

## **2.5 Vektorrechner & Multimedia-Erweiterungen**

Peter Marwedel  
Informatik 12  
TU Dortmund

2012/04/16

---

# Grundlegende Idee

---

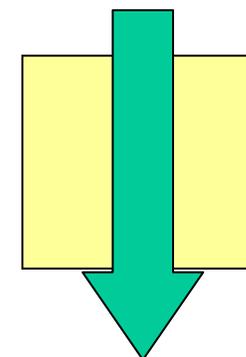
- 1 Befehl/Vektoroperation,  $\neg$  1 Befehl/skalarer Operation
- Ausnutzung der Kenntnis des Kontextes für eine „schlauere“ Ausführung
- Holen von (Abschnitten von) Vektoren
- Besseres Verstecken von Speicherlatenz
- Bessere Nutzung der Speicherbandbreite
- 1 Befehl auf viele Daten anwenden  
☞ *single instruction/multiple data (SIMD)*

Dieser Foliensatz nutzt Material aus Hennessy/ Patterson: Computer Architecture – A Quantative Approach, 5. Auflage, 2011, Kap. 4

# Rechnerklassifikation nach Flynn

**Klassifikation  
von  
Multiprozessor-  
systemen**

		Befehlsströme	
		1	>1
Daten- ströme	1	SISD	MISD
	>1	SIMD	MIMD



SISD	Bislang besprochene Einzelrechner
MIMD	Netze aus Einzelrechnern; sehr flexibel
SIMD	Ein Befehlsstrom für unterschiedliche Daten; identische Befehle bilden starke Einschränkung
MISD	Mehrere Befehle für ein Datum: Kann als Fließband von Befehlen auf demselben Datenstrom ausgelegt werden. Ist etwas künstlich.

Klassifikation hat nur begrenzte Aussagekraft; keine bessere vorhanden.

---

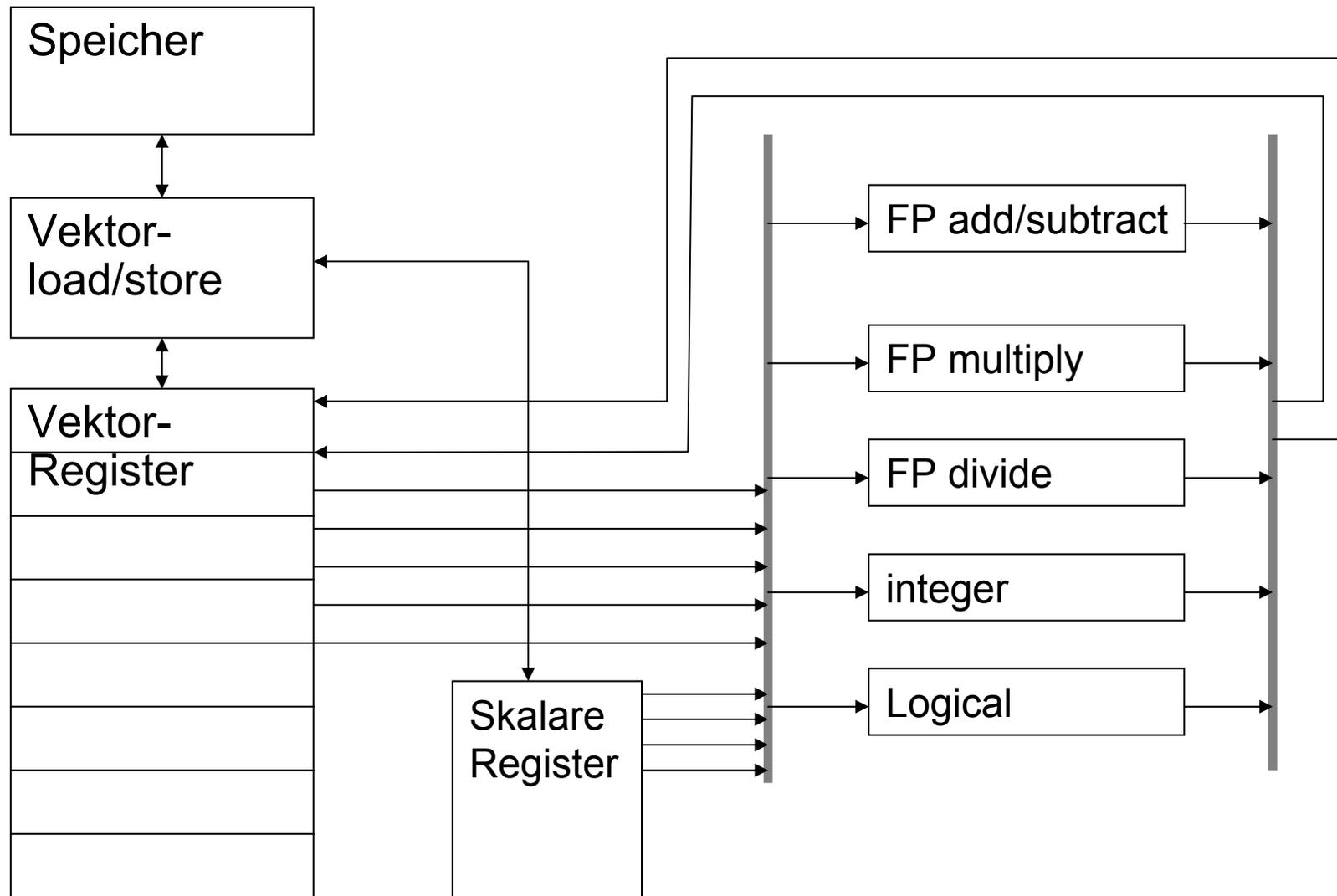
# Beispiel: VMIPS ( $\approx$ Cray-1)

---

Komponenten:

- Vektor-Register:
  - 64 Elemente/Register, 64 Bit/Element
  - Register-File hat 16 Lese- und 8 Schreibports
- Vektor-Funktionseinheiten
  - Fließbandverarbeitung
  - Daten- und Kontrol hazards werden erkannt
- Vektor-Load/Store-Einheiten
  - Fließbandverarbeitung
  - 1 Wort/Taktzyklus nach initialer Latenzzeit
- Skalare Register
  - 32 Allgemeine Register
  - 32 Gleitkomma-Register

# VMIPS-Struktur



# VMIPS-Befehle

## (Auszug; nur *double precision*-Befehle)

Befehl	Operanden	Funktion
ADDVV.D	V1,V2,V3	$\underline{V1} := \underline{V2} + \underline{V3}$ (+ Vektor)
ADDVS.D	V1,V2,F0	$\underline{V1} := \underline{V2} + F0$ (+ Skalar)
SUB/MUL/DIV	Dto.	Dto.
LV	V1,R1	$\underline{V1} := \text{Mem}[R1\dots]$ (ab Adresse in R1)
SV	R1,V1	$\text{Mem}[R1\dots] := \underline{V1}$
LVWS	V1,(R1,R2)	$\underline{V1} := \text{Mem}[R1+i \times R2]$ (mit <i>stride</i> in R2)
LVI	V1,(R1+V2)	$\underline{V1} := \text{Mem}[R1+\underline{V2}[i]]$ ( $\underline{V2}$ : index)
S <sub>x</sub> VV.D	V1,V2	$x \in \{EQ, NE, GT, \dots\}$ ; erzeuge Maske
MTC1	VLR,R1	VL:=R1      Vektorlängenregister
MTVM	VM,F0	VM:=F0      Vektormaskenregister

# MIPS- vs. VMIPS-Code für DAXPY

$\underline{Y} = a \times \underline{X} + \underline{Y};$   
Annahme:  
64 Elemente

MIPS:  
fast 600  
Befehle

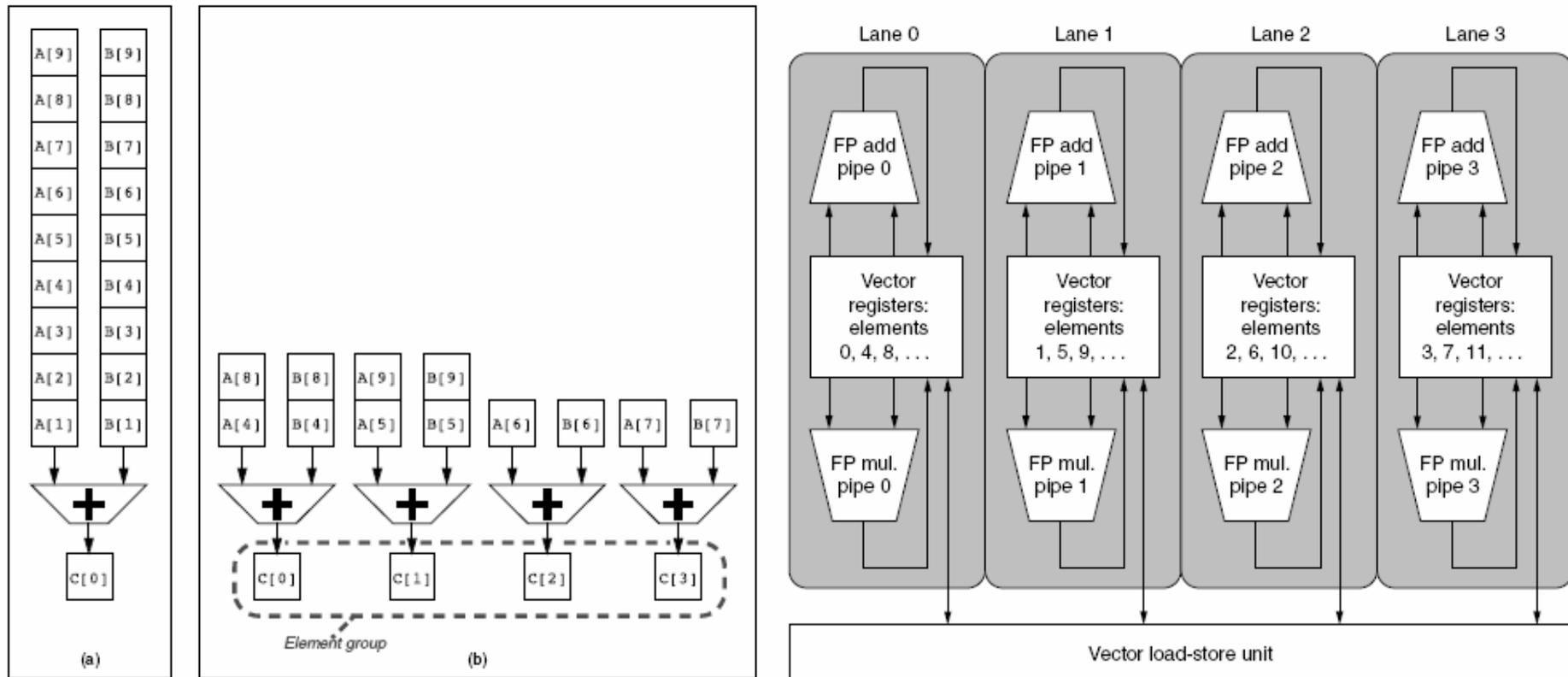
VMIPS:  
6 Befehle

	L.D	F0,a	;lade Skalar
	DADDIU	R4,Rx,#512	;letzte zu ladende Adresse
Loop:	L.D	F2,0(Rx)	;lade X[i]
	MUL.D	F2,F2,F0	;a x X[i]
	L.D	F4,0(Ry)	;lade Y[i]
	ADD.D	F4,F4,F2	;a x X[i] + Y[i]
	S.D	F4,9(Ry)	;speichern in Y[i]
	DADDIU	Rx,Rx,#8	;Inkrementiere Index für X
	DADDIU	Ry,Ry,#8	;Inkrementiere Index für Y
	DSUBU	R20,R4,Rx	;bestimme Grenze
	BNEZ	R20,Loop	;Fertig?

	L.D	F0,a	;lade Skalar
	LV	V1,Rx	;lade Vektor
	MULVS.D	V2,V1,F0	;Multiplikation Vektor/Skalar
	LV	V3,Ry	;lade Y
	ADDVV.D	V4,V2,V3	;addiere
	SV	V4,Ry	;speichere Ergebnis

# Mehrere *lanes*

Element  $n$  des Vektorregisters  $A$  ist fest verdrahtet mit Element  $n$  des Vektorregisters  $B$



# Vektor-Längen-Register + *Strip-mining*

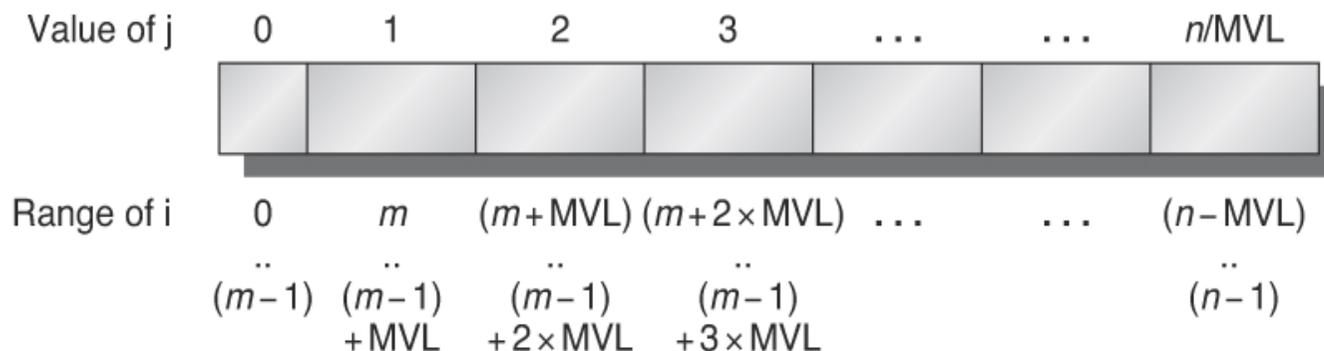
Vektorlänge zur Compilezeit unbekannt ☞ *Vector Length Register (VLR)*

Benutzung von *strip mining* für Vektorenängen > max. Länge:

```

low = 0;
VL = (n % MVL);           /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
    for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
        Y[i] = a * X[i] + Y[i]; /*main operation*/
    low = low + VL;         /*start of next vector*/
    VL = MVL;              /*reset the length to maximum vector length*/
}

```



Alle bis auf den ersten Block haben die Länge MVL und nutzen die ganze Performanz des Prozessors.  
 $m=(n \% MVL)$ .

# Vector Mask Registers

Betrachte:

```
for (i = 0; i < 64; i=i+1)
    if (X[i] != 0)
        X[i] = X[i] - Y[i];
```

Benutze Vektor-Maskenregister, um Elemente auszublenden:

LV	V1,Rx	;load vector X into V1
LV	V2,Ry	;load vector Y
L.D	F0,#0	;load FP zero into F0
SNEVS.D	V1,F0	;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D	V1,V1,V2	;subtract under vector mask
SV	Rx,V1	;store the result in X

GFLOPs-Rate nimmt ab.

---

# Speicherbänke

---

Das Speichersystem muss so entworfen sein, dass Vektor-Lade- und Speicheroperationen effizient unterstützt werden.

Mechanismen dafür:

- Speicherzugriffe auf mehrere Bänke aufteilen
- Unabhängige Kontrolle der Bankadressen
- Laden und speichern von nicht konsekutiven Worten
- Unterstützung mehrerer Vektorprozessoren, die auf gemeinsamen Speicher zugreifen können.

---

# Strides (Schrittlängen)

---

Notwendig, Schrittlängen beim Zugriff auf Speicherzellen zu betrachten:

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

- Je nach Array-Layout (*row major* bzw. *column major order*) erfolgen Zugriffe in innerster Schleife auf A oder B im Abstand von 100 Array-Elementen bzw. 800 Byte bei *double*
- Unterstützung von *strides* in Vektorrechnern ist sinnvoll
- Es kann einen Konflikt beim Zugriff auf eine Speicherbank geben

# Gather-Scatter-Lesen/Schreiben für dünn besetzte Matrizen

Dünn besetzte Matrizen: Elemente evtl. per Index adressiert:

```
for (i = 0; i < n; i=i+1)
```

```
    A[K[i]] = A[K[i]] + C[M[i]];
```

Auf Assemblerebene:

LV	Vk, Rk	;lade K
LVI	Va, (Ra+Vk)	;lade A[K[]], "gather"
LV	Vm, Rm	;lade M
LVI	Vc, (Rc+Vm)	;lade C[M[]]
ADDVV.D	Va, Va, Vc	;addiere
SVI	(Ra+Vk), Va	;speichere A[K[]], "scatter"

- Erfordert evtl. Benutzerannotation für Parallelisierung
- Kann in der Architektur unterstützt werden

# Vektorisierbare Anteile

<i>Benchmark</i>	Ops im Vektor-Modus, Compiler-optimiert	Ops im Vektor-Modus, mit Cray-Experten-Hinweisen	<i>Speedup</i> m. Hinweisen
BDNA	96,1%	97,2%	1,52
MG3D	95,1%	94,5%	~1,0
FLO52	91,5%	88,7%	-
ARC3D	91,1%	92,9%	1,01
SPEC77	90,3%	90,4%	1,07
MDG	87,7%	94,2%	1,49
TRFD	68,8%	73,3%	1,67
DYFESM	68,8%	65,6%	-
ADM	42,9%	59,6%	3,60
OCEAN	42,8%	91,2%	3,92
TRACK	14,4%	54,6%	2,52
SPICE	11,5%	79,9%	4,06
QCD	4,2%	75,1%	2,15

- ☞ Große Variation bei Compiler-Optimierungen
- ☞ Programmierer-Hinweise sind sinnvoll

---

# Roofline-Performance-Modell

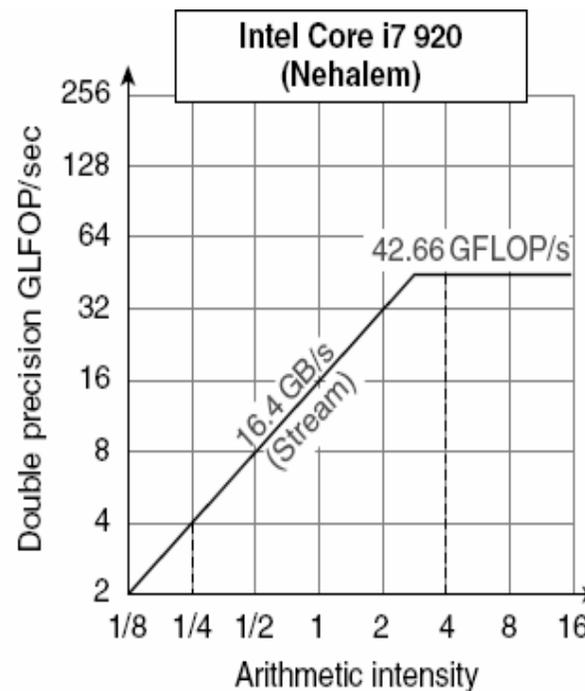
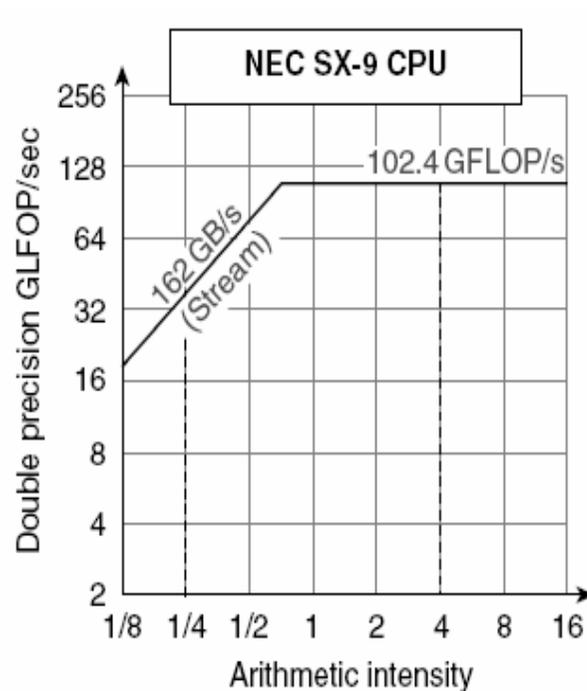
---

Idee:

- Zeichne Spitzen-Gleitkomma-Performanz als Funktion der Arithmetik-Intensität
- Verbindet Gleitkomma- und Speicherperformanz einer Zielmaschine
- Arithmetikintensität =  
# Gleitkomma-Operationen pro gelesenem Byte

# Roofline-Performance-Modell (2): Beispiele

Erreichbare GFLOPs/sec =  
Min (Peak Memory BW × Arithmetic Intensity,  
Peak Floating Point Performance)



- *Unit-stride memory accesses, double-precision floating*
- NEC SX-9: vector supercomputer in 2008 angekündigt,  $x$  M\$
- Stream benchmark
- Gestrichelte vertikale Linien: SX-9 mit 102.4 FLOP/s ist 2.4x schneller als Core i7 mit 42.66 GFLOP/s.

---

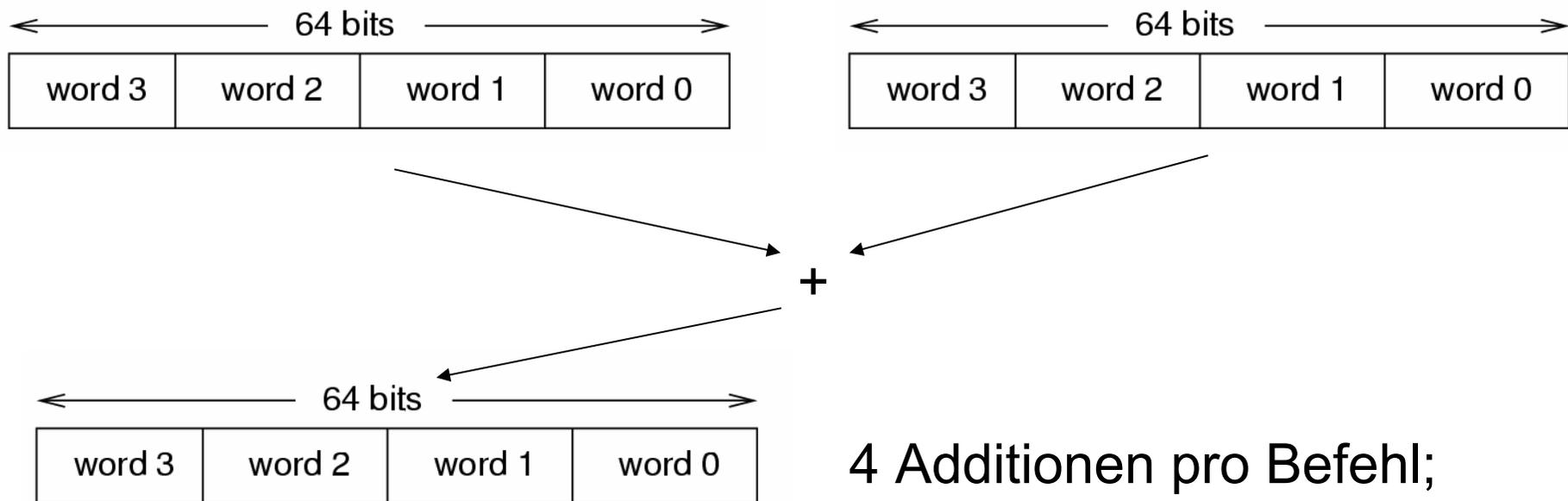
# Multimedia-/SIMD-Befehle

---

- Viele Multimedia-Datentypen benötigen eine geringe Bitbreite (8 Bit bei R/G/B, 16 Bit bei Audio),
  - wohingegen viele Rechner eine große ALU/Registerbreite besitzen (32/64/128 Bit).
  - Dabei gibt es für die Verarbeitung von Multimediadaten hohe Leistungsanforderungen & Realzeitbedingungen.
- ☞ Idee, mehrere Daten mit einem Befehl zu verarbeiten.

# Beispiel

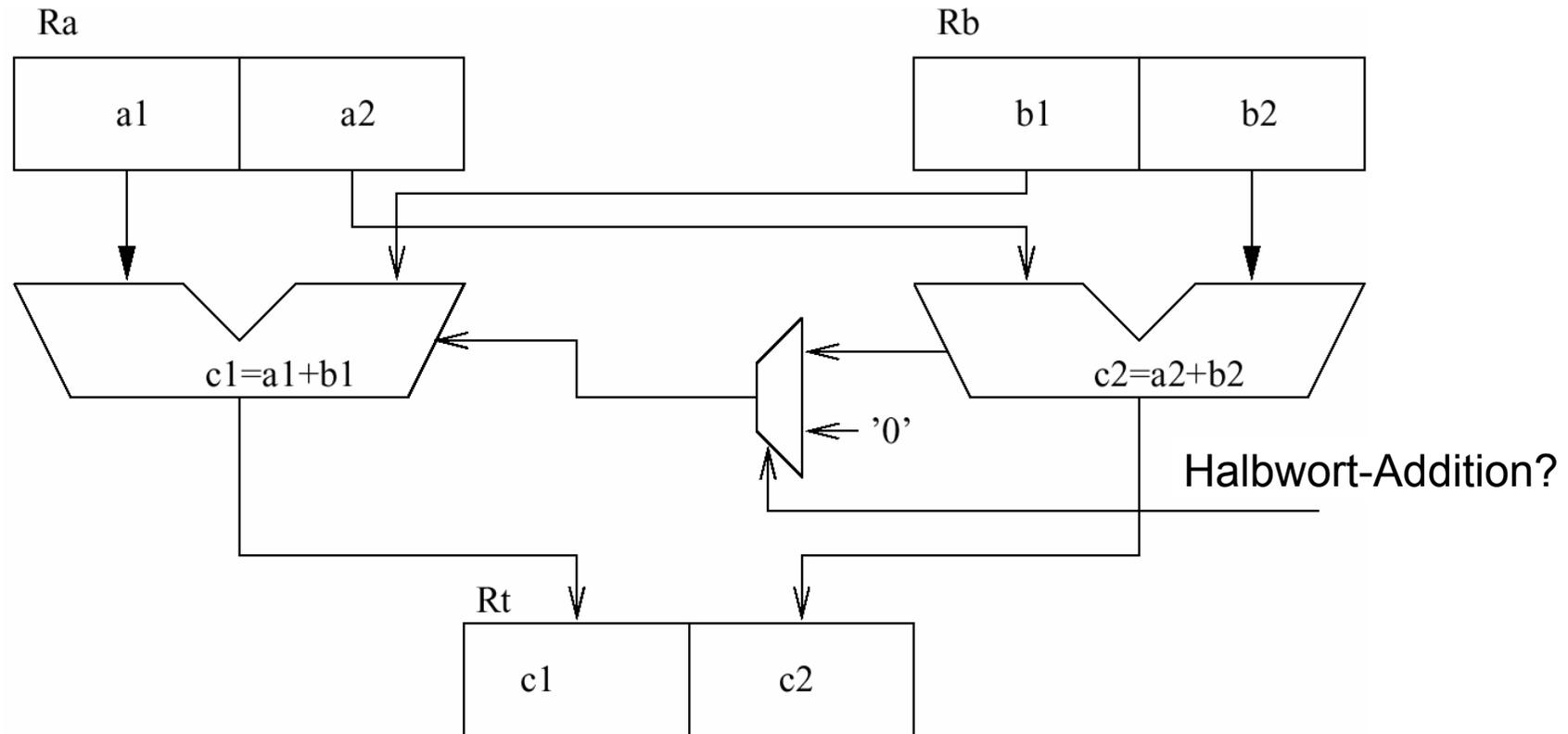
- Speicherung und Verarbeitung von 2-8 Werten in einem langen 64-Bit-Wort:



4 Additionen pro Befehl;  
Überträge an Wortgrenzen  
unterdrückt.

# Frühes Beispiel: HP *precision architecture* (hp PA)

„Halbwort“-Addition **HADD**:



Optionale Sättigungsarithmetik;  
**HADD** ersetzt bis zu 10 Befehle.

# Pentium MMX-Architektur (1)

64-Bit-Vektoren entsprechen 8 Byte-kodierten, 4 Wort-kodierten oder 2 Doppelwort-kodierten Zahlen.

Hier: 1 Wort = 16 Bit; *wrap around/saturating* Option.

Multimedia-Register mm0 - mm7, konsistent mit Gleitkomma-Registern (BS ungeändert).

Befehl	Optionen	Kommentar
Padd[b/w/d] PSub[b/w/d]	<i>wrap around,</i> <i>saturating</i>	Addition/Subtraktion von Bytes, Worten, Doppelworten
Pcmpeq[b/w/d] Pcmpgt[b/w/d]		Ergebnis= "11..11" wenn wahr, "00..00" sonst Ergebnis= "11..11" wenn wahr, "00..00" sonst
Pmullw Pmulhw		Multiplikation, 4*16 Bits, weniger signifikantes Wort Multiplikation, 4*16 Bits, signifikantestes Wort

## Pentium MMX-Architektur (2)

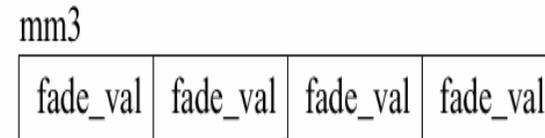
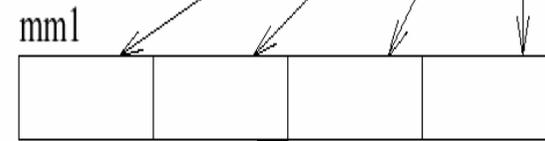
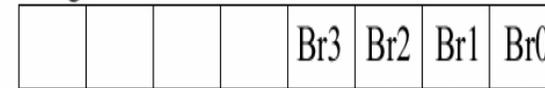
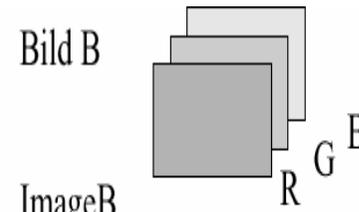
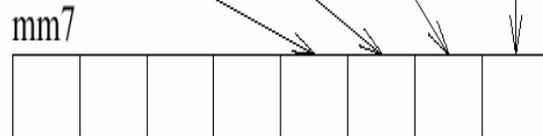
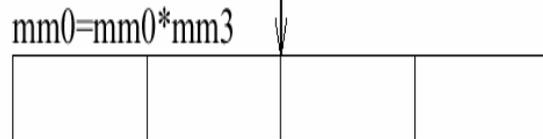
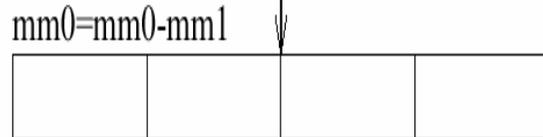
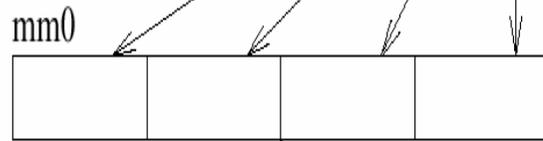
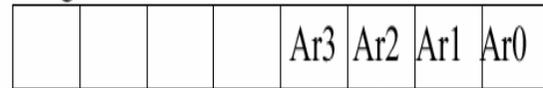
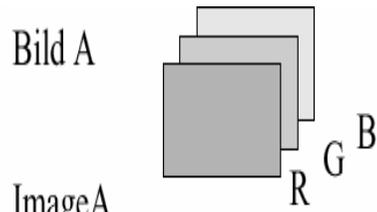
Psra[w/d] Psll[w/d/q] Psrl[w/d/q]	Anzahl der Stellen	Paralleles Schieben von Worten, Doppelworten oder 64 Bit-Quadworten
Punpckl[bw/wd/dq] Punpckh[bw/wd/dq]		<i>Parallel unpack</i> <i>Parallel unpack</i>
Packss[wb/dw]	<i>saturating</i>	<i>Parallel pack</i>
Pand, Pandn Por, Pxor		Logische Operationen auf 64 Bit-Werten
Mov[d/q]		<i>Move</i>

# Appli- kation

Skalierte  
Interpolation  
zwischen  
zwei Bildern

Nächstes  
Byte =  
nächstes  
Pixel,  
dieselbe  
Farbe.

Verarbeitung  
von 4 Pixeln  
gleichzeitig.



```

pxor    mm7,mm7    ;clear register mm7
movq    mm3,fade_val;load scaling value
movd    mm0,imageA ;load 4 red pixels for A
movd    mm1,imageB ;load 4 red pixels for B
unpcklbw mm1,mm7   ;unpack,bytes to words
unpcklbw mm0,mm7   ;upper bytes from mm7
psubw   mm0,mm1    ;subtract pixel values
pmulhw  mm0,mm3    ;scale
paddw   mm0,mm1    ;add to image B
packuswb mm0,mm7   ;pack, words to bytes
    
```

---

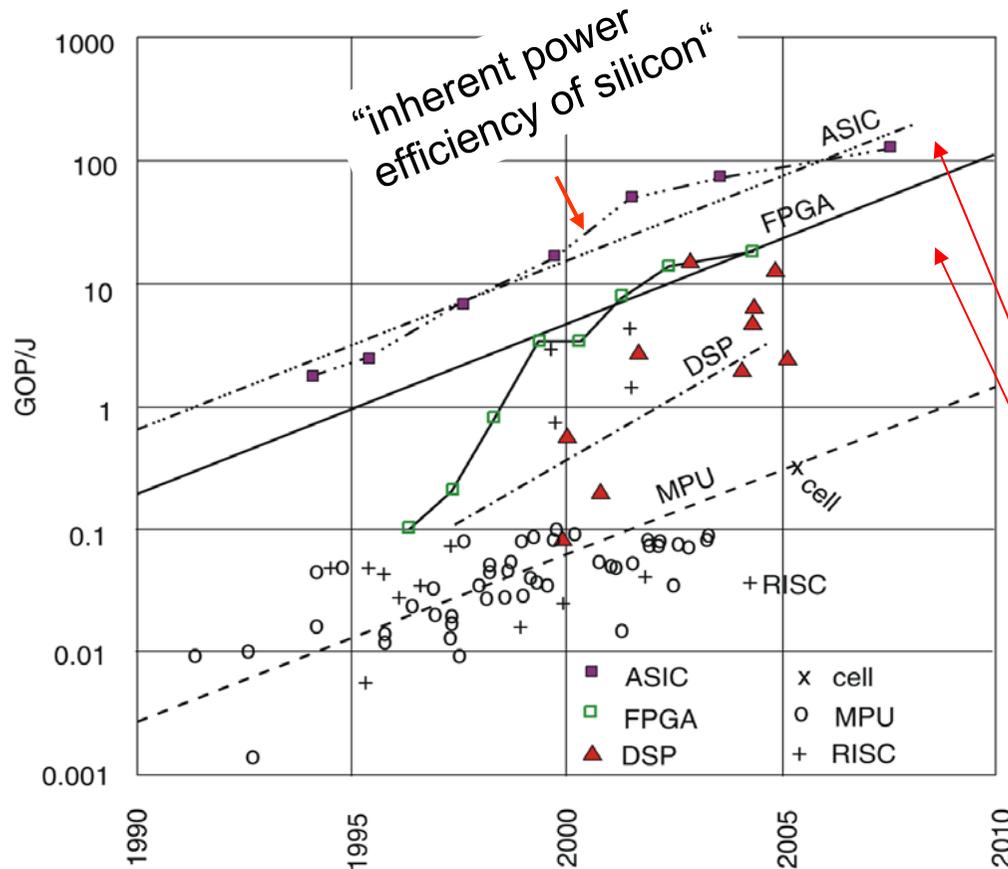
# Bewertung

---

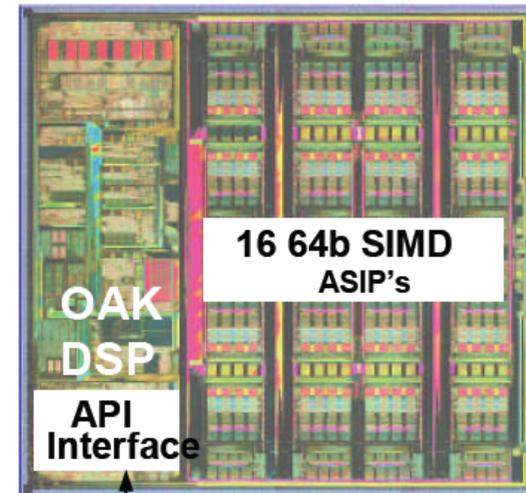
## SIMD

- kann Datenparallelität nutzen, v.a.
  - für wissenschaftliche Berechnungen und
  - für Audio- und Videoverarbeitung,
- ist in der Regel energieeffizienter als MIMD:
  - nur 1x Befehlsholen
  - daher v.a. für mobile Anwendungen geeignet
- kodiert Vektorlänge im Befehl,
- erlaubt, weiterhin sequentiell zu denken,
- erfordert passende Ausrichtung der Speicheroperanden,
- erfordert Compiler, der das Parallelisierungspotential nutzt

# Existenznachweis der Energieeffizienz



## VIP for car mirrors Infineon



**200MHz , 0.76 Watt**  
**100Gops @ 8b**  
**25Gops @ 32b**

Close to power efficiency of silicon

© Hugo De Man: From the Heaven of Software to the Hell of Nanoscale Physics: An Industry in Transition, *Keynote Slides*, ACACES, 2007

## Short vector extensions

Hersteller	Name	Genauigkeit	Prozessor
AMD	3DNow!	Einfach	K6, K6-II, Athlon
Intel	SSE	Einfach	Pentium III/4
Intel	SSE2	Doppelt	Pentium 4
Motorola	AltiVec	einfach	G4
Sun	VIS		Sparc
...	...	...	...

---

# Short vector extensions (1)

---

- MMX: Begrenzt auf integer, Problem der Konsistenzerhaltung mit Gleitkommaregistern
- 3DNow!: 1998 von AMD eingeführt
- *Streaming SIMD Extensions (SSE)*:
  - 1999 von Intel eingeführt
  - 8 neue 128-Bit-Register („XMM 0-7“ )
  - 8 neue 64-Bit-Register („XMM 8-15“ vom AMD64, ab 2004)
  - Unterstützung von Gleitkomma-Datentypen
  - 70 neue Befehle:  
Beispiel: 4 32-Bit-Gleitkomma-Additionen in einem Befehl kodiert
  - Berücksichtigung beim Kontextwechsel
  - Macht MMX überflüssig

---

## Short vector extensions (2)

---

- *Streaming SIMD Extensions 2 (SSE2)*:
  - 2001 von Intel eingeführt
  - 2003 von AMD für Opteron und Athlon übernommen
  - 144 neue Befehle
  - MMX-Befehle können jetzt auf den neuen XMM-Registern arbeiten, MMX wird komplett überflüssig, integer-SIMD und Gleitkomma-Befehle können gleichzeitig bearbeitet werden (geht bei MMX nicht)
  - Cache-Kontrollbefehle
  - Format-Konvertierungsbefehle
  - SSE2-Gleitkomma-Befehle verarbeiten max. 64-Bit-Gleitkommazahlen, skalare Befehle erlauben 80 Bit (!)
  - Schnell nur bei *alignment* auf 16-Byte-Grenzen

---

## Short vector extensions (3)

---

- *Streaming SIMD Extensions 3 (SSE3):*
  - 2004 von Intel eingeführt
  - Von AMD übernommen
  - 13 neue Befehle
  - Addition und Subtraktion von Werten innerhalb eines Registers („Horizontale“ bzw. Reduktions-Operationen)
  - Gleitkomma-Wandlung ohne globale Modifikation des Rundungsmodus
  - *Load*-Befehl für nicht ausgerichtete Daten
  - 2 Befehle für *multi-threading*

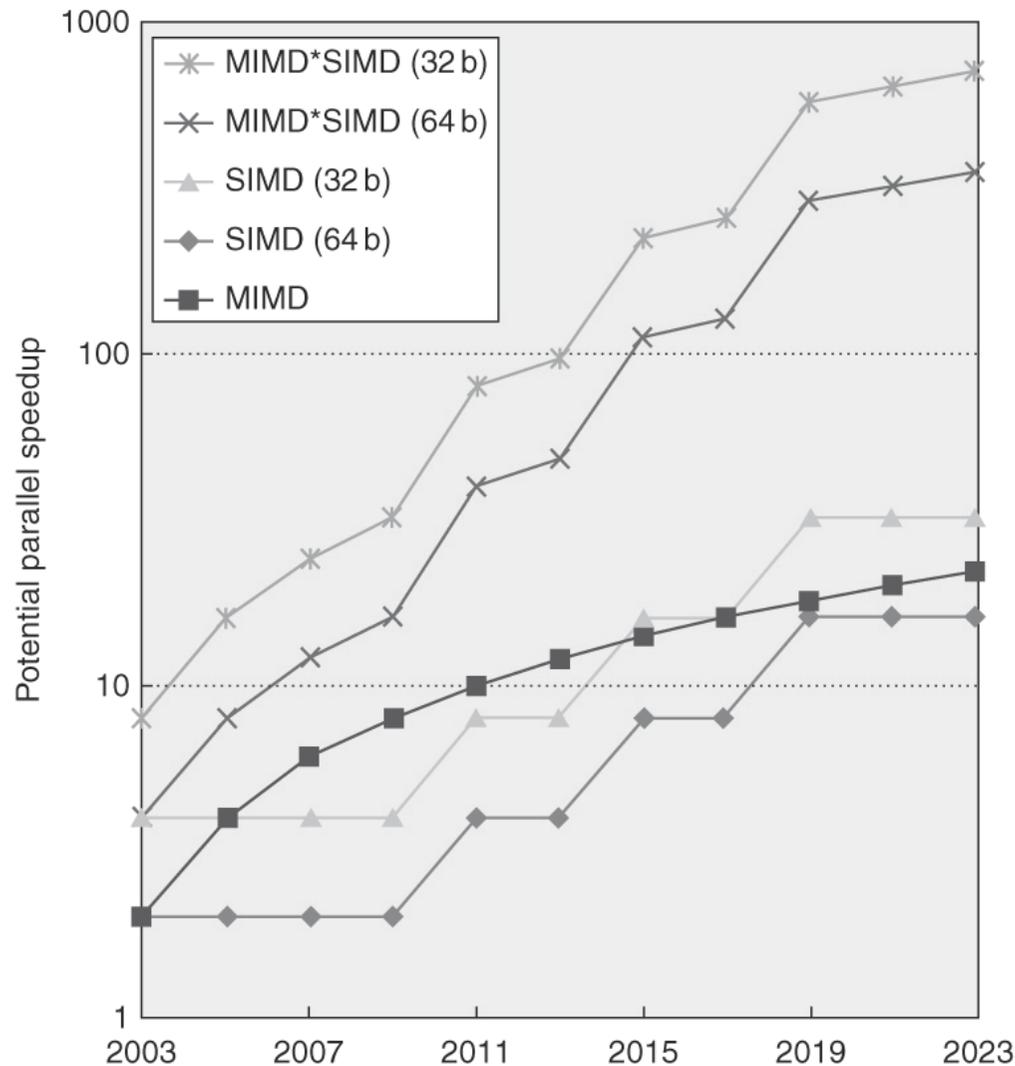
---

# Short vector extensions (4)

---

- *Streaming SIMD Extensions 4 (SSE4), HD Boost:*
  - 2006 von Intel eingeführt
  - 54 neue Befehle, 4 von AMD übernommen
  - CRC32-Befehl, Stringvergleich, Zählen von Einsen, ..
- *Streaming SIMD Extensions 5 (SSE5)*
  - 2007 von AMD vorgeschlagen, u.a. Befehle mit 3 Operanden
  - Ursprüngliche Version zugunsten AVX Kompatibilität geändert
- *Advanced Vector Extensions (AVX)*
  - 2008 von Intel vorgeschlagen, von AMD modifiziert übernommen
  - XMM-Register ➡ **256 Bit**; 512 und 1024 Bit evtl. später
  - 3-Operanden-Befehle
  - Inkompatibel mit SSE5
  - Erfordert BS-Support (Linux 2.6.30, Windows 7 SP1)

# Potentieller *Speedup* durch Parallelität bei MIMD, SIMD, und MIMD&SIMD für x86 Rechner



Zeichnung nimmt an, dass

- bei MIMD pro Jahr 2 Cores hinzugefügt werden und
- die Zahl der Operationen bei MIMD sich alle 4 Jahre verdoppelt,

---

# Zusammenfassung

---

## SIMD-Prinzip

- Vektorrechner
  - *strides, strip-mining, mask-register, gather/scatter*
- Multimedia/SIMD/Streaming SIMD Extensions
  - MMX, SSE1-5, AVX
- *Roofline-Performance-Modell*

---

---

# Reserve

# Chimes

Sequences with read-after-write dependency hazards can be in the same convey via *chaining*

## *Chaining*

- Allows a vector operation to start as soon as the individual elements of its vector source operand become available

## *Chime*

- Unit of time to execute one convey
- $m$  conveys executes in  $m$  chimes
- For vector length of  $n$ , requires  $m \times n$  clock cycles

## Example

```

LV          V1,Rx          ;load vector X
MULVS.D    V2,V1,F0       ;vector-scalar multiply
LV          V3,Ry          ;load vector Y
ADDVV.D    V4,V2,V3       ;add two vectors
SV          Ry,V4         ;store the sum

```

Convoys:

```

1          LV          MULVS.D
2          LV          ADDVV.D
3          SV

```

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 64 element vectors, requires  $64 \times 3 = 192$  clock cycles

# Vector Execution Time

Execution time depends on three factors:

- Length of operand vectors
- Structural hazards
- Data dependencies

VMIPS functional units consume one element per clock cycle

- Execution time is approximately the vector length

*Convey*

- Set of vector instructions that could potentially execute together