Rechnerarchitektur

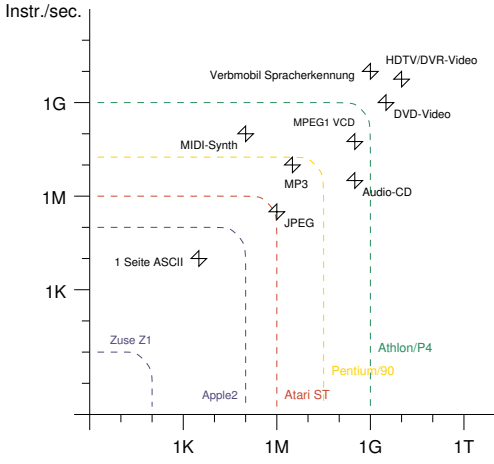
- welche Rechnerarchitektur für "Medienverarbeitung"?
- Anforderungen für Audio, Video, 2D/3D, VR, Speech, ...
- Anforderungen an Server und Netzwerke

- Bewertung und Leistungsanalyse von Rechnern
- Befehlssätze, RISC vs. CISC
- Einfluss der Speicherhierarchie

- ARM-Architektur
- ARM-Erweiterungen: Thumb, Jazelle, XScale
- Single-Chip Multiprozessoren: MAJC, Piranha
- Netzwerkprozessoren

- SIMD-Befehlssätze: MMX, SSE

Medien: Anforderungen



Kriterien:

- MIPS
- MBytes (RAM, Platte)
- Mbps
- MPixel

- viele Anwendungen "saugen" Performance auf: (z.B. 3D/Virt. Realität)

- jede Rechnergeneration erlaubt neue Anwendungen

Rechnerarchitektur: Literatur

D.L.Hennessy & D.A.Patterson, Computer architecture, a quantitative approach, MKP
A.S.Tanenbaum, Computer architecture, 4th ed.,
S.Furber, ARM system-on-chip architecture, Addison-Wesley
M.D.Hill (Ed), Readings in Computer Architecture, MKP 2000

Intel x86, MMX, SSE, SSE2 documentation, www.developer.intel.com
Compaq Piranha, Proc. ISCA 2000
Sun MAJC processor documentation and whitepapers, www.sun.com
Netzwerkprozessoren: Proc. HotChips 12, 2000, www.hotchips.org

Benchmarks: www.spec.org

Medienverarbeitung: Speicherzugriffe

- Mediendaten liegen als 1D-/2D-Arrays vor (Audio, Video, Images, ..)
- große Datenmengen / hohe Datenraten (HDTV: 100 MB/s)
- geringe Genauigkeit der einzelnen Datenworte

- geringe "Lokalität" (Samples/Pixel/Frames unabhängig)
- klassischer Cache wirkungslos
- "Streaming" mit kleinen Puffern

- aber auch "klassische" Algorithmen und Datenstrukturen
- z.B für LZW/BWT/Huffman-Datenkompression
- z.B Hashtabellen/Wörterbücher für Spracherkennung (Verbmobil: 1 GB)

- und natürlich Floating-Point für Simulationen (z.B. Objektphysik in 3D-Spielen, VR, haptische Interfaces)

=> "Medien"-Architekturen müssen alles beherrschen

Medienverarbeitung: I/O

stark unterschiedliche Anforderungen, je nach Medium:

Audio, Wiedergabe (48 KHz, 16bit, 6 Kanäle)	0.6 MB / s
Audio, Studio-Mischpult (100 Kanäle)	9.6 MB / s
Video, 640x480x24, 25fps	23 MB / s
Video, 1280x960x24, 25fps	92 MB / s
3D, Transformation, 100K Dreiecke, 50 fps, je 32-bit	60 MB / s
3D, 1024x768x24, 50fps, trilinear (8/pixel), 3x overdraw:	8490 MB / s
Datenhandschuh, 20 Mestreifen, 8 bit, 25fps	0.0005 MB / s
Force-Feedback, 30 Motoren, 25 fps	

- zum Vergleich: PCI 132 MB/ps, AGP4X 1GB/s
- mehrfache Anforderungen fr Medien-Server

Medienverarbeitung: Server

- Auslieferung statischer Dateien:
- erfordert wenig CPU-Performance
- aber hohen I/O-Durchsatz

- einzelne Client-Anfragen voneinander unabhängig
- im Prinzip vollständig parallele Bearbeitung möglich
- Multiprozessor / SMP-Systeme
- aber: Synchronisation, z.B. wegen Logfiles

- Beispiel "video on demand":

  1 Stream MPEG-2 mit 1 MB/s
  100 Streams mit je 1 MB/s: 100 MB/s

- Beispiel: Compaq Piranha 8xAlpha SOC

Medienverarbeitung: Netzwerk

- Telekommunikation: steigender Anteil Daten vs. Voice
- "packet" vs. "switched" Übertragungen

- exponentiell steigende Datenmengen
- entsprechend komplexeres Routing (erst Recht für ATM)
- neue Anforderungen, z.B. QoS
- potentiell großer Markt

=> neue Geräteklasse "Netzwerkprozessoren"

- effiziente Bearbeitung von Routing-Anfragen
- zugehörige Pufferung / Bearbeitung der Datenpakete
- erweiterte Zugriffs- und Rechtekontrolle (in Echtzeit)
- interessante, parallele Rechnerarchitekturen

(siehe HotChips12 Session)

Medienverarbeitung: Mobilgeräte

- zumehmende Verbreitung von "mobile appliances"
- z.B. Mobiltelefone, PDAs, MP3-Player, ...

- Multimedia-Anwendungen erwünscht (MP3, Video, UMTS, ...)
- erforderliche Rechenleistung wie aktuelle Desktop-Systeme
- 200+ MIPS, 64+ MByte RAM, Hintergrundspeicher
- aber möglichst geringer Stromverbrauch

- spezielle Prozessoren mit zugehörigen Hilfschips
- meistens: RISC mit DSP-Erweiterungen
- Powermanagement, low-voltage, gated-clocks
- spezielle CMOS-Technologie (derzeit 0.13 .. 0.18µm)

- Kriterium: mW / MIPS

Amdahl's Gesetz

"Speedup" durch Parallelisierung? [Gene Amdahl, 1967]

System 1: berechnet Funktion X, zeitlicher Anteil 0<F<1
System 2: Funktion X' ist schneller als X mit "speedup" SX:
SX = Zeitbedarf(X) / Zeitbedarf(X')

Amdahl's Gesetz: $S_{gesamt} = \frac{1}{(1-F) + F/SX}$

=> Optimierung lohnt nur für häufige Operationen !!
=> Beispiele:

```
SX =  10, F = 0.1,  Sgesamt = 1/(0.9 + 0.01)  = 1.09
SX =   2, F = 0.5,  Sgesamt = 1/(0.5 + 0.25)  = 1.33
SX =   2, F = 0.9,  Sgesamt = 1/(0.1 + 0.45)  = 1.82
SX = 1.1, F = 0.98, Sgesamt = 1/(0.02 + 0.89) = 1.10
```


RISC: Designphilosophie

- minimaler, regulärer Befehlssatz
- optimale VLSI-Implementierung
- Compiler erledigt den Rest

- Berücksichtigung von Amdahl's Gesetz
- umfangreiche Performance-Simulationen (Benchmarks)

ursprüngliche RISC Entwurfsentscheidungen:

+ 32-bit Prozessor, 4 GByte Adressraum
+ 32 Universalregister (ausser RISC/SPARC)
+ 32-bit Befehlsworte, wenig Formate

- Pipeline-Abhängigkeiten (delayed branches)
- Spezialregister (MIPS mult/div)

RISC: RISC-I und Mips

ca 1980: 801-Nachfolgeprojekte:

- Berkeley RISC-I "reduced instruction set computer"
- Stanford MIPS "microprocessor w/o interlocked pipeline stages"

- Compiler-gerechte Architektur
- single-Chip VLSI-Implementierung

bessere Performance als 8086/68000:

- sauberer Befehlssatz — RISC

- "hardwired" Controller statt Microcode
- Pipeline
- viele Register, weniger Speicherzugriffe
- gut optimierende Compiler
- Caches, insbesondere I-Cache

auch für CISC möglich!

SPECint: DLX

Instruction	compress	eqntott	espresso	gcc (cc1)	li	Integer average
load	19.8%	30.6%	20.9%	22.8%	31.3%	26%
store	5.6%	0.6%	5.1%	14.3%	16.7%	9%
add	14.4%	8.5%	23.8%	14.6%	11.1%	14%
sub	1.8%	0.3%		0.5%		0%
mul				0.1%		0%
div						0%
compare	15.4%	26.5%	8.3%	12.4%	5.4%	13%
load imm	8.1%	1.5%	1.3%	6.8%	2.4%	3%
cond. branch	17.4%	24.0%	15.0%	11.5%	14.6%	16%
uncond branch	1.5%	0.9%	0.5%	1.3%	1.8%	1%
call	0.1%	0.5%	0.4%	1.1%	3.1%	1%
return, jmp ind	0.1%	0.5%	0.5%	1.5%	3.5%	1%
shift	6.5%	0.3%	7.0%	6.2%	0.7%	4%
and	2.1%	0.1%	9.4%	1.6%	2.1%	3%
or	6.0%	5.5%	4.8%	4.2%	6.2%	5%
other (xor, not)	1.0%		2.0%	0.5%	0.1%	1%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
compare FP						0%
mov reg-reg FP						0%
other FP						0%

"load/store architecture"

keine mul/div Befehle (!)

FIGURE 2.26 DLX instruction mix for five SPECint92 programs. Note that integer register-register move instructions are included in the add instruction. Blank entries have the value 0.0%.

(H&P)

SPECint: x86

Instruction	compress	eqntott	espresso	gcc (cc1)	li	Int. average
load	20.8%	18.5%	21.9%	24.9%	23.3%	22%
store	13.8%	3.2%	8.3%	16.6%	18.7%	12%
add	10.3%	8.8%	8.15%	7.6%	6.1%	8%
sub	7.0%	10.6%	3.5%	2.9%	3.6%	5%
mul				0.1%		0%
div						0%
compare	8.2%	27.7%	15.3%	13.5%	7.7%	16%
mov reg-reg	7.9%	0.6%	5.0%	4.2%	7.8%	4%
load imm	0.5%	0.2%	0.6%	0.4%		0%
cond. branch	15.5%	28.6%	18.9%	17.4%	15.4%	20%
uncond. branch	1.2%	0.2%	0.9%	2.2%	2.2%	1%
call	0.5%	0.4%	0.7%	1.5%	3.2%	1%
return, jmp indirect	0.5%	0.4%	0.7%	1.5%	3.2%	1%
shift	3.8%		2.5%	1.7%		1%
and	8.4%	1.0%	8.7%	4.5%	8.4%	6%
or	0.6%		2.7%	0.4%	0.4%	1%
other (xor, not,...)	0.9%		2.2%	0.1%		1%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
compare FP						0%
mov reg-reg FP						0%
other (abs, sqrt,...)						0%

vgl. DLX: 26% / 9%

FIGURE D.15 80x86 instruction mix for five SPECint92 programs.

(H&P)

SPEC: x86 vs. DLX

	compress	eqntott	espresso	gcc (cc1)	li	Int. avg.
Instructions executed on 80x86 (millions)	2226	1203	2216	3770	5020	
Instructions executed ratio to DLX	0.61	1.74	0.85	0.96	0.98	1.03
Data reads on 80x86 (millions)	589	229	622	1079	1459	
Data writes on 80x86 (millions)	311	39	191	661	981	
Data read/modify/writes on 80x86 (millions)	26	1	129	48	48	
Total data reads on 80x86 (millions)	615	230	751	1127	1507	
Data read ratio to DLX	0.85	1.09	1.38	1.25	0.94	1.10
Total data writes on 80x86 (millions)	338	40	319	709	1029	
Data write ratio to DLX	1.67	9.26	2.39	1.25	1.20	3.15
Total data accesses on 80x86 (millions)	953	269	1070	1836	2536	
Data access ratio to DLX	1.03	1.25	1.58	1.25	1.03	1.23

x86 (CISC): 3% mehr Befehle als RISC...

x86 braucht mehr Lese-Befehle

und 3X mehr Store-Befehle

FIGURE D.17 Instructions executed and data accesses on 80x86 and ratios compared to DLX for five SPECint92 programs.

	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
Instructions executed on 80x86 (millions)	1223	15,220	13,342	6197	6197	
Instructions executed ratio to DLX	1.19	1.19	2.53	2.09	1.62	1.73
Data reads on 80x86 (millions)	515	6007	5501	3696	3643	
Data writes on 80x86 (millions)	260	2205	2085	892	892	
Data read/modify/writes on 80x86 (millions)	1	0	189	124	124	
Total data reads on 80x86 (millions)	517	6007	5690	3820	3767	
Data read ratio to DLX	2.04	2.36	4.48	4.77	3.91	3.51
Total data writes on 80x86 (millions)	261	2205	2274	1015	1015	
Data write ratio to DLX	3.68	33.25	38.74	16.74	9.35	20.35
Total data accesses on 80x86 (millions)	778	8212	7965	4835	4782	
Data access ratio to DLX	2.40	2.05	4.44	4.40	3.44	3.35

20X mehr FP-Stores wegen mangelnder Register (!)

FIGURE D.18 Instructions executed and data accesses for five SPECfp92 programs on 80x86 and ratio to DLX.

(H&P)

SPECfp: DLX

Instruction	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
load	1.4%	0.2%	0.1%	1.1%	3.6%	1%
store	1.3%	0.1%		0.1%	1.3%	1%
add	13.6%	13.6%	10.9%	4.7%	9.7%	11%
sub	0.3%		0.2%		0.7%	0%
mul						0%
div						0%
compare	3.2%	3.1%	1.2%	0.3%	1.3%	2%
load imm	2.2%		0.2%	2.2%	0.9%	1%
cond. branch	8.0%	10.1%	11.7%	9.3%	2.6%	8%
uncond branch	0.9%	0.4%		0.4%	0.1%	0%
call	0.5%	1.9%			0.3%	1%
return, jmp ind	0.6%	1.9%			0.3%	1%
shift	2.0%	0.2%	2.4%	1.3%	2.3%	2%
and	0.4%	0.1%			0.3%	0%
or		0.2%	0.1%	0.1%	0.1%	0%
other (xor, not)						0%
load FP	23.3%	19.8%	24.1%	25.9%	21.6%	23%
store FP	5.7%	11.4%	9.9%	10.0%	9.8%	9%
add FP	8.8%	7.3%	3.6%	8.5%	12.4%	8%
sub FP	3.8%	3.2%	7.9%	10.4%	5.9%	6%
mul FP	12.0%	9.6%	9.4%	13.9%	21.6%	13%
div FP	2.3%		1.6%	0.9%	0.7%	1%
compare FP	4.2%	6.4%	10.4%	9.3%	0.8%	6%
mov reg-reg FP	2.1%	1.8%	5.2%	0.9%	1.9%	2%
other FP	2.4%	8.4%	0.2%	0.2%	1.2%	2%

Adressarithmetik

load / store FP

FP-Arithmetik

FIGURE 2.27 DLX instruction mix for five programs from SPECfp92. Note that integer register-register move instructions are included in the add instruction. Blank entries have the value 0.0%.

(H&P)

SIMD: Flynn-Klassifikation

SISD *"single instruction, single data"*

=> jeder klassische PC

SIMD *"single instruction, multiple data"*

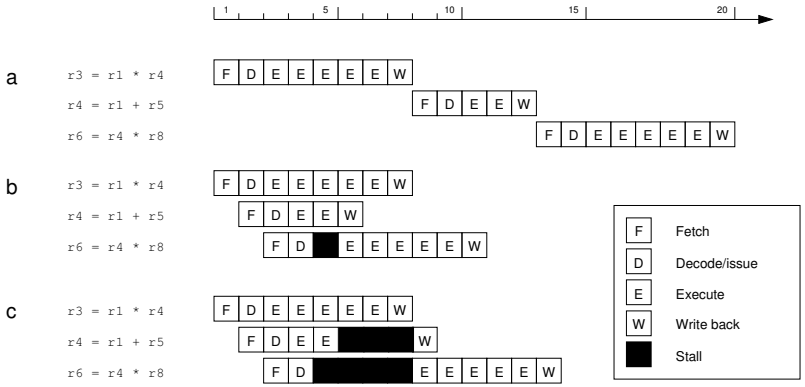
=> Feldrechner/Parallelrechner
=> z.B. Connection-Machine 2: 64K Prozessoren
=> eingeschränkt: MMX&Co: 2-8 fach parallel

MIMD *"multiple instruction, multiple data"*

=> Multiprozessormaschinen
=> z.B. vierfach PentiumPro-Server

MISD :-)

Befehlspipeline: in order / out of order



a) serielle Befehlsbearbeitung
b) pipeline, out-of-order completion
c) in-order-completion

superskalare Prozessoren

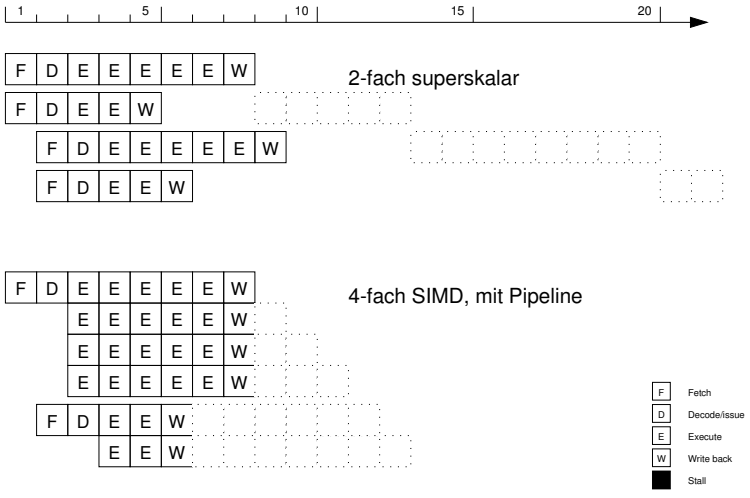
RISC vs. CISC für superskalare Prozessoren:	RISC	CISC
komplexe Befehlsdekodierung		•
mehrfache Funktionseinheiten	•	•
komplexes Steuerwerk (Scoreboard etc.)	•	•
out-of-order execution	•	•
große on-chip Caches	•	•
Speicherzugriffe sind das Nadelöhr	•	•

=> extreme Komplexität für RISC und CISC

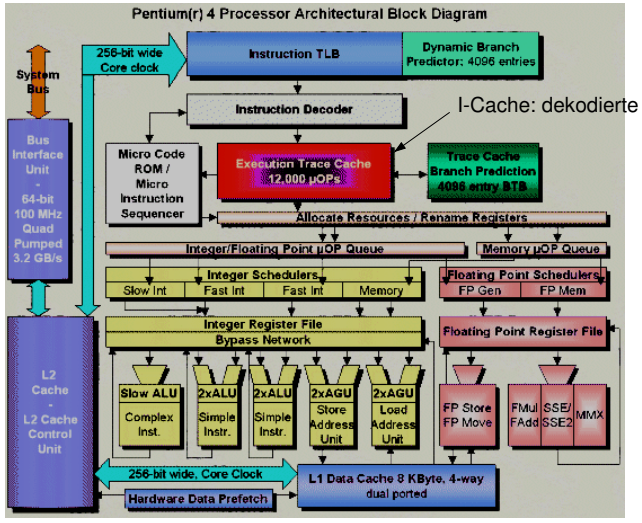
- Marktbedeutung der IA-32 erlaubt große Investitionen
- bessere Chiptechnologie zuerst für x86 (Intel, AMD)
- alle x86-Prozessoren seit Pentium sind superskalar

- vgl. AMD K7 Präsentation (extern)
- K7 verwaltet bis zu 72 "instructions in flight"

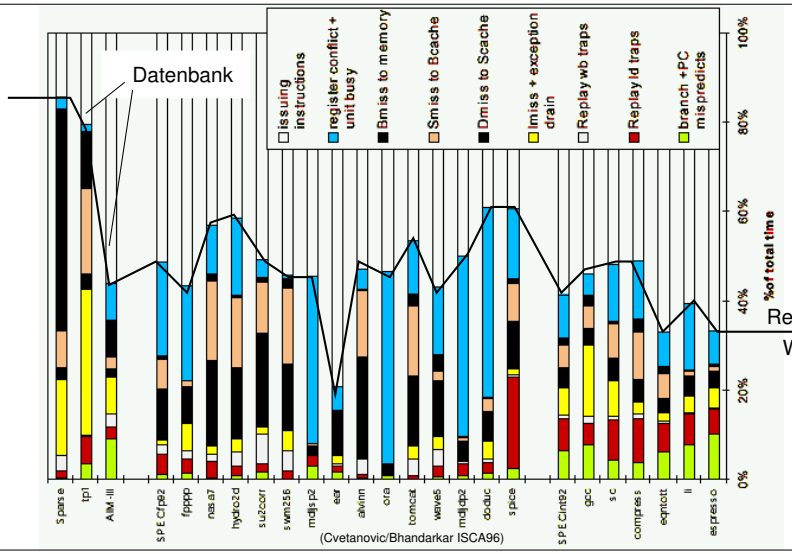
Superskalar, SIMD



Pentium IV

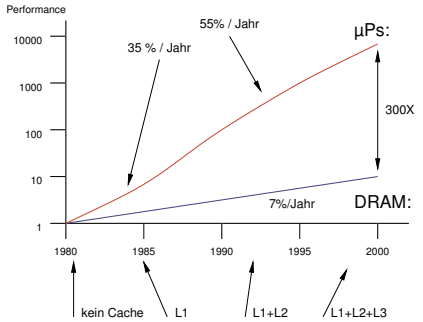


Benchmarks: Beispiel Alpha 21164



(Cvetanovic/Bhandarkar ISCA96)

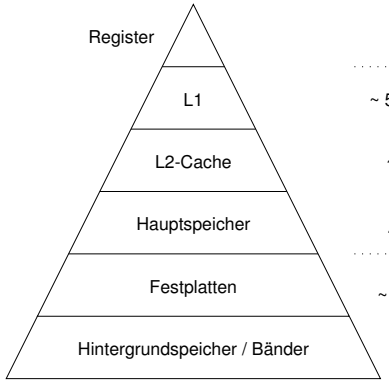
DRAM: Performance Gap



- DRAM-Kapazität: 60% / Jahr, Latenz: 7% / Jahr
- Prozessor-Performance: 55% / Jahr
- Kluft vergrößert sich ständig

=> Speicherhierarchie mit Caches notwendig

Speicherhierarchie



	Größe	Zugriffszeit	Bandbreite
Register	< 1 KB	1 ns	50 GB/s
L1	~ 512 KB	3 ns	10 GB/s
L2-Cache	~ 8 MB	10 ns	2 GB/s
Hauptspeicher	~ 1 GB	100 ns	100 MB/s
Festplatten	~ 50 GB	10.000.000 ns	10 MB/s
Hintergrundspeicher / Bänder	~ 1 TB	1 .. 100 s	1 MB/s

- "Diskontinuität" zwischen Halbleiter- und Magnetspeichern

DRAM: Performance Gap: Was tun?

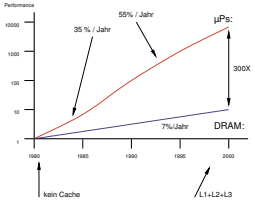
schnellerer Speicher notwendig ...

- aber DRAM inhärent langsam
- SRAM sehr teuer

=> DRAM besser ausnutzen
- SDRAM, SDRAM-DDR
- RAMBUS, SLDRAM

=> Speicherhierarchie verbessern
- größere, schnellere Caches
- bessere Cache-Organisation
- Prefetch-Optimierungen

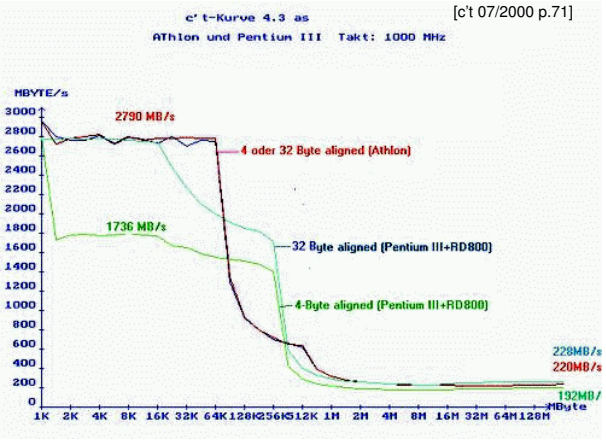
=> neue Konzepte?
- IRAM



"Cache: a safe place for hiding or storing things"
Websters dictionary

x86: ctkurve

[c't 07/2000 p.71]



- Messung der Cache-Transferrate vs. Blockgröße (random)
- Caches deutlich sichtbar: Pentium 16K/256K, Athlon 64K/512K

Cache: Speicherzugriffe

- Speicherzugriffe limitieren Performance (für von-Neumann Rechner)
- effiziente Ausnutzung von Caches erforderlich

aber:

- Cachegröße oft nicht ausreichend, Beispiel Bildverarbeitung:
- 1024x1024x8: 1 MByte / Bild mind. zwei Bilder erforderlich

also:

- sorgfältige Optimierung der Algorithmen / Filter
- möglichst lokale Zugriffe / "Blocking"-Techniken
- Ausnutzen von Prefetch-Befehlen

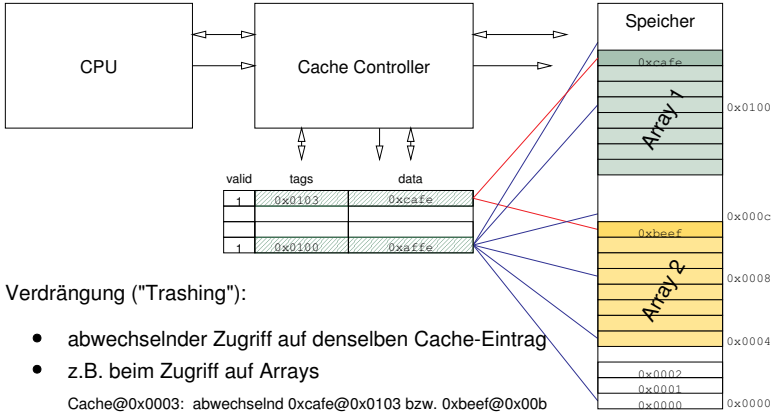
- völlig andere Zugriffsmuster als SPEC/OPC-Benchmarks
- evtl. Probleme mit Direct-Mapped Caches (Verdrängung)

Cache: Compulsory / Capacity / Conflict

3 Arten Cache-Misses:

- compulsory (cold start / first reference)
  erster Zugriff auf einen Block
- capacity Cache zu klein für alle benötigten Blöcke;
  Blöcke müssen ausgetauscht werden
  => Cache vergrössern
- conflict (collision misses / interference misses)
  bei direct mapped / set associative Caches:
  mehrere Blöcke im gleichen Set benötigt
  => Organisation verbessern, etwa 4fach assoz.
  => victim buffers

Cache: Trashing (direct mapped)



Verdrängung ("Trashing"):

- abwechselnder Zugriff auf denselben Cache-Eintrag
- z.B. beim Zugriff auf Arrays
  Cache@0x0003: abwechselnd 0xcafe@0x0103 bzw. 0xbeef@0x00b
- ständiges Neuladen, massiver Performanceverlust
- bessere Cache-Organisation / OS-Unterstützung

Cache: direct-mapped conflict misses

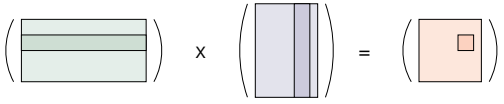
```
static void filterF(char* in1, char* out1)
{
    register int i0,i1,i2;
    register int x, int y;
    register char *in,*out;
    in = in1;
    out = out1;
    for( y=0; y < YRES; y++ ) {
        i0 = (int)in[0];
        i1 = (int)in[1];
        /* ignore boundary pixels, over/underflow for this benchmark */
        for( x=1; x < XRES-1; x++ ) {
            i2 = (int)in[x+1];
            out[x] = (char)( (i0 + (2*i1) + i2) / 4 );
            i0 = i1; i1 = i2;
        }
        in += XRES;
        out += XRES;
    }
} /*filterF*/
```

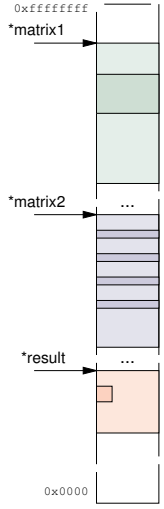
- read a byte from one array, compute, store result in second array, a byte at a time.
- If the arrays line up on top of each other in a direct-mapped cache, there is massive cache-thrashing.

execution time via array size: [comp.arch posting]

SYS	511	512	513	1023	1024	1025	2047	2048	2049	
CRIM	0.2	0.3	0.2	0.8	7.3*	0.9	3.7	33.4*	3.4	D
INDIGO4K	0.2	0.3	0.2	0.8	9.4*	0.8	3.2	37.9*	3.2	D
IN4K-fix	0.2	0.2	0.2	0.8	0.8	0.8	3.3	3.2	3.2	D
HP 720	0.3	0.7	0.3	1.1	2.7*	1.0	4.2	10.8*	4.2	D
HP 735	0.1	0.6*	0.1	0.6	2.7*	0.6	2.4	11.1*	2.6	D
HP 735	0.1	0.7*	0.1	0.6	2.7*	0.6	2.2	10.8*	2.2	D
Gwy486-66	0.3	0.3	0.3	1.3	1.4	1.3	5.5	5.5	5.5	SA?

Cache: Blocking





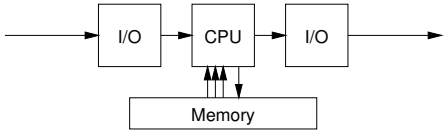
Beispiel Matrixmultiplikation:

- Zugriff auf drei Speicherbereiche
- unterschiedliche Zugriffsmuster
- gesamte Daten passen weder in L1 noch in L2
- Zugriffszeiten Reg/L1/L2/Mem: z.B. 1:2:10:100

=> "Blocking":

- Aufteilung des Algorithmus in Cache-gerechte Häppchen
- aber abhängig von Cache und Applikation
- oft Performancegewinn um Faktor 2..10 (..100) möglich

Rechnerarchitektur? Bsp. Bildverarbeitung

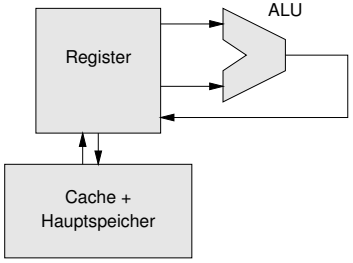


- oft viele Filter oder gar Filterbänke erforderlich
- lokale Operatoren erfordern (m*m) Operationen pro Bildpunkt
- Berechnung muß für jedes Bild wiederholt werden

  Beispiel: 1024x1024x8 Grauwertbild, 50 Filter à 7x7 Pixel:
  1024x1024 x 50 x 7x7 Operationen/Bild = 2.56 GOP/Bild

- Speicherbedarf für einzelne Bilder unproblematisch
- aber enorme Anforderungen an Rechenleistung

Rechnerarchitektur: von-Neumann

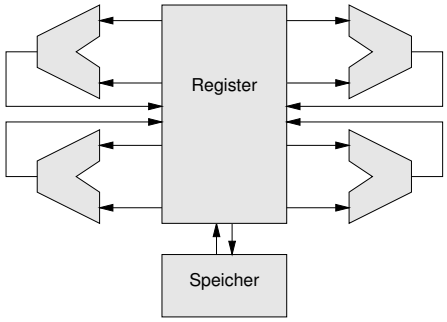


von-Neumann Prinzip:

- CPU mit Registern und ALU
- aber nur wenige (8 .. 64) Register

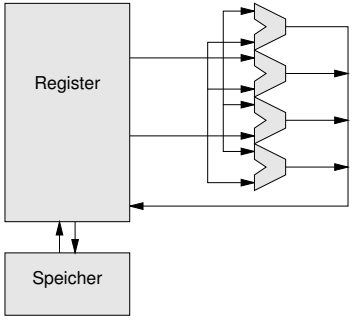
- selbst einzelne Filteroperation erfordert Speicherzugriffe
- extreme Anforderungen an Caching und Speicherverwaltung

Rechnerarchitektur: mehrere ALUs ?



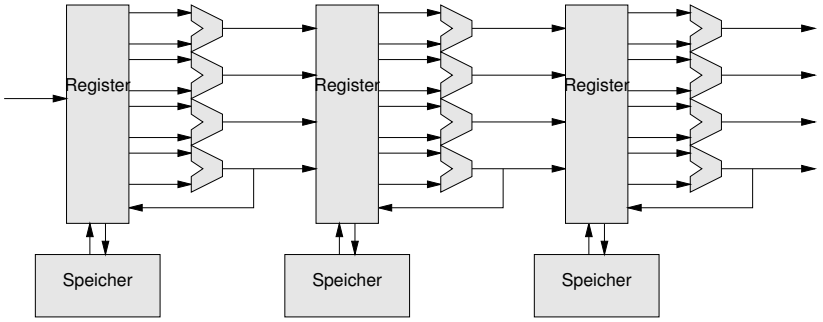
- viele Operationen im Prinzip massiv parallel berechenbar
- kaum Datenabhängigkeiten, wenn separate Quell- / Zielbilder
- scheitert am Speicherzugriff (von-Neumann bottleneck)

Rechnerarchitektur: VIS, MMX, ...



- Datenwortbreite 8 bit pro Pixel, aber 32/64 bit Datenpfade
- SIMD-Parallelverarbeitung, z.B. 8 Pixel parallel in einem Register
- keine Mehrbelastung für das Speicherinterface
- Details später

Rechnerarchitektur: Streaming ...



- höhere Performance nur mit höherer Parallelität
- lokale Speicher/Register erforderlich
- "Streaming"-Architekturen, Realisierung z.B. mit FPGAs
- Details später

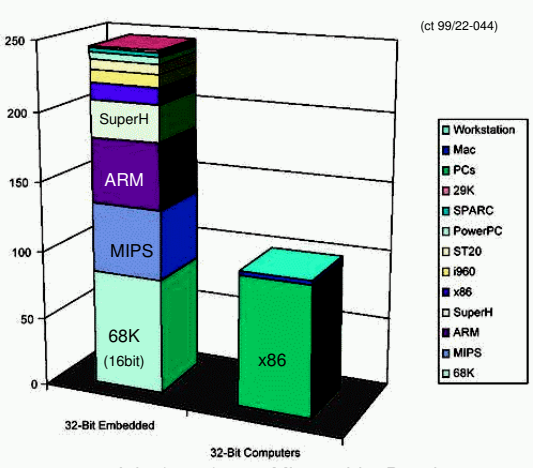
ARM

- Marktführer für 32-bit Mikroprozessoren / Mikrocontroller
- derzeit ca. 60% Marktanteil: vor MIPS (30%) und anderen
- StrongArm ausgewählt für WindowsCE/PocketPC
- einzige relavante europäische Prozessorfirma:
  www.arm.com

- "fabless" keine eigenen Fabriken
- "IP" Lizensierung der Prozessoren als "intellectual property"
- "cores" fertige Designs für Integration in Chips

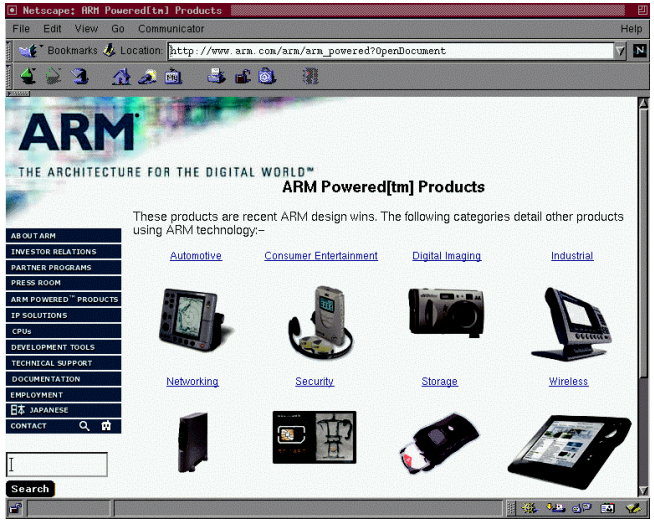
- ARM entwickelt von "Acorn Computers, Ltd", 1983-1985
- für den Nachfolger des BBC-Homecomputers (mit 6502 CPU)
- weil damalige 16-bit CPUs zu langsam (insb. für Interrupts)
- vollwertiger, aber sehr billiger RISC-Prozessor

Mikroprozessoren: 32-bit Markt '99

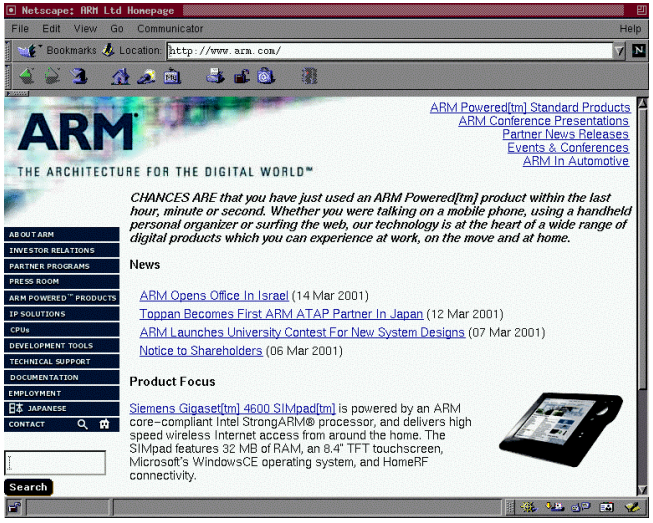


- pro Jahr (1999): 250 Mio. 32-bit µPs, plus ca. 100M in PCs
- zusätzlich ca. 5 Mrd. 4/8/16-bit Microcontroller

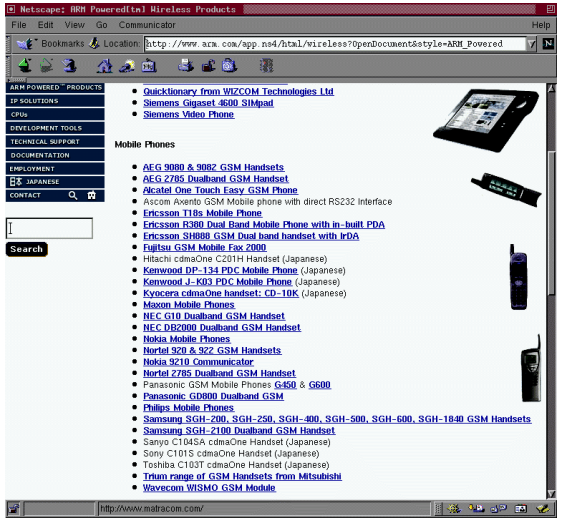
ARM: ARM powered products (3/2001)



ARM: "Advanced RISC Maschines"



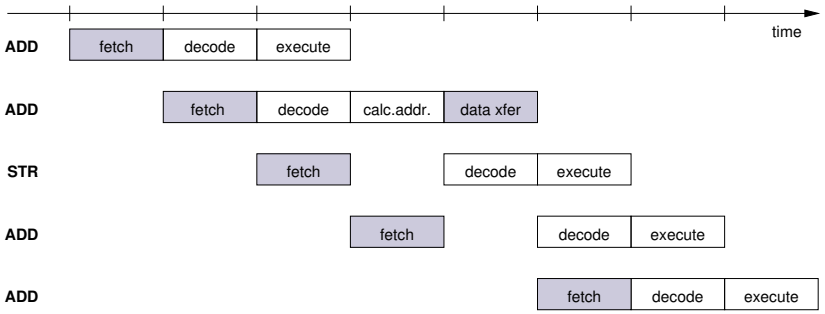
ARM: im Handy-Markt



ARM: Architektur

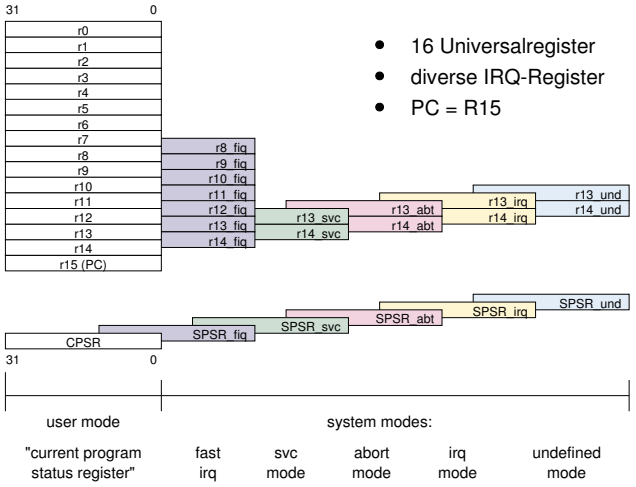
- 32-bit RISC - nach Vorbild von RISC-I bzw. MIPS
  - load/store Architektur
  - 16 Universalregister
  - 3-Adress-Befehle, ein festes 32-bit Format
  - Ausführung aller Befehle in einem Takt (soweit möglich)
- aber kein Cache (aus Kostengründen)
- 3-stufige Pipeline
- erlaubt optimale Ausnutzung der Speicherbandbreite
- interessanter Befehlssatz (z.B. "predication", PC=R15, ...)
- Spezialregister für schnelle Interrupt-Behandlung
- Koprozessor-Interface für spätere Erweiterungen

ARM: 3-stufige Pipeline



- Konzept mit 3-stufiger Pipeline, ohne Cache
- Lade-/Speicherbefehle benötigen 4 Takte:
  fetch / decode / address calc. / data transfer
- Beispiel für Nutzung der Pipeline

ARM: Register



- 16 Universalregister
- diverse IRQ-Register
- PC = R15

(Furber, S39)

ARM: Befehlssatz (1)

- übliche 3-Adress-Befehle:

```
ADD  r0, r1, r2   ; r0 := r1 + r2
ADC  r0, r1, r2   ; r0 := r1 + r2 + C      (add with carry)
SUB  r0, r1, r2   ; r0 := r1 - r2
RSC  r0, r1, r2   ; r0 := r2 - r1 + C - 1 (reverse sub)
...
AND  r0, r1, r2   ; r0 := r1 AND r2
EOR  r0, r1, r2   ; r0 := r1 XOR r2
...
MVN  r0, r2       ; r0 := NOT r2           (move negated)
...
CMP  r1, r2       ; set CC on (r1 - r2)    (compare)
TST  r1, r2       ; set CC on (r1 AND r2)  (bit test)
...
```

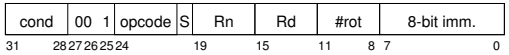
ARM: Befehlssatz (2)

- optional: Verschiebung des zweiten Operanden:

```
ADD    r3, r2, r1, LSL #3;   r3 := r2 + r1<<3
ADD    r5, r5, r3, LSL r2;   r5 := r5 + r3<<(2^r2)  (!)

mögliche Shifts: LSL, LSR, ASL, ASR, ROR, RRX
```

- Immediate-Operanden: (0 .. 255) * 2^(2*#rot)
- deckt viele wichtige Immediate-Werte ab

```
ADD    r3, r3, #1        ; r3 := r3 + 1
AND    r8, r7, #&FF      ; r8 := r7[7:0]
```



ARM: Beispielcode

```
      // for (i=0; i < 10; i++) { a[i] = 0; }
      MOV    r1, #0           ; value to store in a[i]
      ADR    r2, a[0]         ; r2 points to a[0]
      MOV    r0, #0           ; i=0
LOOP: CMP    r0, #10          ; i<10?
      BGE    EXIT             ; if i>=10 goto exit
      STR    r1,[r2,r0,LSL #2]; a[i]=0
      ADD    r0, r0, #1       ; i++
      B      LOOP
EXIT  ...

      ... // store and load multiple registers:
      STMFD  r13!, {r0-r2,r14}; save work regs & link reg.
      BL     SUBROUTINE       ; branch and link
      ...
      LDMFD  r13!, {r0-r2,pc} ; restore work regs & return
```


ARM: Befehlssatz (3)

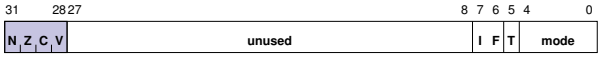
- 64-bit Addition (r1,r0) + (r3,r2)

```
ADDS   r2, r2, r0        ; r2:=r2+r0, S=set CC (carry)
ADC    r3, r3, r1        ; r3:=r3+r1+c
```

- Unterprogramm für (10*r0)

```
       MOV    r0, #3
       BL     TIMES10           ; Funktionsaufruf
       ...
TIMES10: MOV  r0, r0, LSL #1     ; X2 r0:=r0<<1
       ADD    r0, r0, r0, LSL #2 ; X5 r0:=r0+r0<<2
       MOV    pc, r14           ; Rücksprung
```


ARM: Predicated Instructions



Current Program Status Register

- bedingte Ausführung aller (!) Befehle
- abhängig von den Statusflags (negative, zero, carry, overflow)
- 16 Varianten: EQ NE .. GE LT .. ALways NeVer

```
...
CMP    r0, #5  ;  if (r0 < 5)
BLLT   SUB1    ;  then call SUB1
BLGE   SUB2    ;  else call SUB2
...
```

ARM: Exceptions

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instr. memory fault)	Abort	0x0000000C
Data abort (data access mem.fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

- Exceptions speichern Werte von PC und CPSR
- verzweigen zu den Vektoren (dort dann Sprung zum Handler)
- spez. Befehle zum Rücksprung

- zusätzliche Arbeitsregister für FIQ, Handler ab 0x0000001C

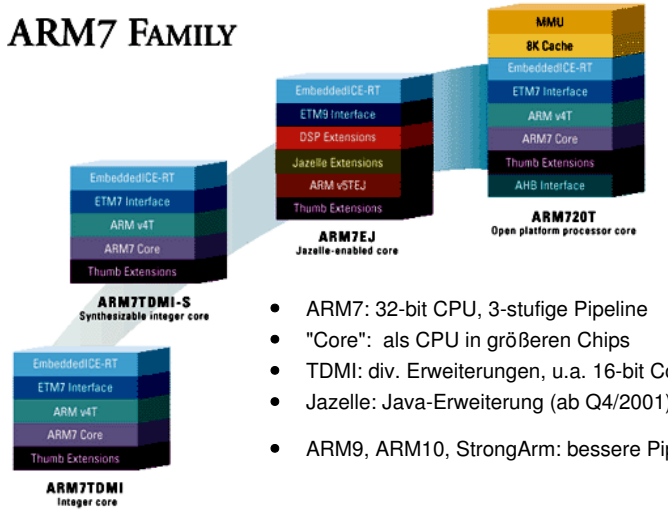
ARM: TDMI. . .

diverse Architekturvarianten:

Thumb	16-bit Befehlssatz, für minimale Programmgröße
Debug	Debug-Interface
Multiplier	Multiplizierer, 32x32 -> 64 bit
ICE	In-Circuit Emulator: Echtzeit-Breakpoints usw.
Piccolo	DSP-Erweiterung mit Puffern und schnellen MAC
Jazelle	Hardware zur direkten Ausführung von Java-Bytecodes
AMBA	"advanced microcontroller bus architecture"
ARM 7	einfache 3-stufige Pipeline, kein Cache
ARM 9	klassische 5-stufige Pipeline, I/D-Caches
StrongARM	mehrstufige Pipeline, hohe Taktraten (Compaq/intel)
. . .	

(Furber, www.arm.com)

ARM: Roadmap



- ARM7: 32-bit CPU, 3-stufige Pipeline
- "Core": als CPU in größeren Chips
- TDMI: div. Erweiterungen, u.a. 16-bit Code
- Jazelle: Java-Erweiterung (ab Q4/2001)

- ARM9, ARM10, StrongArm: bessere Pipeline

(www.arm.com, Furber)

ARM: Thumb-Befehlssatz

- starker Kostendruck für eingebettete / mobile Geräte
- möglichst kleiner Programmspeicher wünschenswert
- auch zu Lasten von Performance

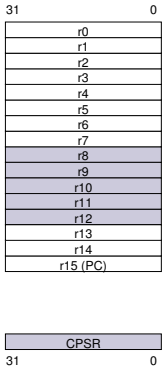
=> "Thumb"-Modus mit kompakterer Befehlskodierung

- 16-bit Befehlskodierung mit 2-Adress-Befehlen
- eingeschränkter Befehlssatz, Zugriff nur auf r0..r7, r13, r14, pc
- Umschalten zum ARM-Modus jederzeit möglich:
- für Exceptions / Betriebssystem / vollen Befehlssatz

Thumb vs. ARM-Befehlssatz:

- ca. 40% mehr Befehle pro Programm
- ca. 70% der Programmgröße
- ca. 30% weniger Stromverbrauch für Speicherzugriffe
- ca. 45% schneller (16-bit Speicher) / 40% langsamer (32-bit Speicher)

Thumb: Register



- Lo/Hi Register: r0 .. r7 / r8 .. r15
- Hi-Register nur eingeschränkt benutzbar
- r13 Stackpointer
- r14 Linkregister
- r15 Programmzähler

- 16-bit 2-Adress Befehlsformat
- alle Befehle setzen Condition Codes

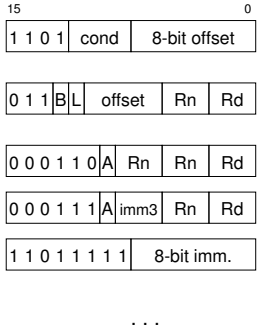
- Implementierung durch Umkodierung in äquivalente ARM-Befehle

- wenig reguläre Befehlskodierung
- um 16-bit Raum voll auszunutzen

ARM: Thumb-Beispiel

```
        ...
        CODE32              ; program starts in ARM mode
        ADR     r0, START+1; thumb entry address
        BX      r0          ; jump to thumb area
        CODE16              ;
START   ADR     r1, TEXT    ; r1 -> "Hello, world"
 LOOP   LDRB    r0, [r1]    ; get the next byte
        ADD     r1, r1, #1  ; increment pointer **T
        CMP     r0, #0      ; check for end of text
        BEQ     DONE        ; finished?
        SWI     SWI_putchar; if not, print char (ARM mode!)
        B       LOOP;       ; ... next iteration
 DONE   SWI     SWI_exit    ;
        ...
 TEXT   DATA = "Hello, world", &0a,&0d,&00
```

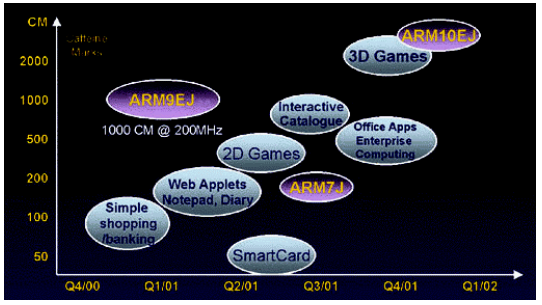

Thumb: Beispiele für Befehle




```
B<cond> <label>

LDR|STR{B} Rd,[Rn,#offset5]

ADD|SUB rd,rn,rm

ADD|SUB rd,rn,#imm3

Thumb software interrupt
```

. . .

ARM: Jazelle



- Java ideal zur Erweiterung von mobilen/eingebetteten Geräten
- aber Java-Interpreter zu langsam / JIT zu speicheraufwendig

"Jazelle":

- Befehlssatzerweiterung, direkte Ausführung von Java-Bytecode

ARM: Jazelle Konzept

direkte Ausführung der wichtigsten Java-Bytecodes:

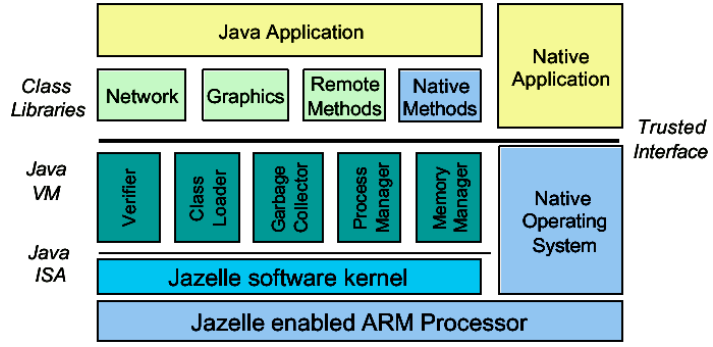
- zusätzlicher Betriebsmodus (ARM | Thumb | Jazelle)
- Befehlsdecoder für Bytecodes (ca. 12.000 Gatter)
- ALU angepaßt an Java-Datentypen und Stack-Adressierung

- Unterstützung für "komplexe" Bytecodes
- z.B. Funktionsaufrufe ("invokevirtual", "invokestatic")

- Exceptions und OS-Aufrufe wechseln in ARM-Modus
- bestehende ARM Betriebssysteme können verwendet werden
- Mischung aus Java / C / Assemblercode problemlos möglich

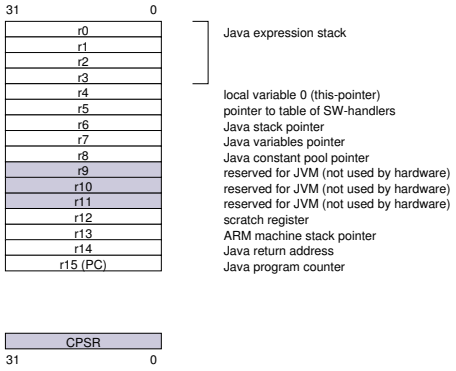
- hohe Performance, kompakter Code, kein JIT-Compiler notwendig

ARM: Jazelle, Schichten



- Software-Kernel implementiert die "komplexen" Java-Bytecodes
- obere Softwareschichten (Class Libraries, Apps) voll portabel
- Jazelle-Befehlssatz auch im "Armulator" und on-chip Debugging

ARM: Jazelle Register



- alle ARM Register übernehmen Spezialaufgaben
- vier Register (r0..r3) als Rechenregister ("stack cache")
- Auslagern / Einlagern der Register auf den Stack automatisch

ARM: Jazelle und Umfeld

	Execution Performance CM/MHz	Real-time System Performance	Memory Cost	Hardware Implementation Cost	Legacy Code / RTOS support
Software Emulation (SUN JDK, ARM9)	0.67		~16kbyte	-	Yes
Software Emulation (ARM JDK, ARM9)	1.7		~16kbyte	-	Yes
JIT	6.2*	Poor	>100k byte	-	Yes
Co-processor (eg Jedi Tech. JSTAR)	2.9		-	~25k gates	Yes
Dedicated Processor	3		-	20-30k gates	No
ARM with architecture extensions	5.5	Excellent	~8kByte	~12k gates	Yes

The only solution to meet all of the performance & application requirements.

*Note: JIT performance excludes compilation overhead.

- guter Kompromiß aus HW/SW-Aufwand und Performance
- JIT-Compiler besser wegen zusätzlicher Optimierungen

(ARM Jazelle Whitepaper)

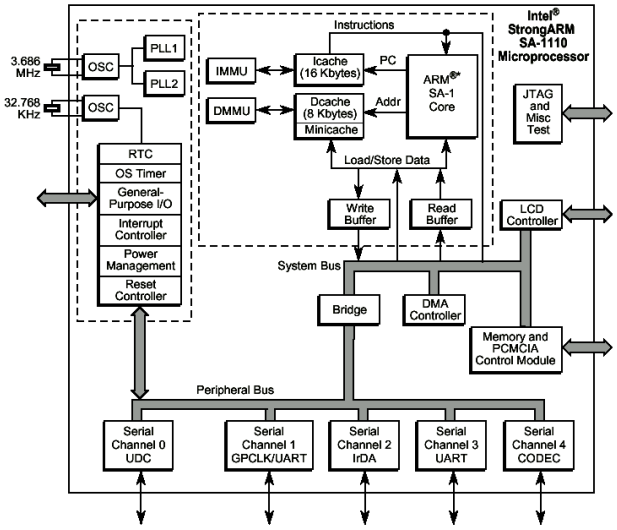
StrongARM

- gemeinsam entwickelt von DEC und ARM
- Harvard-Architektur mit separaten I/D-Caches
- klassische 5-stufige Pipeline
- mit Forwarding / Bypass
- on-Chip MMU für Multitasking-Betriebssysteme

- Design ausgelegt für sehr hohen Takt
- bei geringem Stromverbrauch
- SA-110: 2.5 Millionen Transistoren, 200 MHz Takt, < 1 Watt
- zugehöriger Supportchip liefert weitere Schnittstellen

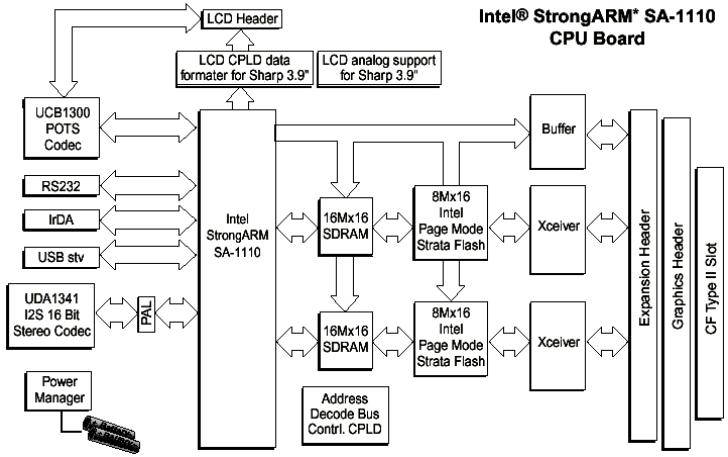
- eingesetzt in den Compaq "Itsy" PDA-Prototypen
- derzeit u.a. in den meisten Windows-CE Organizern
- von Microsoft als einzige CPU für "Pocket-PC 2002" ausgewählt

StrongARM: Blockschaltbild SA-1110



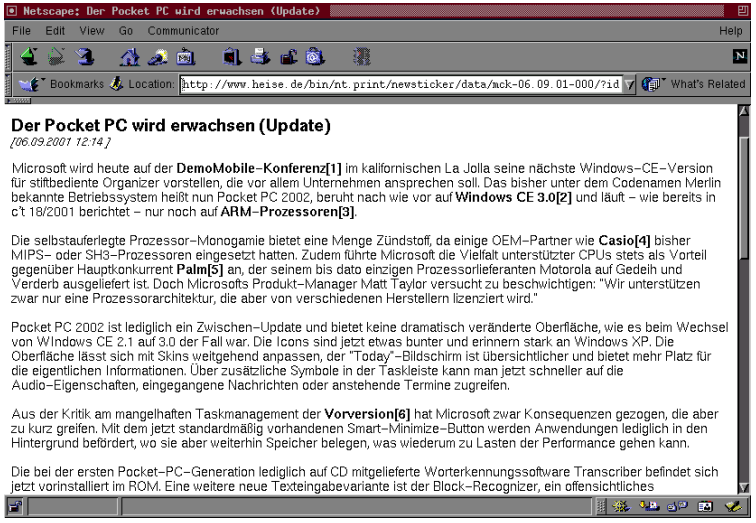
(Intel SA1110 Datenblatt)

StrongARM: Intel SA1110



CPU mit vielen Schnittstellen on-chip, Gesamtsystem: nur CPU+Speicher+IO

StrongARM: fuer PocketPC



Xscale: Übersicht

- Intels neueste Version der ARM-Architektur

- 7-stufige, komplexe Pipeline für höhere Taktrate
- zusätzlicher ALU (MAC) für Signalverarbeitung
- zugehörige neue (SIMD) Befehle

- dreifacher Cache: 32 KB I, 32 KB D, 2 KB "mini data"
- getrennte Befehls- und Daten-MMU
- diverse Pufferspeicher, separate 32-bit Load/Store-Busse

- flexibles Power-Management
- "speed step" Technik regelt Spannung vs. Taktrate

- Fertigung in 0.18µ

(intel XScale whitepaper, und SA 1110 Datenblatt)

Xscale: Signalverarbeitung

```
SMLAxy          32 <= 16x16 + 32
SMLAWy          32 <= 32*16 + 32
SMLALxy         64 <= 16*16 + 64
SMULsy          32 <= 16*16
SMULWy          32 <= 32*16
QADD            rz <= saturate( rx + ry )
QDADD           rz <= saturate( rx + 2*ry )
QSUB            ...
. . .
```

- Auswahl der DSP-Befehle
- Intel liefert "integrated performance primitives" Bibliotheken, "filtering, ..., transforms, MP3, H.263, MPEG-4, GSM, JPEG"
- Konzept und Details zu SIMD-Befehlssätzen s.u.

(Intel IPP datasheet)

Xscale: Performance

Cache blocking techniques, such as strip-mining, are used to improve temporal locality of the data. Given a large data set that can be reused across multiple passes of a loop, data blocking divides the data into smaller chunks which can be loaded into the cache during the first loop and then be available for processing on subsequence loops thus minimizing cache misses and reducing bus traffic.

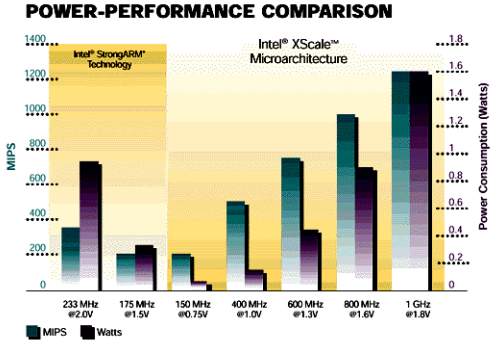
As an example of cache blocking consider the following code:

```
for(i=0; i<10000; i++)
    for(j=0; j<10000; j++)
        for(k=0; k<10000; k++)
            C[j][k] += A[i][k] * B[j][i];
```

The variable A[i][k] is completely reused. However, accessing C[j][k] in the j and k loops can displace A[i][j] from the cache. Using blocking the code becomes:

```
for(i=0; i<10000; i++)
    for(j1=0; j<100; j++)
        for(k1=0; k<100; k++)
            for(j2=0; j<100; j++)
                for(k2=0; k<100; k++)
                {
                    j = j1 * 100 + j2;
                    k = k1 * 100 + k2;
                    C[j][k] += A[i][k] * B[j][i];
                }
```


Xscale: Performance



- XScale mit 0.13µm CMOS-Prozeß, 1 GHz Takt, < 2 Watt
- gleichzeitig bessere Performance durch bessere Architektur
- klarer Kandidat für zukünftige Mobilgeräte

(www.developer.intel.com)