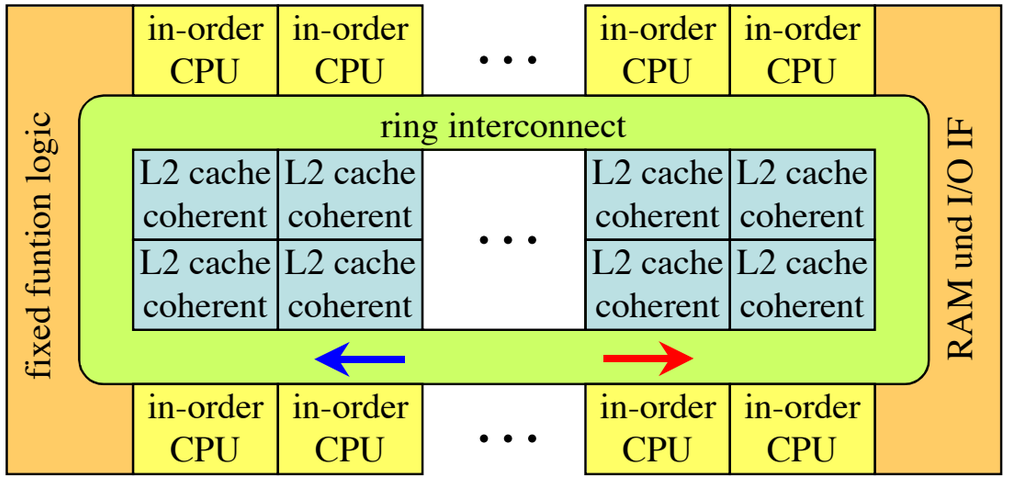
- QPI Link
  - flit = 4 phits: 80 Bit (64 Daten, 8 CRC, 8 Header)
  - auch 8 oder 16 phit auf schmaleren Links
  - Header: 14 Messages, 6 definiert
  - Data Response, no Data Response
  - home, snoop, non-coherent Standard, nc-Bypass
  - kreditbasierte Flusskontrolle
  - CRC über ganzes Flit
  - Retry mit cycle control flit
- QPI Routing: Routing Table in firmware
- QPI Transport: optional
- QPI Protocol
  - MESIF = MESI + F-State
  - Forward: genau eine clean copy
  - source snoop
  - home snoop (home hat Verzeichnis)



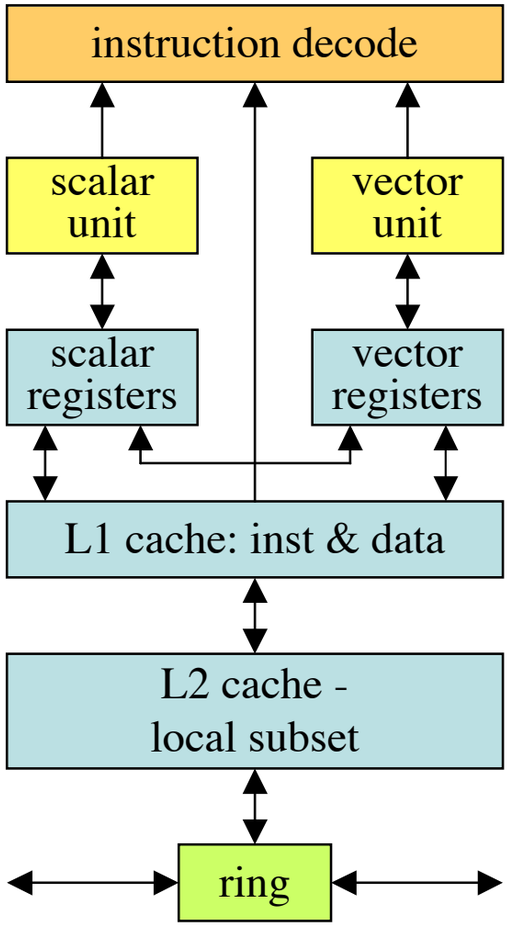


- MESIF: Forward State

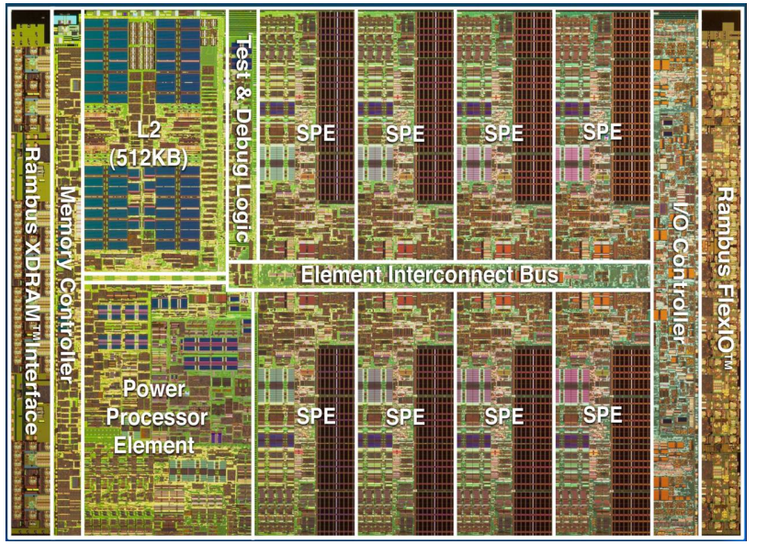
- Many-Core: Intel Larrabee [Seiler et al., 2008]
  - viele (z.B. 80) in-order Kerne
  - einfache Pipeline
  - modifizierter X86 Befehlssatz (extended …)
  - breite Vektor-Instruktionen
  - Spezial-Hardware für Texture-Filter
  - als GPU-Ersatz geplant
  - 2013: "Knights Corner" MIC (Many Integrated Core)
  - 2016: "Knights Landing"
- Vektor-Einheit
  - 16 Vektorelemente - 32 bit float
  - (= 16 Pixel von einem Farbkanal)
  - 8 Vektorelemente - 64 bit float
  - ähnlich VEX bis zu 3 Operanden
  - ein Operand aus L1-Cache

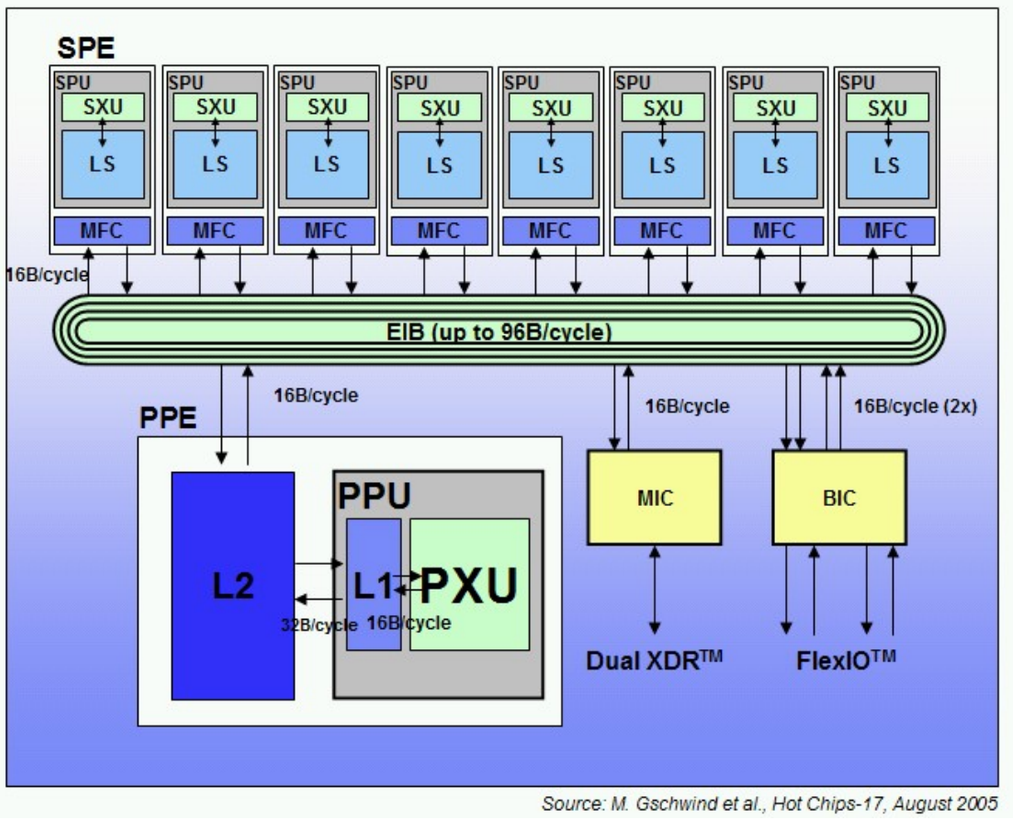


- Caches
  - 32 KB L1 Cache Instruktionen
  - 32 KB L1 Cache Daten
  - ähnlich Register, bes. für Vektoren
  - L1 gemeinsam für Threads im Kern
  - L2 cache hat 256 KB Subsets pro Kern
  - Direktzugriff für Kerne, parallel
- Cache Control Instruktionen
  - pre-fetch in L1 oder L2
  - lines markieren für 'early eviction'
  - L2 wird zu Zwischenspeicher
- Ring zur Kern-Kommunikation
  - bi-direktional, 512 bit
  - Zwischenspeichern im Kern
  - even clock Empfang von links, odd clock von rechts
  - Kohärenzprotokoll zwischen L2 caches
  - Laden-Speichern von L2 cache lines



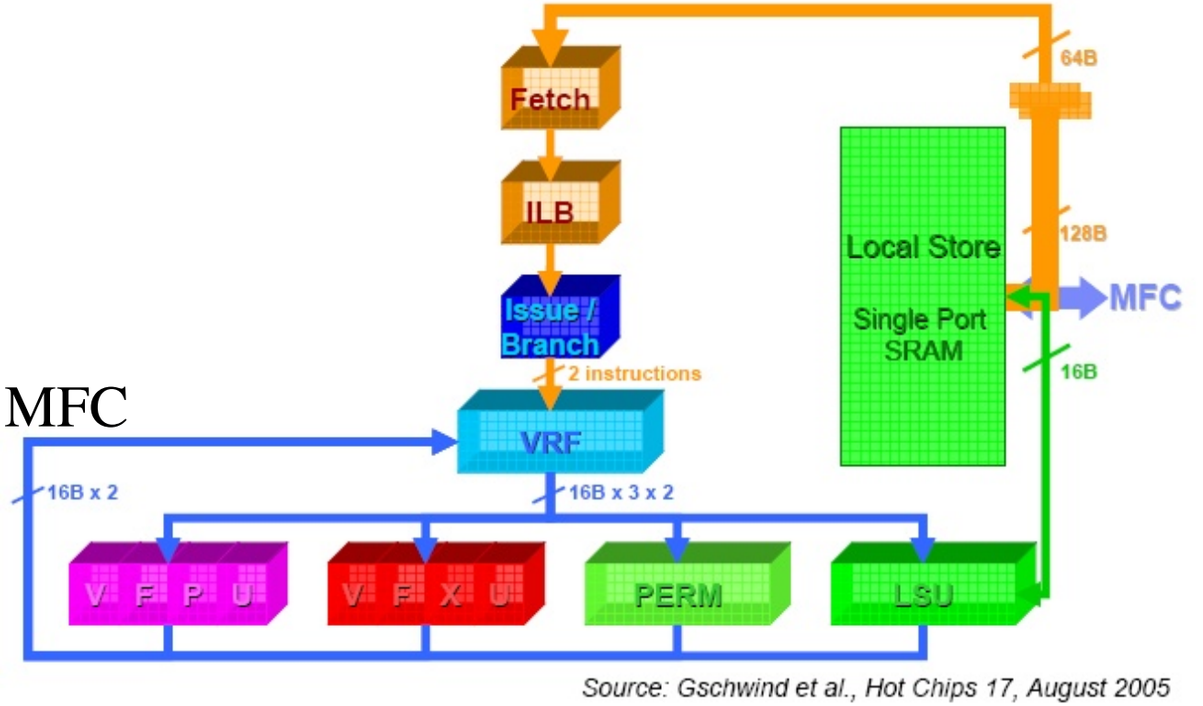
- Cell Broadband Engine (IBM)
  - heterogene Prozessorkerne
  - PowerPC evtl. mit Altivec
  - SPE: Synergistic Prozessor Elements
  - Playstation 3, HPC Roadrunner
- Architektur
  - n SPEs (z.B. 8)
  - lokaler Speicher (z.B. 256 Kbytes)
  - m 64 Bit PPE
  - EIB: Element Interconnect Bus
  - Hauptspeicher am EIB
- PPE
  - normaler PPC
  - normaler Speicherzugriff
  - Betriebssystem und I/O
  - zwei threads
  - L1 32K Daten und 32 K Inst
  - L2 512K unified





Source: M. Gschwind et al., Hot Chips-17, August 2005

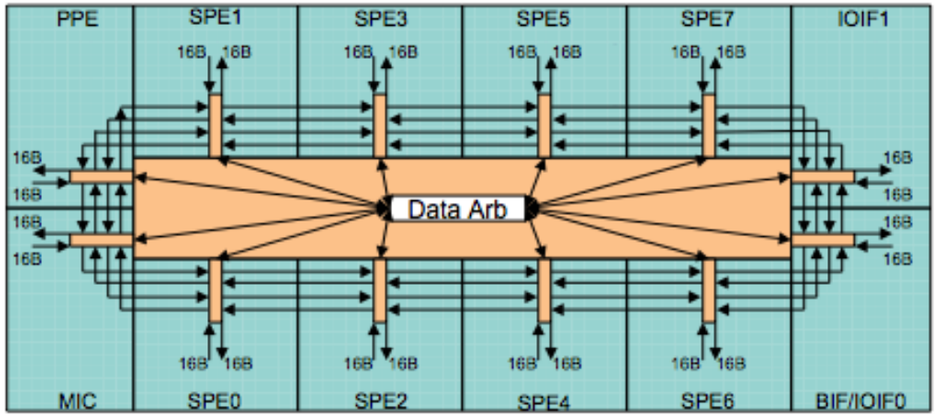
- SPE
  - einfacher Instruktionssatz
  - vektorisiert
  - Register File: 128 * 128 bit
  - 256 Kbytes lokales SRAM
  - Hauptspeicherzugriff durch MFC
- Memory Flow Controller
  - 16 asynchrone DMAs
  - 'memory level parallelism'
  - Scatter/Gather Listen
  - eine MFC pro SPE
  - kontrolliert vom SPE, PPC oder anderen SPEs
  - message-basiert, Channels
  - LS mapped auf main memory
  - oder copy-in / copy-out



Source: Gschwind et al., Hot Chips 17, August 2005

```
spe_sum_all(float*a)
{ float local_a[MAX]  __attribute__ ((aligned (128)));
  mfc_get(&local_a[0], &a[0], sizeof(float)*MAX, 31, 0, 0);
  mfc_write_tag_mask(1<<31);
  mfc_read_tag_status_all();
  for (i=0; i<=MAX; i++) sum += local_a[i];
} //Gschwind, 2006
```

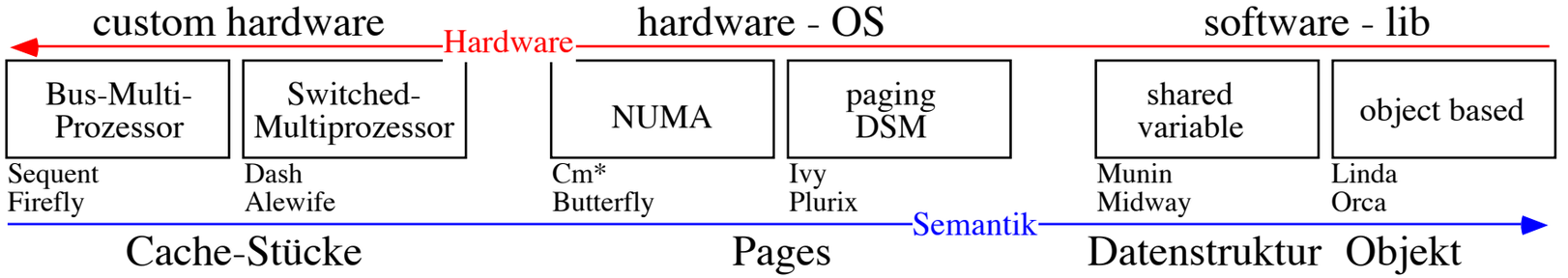
- Element Interconnect Bus
  - 4 Ringe a 16 byte
  - 128 byte cache line/Transaktion
  - Crossbar möglich
  - 204.8 GB/sec
- Roadrunner
  - Hybrid Compute Node mit Blades
  - 1 Infiniband Switches in Connected Unit: 384 GB/s
  - 8 Infiniband switches zwischen CUs



Source: Clark et al., Hot Chips 17, 2005

	Hybrid Compute Node	Connected Unit	System
PowerXCell8i	4	720	12960
RAM	16 GB	2.82 TB	50.6 TB
DP Flops	408 GF	73.4 TF	1.32 PF
Operton Dual Core	2	360	6480
RAM	8 GB	1.41 TB	25.3 TB
DP Flops	15.3 GF	2.77 GF	49.8 TF
gesamt	423 GF	76.2 TF	1.37 PF

1.3 Speichergekoppelte Multiprozessoren



- Blockbasiert
  - Maschinenbefehle greifen auf Speicher zu
  - physisch gemeinsamer Speicher, kontrollierter Zugriff
  - Caching, cache-consitency
- Seitenbasiert
  - Memory Management Unit (MMU) übersetzt Adresse
  - lokale Adresse: reale Adresse im RAM, evtl. page-fault + Paging
  - remote-Adresse: page-fault + Seite holen und einblenden
- Spracherweiterungen
  - gemeinsame Variablen bzw. Objekte
  - keine Programmiertransparenz
  - mehr Wissen über gemeinsame Objekte

1.3.1 Konsistenzmodelle

- Sequentielles Programm A:
  - ergibt 30 als Resultat in der Variablen "c"

Statements	Einfacher Stackcode
`a:= 0;`	`const(0); write(a)`
`b:= 0;`	`const(0); write(b)`
`a:= 10;`	`const(10); write(a)`
`b:= 20;`	`const(20); write(b)`
`c:= a+ b;`	`read(a); read(b); add; write(c)`

- Nebenläufige Ausführung ohne Synchronisierung
  - verschiedene Resultate möglich für c
  - 120 mögliche Permutationen (5*4*3*2)
  - zum Beispiel für drei Prozessoren sei c = { 0, 10, 20, 30 ... }

CPU #1	CPU #2	CPU #3
a:=0;	*b:=0;*	*a:=10;*
b:=20;	*c:=a+b;*	

- Inkorrekte Ausführungssequenzen
  - Verzögerung von Nachrichten im Netz
  - präemptive Prozessumschaltungen
  - die Reihenfolge der Lese- und Schreib-Operationen unbestimmt

c = 0:

CPU #1	CPU #2	CPU #3
		a:=10;
a:=0;		
b:=20;		
	b:=0;	
	c:=a+b;	

c = 10:

CPU #1	CPU #2	CPU #3
a:=0;		
	b:=0;	
		a:=10;
	c:=a+b;	
b:=20;		

- Einige Sequenzen mit korrektem Resultat:
  - pro Variable Zugriff in der richtigen Sequenz garantieren

a:=0;
b:=0;
b:=20;
a:=10;
c:=a+b;
	a:=0;
	b:=0;
	a:=10;
	b:=20;
	c:=a+b;
		b:=0;
		a:=0;
		b:=20;
		a:=10;
		c:=a+b;
			b:=0;
			a:=0;
			a:=10;
			b:=20;
			c:=a+b;
				a:=0;
				a:=10;
				b:=0;
				b:=20;
				c:=a+b;

=30	=30	=30	=30	=30

- Nicht sequenzerhaltend, aber korrektes Resultat

b:=0;
b:=20;
a:=0;
a:=10;
c:=a+b;
	a:=0;
	b:=20;
	a:=10;
	c:=a+b;
	b:=0;
		b:=0;
		b:=20;
		a:=10;
		c:=a+b;
		a:=0;
			b:=20;
			a:=10;
			c:=a+b;
			a:=0;
			b:=0;
				a:=10;
				b:=20;
				c:=a+b;
				b:=0;
				a:=0;

=30	=30	=30	=30	=30

  - für kausal nicht zusammenhängende Operationen kann die Reihenfolge auch innerhalb einer CPU vertauscht werden

- Pro Variable Zugriff in der richtigen Reihenfolge garantieren
  - z.B. Locks

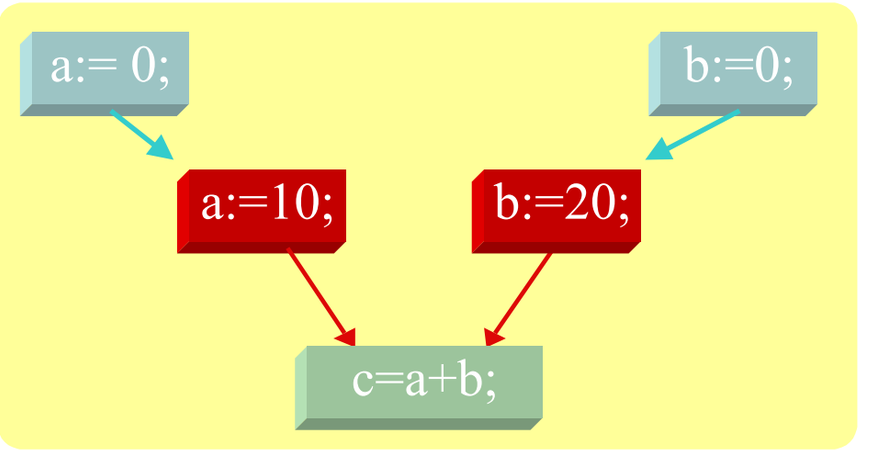
30:

CPU #1	CPU #2	CPU #3
a:=0;		
		a:=10;
	b:=0;	
b:=20;		
	c:=a+b;	

- Kausaler Zusammenhang zwischen:

  "a:=0;" und "a:=10;"
  "b:=0;" und "b:=20;"
  "a:=10;" und "c:=a+b;"
  "b:=20;" und "c:=a+b;"

  - „a:=0“ und „b:=0“ überflussig …



- Datenflussgraph
  - Abhängigkeiten zwischen Instruktionen
  - berücksichtigt „Happens before“ Relation
  - Operationen ohne Abhängigkeit können nebenläufig stattfinden
  - weitergehende Nebenläufigkeit bzw. Parallelisierung kaum realisierbar

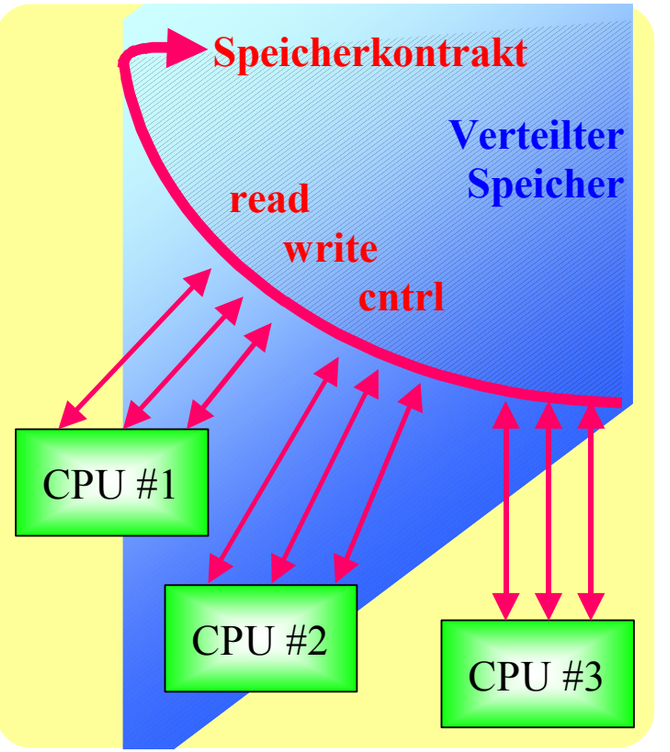
- Relaxierung:
  - Beispiel vertauscht die Reihenfolge von Instruktionen
  - trotzdem korrektes Resultat, da der Datenflussgraph berücksichtigt

30:

CPU #1	CPU #2	CPU #3
	b:=0;	
b:=20;		
a:=0;		
		a:=10;
	c:=a+b;	

- Kausal nicht zusammenhängende Operationen
  - Reihenfolge kann auch innerhalb einer CPU vertauscht werden
  - auch durch parallelisierende Compiler ausgenutzt
  - es gab Prototypen von Datenflussmaschinen
  - Abhängigkeiten zwischen Instruktionen an die Hardware delegieren
- Multiprozessorkonsistenz
  - aus Sicht der einzelnen CPUs Speicherzugriffe streng sequentiell
  - die Sicht auf den Speicher bleibt "konsistent"
  - ausgefeiltes Bus-Protokoll sorgt für scheinbar sequentiellen Zugriff

- Speicherungsmechanismus
  - Sichtbarkeit von Schreiboperationen nach verschiedenen Regeln
- Beispiele
  - sofort, bzw. getaktet
  - sichtbar nach Anforderung
  - gleiche Sicht für alle Stationen
  - konsistent aus der Sicht einer Station
  - entsprechend einer verteilten Zeitskala
  - geordnet nach Zeitstempelverfahren ...



- „Kohärenz“:
  - bezüglich einzelner Cache-Zeilen
  - bezüglich einzelner Speicherwörter
  - einheitliche Sicht nur auf einen Teil des Systemzustandes
- Was geschieht mit Variablen in Registern bei einer Prozessumschaltung ?
- Konsistenzmodell: 'Vertrag zwischen Programm und Speicher'
  - Garantien für Ergebnis des konkurrierenden Datenzugriffs
  - Ergebnismenge für gegebene Zugriffssequenz

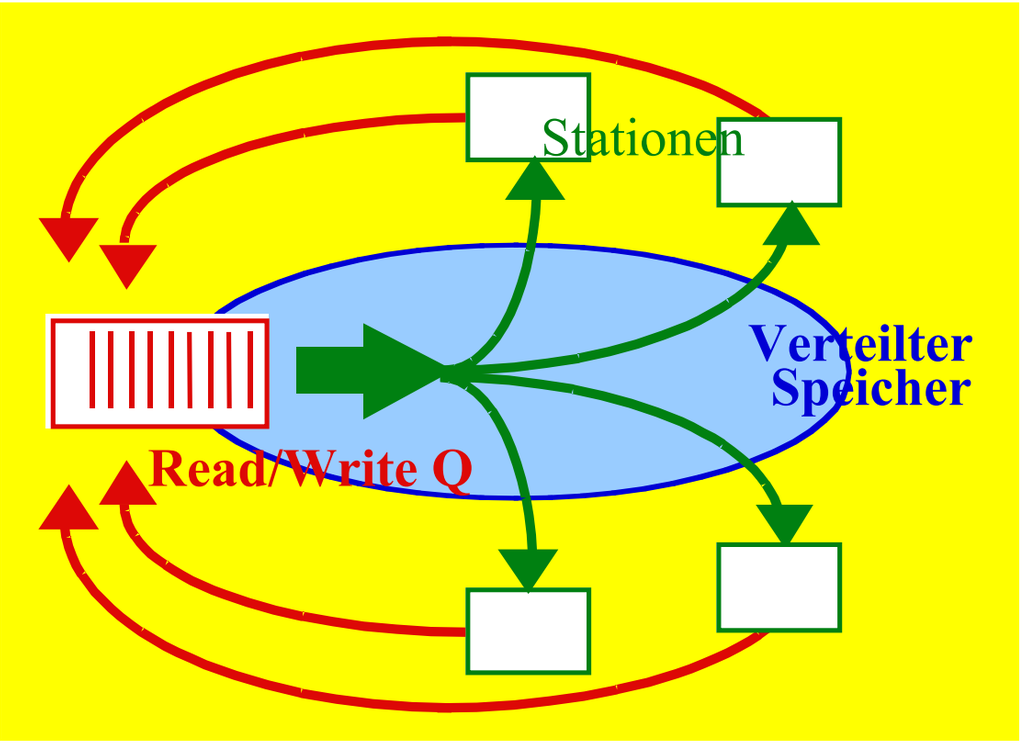
- Strikte Konsistenz
  - alle Leseoperationen liefern den 'zuletzt' geschriebenen Wert
  - unabhängig davon welcher Prozess geschrieben hat
  - globale Zeit im verteilten System?-> Begriff "zuletzt" ist unscharf
  - Lese- und Schreiboperationen in Warteschlange
  - atomare Lese- und Schreiboperationen
  - Instruktionsausführung /Warteschlangeneintrag in starrem Zeitraster

  ~ nicht verteilt

P1:	W(x,1)		
P2:		R(x,1)	R(x,1)

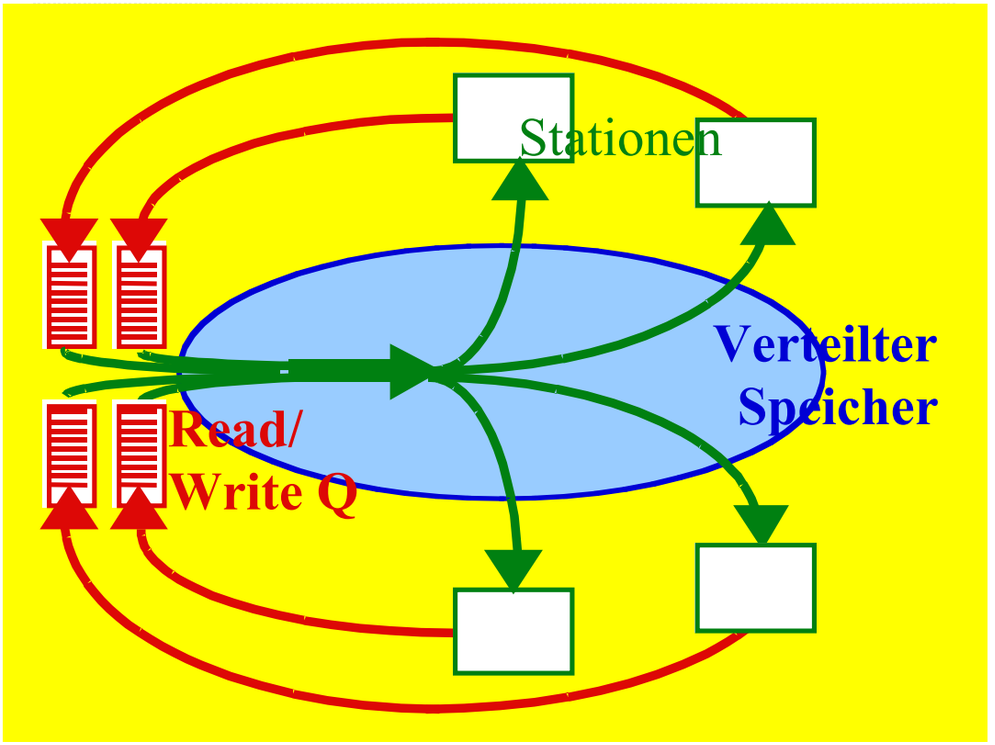
P1:		W(x,1)	
P2:	R(x,0)		R(x,1)

P1:	W(x,1)		
P2:		R(x,0)	R(x,1)

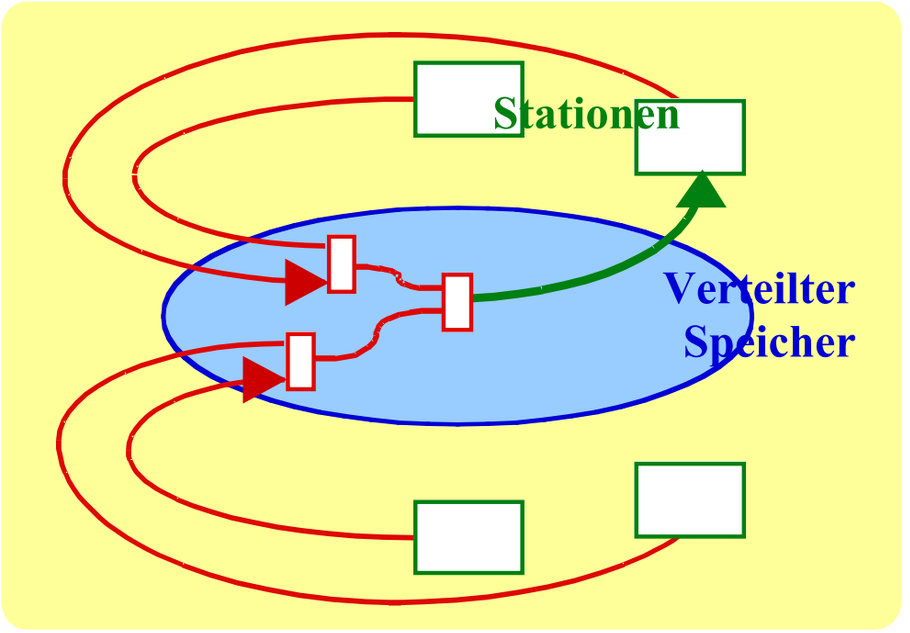


- Sequentielle Konsistenz [Lamport]
  - 'Multiprozessor erscheint wie multitasking Monoprozessor'
  - alle Zugriffe in der gleichen *Reihenfolge* wahrgenommen
  - zeitliche Verschränkung unbestimmt
  - minimale Knotenkommunikationszeit t, Lesen r, Schreiben w:
  - $r + w \geq t$

P1:	W(x,1)		
P2:		R(x,0)	R(x,1)



- Kausale Konsistenz
  - nur kausal voneinander abhängige Operationen global geordnet
  - Semantik des sequentiellen Programmes -> kausale Abhängigkeit
  - evtl. explizite Synchronisierung eines parallelen Programmes
  - Rechnersystem muß Datenflussgraphen realisieren
  - siehe Vektorzeit

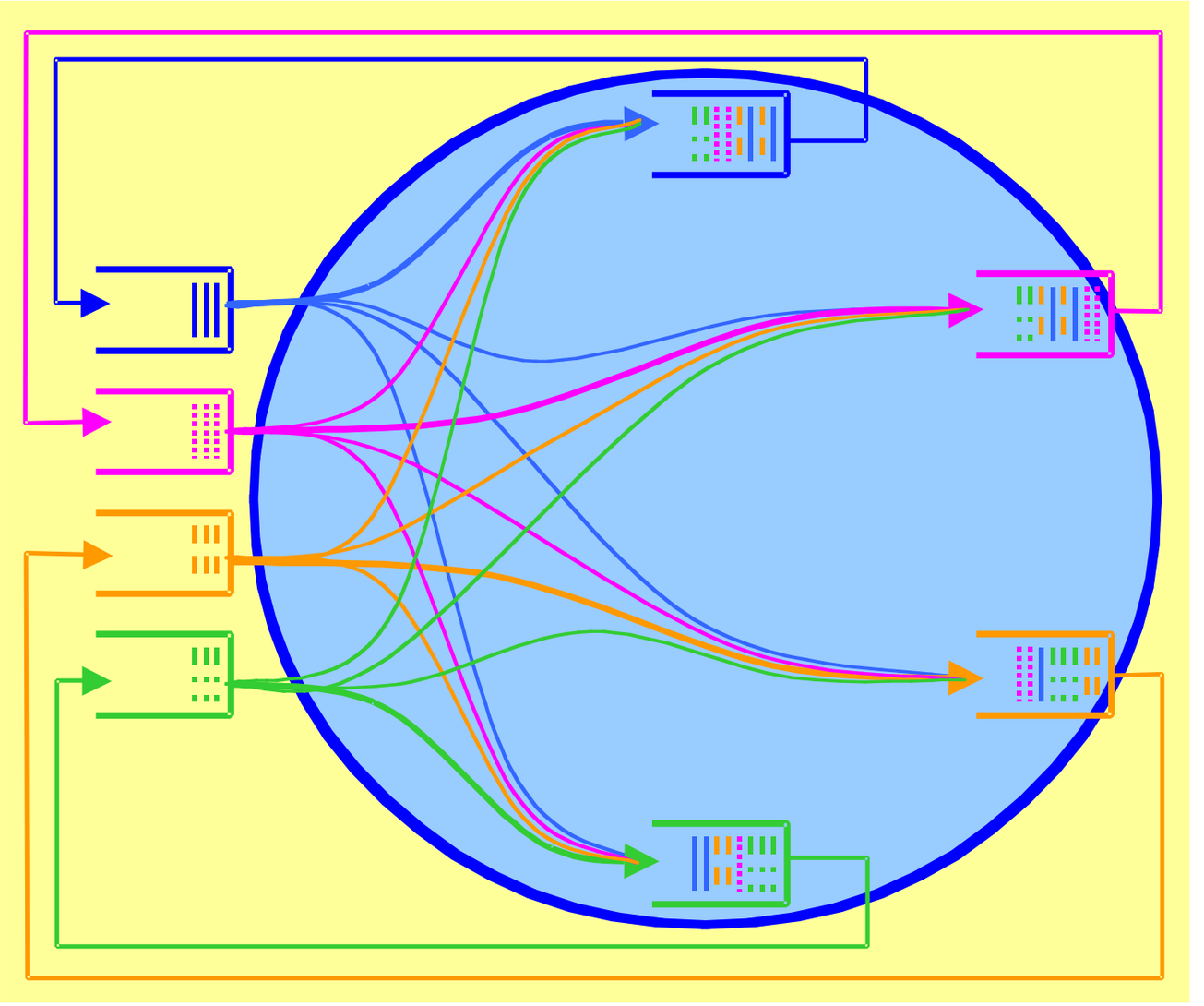


- PRAM Konsistenz: Pipeline RAM (FiFo Konsistenz)
  - Schreibreihenfolge innerhalb eines Prozesses unverändert
  - von allen gleich wahrgenommen
  - Schreiben aus versch. Prozessen unterschiedlich verschränkbar

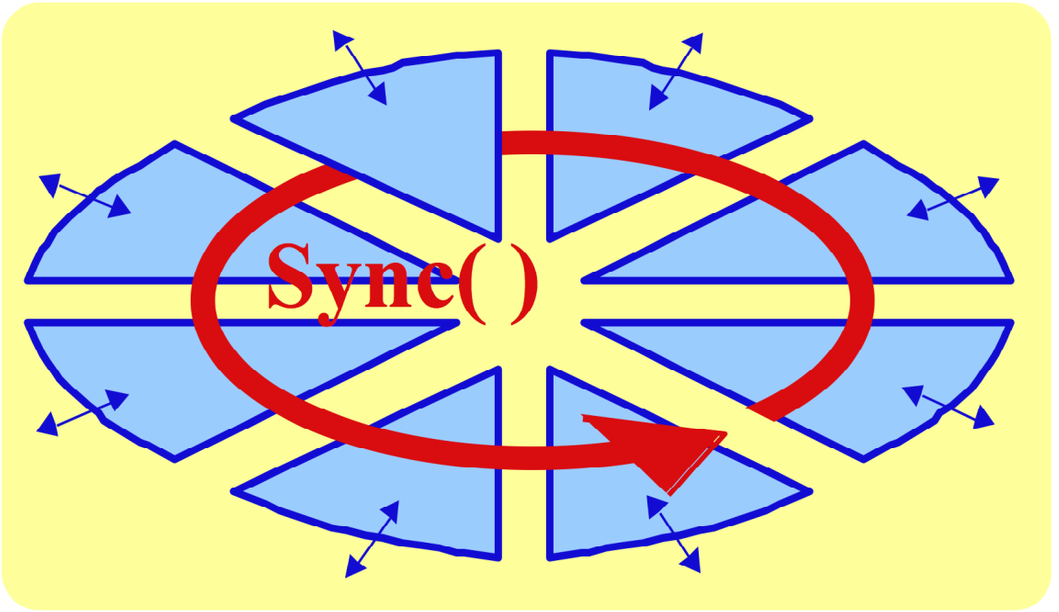
P1:	W(x,1)	W(x,2)		
P2:			R(x,2)	R(x,1)

P1:	W(x,1)	W(x,2)		
P2:			R(x,1)	R(x,2)

P1:	W(x,1)		W(x,2)	
P2:		R(x,1)		R(x,2)

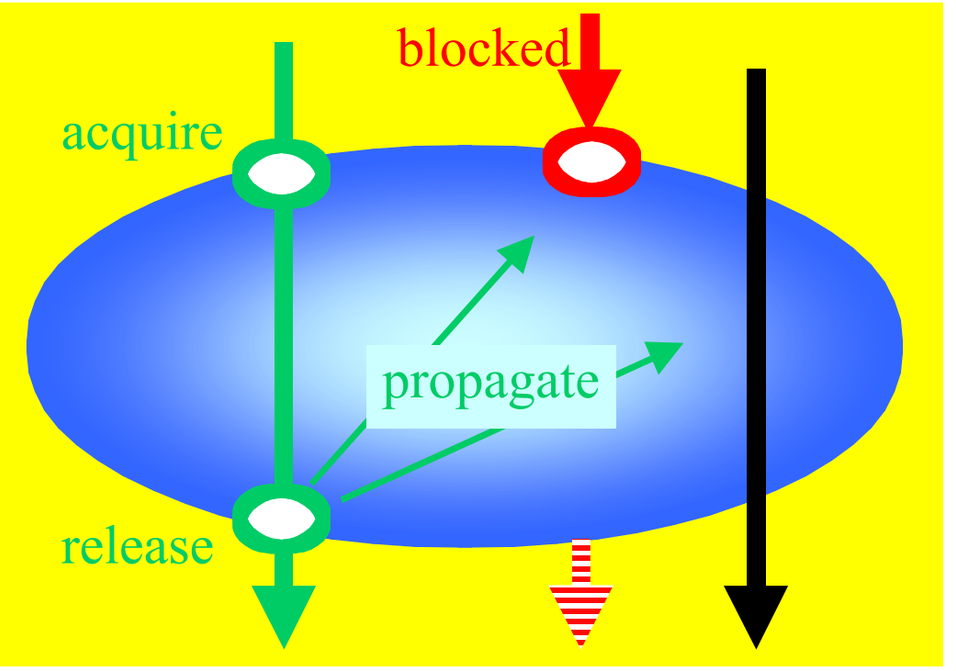


- Schwache Konsistenz
  - beteiligten Prozesse sehen Schreibzugriffe in beliebiger Reihenfolge
  - Synchronisierte Variablen:
    - Zugriff sequentiell konsistent
    - nur Zugriff nach Ausführung aller vorhergehender Schreibops.
    - kein Schreiben vor vorhergehendem Lesen

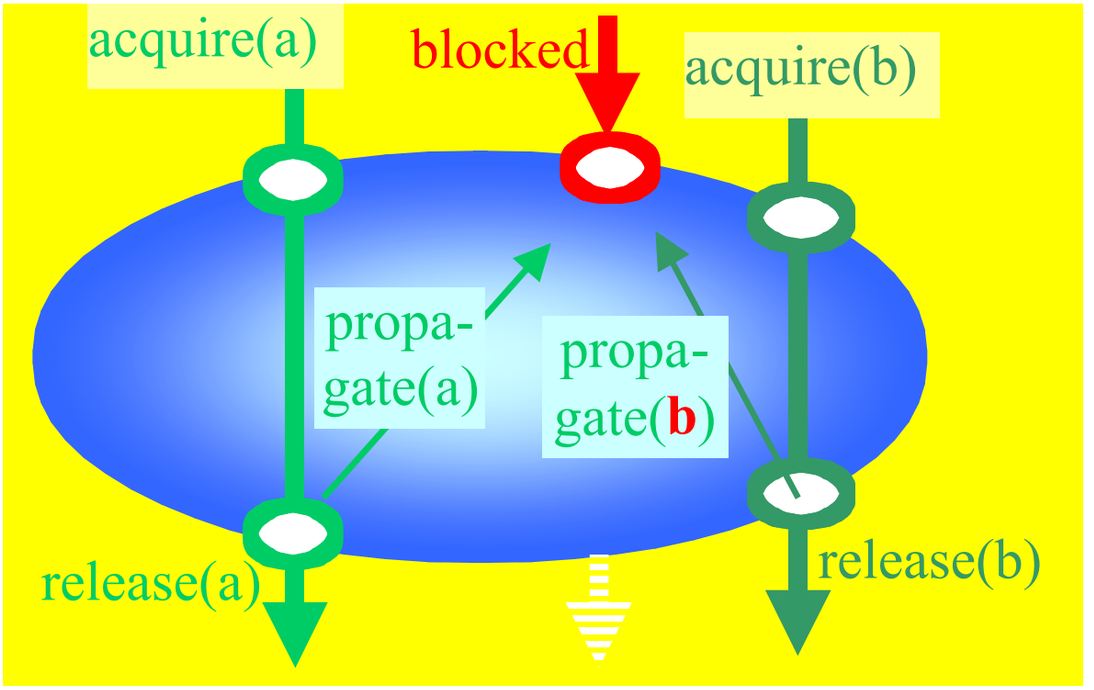


  - Update: schreibender Prozeß verschickt seine Änderunge (propagate)
  - Invalidate: schreibender Prozeß erklärt Daten für ungultig
  - bei Bedarf werden Daten angefordert

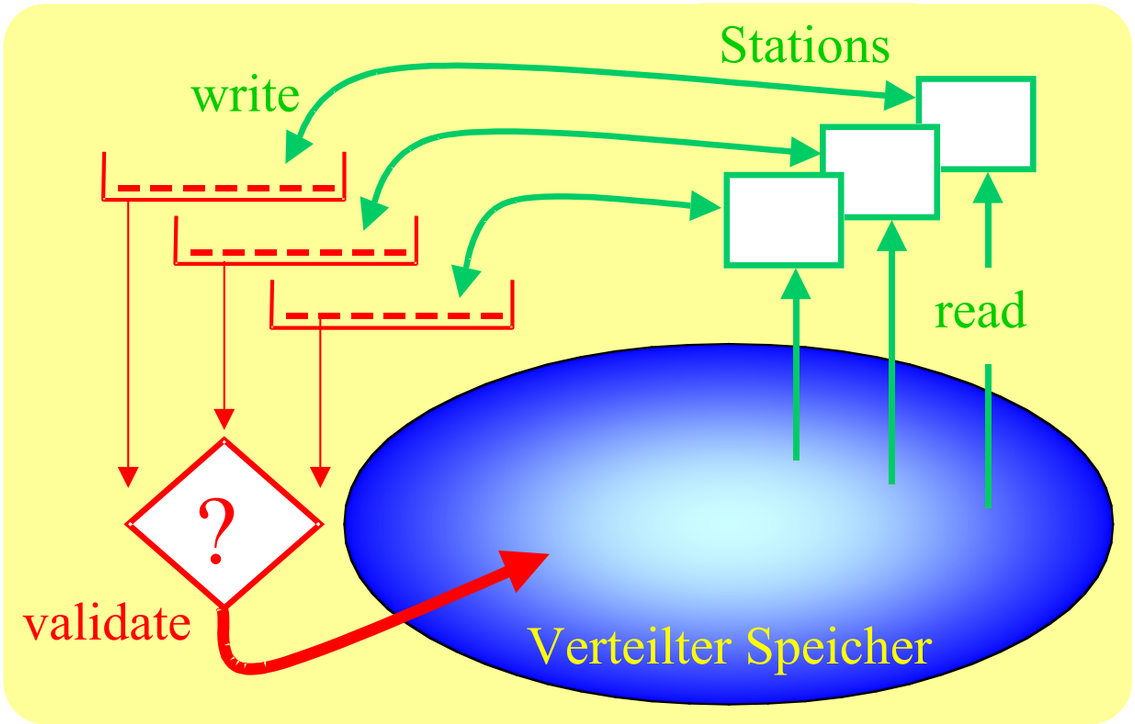
- Release-Konsistenz
  - "acquire" versucht den Eintritt in eine kritische Region
  - "release" verlässt kritische Region
  - "acquire" blockiert, falls kritische 'Region besetzt'
- Teilnehmender Prozess tritt mit "acquire" in kritische Region ein
  - arbeitet mit gemeinsamen Variablen
  - propagieren der Variablen
  - "release" rufen
- Lazy Release: nächster acquire besorgt Variablen



- Entry-Konsistenz
  - Zugriffsrechte auf gemeinsame Variablen
  - separat pro Variable
  - exklusiv zum Schreiben
  - nicht exklusiv nur zum Lesen
  - Vergabe geschützt mit Synchronisierungsvariablen (Semaphore etc.).

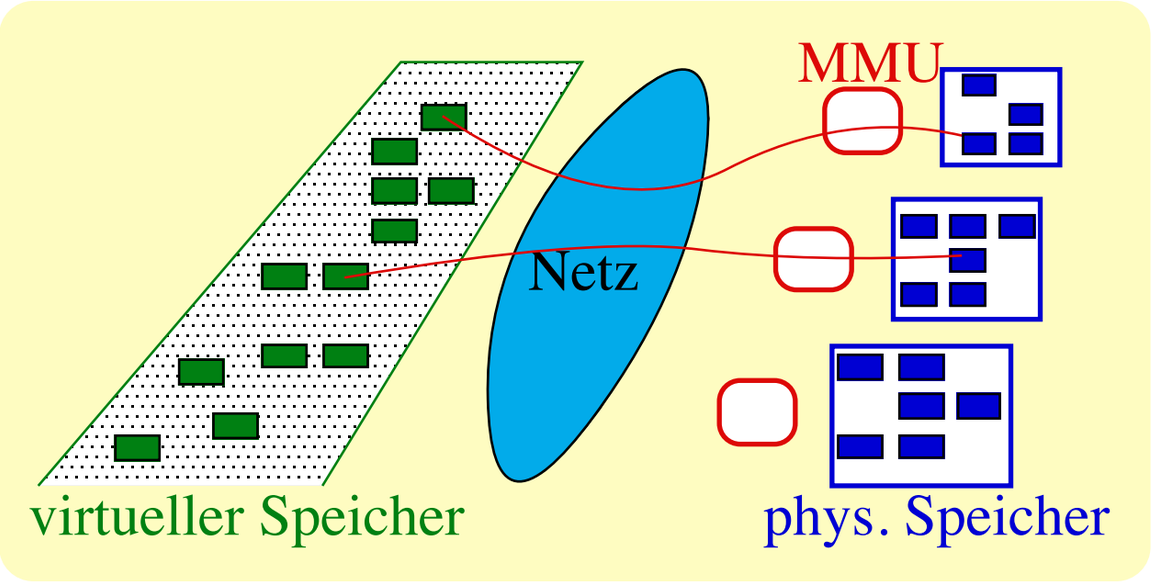


- Transaktionale Konsistenz
  - alle Schreiboperationen vorerst im lokalen Speicher
  - Berechungen in rücksetzbare Transaktionen aufteilen
  - Transaktionsende: aktualisieren bzw. synchronisieren
  - Schreib/Lesekonflikte bei der Aktualisierung: zurücksetzen
  - wahlweise update- oder invalidate-Protokoll zur Synchronisierung
  - „Optimistische Synchronisierung“



1.3.2 MMU-basiertes DSM

- Abbildung des logischen Adressraumes auf Stationen im Cluster



  - hardwaremässig mithilfe der MMU und Paging
  - evtl. Lese-& Schreibbefehle über das Netz transportieren

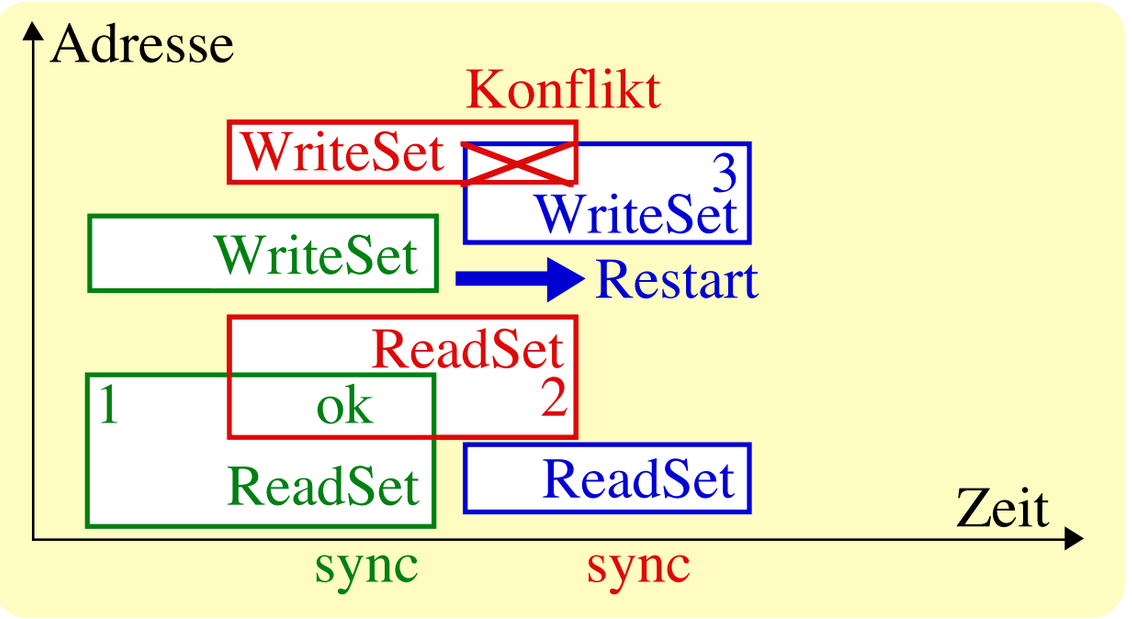
- Vorteil einfaches Programmiermodell
  - "single-system" Perspektive, keine Serialisierung
- Beispiel Plurix/Rainbow

- Problem ist die (relativ) hohe Verzögerung im Netz
- Keine Kopien im Netz => keine Replikation
  - Eigentümer einer Seite kann sofort zugreifen
  - Kommunikationspartner brauchen länger
  - ortsfeste oder migrierende Seiten
- Kopien im Netz vorhanden => Replikation
  - lesen einer Seite ist sofort möglich
  - entweder auf alle Kopien schreiben (update)
  - oder alle Kopien löschen und nur auf Original schreiben (invalidate)
- Matrixdarstellung:

	ohne Replikation	mit Replikation
Ortsfeste Seiten, R/W-Operation transportieren	Verzögerung für alle, außer für Eigentümer	sofort lesen, überall schreiben, **"write update"**
Migrierende Speicherteile, lokale Operation	langsam lesen, langsam schreiben, Seitenflattern	aus Cache lesen, Original schreiben, **"write invalidate"**

- Nach "write invalidate" Seiten von anderen Lesern erneut anfordern

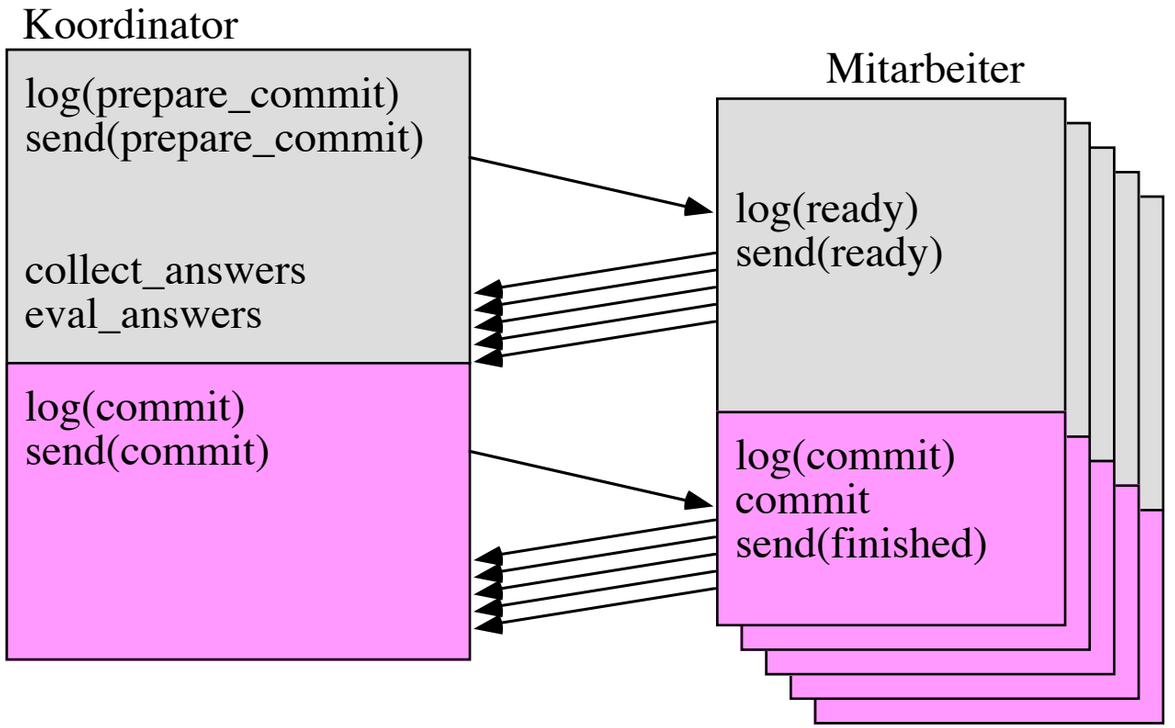
- Transaktionssemantik
- Atomare Operationen
  - Transaktion
  - ACID
- Atomicity
  - alle Teile oder keiner ausgeführt
  - commit oder abort
- Consistency
  - Zustandsüberführung konsistent -> System konsistent
- Isolation
  - Serialisierbarkeit
  - erlaubt Nebenläufigkeit
- Durability
  - Persistenz des Resultates => Speicherung
- Operationen
  - **`tid beginTransaction`**
  - **`result endTransaction(tid)`**
  - **`val get(tid,attribut); set(tid,attribut,val)`**
  - **`abort(tid)`**



- Fehlerverhalten des Servers
  - recovery nach dem Restart des Servers
  - roll-back nach Klienten-Absturz
  - time-outs bis zum Abschluß einer Transaktionen
- Klienten-Verhalten
  - Operation läuft in time-out
  - Operation kommt mit Fehler zurück nach Server-restart
  - Rückfrage beim Benutzer?
- Private Arbeitskopie
  - Operationen auf shadow copy
  - commit: atomares Zurückschreiben
  - Vergleich Orginal mit Anfangs-Shadow

- Intention-list
  - Server zeichnet Operationen einer Transaktion auf
- Optimistisch
  - Undo-Liste
  - Lesen einfach
  - Commit: Undo-Liste löschen
  - Abort: Rückabwickeln (rollback)
  - Lesen für andere Prozesse schwer
- Pessimistisch:
  - Do-Liste sammeln
  - Lesen komplex: Orginal und Do-Liste mergen
  - Commit: atomar Ausführen
  - Abort: Do-Liste löschen
  - Lesen für andere Prozesse einfach

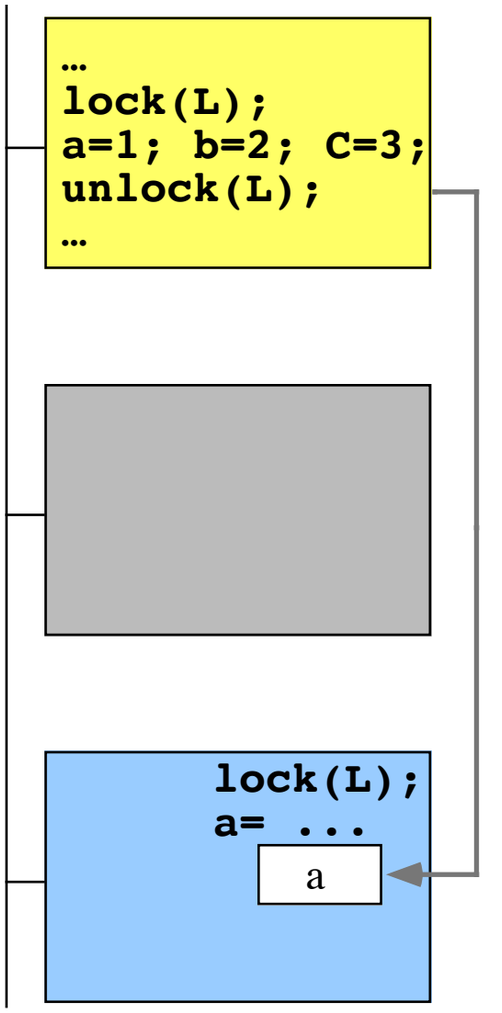
- 2-Phasen Commit Protocol
  - Koordinator sendet 'Prepare' vor dem Commit
  - Mitarbeiter antworten Ready, wenn Commit OK
  - Stimmen zählen
  - Commit durchführen



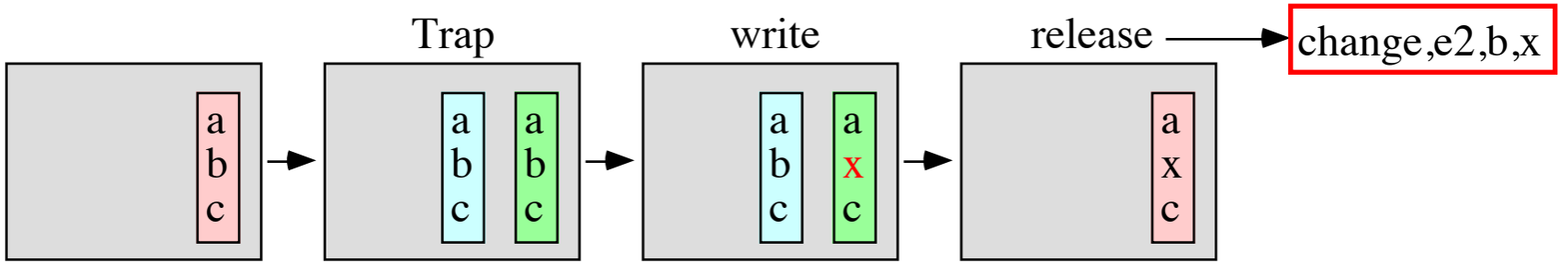
1.3.3 Shared Variables

- Verteilung kleinerer Objekte
  - weniger 'false sharing'
  - Variablen und Datenstrukturen
  - 'verteilte Datenbank'
  - Replikation
- Munin [Bennett, Carter, 1990-1993]
  - verteilte Objekte auf Seiten
  - MMU kontolliert Zugriff
  - mehrere Konsistenzmodelle: Parametrisierung durch Programmierer
  - update, invalidate, Zeitpunkt
  - Anzahl Replikate, Eigentümer, Nutzung, …
- Spezialcompiler
  - Schlüsselwort 'shared' bei der Variablendeklaration
  - evtl. Benutzungsinformation
  - vorgefertigte Modelle: read-only, migratory, write-shared, conventional
  - default: eine Seite pro Variable
  - Programmierer kann Variablen zusammenfassen

- Shared Variablen
  - read MMU-kontrolliert
  - write: nur in kritischen Regionen
  - synchronization: Zugangsprozeduren
- Kritische Regionen
  - exklusiver Zugang mit lock()
  - unlock() stößt Update der Replikate an
- Read-only Variable
  - Page-fault
  - Munin sucht im Verzeichnis
  - fordert vom Besitzer die Seite an
  - MMU verhindert Schreibzugriff
- Conventional
  - eine Schreibkopie, viele Lesereplikate, invalidate
- Migratory Variable
  - Benutzung in kritischen Regionen
  - nur eine Kopie, keine Replikation
  - page-fault - Seite holen - alte Kopie löschen - benutzen



- Write-shared Variable
  - Bsp. Arrays: Elemente von verschiedenen Prozessen zugreifbar
  - Trap beim Write: twin anlegen
  - Release: vgl. write-copy und twin
  - Differenz verteilen
  - Empfänger vergleichen write-copy, twin, Differenzliste
  - run-time Fehler bei Konflikten



- Barrieren sorgen für Synchronisation

P1

```
wait_at_barrier(b);
for (i=0;i<n;i+=2)
    a[i]+=x;
wait_at_barrier(b);
```

P2

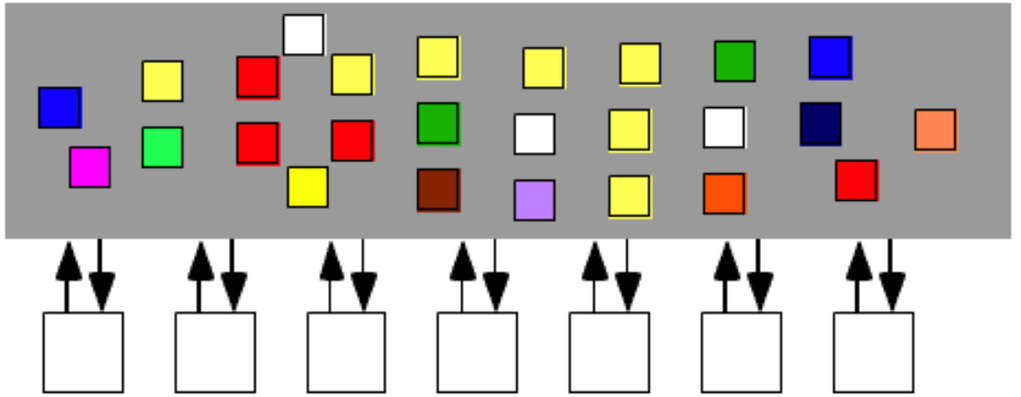
```
wait_at_barrier(b);
for (i=1;i<n;i+=2)
    a[i]+=y;
wait_at_barrier(b);
```

1.3.3 Objektbasierter Verteilter Speicher

- Objekte verstecken Zugang zu Feldern
  - Interna bleiben Implementierungssache
  - Linda, Orca, Amber, Emerald, Cool
- Linda [Gelernter, 1985]
  - kleine Bibliothek
  - C, FORTRAN, Modula, …
  - JavaSpaces
  - Präprozessor
- Tupel
  - Gruppe von Elementen (Felder)
  - Basistypen
  - C: integer, long integer, float, …, arrays, structures
  - kein Tupel im Tupel

  ("abc",2,5)

  ("konrad","froitzheim",45054)
- Tupel-Space
  - global im verteilten System
  - insert und remove



- out("konrad","froitzheim",m);
  - schreibt Tupel in den Tupel-Space
  - Konstante, Variablen, expressions
  - Tupel werden nicht überschrieben => mehrere Kopien
- in("konrad","froitzheim",?i);
  - inhaltsadressiert
  - i erhält Wert des dritten Tupelelementes
  - Tupel wird entfernt
  - matching und entfernen atomar
  - blockiert falls kein passendes Tupel da
- Beispiel: Semaphor
  - out("mein kleiner Sema"); gibt frei
  - in("mein kleiner Sema"); => lock
- read()
  - findet match
  - entfernt Tupel aber nicht

- Beispielproblem: Ray-tracing
  - Dispatcher

```
out("tasks",L [1],1);
out("tasks",L [2],2);
…;
out("tasks",L [n],n);
```

  - Bearbeiter

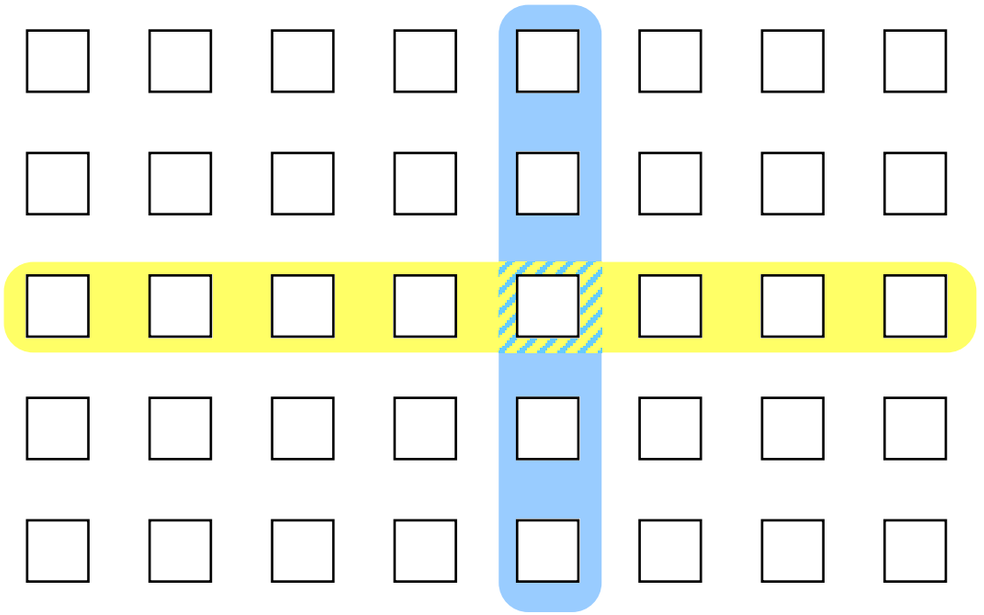
```
in("tasks",?Line,?nr);
Rechnen(Line);
out("fertig",Line,nr);
```

  - Anzeige

```
in("fertig",?disp,?nr);
VRAM[nr]=disp;
```

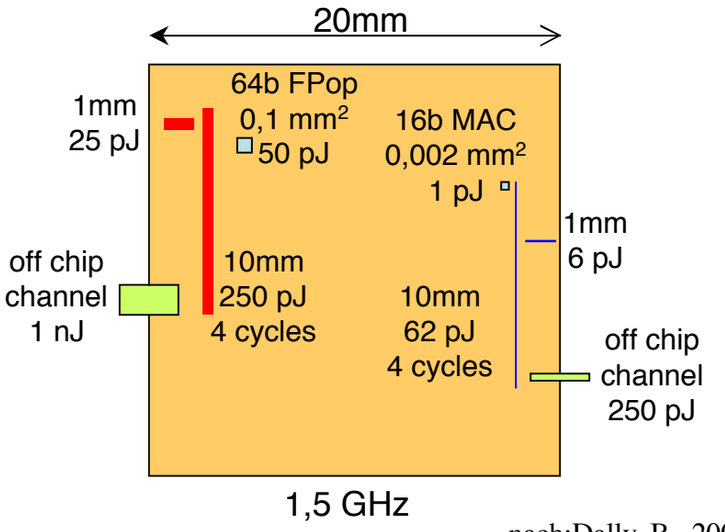
- Implementierung
  - assoziativer Speicher teuer
  - volle Suche …
  - Verteilung?
- Idee: Hierarchie
  - erstes Tupelelement teilt in Subspaces
  - Konvention: erstes Element String
  - Konstante im ersten Feld => subspace zur Übersetzungszeit bestimmen
  - subspaces => Verteilung
  - Hashing auf den anderen Feldern
- Multiprozessor
  - tuple subspace => hash-table
  - im globalen verteilten Speicher
  - lock subspace - enter/remove - unlock subspace

- Multicomputer mit Hochleistungsnetz
  - out() => broadcast, in() => lokale Suche
  - entfernen mit delete protokoll: 2-phase-commit
  - Probleme bei großen Systemen?
- Linda im LAN [Carriero]
  - out() lokal, in() mit Broadcast
  - keine Antwort: erneuter Broadcast
  - Treffer wird entfernt ohne besondere Probleme
  - in() blockiert evtl.
- Kombination [Krishnaswamy, 1993]
  - Prozessoren im Gitter
  - out() broadcast in der Zeile
  - in() broadcast in der Spalte
  - Kreuzung eindeutig

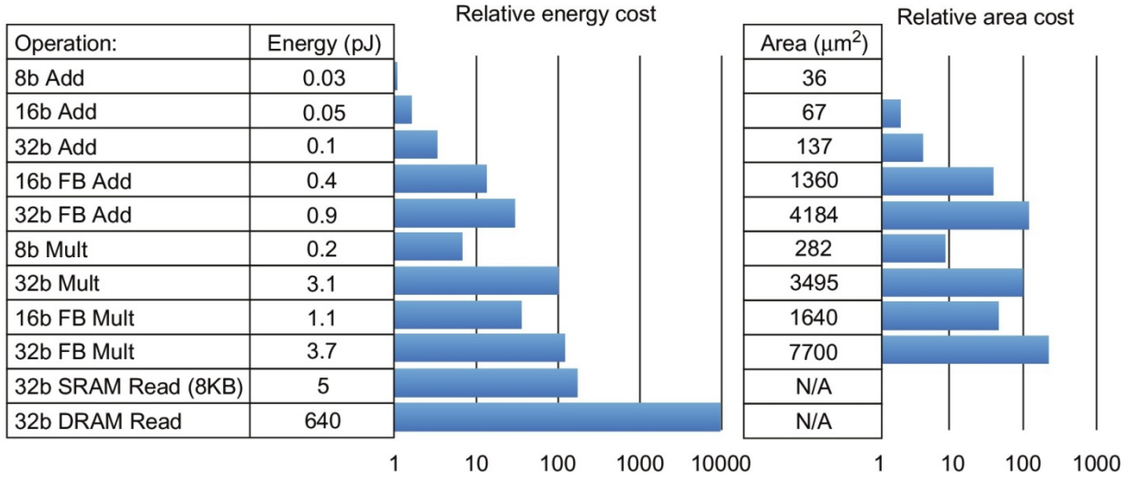


1.4 Throughput Oriented Architectures

- the power wall again
  - 20*20 mm
  - 4.000 64b FPUs
  - 200.000 16b MACs
- Latenz optimieren?
  - typische CPU
  - Operationen schedulen
  - ILP ausbeuten
  - Nutzen/Aufwand sinkt
- Durchsatz optimieren
  - hoher Datenstrom
  - Arithmetik 'billig'
  - Datenlokalität
- Anspruchsvolle Probleme
  - Bildverarbeitung, Mustererkennung, Simulation
  - hohe arithmetische Dichte
  - viele Daten, viel Parallelität
  - Lokalität hoch



nach:Dally, B., 2009

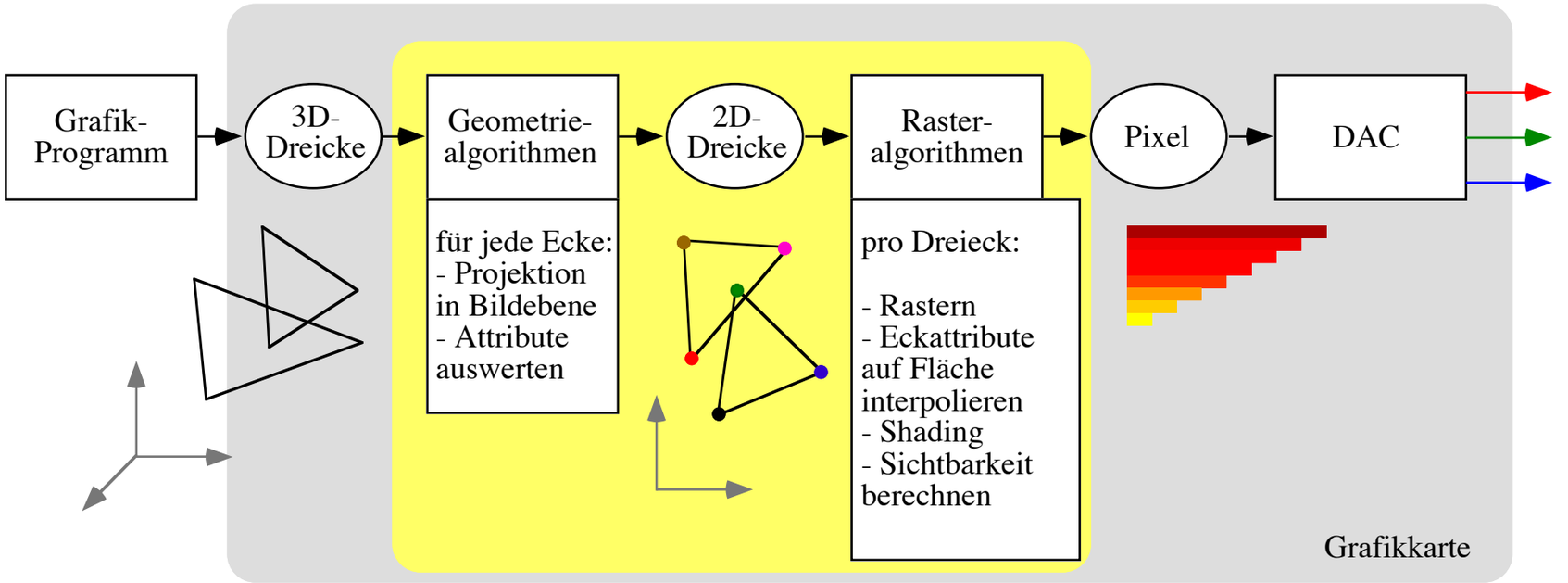


Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FB Add	0.4	1360
32b FB Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FB Mult	1.1	1640
32b FB Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

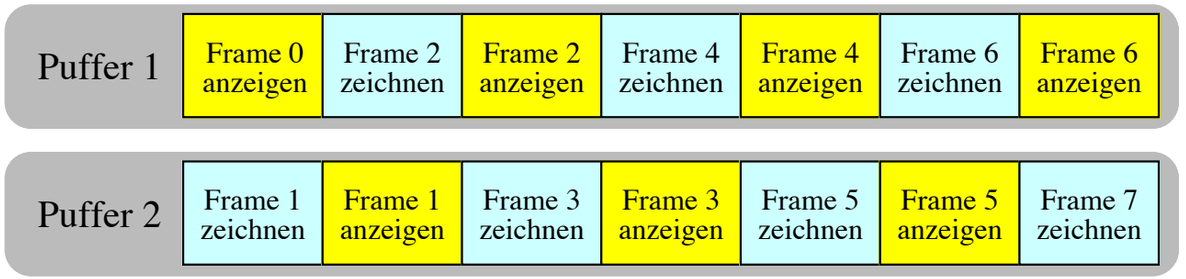
Energy numbers are from Mark Horowitz *Computing's Energy problem (and what we can do about it)*. ISSCC 2014
Area numbers are from synthesized result using Design compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Patterson, Hennessy: Computer Architecture

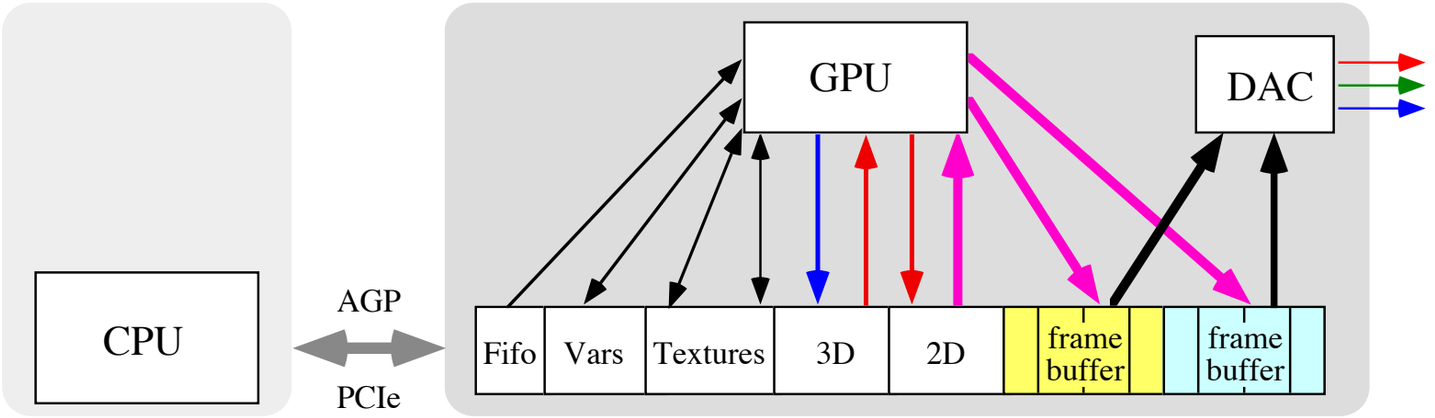
- Beispielaufgabe 3D-Grafik
  - Transformationen: drehen, verkleinern, …
  - 3D-Grafik: Dreicke im Raum beschreiben Oberflächen
  - 3D-Koordinaten auf 2D-Ebene (Bildschirm) abbilden
  - Eckepunkte einfärben (shader)
  - Flächenpunkte einfärben (shader)
  - Füllen mit Texturen
  - Raster-Operationen



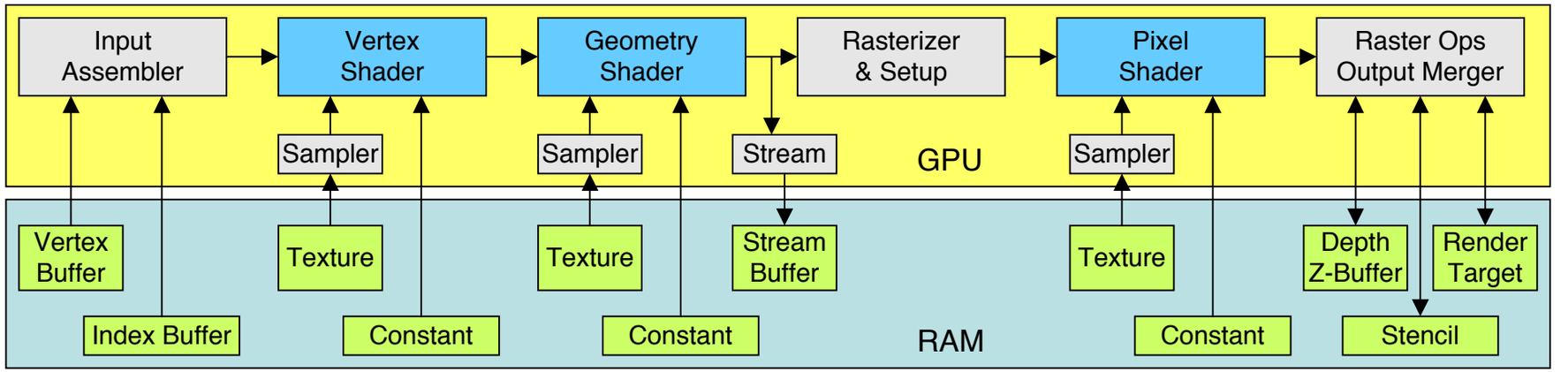
- Grafik-Verarbeitungssequenz
  - gut pipelineisierbar
  - gut parallelisierbar: "für jede Ecke", "pro Dreieck"
  - aber: flexible Speicherpartitionierung
- Moderne Grafikkarten
  - viel Speicher mit Controller zur Bankeinteilung
  - Kommandos, Texturen, Pixel, Datenstrukturen, …
  - schnelle Vermittlung zwischen GPU und Speicher



- Datenstöme in der Grafikkarte



- Rendering Pipeline



- Vertex Shader
  - Punkte aus 3D nach 2D-Fläche projezieren
  - Szene -> Bildschirm
- Geometry Shader
  - Primitive transformieren und erzeugen: Linien …
  - Polygon -> Dreieck: Tesselation
  - Shadow Volumes
- Rasterizer
  - Pixel Fragmente generieren
  - Teile von geometrischen Primitiven

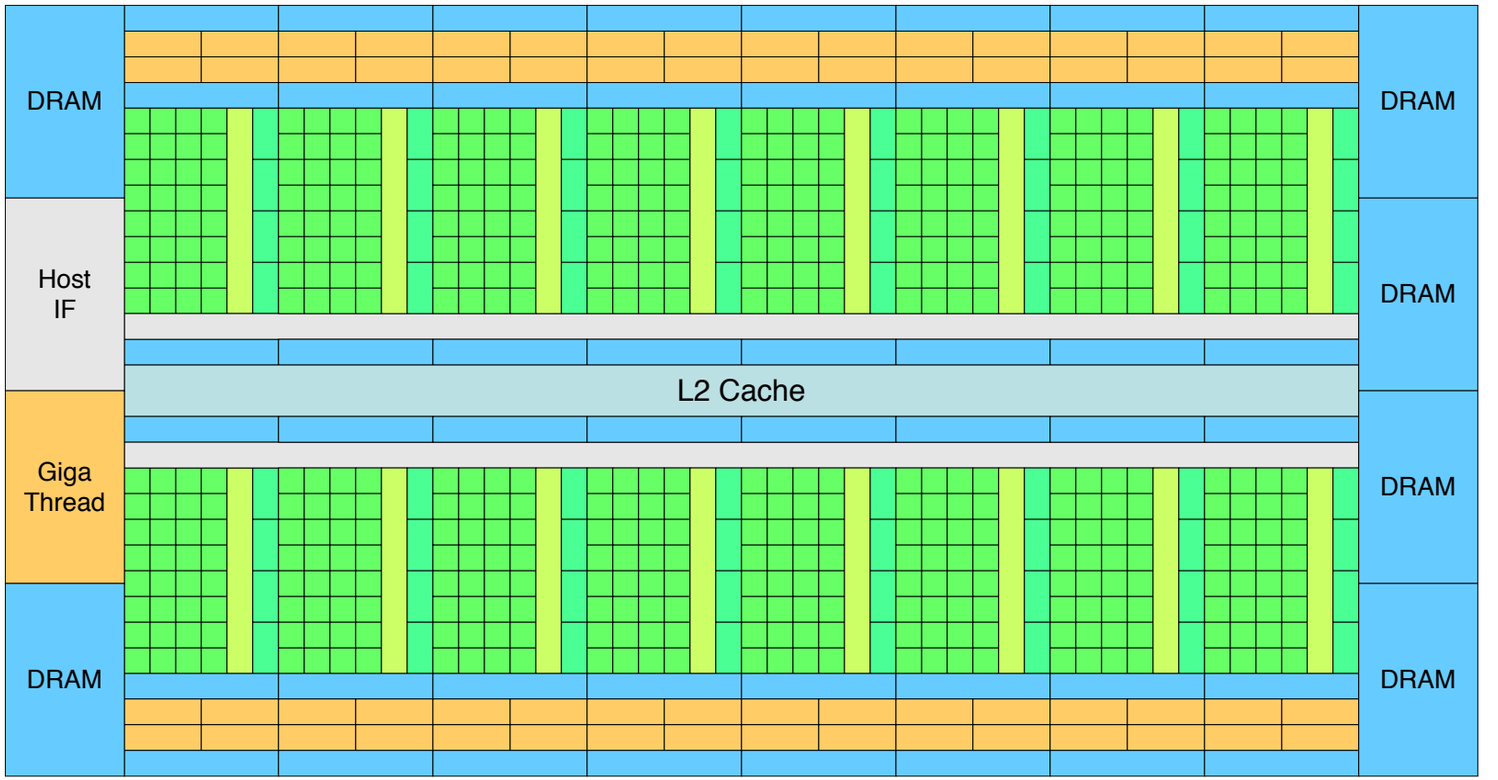
- Pixel Shader berechnet Farbe
  - pro Pixel
  - Renderer impliziert 'Formel'
  - Beispiel Cg, 1400 Instuktionen

```
void reflect(float2 texCoord : TEXCOORD0,
             float3 refl_dir : TEXCOORD1,
             out float4 color : COLOR,
             uniform float shiny,
             uniform sampler2D surfaceMap,
             uniform samplerCUBE envMap    )
float4 surfaceColor = tex2D(surfaceMap, texCoord);
float4 reflectedColor = texCube(envMap, refl_dir);
color = lerp(surfaceColor, reflectedColor, shiny);
```

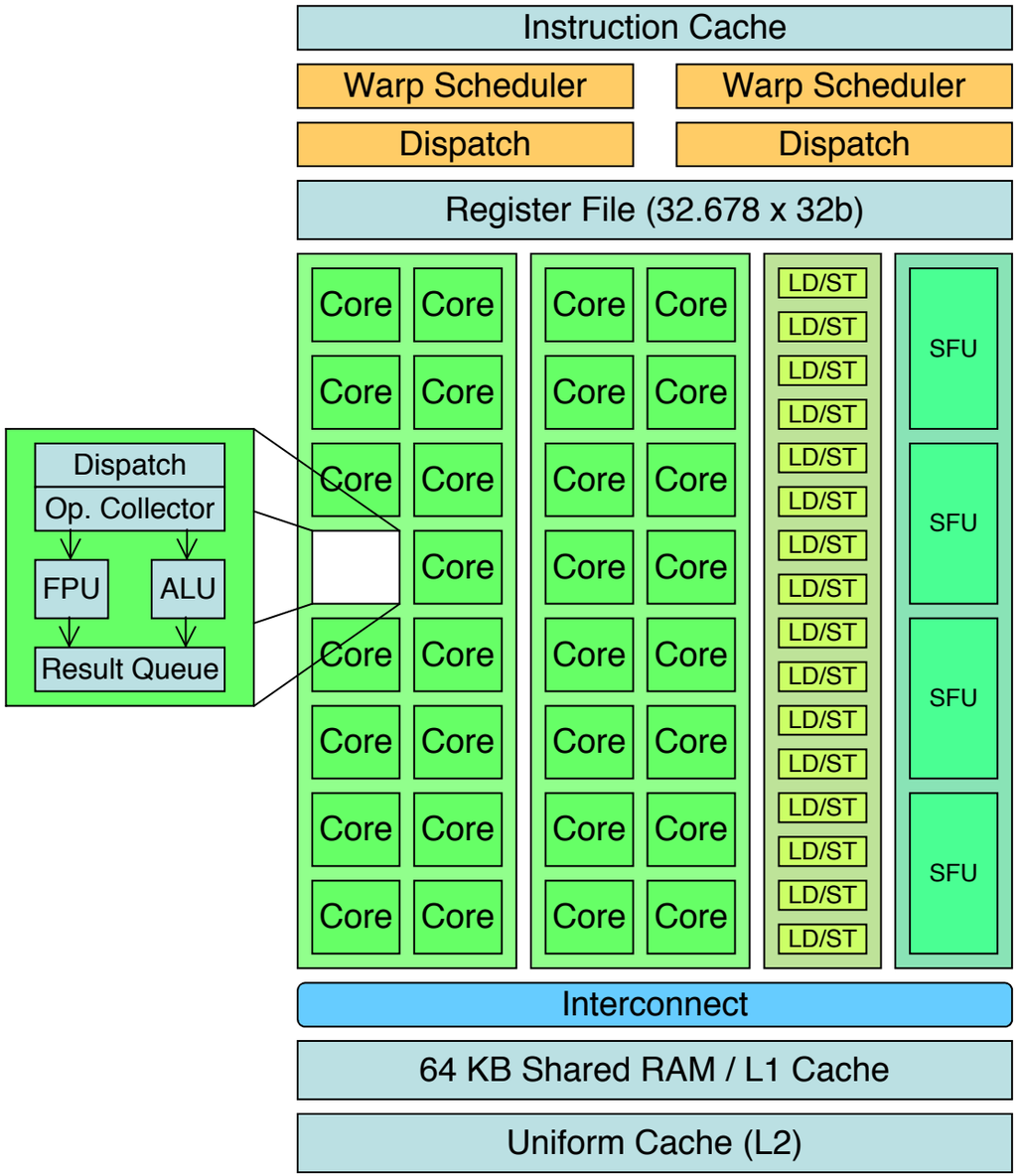
- Shader-Programme
  - geometrische Transformationen
  - projektive Abbildung
  - besondere Programmiersprachen: HLSL, GLSL, Cg, …

$$\begin{bmatrix} v_1 & v_2 & v_3 & 1 \end{bmatrix} \begin{bmatrix} xf_{11} & xf_{12} & xf_{13} & 0 \\ xf_{21} & xf_{22} & xf_{23} & 0 \\ xf_{31} & xf_{32} & xf_{33} & 0 \\ xl_1 & xl_2 & xl_3 & 1 \end{bmatrix}$$

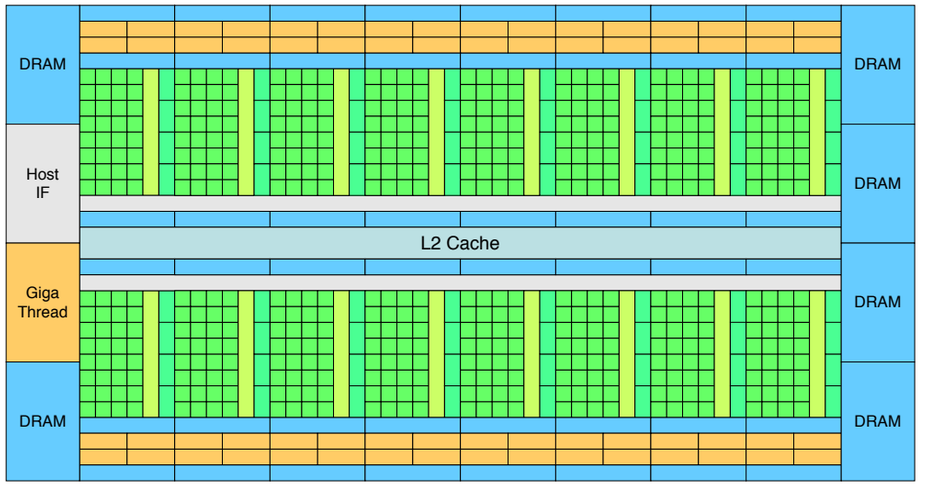
- GPU besteht aus vielen einfachen Verarbeitungseinheiten
  - vertex shader, pixel pipelines, raster operations engines
  - alle mit kleinen Programmstücken
  - hochgradig parallel
- NVIDIA Fermi GPGPU
  - 512 Cores
  - 16*32
  - 384 Bit Speicherbus
  - RAM 6*64 Bit Partitionen
  - L2 Cache
  - Interconnect
- GigaThread
  - verteilt Thread-Blöcke

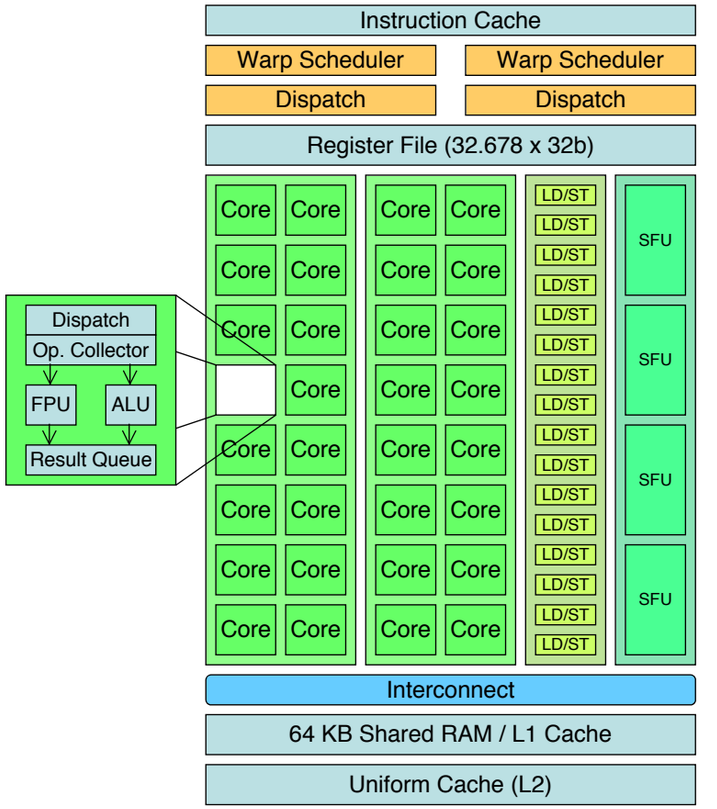


- Streaming Multiprocessor
  - 32 CUDA Prozessoren (SP)
  - Floating Point ALU
  - double precision
  - IEEE 754-2008
  - Fused Multiply Add
  - Integer ALU 32bit
- Load/Store Unit
  - für 16 Threads
- Special Function Unit
  - Sinus, Cosinus, …
  - sqrt, 1/x
- Lokaler Speicher
  - shared für SPs
  - 48 KB shared / 16 KB Cache
  - 16 KB shared / 48 KB Cache
  - Thread-Kommunikation
- Warp-Scheduler
  - 2 Warps gleichzeitig, 2 Zyklen/Warp
  - Warp an SPs, SFUs oder LD/ST

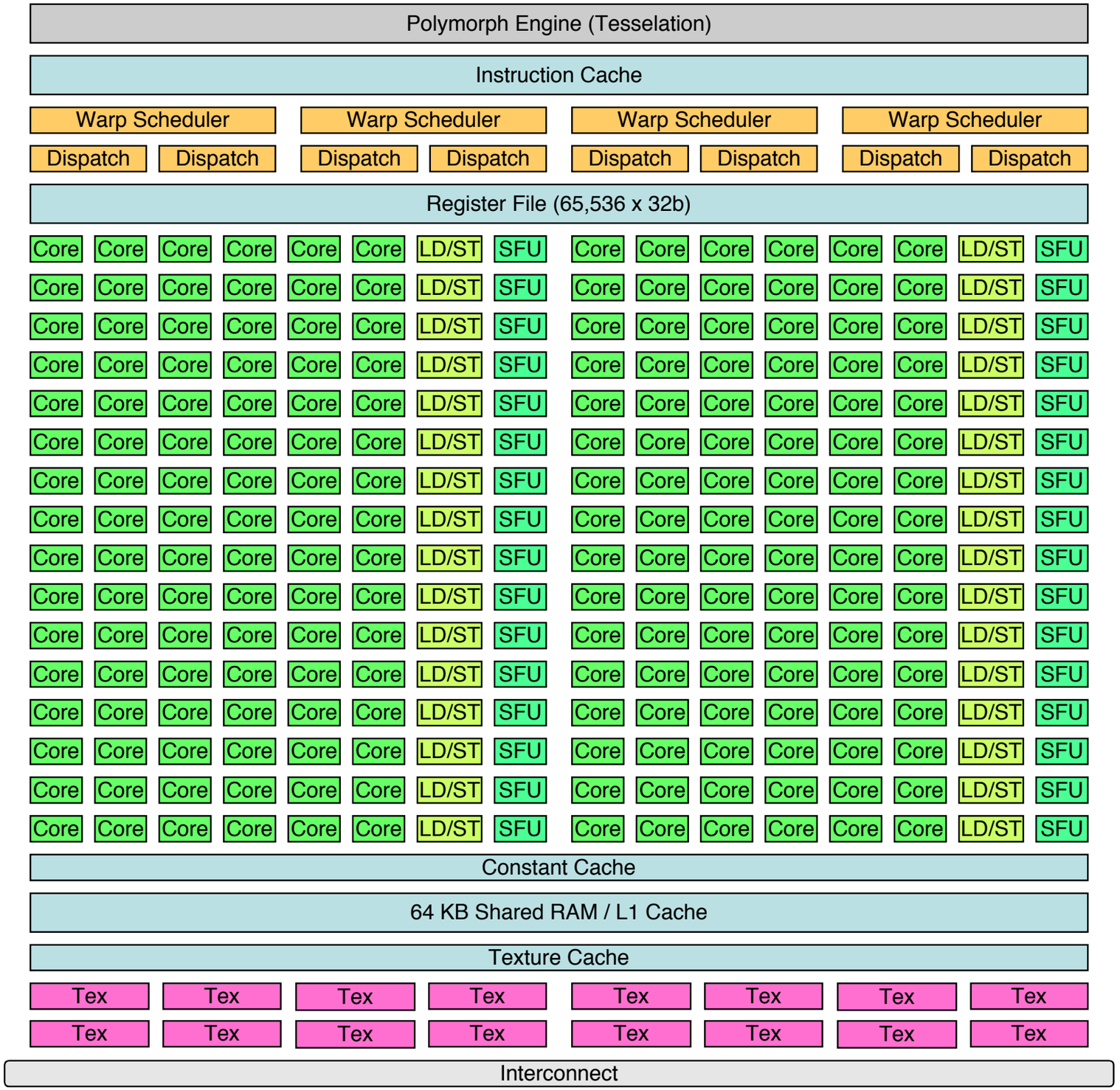


- 512 Prozessoren (Fermi)
  - wie bekommt man genug Threads?
  - SP in-order
  - Hardware thread-scheduler
  - ohne Datenflussanalyse
- Feinkörnig datenparallel
  - serieller Code pro Ecke, pro Pixel
  - serieller Code für jedes Vektor-Element
  - "Millionen" threads pro frame
- Skalierbarkeit
  - Anzahl SM unterschiedlich
  - billige - teure Grafikkarte
  - Laptop-Grafik
- Lokalisierbarkeit der Daten
  - lokaler Speicher
  - Kommunikation "benachbarter" threads
- Thread-Management
  - verteilte Scheduler
  - örtlich und zeitlich schedulen
  - Speicher-Latenz berücksichtigen

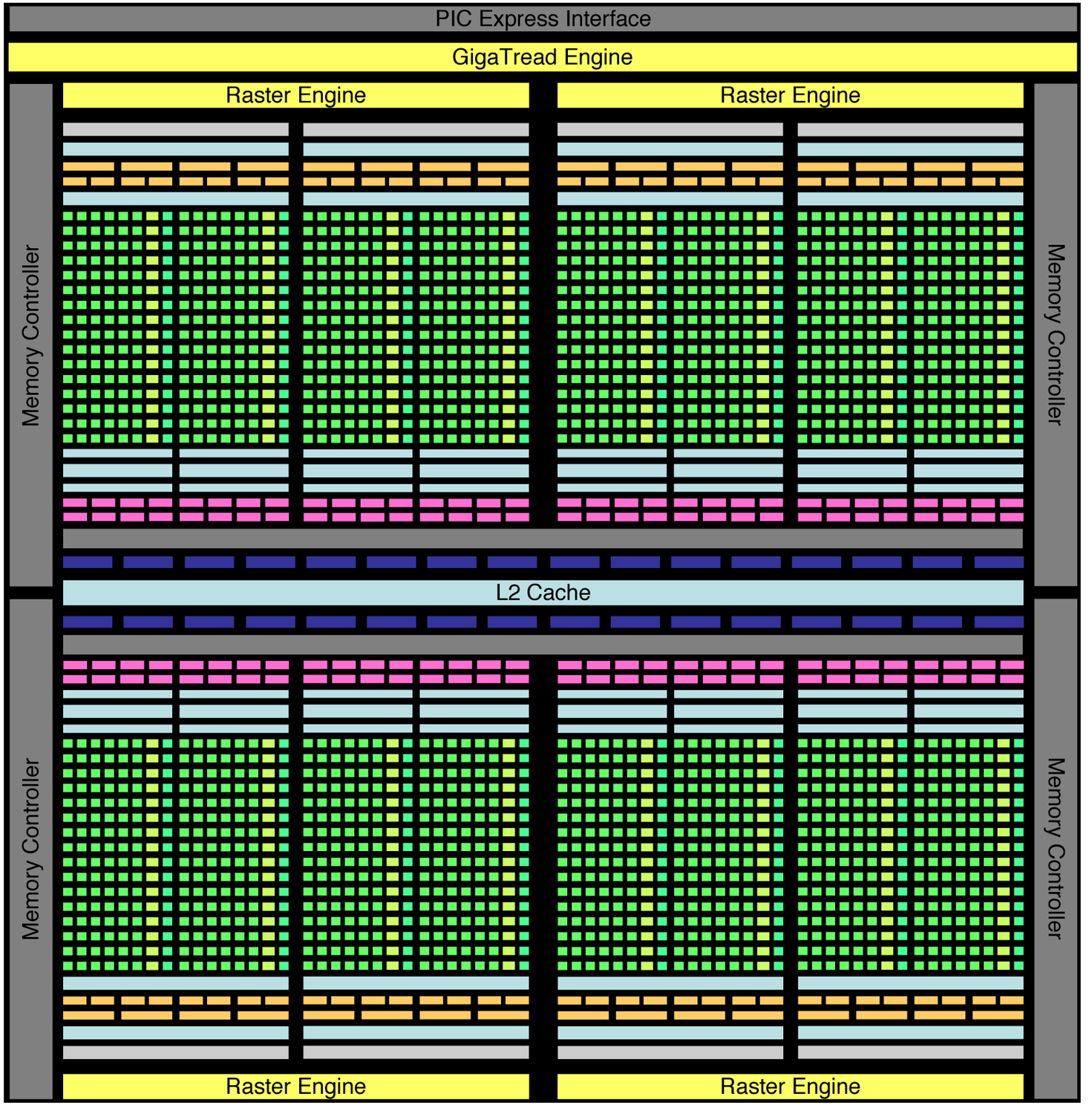




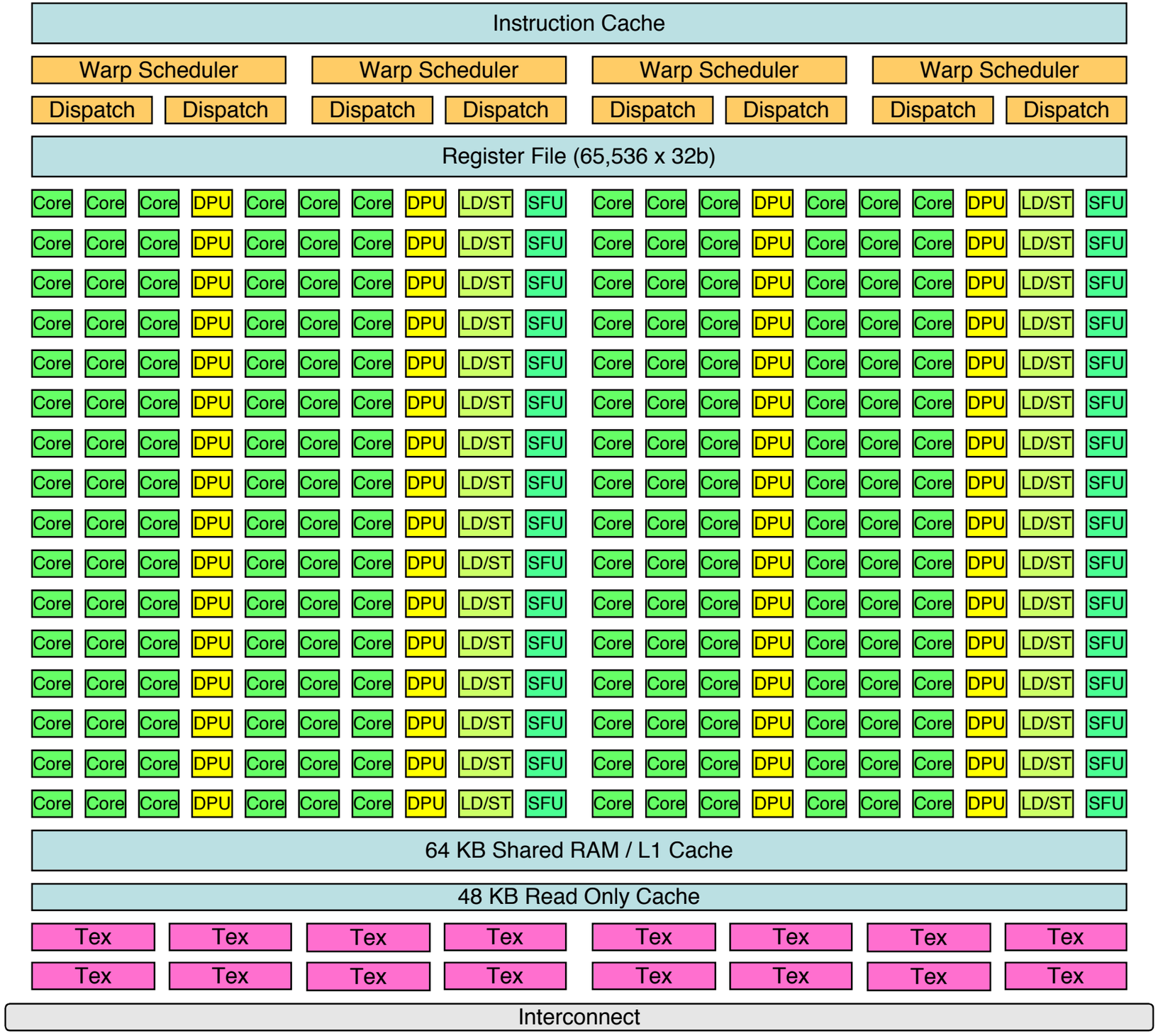
- Update: Kepler
  - 32 bit cores
  - Special Function Units
  - Load/Store Units
  - Double Prec. Units
  - Texturizer
  - Polymorph Engines
- Scheduler
  - Hints vom Compiler
  - Latenz Rechen-Ops



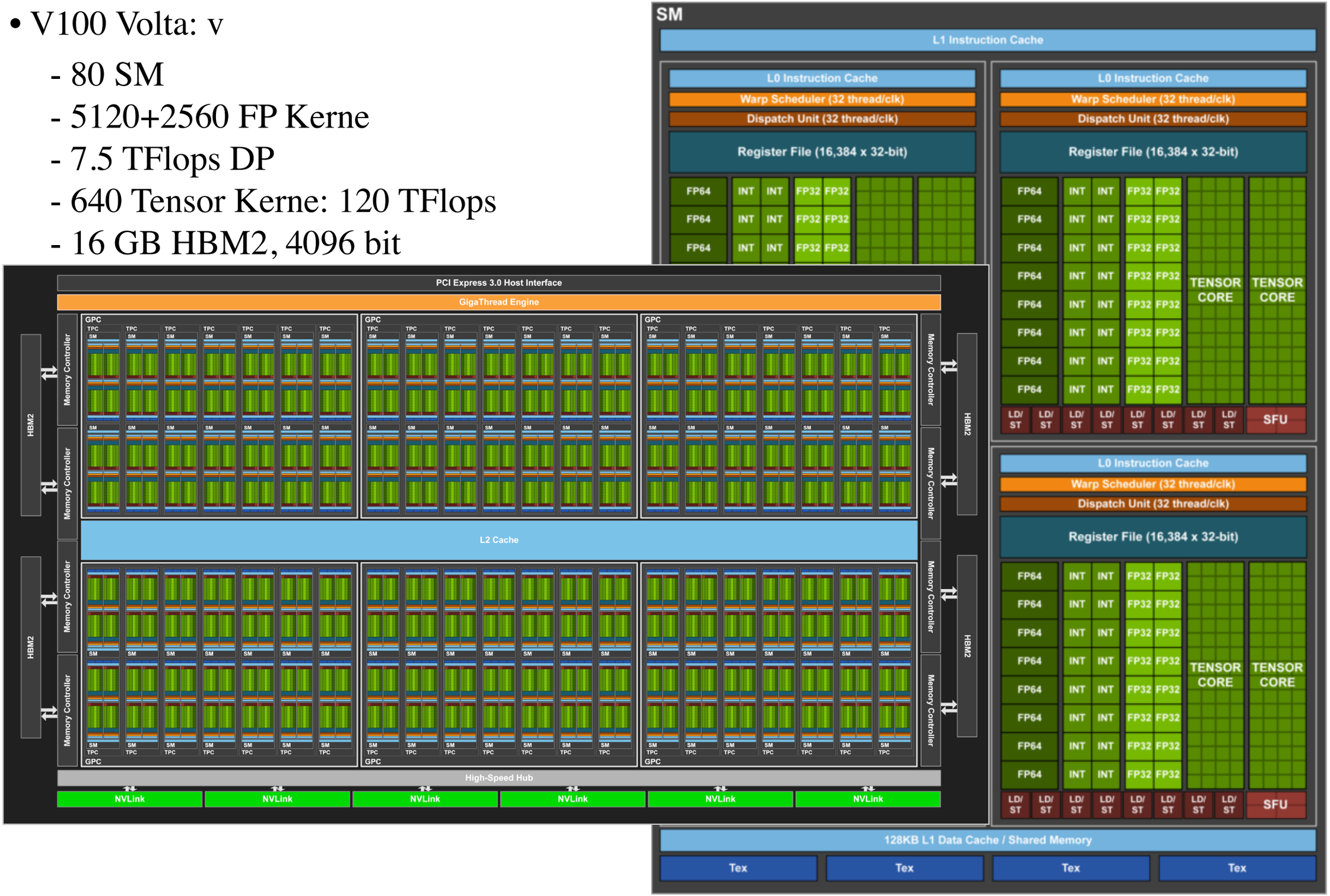
- GK 104
  - 1536 Kerne
  - 3 Tflops (SP)
  - 128 Gflops (DP)
  - Speicher@192 GB/sec
  - 195W
  - für Grafikkarten
  - 3.5 E9 Transistoren
  - 8 SMX in 4 GPU Cores
  - Caches, Register, RAM
  - ca. 1 Ghz



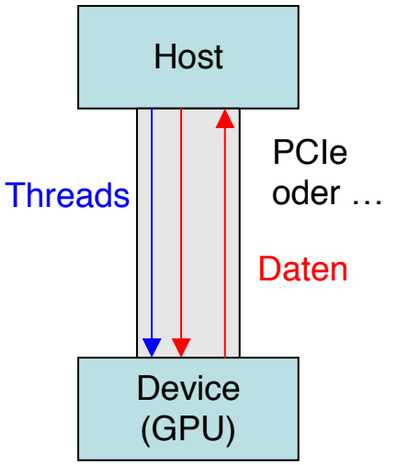
- GK110
  - Compute
  - 2880 Kerne
  - 3.5 Tflops (SP)
  - z.B. 15 SMX
  - 7.1 E9 Transistoren
  - 6 SpeicherIF
  - Double Prec. Units
  - 960 DPUs
  - 2 (?) Tflops (DP)
  - Warps erzeugen
  - RDMA

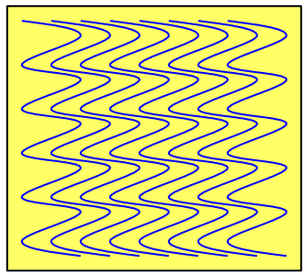


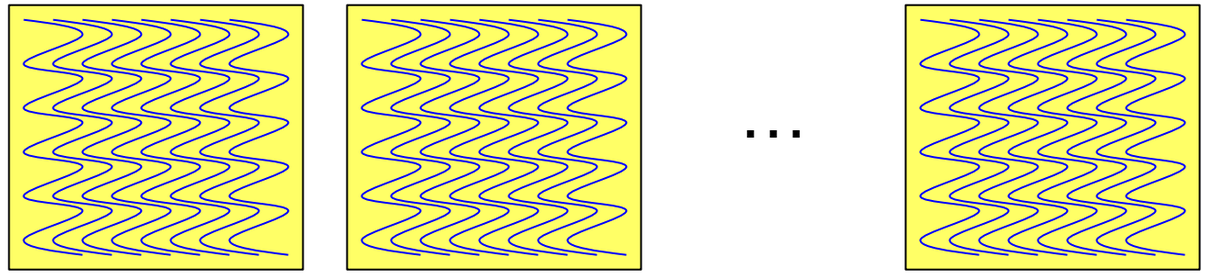
- V100 Volta: v
  - 80 SM
  - 5120+2560 FP Kerne
  - 7.5 TFlops DP
  - 640 Tensor Kerne: 120 TFlops
  - 16 GB HBM2, 4096 bit



- CUDA: Compute Unified Device Architecture
  - Erweiterung von C/C++ mit C-Runtime
  - Java: JaCuda
  - Wrapper für Python und Fortran
  - …
- Kernel
  - serielles Programm
  - datenparallel
  - einfach oder komplex
  - wird zu Thread
- Threadblöcke
  - nebenläufige Threads
  - Barrieren zur Synchronisation
  - privater Speicherblock
- Grid
  - parallel Threadblöcke
  - unabhängig
  - globaler Speicher
  - Blöcke auf SMs verteilt







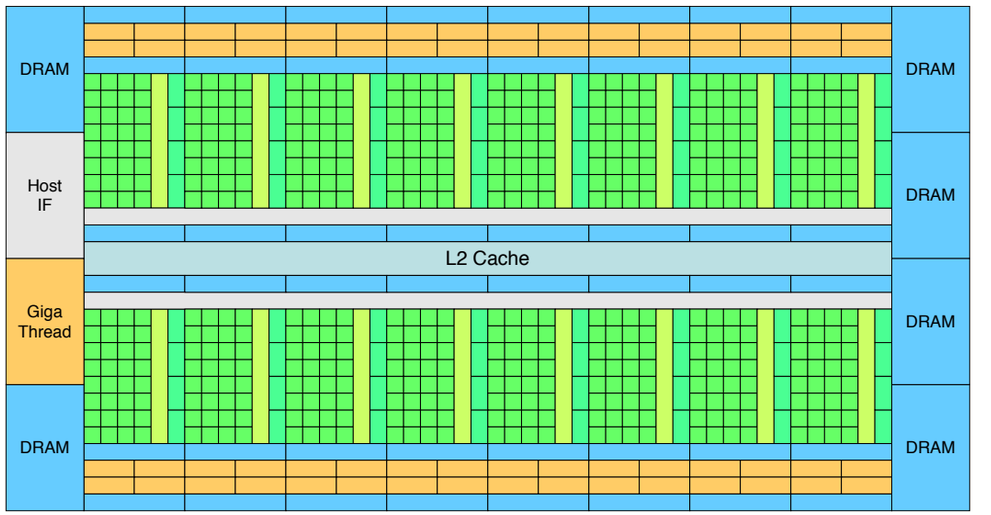
- Parametrisierung
  - dimBlock: Threads/Block
  - dimGrid: Blocks/Grid
  - **`kernel<<<dimGrid, dimBlock>>>(… Funktionsparameter …);`**
- Beispiel DAXPY

```
void daxpy_serial(int n, double a, double *x, double *y)
{  for(int i=0; i<n; ++i)
       y[i]= a*x[i]+y[i];
}
daxpy_serial(n,1.7,x,y);
```

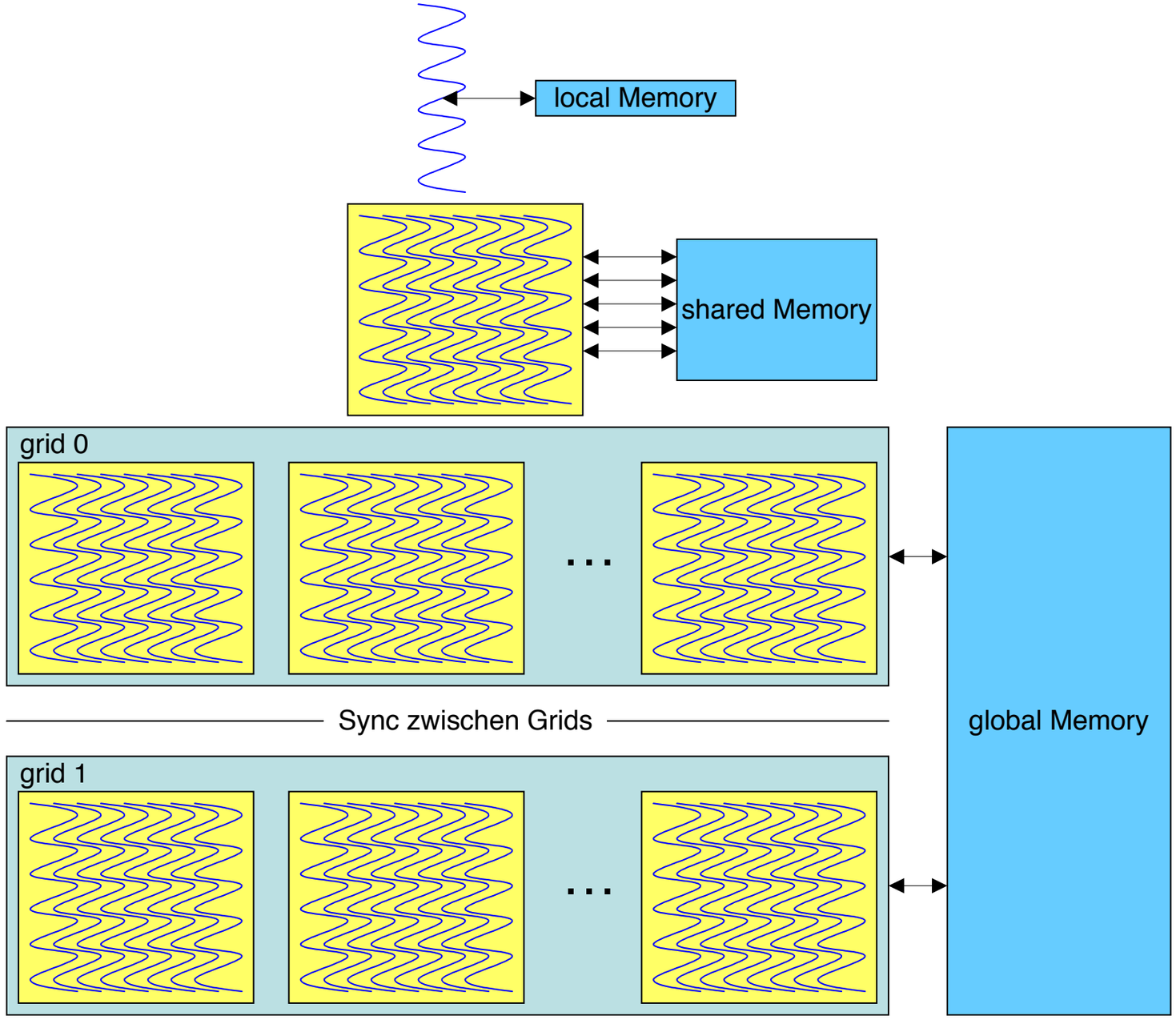
```
_global_
void daxpy_parallel(int n, double a, double *x, double *y)
{  int i=blockIdx.x*blockDim.x+threadIdx.x;
   if (i<n) y[i]= a*x[i]+y[i];
}
daxpy_parallel<<<(n+255)/256,256>>>(n,1.7,x,y);
```

- threadId und blockId
  - bis 3 Dimensionen
  - .x,.y,.z

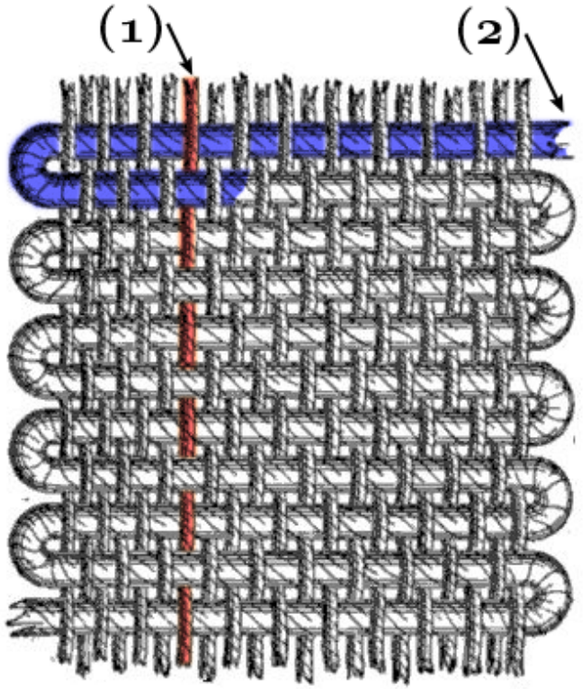
- Synchronisation mit Barrieren
  - **`_syncthreads()`**
  - threads im Block warten bis alle anderen da
  - Schreiben im gemeinsamen Speicher nachher für alle sichtbar
- Threadblock in einem SM
  - Bsp: 256 Threads/Block
  - 32 SPs im SM
  - Threadblock aufteilen
  - Teile sequentiell ausführen
  - evtl. 'interleaved' <=> Speicherlatenz
  - Cooperative Thread Array CTA

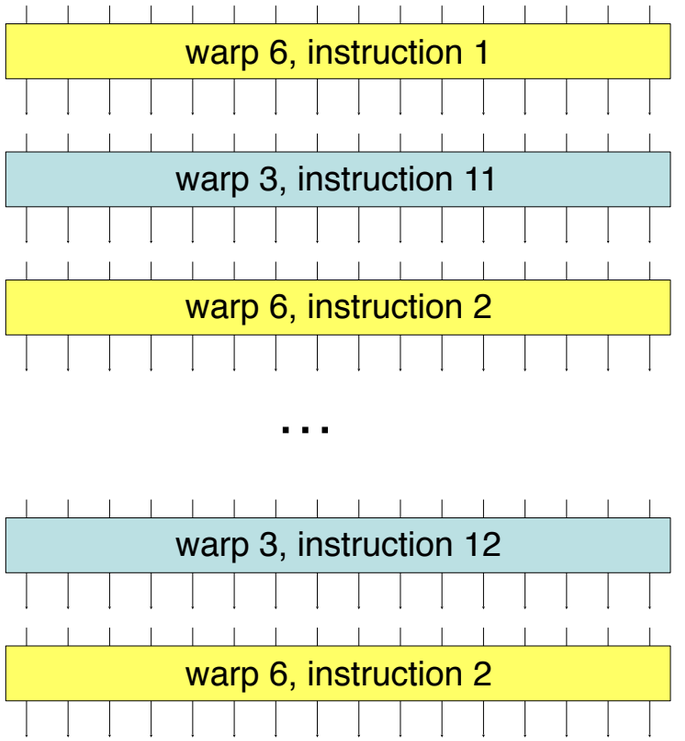


- Speicher
  - lokal on chip: **`_shared_`**
  - global im Cache/DRAM: **`_device_`**
- Kommunikation zwischen Threadblocks
  - im globalen Speicher
  - atomare Operationen
  - keine Sync-Barrieren
- Speichermanagement
  - **`cudaMalloc()`**, **`cudaFree()`**, **`cudaMemcpy()`**

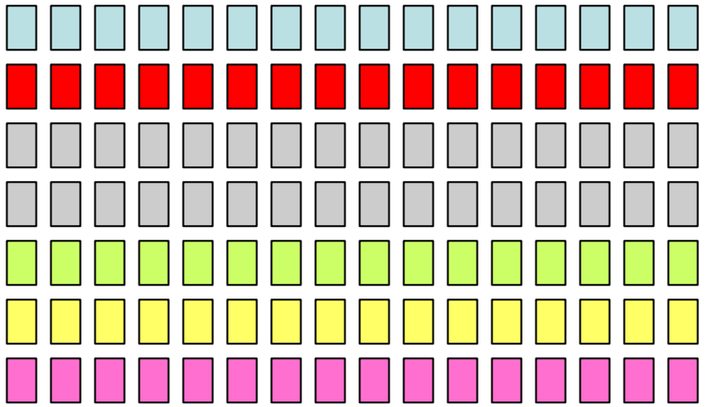

local Memory
shared Memory
grid 0
...
Sync zwischen Grids
grid 1
...
global Memory

- Warp
  - Weberei: Kette
  - Thread als Kettfaden
  - SM fasst Threads zu Warps zusammen
  - Threadblock evtl. mehrere Warps
  - Warps eines Threadblocks in einem SM
  - Bsp: 32 Threads/Warp
  - 256 Threads im Block => 8 Warps
- Multithread-Scheduler
  - wählt einen Warp aus pro Zyklus
  - Warp muss 'bereit' sein
  - eine Instruktion aus dem Warp
  - Analogie Schuss beim Weben
  - nur gleiche Operationen
  - andere Threads inaktiv/SP inaktiv
  - auch für Special Function Unit
  - maximal 8 Thread-Blöcke/SM

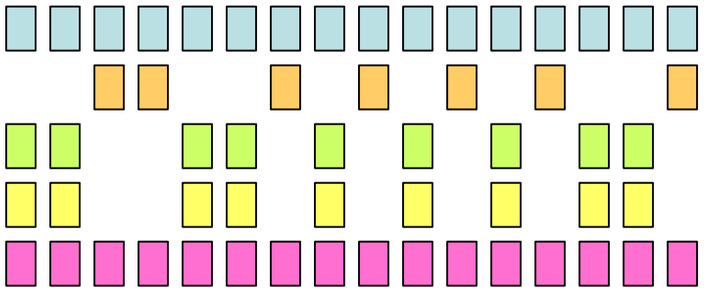




- Single Instruction Multiple Thread (SIMT)
  - Warp blockiert wenn eine Op wartet
    - Speicherlatenz, Branch delay slots
    - pipelinisierte Float Arithmetik
  - andere Warps schedulen
  - breite Thread-Blöcke (->SIMD)
  - viele Thread-Blöcke (->SIMT)
- PC und Registersatz pro Thread
  - begrenzt Anzahl warps/SM
- Thread-Divergenz
  - Verzweigung
  - pro Takt und SM nur ein Op-Typ
  - evtl. Thread in mehreren Zyklen bearbeiten
  - Pfad-basiertes Scheduling
  - keine Divergenz -> Effizienz
- Throughput global betrachten
  - Thread-Latenz sekundär
  - klassisch: viele Instruktionen in einem Thread rechnen
  - GPU: viele Instruktionen!



```
__kernel void vector_add_gpu (
      __global const float* src_a,
      __global const float* src_b,
      __global float* res,
      const int num)
{  const int idx = get_global_id(0);
   if (idx < num)
      res[idx] = src_a[idx] + src_b[idx];}
```



- SP-Instruction Set Architecture
  - PTX: Parallel Thread Execution
  - Virtuelle Maschine
  - Optimierung/Übersetzung auf GPU …
  - PTX-to-GPU im Treiber
- Operanden
  - Bits: .b8, .b16, .b32, .b64
  - unsigned int .u8,…, .u64
  - signed int .s8, …, .s64
  - float .f16, .f32, .f64
- Instruktionen mit bis zu 4 Operanden
  - destination, op1, op2
  - op3 für MAC bzw. FMA
  - Speichertyp (shared, global, constant, texture)
- Datentransport
  - ld.space.type: **`ld.global.b32 reg,[a+offs]`**
  - st.space.type: **`st.shared.b32 [a+offs],reg`**
  - atom.space.op.type: **`atom.gobal.cas.b32 dest,[a],b,c`**
  - compare and swap, exchange, add, inc, dec, min, max, and, or, xor
  - texture Zugriff 2D mit Interpolator

- Arithmetik
  - add, sub, mul, mad, fma, min, max, cvt, …
  - special function: sqrt, sin, cos, lg2, ex2, …
  - logical: and, or, xor, not, cnot, shl, shr
- Flusskontrolle
  - branch mit Prädikat-Register: **`@p bra target`**
  - call mit Resultat und Parametern: **`call (res) rechnewas, (a,b,c)`**
  - Argumente 'by-value'
  - ret
  - exit: Thread beenden
- Barriere mit verschiedener Semantik
  - sync: warten bis alle (oder Anzahl b) Threads da
  - arrive: markiert Ankunft, blockiert nicht
  - reduction: zählt Threads mit Prädikat p (true, false)
  - **`bar.sync a {,b}`**: wartet bei Barriere a auf b threads, 0 = alle
  - **`bar.red p, a {,b}, {!}c`**: Prädikat auswerten, Bedingung: and, or, popc
  - **`bar.arrive a,b`**
  - bar.sync und bar.red synchronisieren Speicher

- Beispiel Matrixmultiplikation

  - konventionell

```
void MatrixMul(float* N, float* M, float* P, int width)
{ for (int i = 0; i<width; i++)
      for (int j = 0; j<width; j++) {
          float sum =0;
          for (int k = 0; k<width; k++)
              sum += M[i*width + k] * N[k*width + j];
      }
      p[i*width + j] = sum;
}}
```

  - thread

```
_global_ void MaMulKernel(float* Nd, float* Md, float* Pd, int width)
{ int col = threadIdx.x;
  int row = threadIdx.y;
  float sum = 0;
  for (int k = 0; k<width; k++)
      sum += Md[row*width + k] * Nd[k*width + col];
  Pd[row*width + col] = sum;
}
```

- Verteilung auf Threads ersetzt Schleifen
  - 2 Schleifen: zweidimensionales Thread-Array

```
dim3 dimBlock(width,width);
dim3 dimGrid(1,1);
MaMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

- Hostprogramm

```
void MatrixMul(float* N, float* M, float* P, int width)
{ int size = width * width*sizeOf (float);
  float *Md, *Nd, *Pd;
  dim3 dimBlock(width,width);
  dim3 dimGrid(1,1);

  cudaMalloc((void**) &Nd, size);
  cudaMemcpy(Nd,N,size,cudaMemcpyHostToDevice);
  cudaMalloc((void**) &Md, size);
  cudaMemcpy(Md,M,size,cudaMemcpyHostToDevice);
  cudaMalloc((void**) &Pd, size);

  MaMulKernel<<<dimGrid, dimBlock>>>(Md,Nd,Pd,width);

  cudaMemcpy(Pd,P,size,cudaMemcpyDeviceToHost);
  cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

• Problem: Threadblock hat max. 1024 Threads (Fermi)

- MaMulKernel benutzt nur threadIdx
- hier also nur 32*32 Matrix
- blockIdx auch benutzen
- Matrix in Kacheln aufteilen

Pd[0,0]	Pd[1,0]	Pd[0,1]	Pd[1,1]
Md[0,0]* Nd[0,0]	Md[0,0]* Nd[1,0]	Md[0,1]* Nd[0,0]	Md[0,1]* Nd[1,0]
Md[1,0]* Nd[0,1]	Md[1,0]* Nd[1,1]	Md[1,1]* Nd[0,1]	Md[1,1]* Nd[1,1]
Md[2,0]* Nd[0,2]	Md[2,0]* Nd[1,2]	Md[2,1]* Nd[0,2]	Md[2,1]* Nd[1,2]
Md[3,0]* Nd[0,3]	Md[3,0]* Nd[1,3]	Md[3,1]* Nd[0,3]	Md[3,1]* Nd[1,3]

```
_global_ void MaMulKernel(float* Nd, float* Md, float* Pd, int width)
{ int row = blockIdx.y*c_tile_width + threadIdx.y;
  int col = blockIdx.x*c_tile_width + threadIdx.x;
  float sum = 0;
  for (int k = 0; k<width; k++)
      sum += Md[row*width + k] * Nd[k*width + col];
  Pd[row*width + col] = sum;
}

dim3 dimGrid(width/c_tile_width, width/c_tile_width);
dim3 dimBlock(c_tile_width, c_tile_width);

MaMulKernel<<<dimGrid,dimBlock>>>(Md,Nd,Pd,width);
```

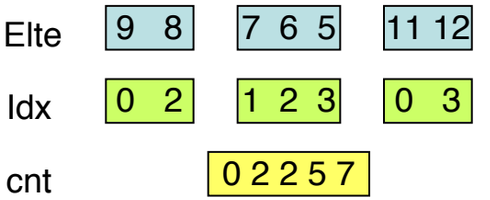
- Speicher
  - Register (on chip)
  - Cache bzw. Shared Memory/SM (on chip)
  - Constant Cache/SM (road only)
  - Texture Cache/SM (on chip)
  - Global Memory für alle SMs, extern
  - local: extern, nur thread
- Speicherdurchsatz kritisch
  - Durchsatz zum global Memory fest
  - Fermi z.B. 144 GByte/sec = 36G Single Precison Loads
  - Verhältnis load/store zu ALU 1:1?
  - Speicherduchsatz << ALU-Durchsatz
  - Bsp. MaMult: 2 Speicherzugriffe und 2 Alu-Ops pro Schleife
  - Single Precision: 36 GFlops/1030 GFlops ~ 1/28
- 'Gemeinsam Laden'
  - on-Chip Speicher benutzen
  - Wiederverwendbarkeit
  - Speicher in Kacheln aufteilen
  - Elemente tile_width-oft verwendet

- Jeder Thread lädt ein Md und ein Nd Element
  - viele Load-Store Units
  - alle blockiert beim Laden
  - in Shared Memory laden: Nds und Mds

```
_global_ void MaMulKernel(float* Nd, float* Md, float* Pd, int width)
{ _shared_float_Mds[c_tile_width][c_tile_width];
  _shared_float_Nds[c_tile_width][c_tile_width];
  int bx = blockIdx.x; by = blockIdx.y;
  int tx = threadIdx.x; ty = threadIdx.y;
  int row = by * c_tile_width + ty;
  int col = bx * c_tile_width + tx;
  float sum = 0;
  for (int m=0; m < width/c_tile_width; m++) {
      Mds[ty][tx] = Md[row*width + m*c_tile_width + tx];
      Nds[ty][tx] = Nd[(m*c_tile_width+ty)*width + col];
      _syncthreads();
      for (int k = 0; k< c_tile_width; k++)
          sum += Mds[ty][k] * Nds[k][tx];
      _syncthreads();
  }
  Pd[row*width + col] = sum;
}
```

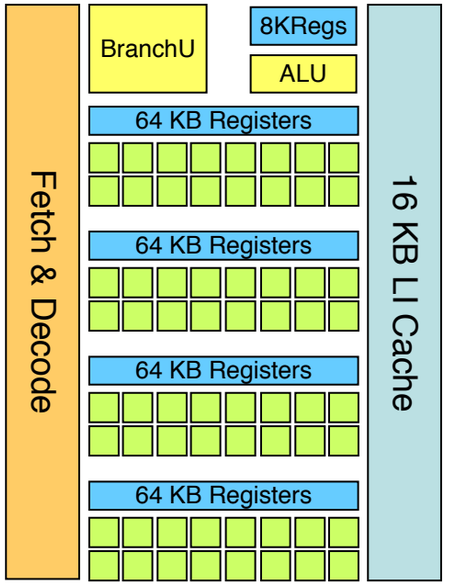
- Sparse Matrix-Multiplikation
  - Compressed Sparse Array CSA
  - Vektor mit Elementen
  - Vektor mit Zeilenindizes
  - Vektor mit Element-Anzahl

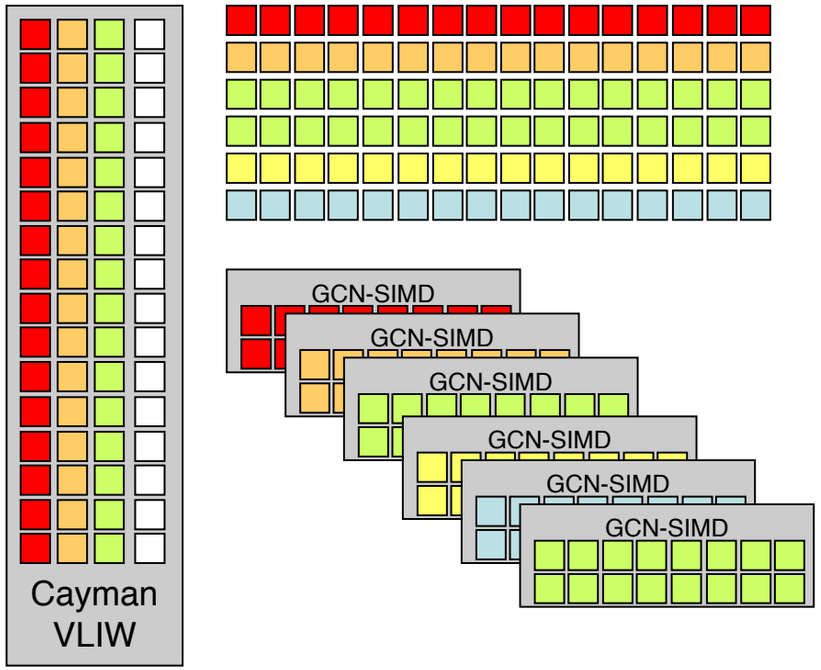
$$\begin{bmatrix} 9 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 7 & 6 & 5 \\ 11 & 0 & 0 & 12 \end{bmatrix}$$




```
_device_ float multrow(int rowsize, int *Idx, float *Elte, float *x)  {
   float sum = 0;
   for (int col = 0; col<rowsize; col++)
       sum += Elte[col] * x[Idx[col]];
   return sum;                                                    }

_global_ void csamul_k(int *cnt, int *Idx, float *Elte, int rowcnt,
                       float *vec_in, float *vec_out) {
   int row = blockIdx.x*blockDim.x + threadIdx.x;
   if (row < rowcnt)
      int rbegin = cnt[row]; int rend = cnt[row+1];
      y[row]=mult_row(rend-rbegin,Idx+rbegin, Elte+rbegin, vecin); }}

int blocksize = 512;
int nblocks = (num_rows+blocksize-1)/blocksize;
csamul_k<<<nblocks,blocksize>>>(cnt,Idx,Elte,num_rows,x,y);
```

- ATI Graphics Chip (später AMD)
  - Radeon 6900 (Cayman): VLIW4
  - Compiler (auch im Grafiktreiber)
  - statisches Scheduling
  - Grafik und eingeschränkt Compute
  - VLIW
- AMD Fusion
  - CPU und GPU auf einem Chip bzw Die
  - AMD x86 Kerne
  - Graphics Core Next
  - (CU + scalar_ALU + L1_Cache) * n + L2 Cache
- GCN Compute Unit
  - 4 * 16-fach SIMD
  - Scheduler für Wavefronts
  - 4*10 Wavefronts
  - Bitmaske für SIMD-Vektoren



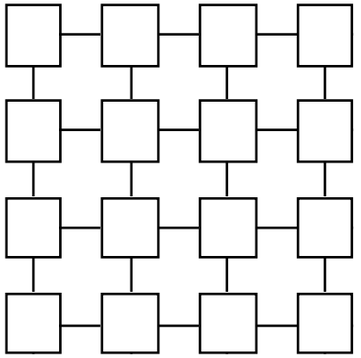


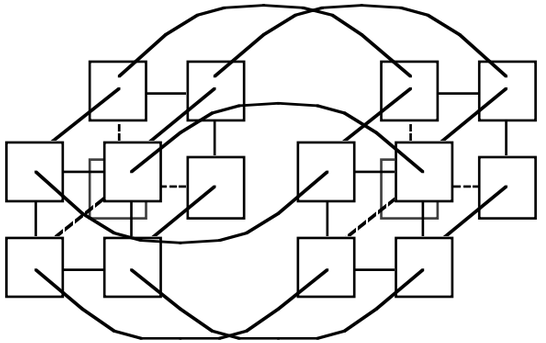
- Sprachen für GPU Computing
- Ct [Intel]
- Cuda [NVIDIA]
- OpenCL [Apple, Khronos]
  - C99
  - Speicher-Spezifikation: _global, _local, _constant, _private
  - keine Funktionspointer, Rekursion, 'variable' Arrays
  - Vektoroperationen
  - Synchronisation, atomare Funktionen
- OpenCL Laufzeitsystem
  - Device, Context
  - Command Queue
  - Kommunikation mit Puffern
  - Kernel-Argumente bereitstellen

```
error = clSetKernelArg(vector_add_k, 0, sizeof(cl_mem), &src_a_d);
error |= clSetKernelArg(vector_add_k, 1, sizeof(cl_mem), &src_b_d);
error |= clSetKernelArg(vector_add_k, 2, sizeof(cl_mem), &res_d);
error |= clSetKernelArg(vector_add_k, 3, sizeof(size_t), &size);
assert(error == CL_SUCCESS);
const size_t local_ws = 512;
const size_t global_ws = shrRoundUp(local_ws, size);   // Total number of work-items
error=clEnqueueNDRangeKernel(queue, vector_add_k, 1,NULL,&global_ws,&local_ws,
```

1.5 Netzwerkbasierte Multicomputer

- Prozessoren mit RAM an den Knoten
  - Nachrichten-Vermittlung als Nebenaufgabe
  - 2-D Gitter (Transputer): Bildverarbeitung, 2-D-Probleme





  - Hyperkubus: Intel's Hypercube: Verbindung $O(\log_2 n)$

- Connection Machine [D. Hillis, 1986]
  - 65536 Prozessoren, jeweils 4096 Bit Speicher
  - 1-Bit, 4096 Bit Speicher
  - 16 Prozessoren pro Chip
  - paarweise verbunden
  - 12-Hypercube zwischen Chips
  - hierarchisches Routing mit 12 Bit Adressen
  - Thinking Machines Corporation
  - CM-1, CM-2

- Transputer [Inmos, 1984]
  - RISC/CISC Architektur
  - T2 16 Bit, T4 32 Bit, T8 float
  - system on a chip
  - 4 serielle Links (5, 10, 20 Mbit/s)
  - später auch Switches
  - Hardware Task Scheduler
  - Workspace Pointer
  - FPS, Parsytec
- Infiniband
  - bidirektional, Punkt-zu-Punkt
  - 1,4,12 Lanes
  - 8B/10B Codierung: SDR, DDR, QDR
  - SDR: 2,5 Gbits/s
  - FDR, EDR 64B/66B
  - SDR-Switches mit 200 nsec Verzögerung
  - 4KB-Pakete
  - Remote DMA
  - Multicast

2. Software-Infrastruktur

2.1 Algorithmische Komponenten

2.1.1 Zeit

- Lamport Algorithmus
  - jeder Prozess hat eigene Uhr = Zähler
  - lokale Ereignisse:

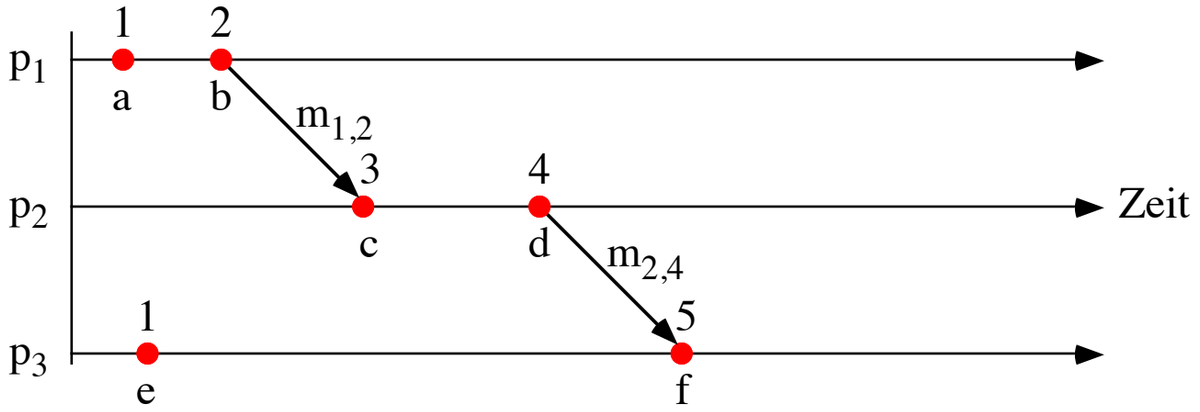
```
c_local++;
```

  - Sendeereignis:

```
c_local++; send(message,c_local);
```

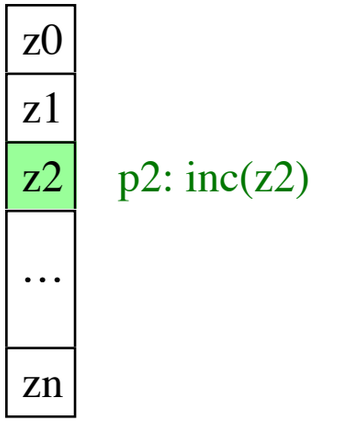
  - Empfangsereignis:

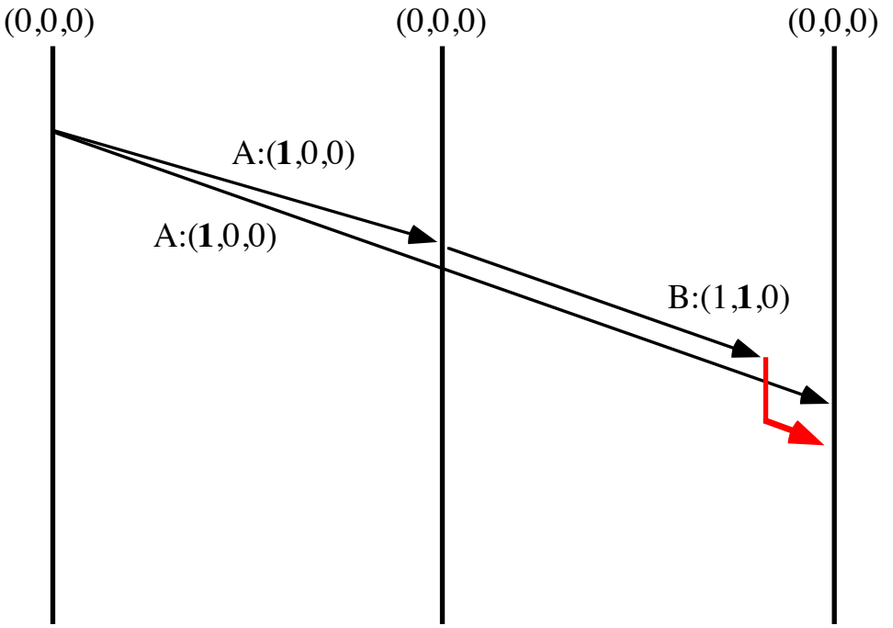
```
receive(message,c_remote);
c_local = max(c_local,c_remote)+1;
```



- C(a)<C(b): a→b oder allb
  - keine totale Ordnung

- Zeitstempel
  - jeder Prozess p_i hat *logische* Zeit z_i
  - bei Nachrichtenversand inkrementieren
- Vektorzeit
  - Zeitstempel-Vektoren
  - jede Station hält lokalen Vektor
  - Nachrichten enthalten Vektoren
  - ->Vektor der Quelle beim Senden
  - Empfang: Vergleich mit lokalem Vektor
  - Empfänger verzögert evtl. Nachrichten





- Implementierung der Vektorzeit
  - Sender k packt Zeitstempel-Vektor in Nachrichten
  - Empfänger vergleicht Stempel-Vektor V mit lokalem Vektor L
  - accept if $(V_k = L_k + 1)$ and $(V_i \leq L_i \; \forall \; i \neq k)$

p0	p1	p2	p3	p4	p5
4	3	3	3	2	3
6	7	5	7	6	7
8	8	8	8	8	8
2	2	2	3	2	3
1	1	1	1	1	1
5	5	5	5	5	5

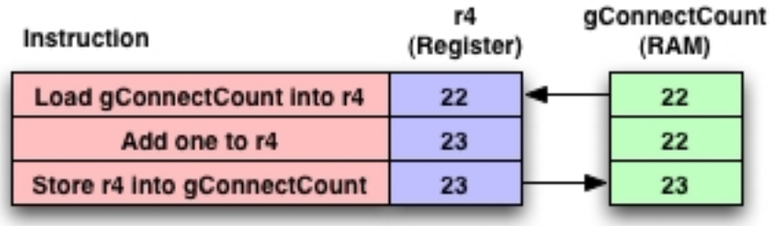
p0	p1	p2	p3	p4	p5
4	4	3	4	2	4
6	7	5	7	6	7
8	8	8	8	8	8
2	2	2	3	2	3
1	1	1	1	1	1
5	5	5	5	5	5

Nachricht von p0 an alle

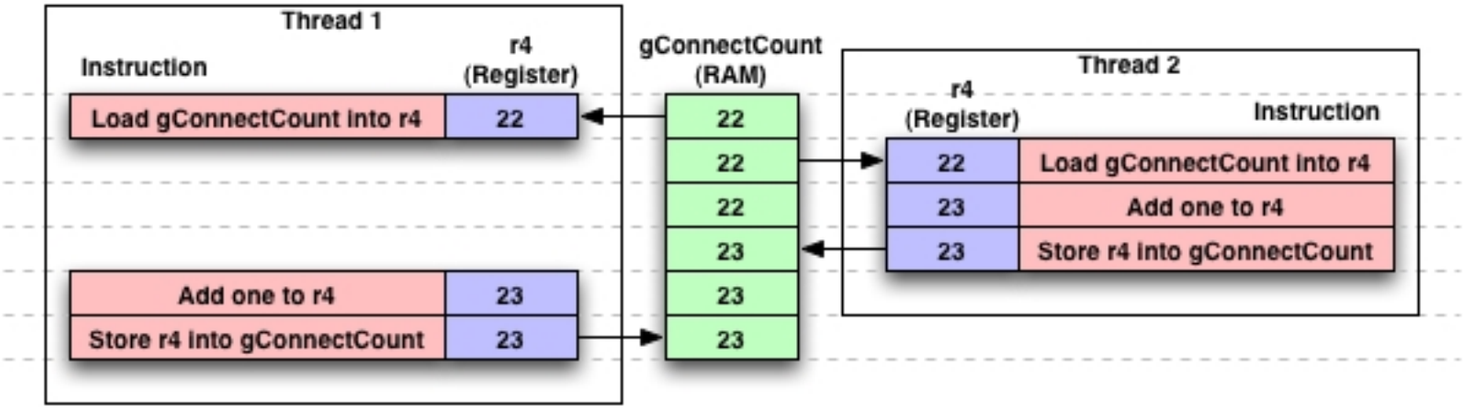
4	6	8	2	1	5	Daten

p0	p1	p2	p3	p4	p5
4	3	3	3	2	3
6	7	5	7	6	7
8	8	8	8	8	8
2	2	2	3	2	3
1	1	1	1	1	1
5	5	5	5	5	5
	accept	delay	accept	delay	accept

2.1.2 Koordination



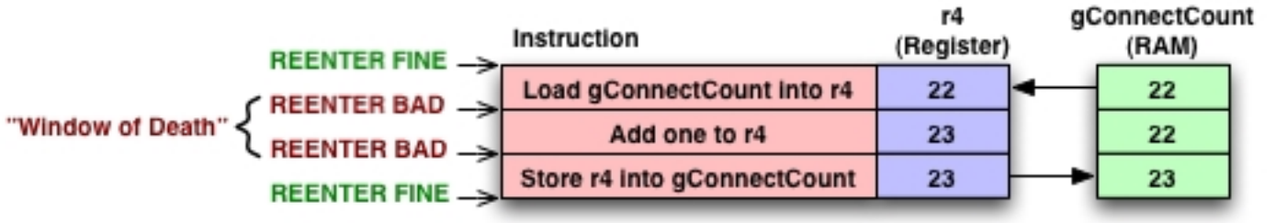
- Nebenläufigkeit der Prozesse
  - in einem Rechner durch Interrupts
  - im Parallelrechner mit Shared Memory
  - im Verteilten System
- Mehrere Prozesse wollen eine Ressource benutzen
  - Variablen (bool, Zähler), Objekte, Geräte
  - Bsp.: Kontostand, Drucker, Verwaltungsvariablen Ringpuffer



- Zugangskontrolle nötig
  - Schutzvariablen?
  - auch der Zugriff auf Schutzwariablen kann überlappen
  - Operationssequenz: Lesen-Testen-Setzen-Schreiben
  - Testen - *Testen - Setzen* – Setzen

2.1.2.1 Mutual Exclusion

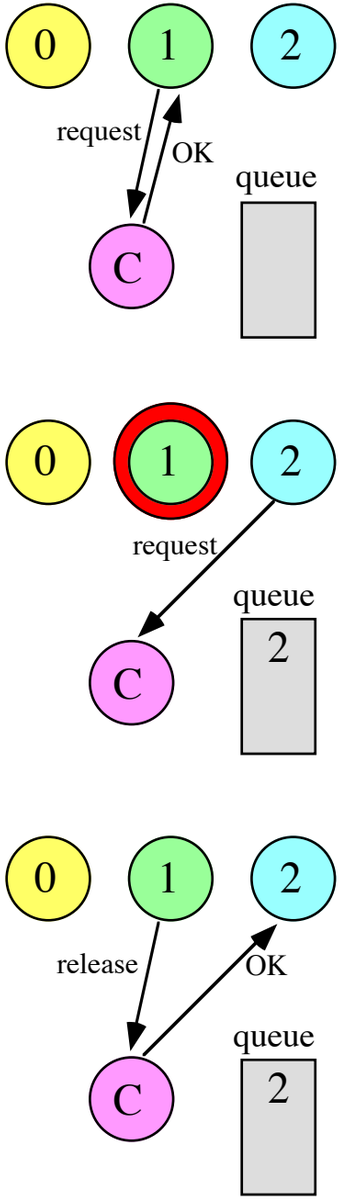
- Lock schützt kritisch Region
  - Betriebssytemkonzept
  - 'Schutzbit' kontrolliert Eingang
  - Testen - Setzen - (Betreten - Arbeiten - Verlassen) – Rücksetzen



  - unteilbare Operationen: Read-Modify-Write
  - 68000: TAS (Test and Set), IA: XCHG
- PPC: lwarx+stwrx
  - Load Word And Reserve indeX
  - STore Word Conditional indeX

```
retry: lwarx  r4,0,r3   //integer lesen und reservieren
       addi   r4,r4,1   //increment
       stwcx. R4,0,r3   //Versuch, integer zurückzuschreiben
       bne-   retry     //Falls fehlgeschlagen – nochmal versuchen
```

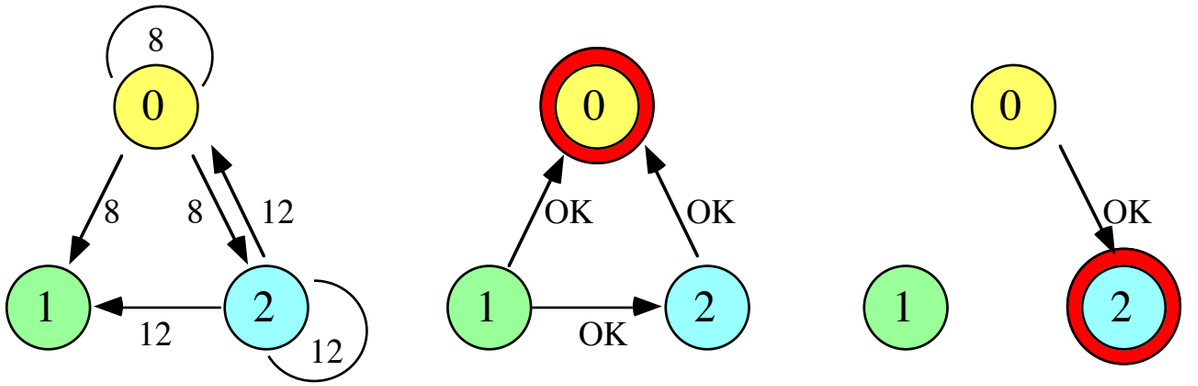
- Singleton pattern: Objektinstanziierung als Lock
  - höchstens eine Instanz existiert
- Verallgemeinerung: Semaphor [Dijkstra, 1970]
  - Passeeren (`m--`) und Vrijgeven (`m++`)
  - 'Zählende' Locks (`if (m<=0) wait;`)
  - Locks werden auch binäre Semaphore oder Mutex genannt
- Allgemeines Verteiltes System
- Zentraler Verwalter
  - Request vom Klienten an den Verwalter
  - Permission wenn frei
  - Verzögern oder Ablehung wenn 'Besetzt'
  - Warteschlange
- Bewertung
  - einfach zu implementieren
  - fair
  - zuverlässiges Protokoll vorausgesetzt
  - 'single point of failure'



- Verteilter Algorithmus [Ricart, Agrawal 1981]
  - Request an alle: CR, Prozess#, Lamport-Zeit

```
if (requestin)
    if (inCritRgn == request->Rgn) queue(request);
    else if (ownrequest_pending)
            if (lowesttime(allRequests)->process == self)
            {  queue(request); sendOK(self); }
            else sendOK(request->process);
         else sendOK(request->process);
if (OKIn) {
inCritRgn = OKIn.critRgn; DoCritRgn; inCritRgn = NULL;}
sendOK(dequeue(request)->process);
```

- n points-of-failure
  - immer antworten
  - timeouts



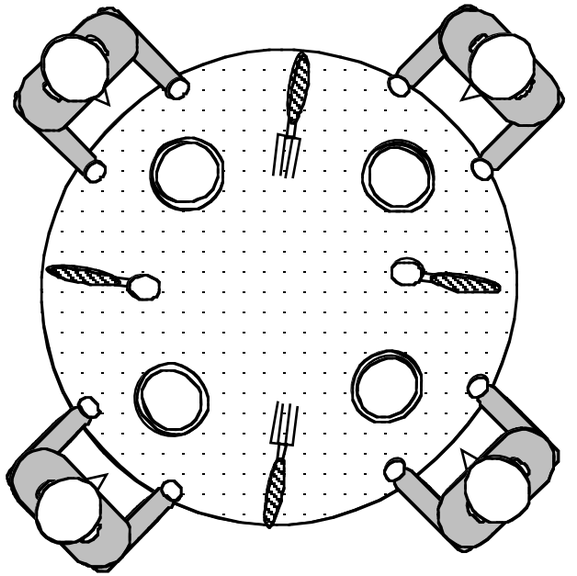
- Token Ring
  - logische Ringstruktur
  - Token = Erlaubnis, kritische Region einmal zu betreten
  - an logischen Nachfolger weiterreichen
  - Bestätigung über Empfang
  - keine Bestätigung -> nächsten Nachfolger
  - komplettes Topologiewissen nötig
- Vergleich

Algorithmus	Nachrichten/Vorgang	Verzögerung	Probleme
Zentral	3	2	Koordinator abgestürzt
Verteilt	2(n-1)	2(n-1)	irgendeiner abgestürzt
Token passing	1 - ∞	0 bis n-1	Tokenverlust Prozess abgestürzt

=> Verteiltes Verfahren ist schlecht

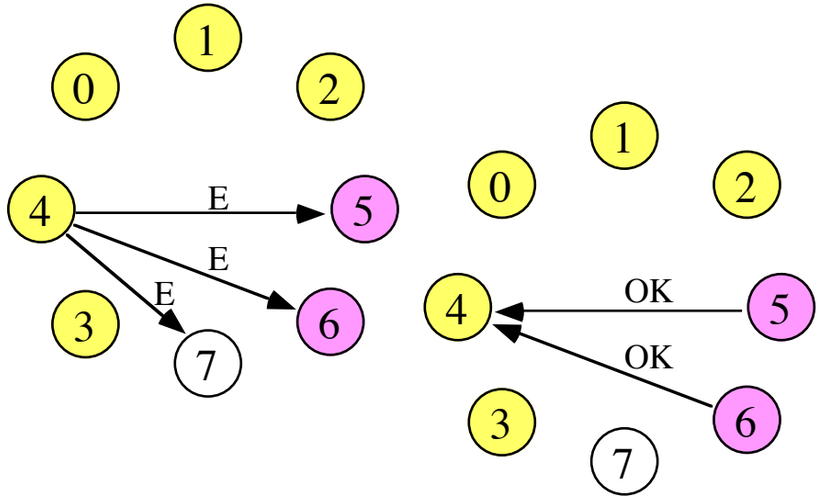
2.1.2.2 Deadlocks

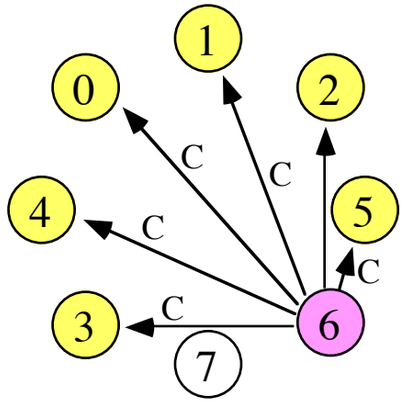
- Mutex besetzt: warten
  - dining Philosophers
  - Abhängigkeiten und Warten erzeugen Wartegraph
  - Zyklen im Wartegraph => Verklemmung
- Entdecken und Lösen
  - Prozess beenden
  - Transaktion abbrechen
  - zentral oder verteilt
  - Graph aufbauen und Zyklen suchen
- Verhindern
  - nur eine Ressource pro Prozess
  - release then lock
  - Ressourcenordnung
  - Zeitstempel: junge und alte Transaktionen
- Vermeiden (Wissen?)



2.1.2.3 Wahlverfahren

- Zentrale Leitung verteilt 'wählen'
  - alle Prozesse haben Nummer
  - alle Prozesse sind einander bekannt
  - kein Wissen über Aktivität
- Bully-Algorithmus (bully = Tyrann)
  - Station P merkt, daß Koordinator weg
  - P sendet Election(P) an 'höhere' Prozesse
  - Anwort 'OK' => Abbruch
  - keine Antwort => P neuer Koordinator
  - Coordinator(P) an alle
- Q empfängt Election(P)
  - tolerieren oder
  - OK an P + Election(Q) an alle höheren
- Falls ein Koordinator X wieder aufwacht:
  - Coordinator(X) an alle
- Ringalgorithmus auch möglich



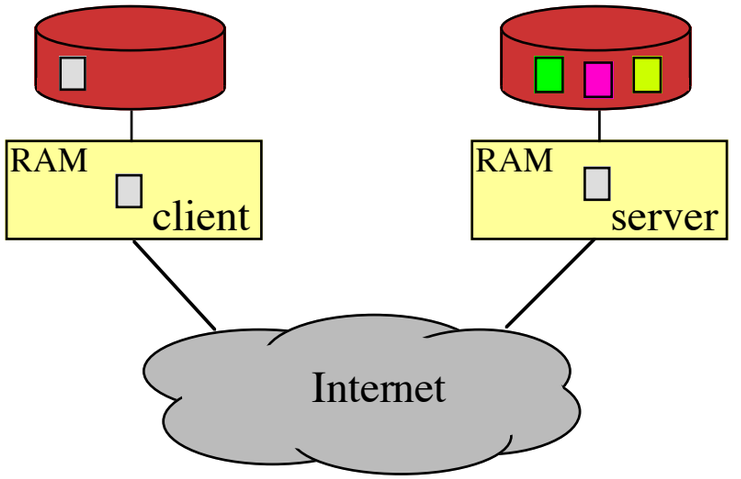


2.1.2.4 Effizienz beim verteilten Zugriff

- Daten gemeinsam nutzen
  - Koordination: Semaphore, Locking
  - Konsistenz: Datenbanken, Transaktionen
  - Effizienz: Caching, verteilte Kopien
- Entfernter Zugriff kostet Zeit
  - Transport im Netzwerk
  - TCP-Verbindung, …
  - Lesen und besonders Schreiben
  - zeichenweiser Zugriff auf Strings …
- Caching
  - lokale Kopie -> schneller Zugriff
  - dynamisch, heuristisch
  - Ersetzungsalgorithmen
  - besonders häufig in (verteilten) Dateisystemen
- Replikation
  - Verfügbarkeit
  - Fehlertoleranz
- Problembereiche: Konsistenz und Transparenz

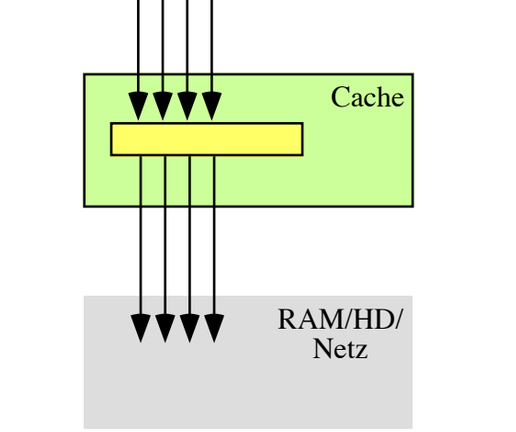
2.1.2.4.1 Caching

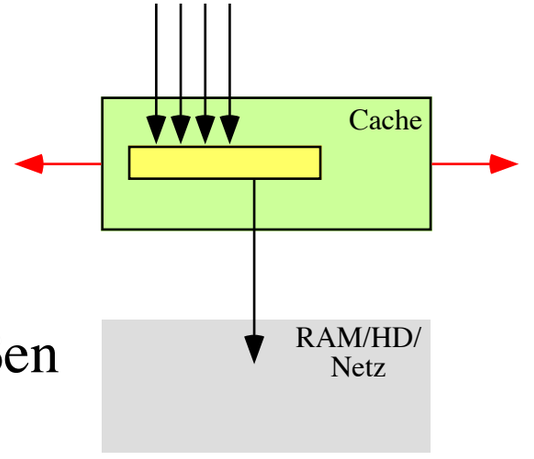
- Objekte 'in der Nähe' halten
  - Speicherzellen, Seiten, Sektoren, Tracks, Files, …
  - Cache-RAM, RAM, Controller, …
  - Zugriffskosten reduzieren
- Sonderfall verteiltes Dateisystem
  - Files
  - Server: RAM
  - Klient: Festplatte, RAM
  - Prozess, OS, Caching-Prozess



- Aufgaben
  - Einheit des Cachings: Blöcke oder Files?
  - Cache kleiner als Speicher: was soll 'gecacht' werden?
  - Prefetching?
- Last recently used
  - Element das am längsten nicht mehr benutzt wurde ersetzen
  - File-Cache-Zugriff selten vgl. CPU-Cache
  - verkettete Liste als Verwaltungsstruktur

- Write-through
  - Schreiben im Cache
  - Schreiben auf Orginal
  - Validierung späterer Zugriffe: Zeitstempel,Prüfsummen
- Delayed write
  - Netzwerk-Nachricht bei jedem Schreiben aufwendig
  - Sammeln der Updates und Burst
  - unklare Semantik
- Write on close
  - Session-Semantik
  - Öffnen - lokale Updates - komplett-Update beim Schließen
  - Locking bzw. Transaktionssemantik
  - evtl. mit Verzögerung: Löschen häufig nach Close
- Write-invalidate: andere caches konsistent halten
- Zentrale Kontrolle
  - alle Zugriffe anmelden: read, write, update
  - Remove-from-Cache bei konkurrierenden Zugriff
  - unsolicited-message



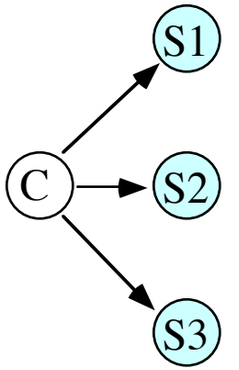


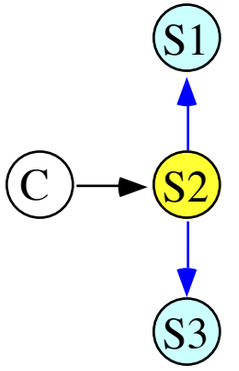
2.1.2.4.2 Replikation

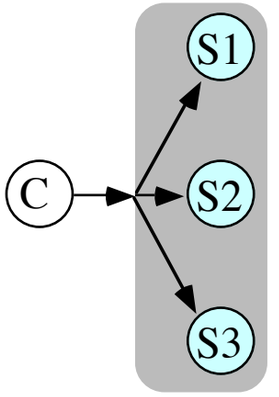
- z.B. Mirror-Server
  - ftp, WWW, …
  - Usenet News
- Client/Server, Peer-to-peer
  - Nachrichtenaustausch
  - Lesen Regelfall
  - Schreiben Ausnahme
  - besondere Vorkehrungen beim Schreiben
- Verteilter gemeinsamer Speicher (DSM)
  - Speicherobjekte: Variablen
  - Seiten
- Replikationsmanagement
  - Konsistenzmodell
  - Verhältnis Lesen-Schreiben
  - Aufwand

- Asynchrones Replikationsmanagement
  - Lesen immer lokal
  - Schreiben lokal
  - Update anderer Kopien periodisch
  - inkohärenz bis zur Synchronisation
  - entkoppelt und einfach
  - Bsp. News: Antwort vor der Frage
- Kausal geordnetes Replikationsmanagement
  - kausale Ordnung für abhängige Veränderungen
  - zeitliche Kohärenz
  - nur von einem benutzte Replikate inkohärent
- Synchrones Replikationsmanagement
  - alle Replikate atomar geändert
  - volle Konsistenz
  - hoher Aufwand

- Erzeugung von Replikaten
  - nicht alles muß repliziert werden
  - Auswirkungen auf Konsistenzerhaltung
- Explizite Replikation
  - klientengesteuert: Anforderung an mehrere Server
  - Zugriff auf ein Replikat
  - Konsistenzmanagement beim Klienten
- Lazy Replikation
  - Objekterzeugung durch Server
  - Server legt Replikate bei anderen Servern an
  - Konsistenzmanagement durch Server
- Servergruppe
  - Gruppenkommunikation
  - Objektanforderung bewirkt n Replikate
  - Schreibzugriff an Gruppe
  - Konsistenz abhängig von Gruppensemantik



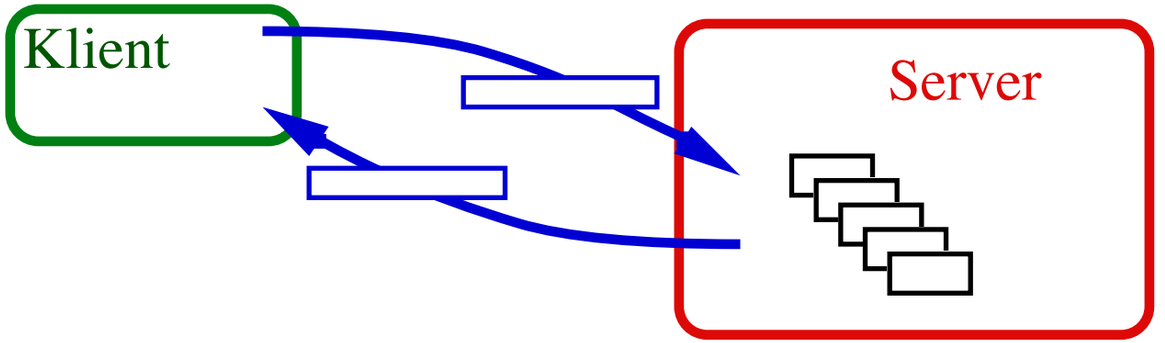




- Primärkopien
  - 'Orginal' im Primärserver
  - Replikate auf Sekundärserver
  - Lesen überall, Schreiben nur auf Orginal
  - Update der Replikate in der Servergruppe
  - Primärserver-Ersatz siehe Election
- Abstimmung
  - N Server
  - N/2+1 Replikate beschreiben mit Zeitstempel
  - N/2+1 Replikate lesen
- Verallgemeinerung: Quota
  - $N_l+N_s > N$
  - N_l und N_s entsprechend Verhältnis Lesen/Schreiben optimieren
- Lazy Update
  - Server sammeln Updates
  - gossip-Nachrichten
  - schwache Konsistenz
  - News
- Peermodell als Variante

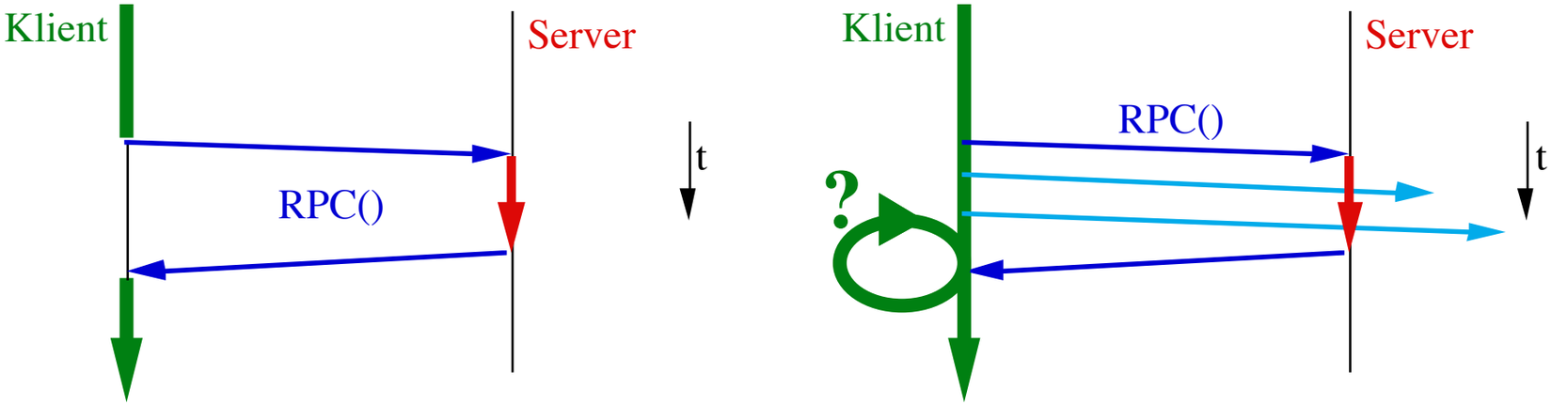
2.2.1 Remote Procedure Call

- Nachrichtenbasierte Kommunikation zwischen Knoten im Netz



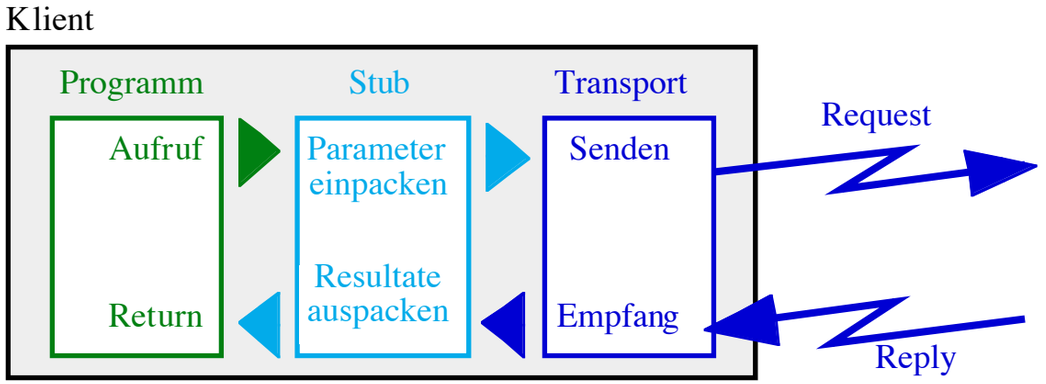
- Teilweise vergleichbar einem lokalen Prozeduraufruf
  - Parameter an die ferne Prozedur übergeben
  - Resultate an Klienten zurückliefern (Return)
  - eventuell warten
- Probleme im Klienten-Programm
  - globale Variablen
  - äußere Prozeduren
  - Zeiger auf Datenobjekte im Heap
  - komplexe Datenstrukturen schwierig

- Synchroner RPC
  - blockierend
  - kein explizites Warten auf Antwort
  - sicherheitshalber Time-Out bei Netzstörung ...



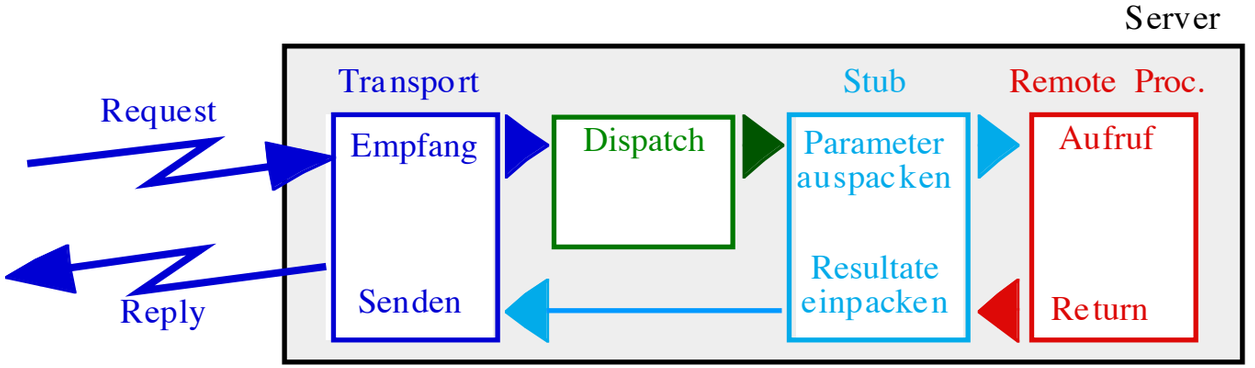
- Asynchroner RPC:
  - Klient läuft weiter
  - kann weitere RPCs absetzen
  - explizites Abfragen, ob Antwort schon da
  - oder "Completion routine" oder HW-Interrupt ...

- Besondere Programmiersprachen
  - integrierter RPC
  - Cedar, Argus, …
- Interface Description Languages
  - Präprozessor
  - XDR, ANSA Testbench, Matchmaker
  - Signatur für Prozeduren
- Software im Klienten-Rechner
  - "Stub" als Rumpfprozedur und lokaler Platzhalter



  - Verpacken der Daten für den Versand

- Software in der Server-Maschine
  - Dispatcher: Identifikation der gewünschten Prozedur



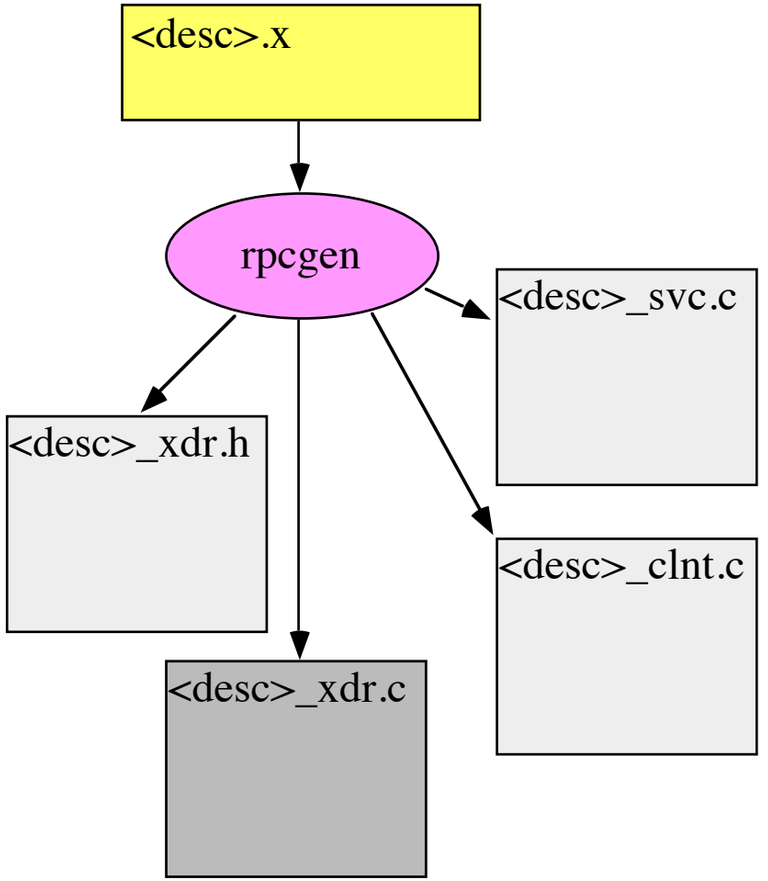
  - lokale Datendarstellung der Parameter (auspacken)

- Rückgabe der Resultatparameter
  - netzkonforme Darstellung herstellen (einpacken)
  - an Transportsystem übergeben
- Zuverlässigkeit

retransmit request	duplicate filtering	Reaktion	Semantik
Nein	-	-	maybe
Ja	Nein	re-execute	at-least-once
Ja	Ja	re-transmit	at-most-once

- RPC/XDR-Beispiel
- Schnittstellenbeschreibung: add.x

```
struct i_result {
    int x
    };

struct i_param {
    int i1;
    int i2;
    };

program ADD_PROG {
    version ADD_VERS {
       i_result ADDINT(i_param) = 1;
       } = 1;
    } = 21111111;
```



• Generiertes Headerfile: add.h

- in Server und Klient-Programm einbinden

```
typedef struct {
    int x
    } i_result;

bool_t xdr_i_result ();

typedef struct {
    int i1;
    int i2;
    } i_param;

bool_t xdr_i_param ();

#define ADD_PROG ((u_long) 21111111)
#define ADD_VERS ((u_long) 1)
#define ADDINT   ((u_long) 1)

extern i_result *addint_1();
```

• XDR-Konvertierungsroutinen: add_xdr.c (3)

```
#include <rpc/rpc.h>
#include "add.h"

bool_t xdr_i_result (xdrs, i_resultp)
    XDR *xdrs;
    i_result *i_resultp;
{
    if (!xdr_int (xdrs, &i_resultp)) return (FALSE);
    return (TRUE);
}

bool_t xdr_i_param (xdrs, i_paramp)
    XDR *xdrs;
    i_param *i_paramp;
{
    if (!xdr_int (xdrs, &i_paramp->i1)) return (FALSE);
    if (!xdr_int (xdrs, &i_paramp->i2)) return (FALSE);
    return (TRUE);
}
```

- Client-Stub. add_clnt.c (2)

```
/* Please do not edit this file.
 * It was generated using rpcgen */

#include <rpc/rpc.h>
#include <sys/time.h>
#include "add.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = {25, 0};

i_result *addint_1(i_param *argp, CLIENT *clnt) {
   static i_result res;

   bzero ((char *)&res, sizeof(res));
   if (clnt_call (clnt,
                  ADDINT,
                  xdr_i_param, argp,
                  xdr_i_result, &res,
                  TIMEOUT)   != RPC_SUCCESS) {
       return (NULL);
       }
   return (&res);
   }
```

• Server-Schleife: add_svc.c (4)

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "add.h"
static void add_prog_1();

main() {
    SVCXPRT *transp;

    (void) pmap_unset (ADD_PROG, ADD_VERS);
    transp = svcudp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
       (void) fprintf (stderr, "cannot create udp service.\n");
       exit (1);
       }
    if (!svc_register (transp, ADD_PROG, ADD_VERS,
         add_prog_1, IPROTO_UDP)) {
       (void) fprintf (stderr,"unable to register.\n");
       exit (1);
       }
    svc_run();
    exit(1); /* not reached */
    }
```

```
void add_prog_1 (struct svc_req *rqstp, SVCXPRT *transp) {
   union {
      i_param addint_1_arg;
   } argument;
   char *result;
   bool_t (*xdr_argument)(), (*xdr_result)();
   char *(*local)();

   switch (rqstp->rq_proc) {
      ...
      case ADDINT:
         xdr_argument = xdr_i_param;
         xdr_result = xdr_i_result;
         local = (char *(*)()) addint_1;
         break;
      ...
      }
   bzero ((char *)&argument, sizeof(argument));
   svc_getargs (transp, xdr_argument, &argument);
   result = (*local)(&argument, rqstp);
   svc_sendreply (transp, xdr_result, result);
   }
```

(5)

• Server-Prozedur (selbst zu programmieren) (6)

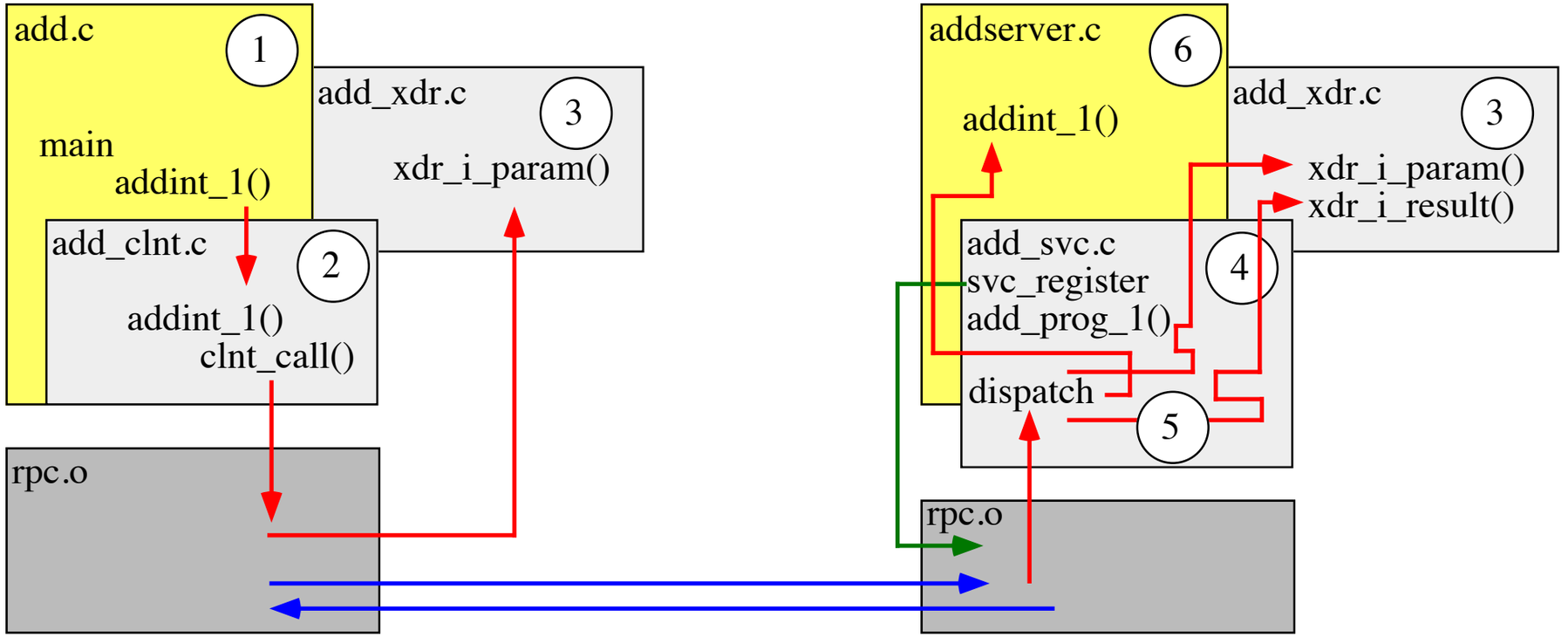
```
#include <rpc/rpc.h>
#include "add.h"
i_result *addint_1(i_param *p) {
    static i_result result;
    result.x = p->i1 + p->i2;
    return (&result);
}
```

• Client-Hauptprogramm (1)

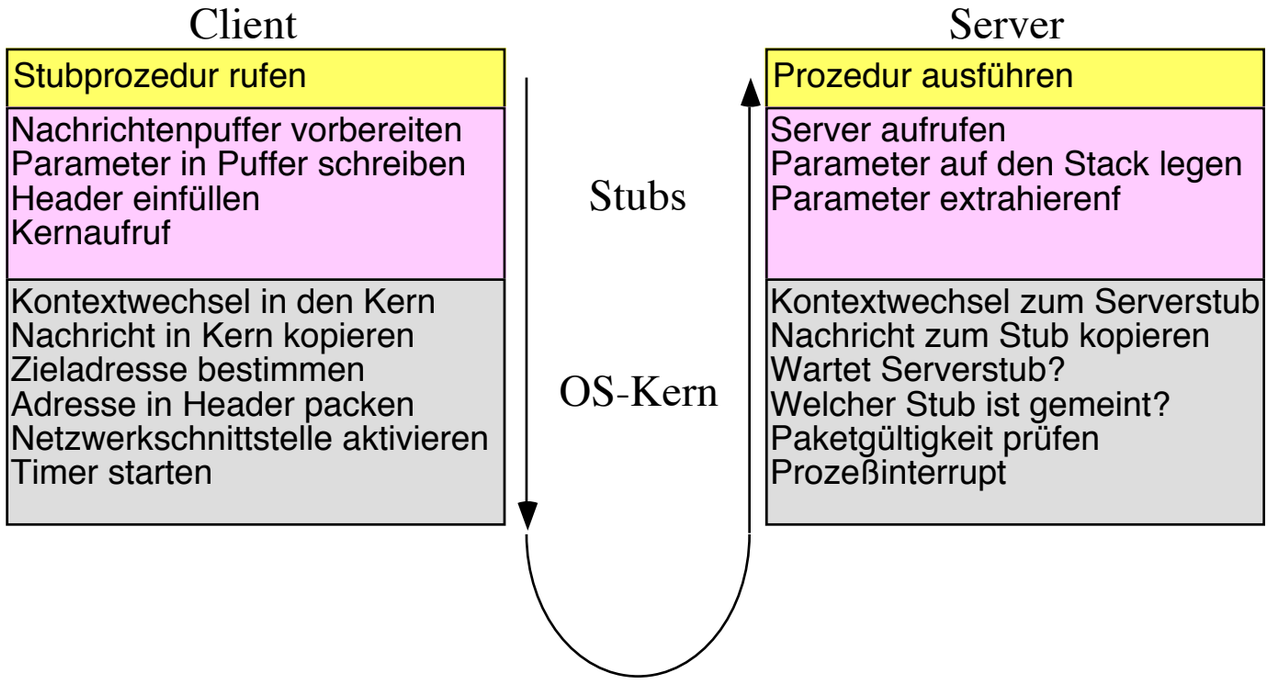
```
main (argc, argv)
    int argc; char *argv[];
{   CLIENT *cl;
    char *server;
    i_param parameter;
    i_result *ergebnis;

    server = argv[1]; /* Serveradressierung durch 1. Parameter */
    cl = clnt_create (server, ADD_PROG, ADD_VERS, “udp”);
        /* Fehlerbehandlung? */

    parameter.i1 = 12; parameter.i2 = 30;
    ergebnis = addint_1 (&parameter, cl); /* transparency? */
    printf (“Summe ist %d\n”, ergebnis->x);
}
```

- Ein Ausschnitt aus xdr.h

```
enum xdr_op {
  XDR_ENCODE = 0,
  XDR_DECODE = 1,
  XDR_FREE = 2
};
typedef struct XDR XDR;
struct XDR
{ enum xdr_op x_op;    /* operation; fast additional param */
  struct xdr_ops
  { bool_t (*x_getlong) (XDR *_xdrs, long *_lp);
    /* get a long from underlying stream */
    bool_t (*x_putlong) (XDR *_xdrs, _const long *_lp);
    /* put a long to " */
    bool_t (*x_getbytes) (XDR *_xdrs, caddr_t _addr, u_int _len);
    /* get some bytes from " */
    bool_t (*x_putbytes) (XDR *_xdrs, _const char *_addr, u_int _len);
  /* more functions ... */
  } *x_ops;
/* ... */
};
```


add.c
1
main
addint_1()
add_clnt.c
2
addint_1()
clnt_call()
add_xdr.c
3
xdr_i_param()
rpc.o
addserver.c
6
addint_1()
add_svc.c
svc_register
add_prog_1()
4
dispatch
5
add_xdr.c
3
xdr_i_param()
xdr_i_result()
rpc.o

- Aufwand
  - Marshalling
  - Paketisierung
  - Protokoll
  - Dispatch
  - Prozeßwechsel



- Anzahl Datenkopien kritisch

- Sun XDR (eXternal Data Representation, [RFC 1832])
  - für normale Datentypen
  - keine Typinformation in der Nachricht
  - ASCII, Integer, U…, …
  - Arrays und Strings mit Längenfeld
- CORBA
  - CDR: Common Data Representation
  - Interface Definition Language: Code generieren
- Java Object Serialization
- Mach: Matchmaker mit type-tags
- Xerox Courier

```
Person : TYPE = RECORD [
    name, wohnort : SEQUENCE OF CHAR;
    kontostand : CARDINAL
]
```

- Abstract Syntaxt Notation ASN.1 [CCITT, 1988]
  - Definition von Datentypen in der Nachricht
  - implizite Typen oder type-tags
  - siehe Vorlesung Kommunikationsdienste

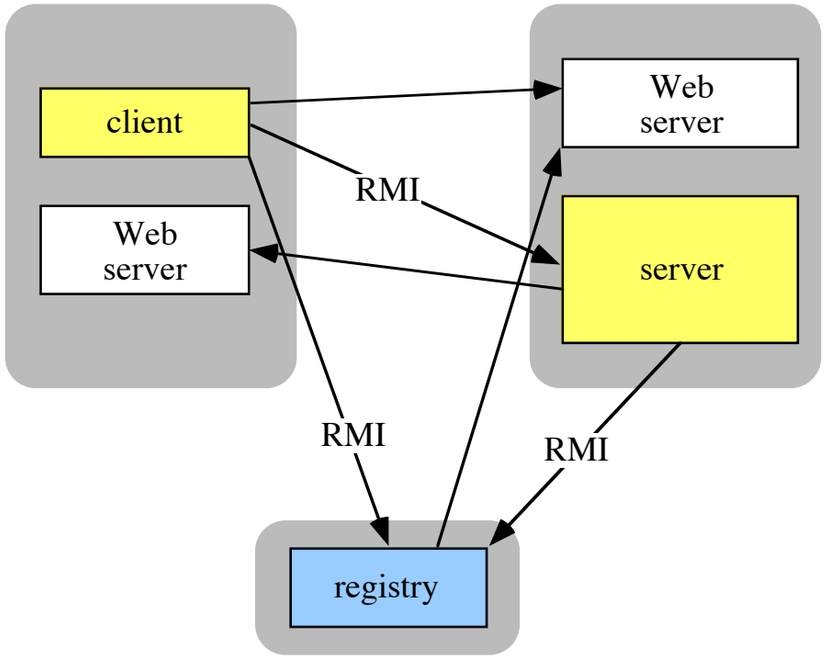
- Marshalling
  - Vorgang des Ein- und Auspackens
  - in der verteilten Anwendung

```
char *name = "Meier", *wohnort = "Freiberg";
char kontostand = -5123456;
sprintf(message,"%d %s %d %s %d",
    strlen(name),name,strlen(wohnort),wohnort,kontostand);
```

  -> Ausgabe: 5 Meier 8 Freiberg -5123456

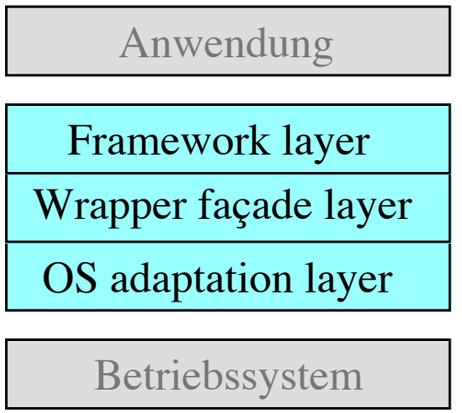
- Automatische Generierung des Interfacecodes
  - Beschreibungssprache
  - Präpozessor
- Aufwand oft beträchtlich

- Sonderfall objektorientiertes Programmieren
  - Selektor an entferntes Objekt schicken
  - remote method invocation: RMI
  - Serializable Java object
  - keine IDL
- Distributed Object Application
  - RMIRegistry
  - WWW-Transfer des Bytecodes
- Remote Interface
  - macht Objekte benutzbar
  - **`java.rmi.Remote`** erweitern
  - Stub wird übergeben statt Objekt

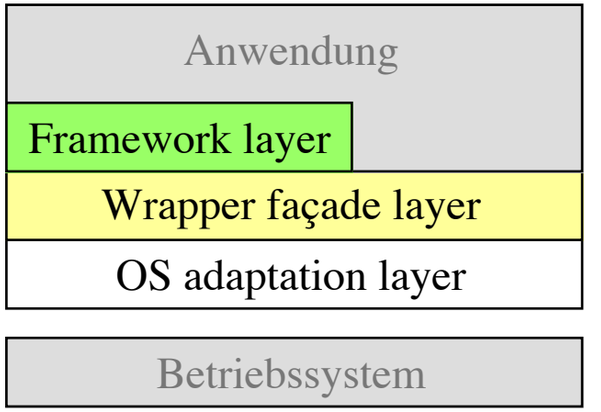


2.2.2 ACE

- ACE: Adaptive Communication Environment [Schmidt]
  - Netzwerkprogrammier-Library
  - Plattformabstraktion, Portabilität
  - 'Wrapper'
  - Framework-Metapher
  - Patterns
- Unterstützung für nebenläufiges Programmieren
  - Threads und Thread-Management
  - Reactor, Proactor
  - Pattern zur Synchronisation
- Wrapper für Socket API
  - Socket API in fast allen OS zentrales Netzwerk-API
  - betriebssytemspezifische Varianten
  - viele Konzepte in einem API
  - objektorientierter ACE-Wrapper für Socket API



- Weitere Wrapper 'Façades'
  - Event Demultiplex
  - Process, Threading
  - Synchronization: Gurad, Mutex, Locks, Semaphor
- Events: Anreize für Programm
  - Bsp: Paket, Mouse-Click, Request
  - klassische Programme haben 'Eventloop'
  - Eventloop verteilt Events an Service-Funktionen
  - Eventloop auch ein Pattern
- Framework (nach Schmidt)
  - erhält Events
  - sucht und ruft Handler (dispatch)
  - Programm wird Handler-Menge



- Design-Patterns
  - Entwurfsmuster, Standard-Vorgehensweise
  - Schmidt et al. haben Patterns erarbeitet, in ACE implementiert
- Reactor
  - register_handler(handler: ACE_Event_Handler *, mask: …):int
  - Programm übergibt Kontrolle: handle_events(): synchron
- Proactor
  - asynchroner I/O
  - ACE hilft beim dispatch des Resultats
- Acceptor/Connector
  - acceptor: passiver Verbindungsaufbau
  - connector: aktiver Verbindungsaufbau
  - synchron und asynchron benutzbar
  - Timer möglich
- Task-Framework
  - half-sync/half-async
  - 'active object'