## x86 Prozessoren: Inhalt

Architektur der Intel x86-Familie:

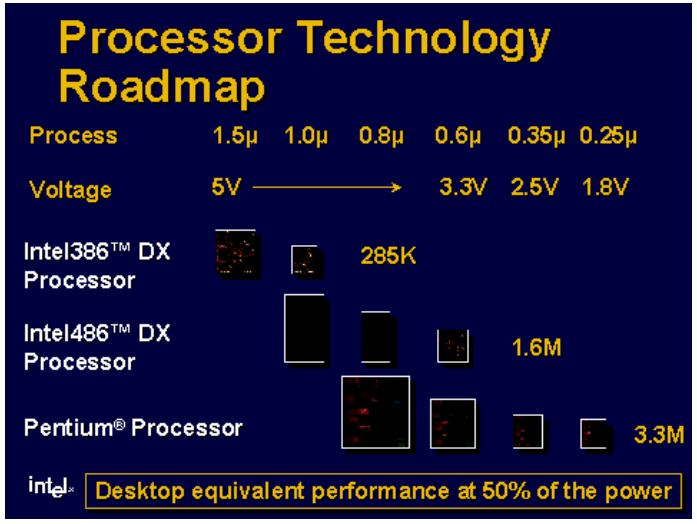
- Historie — 8008 -> Pentium III
- Register
- Befehlssatz — nur Übersicht
- Speichermodell — real / protected / virtual 8086 / ...
- Programmbeispiele

- RISC vs. CISC - Debatte
- Instruction Level Parallelism
- Aktuelle Implementation — AMD Athlon

- Ausblick auf IA-64 und AMD x86-64



## x86: Halbleitertechnologien . . .




[intel IDF98]



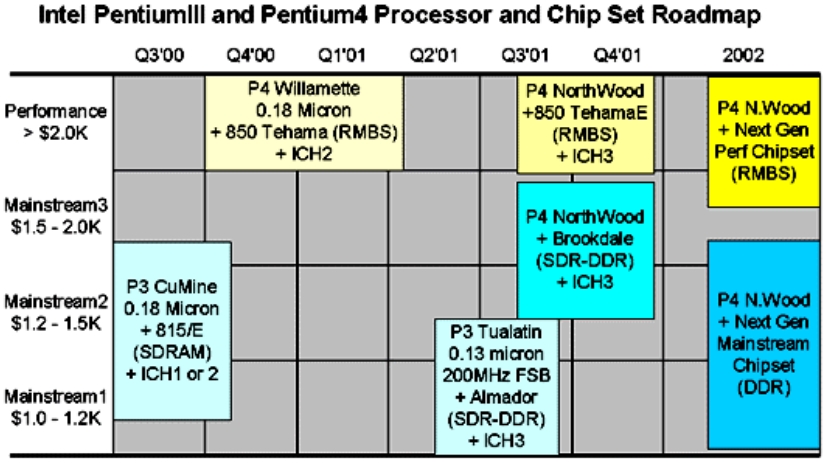
## x86: Evolution ...

| Intel Processor | Date of Product Intro-duction | Perform-ance in MIPs[1] | Max. CPU Frequency at Intro-duction | No. of Transis-tors on the Die | Main CPU Register Size[2] | Extern. Data Bus Size[2] | Max. Extern. Addr. Space | Caches in CPU Pack-age[3] |
|---|---|---|---|---|---|---|---|---|
| 8086 | 1978 | 0.8 | 8 MHz | 29 K | 16 | 16 | 1 MB | None |
| Intel 286 | 1982 | 2.7 | 12.5 MHz | 134 K | 16 | 16 | 16 MB | Note 3 |
| Intel386™ DX | 1985 | 6.0 | 20 MHz | 275 K | 32 | 32 | 4 GB | Note 3 |
| Intel486™ DX | 1989 | 20 | 25 MHz | 1.2 M | 32 | 32 | 4 GB | 8KB L1 |
| Pentium® | 1993 | 100 | 60 MHz | 3.1 M | 32 | 64 | 4 GB | 16KB L1 |
| Pentium® Pro | 1995 | 440 | 200 MHz | 5.5 M | 32 | 64 | 64 GB | 16KB L1; 256KB or 512KB L2 |
| Pentium II® | 1997 | 466 | 266 | 7 M | 32 | 64 | 64 GB | 32KB L1; 256KB or 512KB L2 |
| Pentium® III | 1999 | 1000 | 500 | 8.2 M | 32 GP 128 SIMD-FP | 64 | 64 GB | 32KB L1; 512KB L2 |

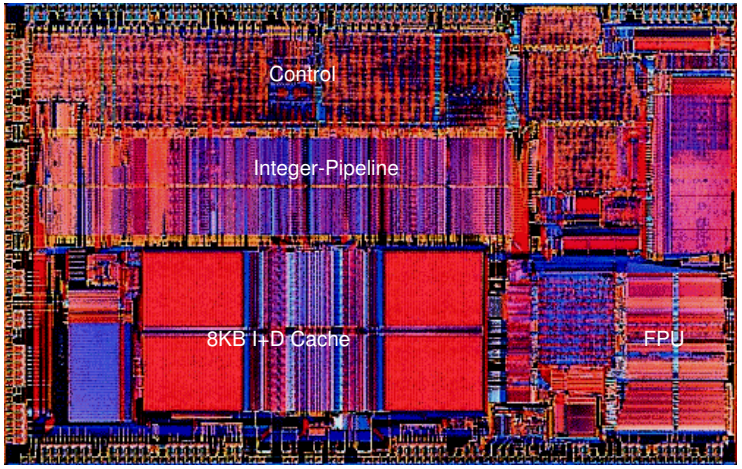
[Intel Pentium-III databook]



## x86: Intel Roadmap Q3/00

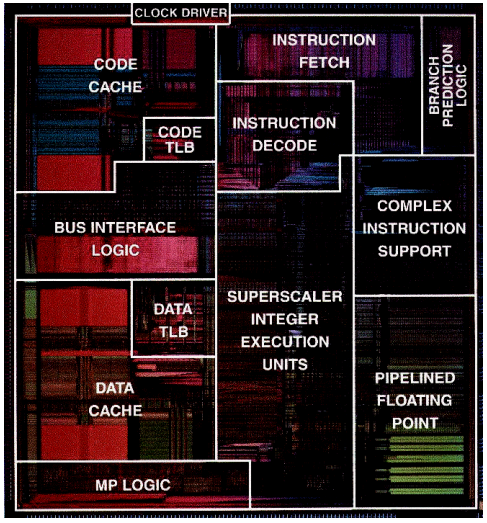


## x86: Chiplayout 486DX






## x86: Chiplayout Pentium (P54C)

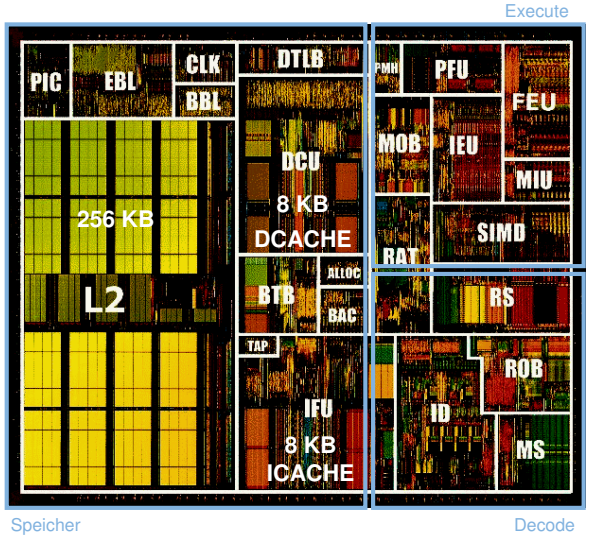



[www.intel.com]

- ~ 40% Speicher
- ~ 60% Execute:
- ~ 15% FPU
- ~ 10% APIC/MP

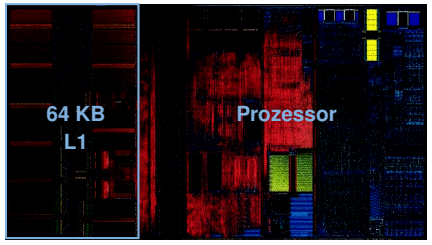


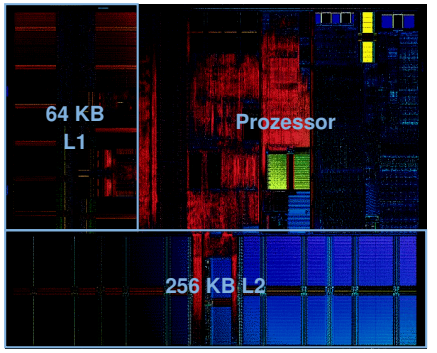
## Pentium III: ChipLayout






## AMD K6: Layout K6-2 vs. K6-III




- gleicher Prozessorkern, 32K I$, 32K D$, 256K L2
- ca. 30% bzw. über 50% Chipfläche für Speicher

## x86: Pentium-Klasse

**Moderne x86-CPUs im Vergleich**

| Typ | Pentium | P6 | K86 | M1 | Nx586 |
|---|---|---|---|---|---|
| Hersteller | Intel | Intel | AMD | Cyrix | NexGen |
| Interner Takt | 100 MHz | 133 MHz | 100 MHz | 100 MHz | 93 MHz |
| Daten-Cache | 8 KByte | 8 KByte | 16 KByte | 16 KByte | 16 KByte |
| Befehls-Cache | 8 KByte | 8 KByte | 8 KByte | unified | 16 KByte |
| L2-Cache-Interface | – | ja | – | – | ja |
| L2-Cache | – | 256 KByte | – | – | – |
| Dispatcher-Rate | 2 Befehle | 3 Befehle | 2–3 Befehle | 2 Befehle | 2 Befehle |
| Parallele Einheiten | 3 Einheiten | 5 Einheiten | 7 Einheiten | 2 Einheiten | 3 Einheiten |
| Out-of-Order | – | 40 Befehle | 16 Befehle | – | – |
| Renaming-Register | – | 40 Register | 16 Register | 32 Register | – |
| IC-Prozeß | 0,6 µ BiCMOS | 0,6 µ BiCMOS | 0,5 µ CMOS | 0,65 µ CMOS | 0,5 µ BiCMOS |
| Metall-Layer | 4 | 4 | 3 | 3 | 4 |
| Logic Transistoren | 2,4 Millionen | 4,5 Millionen | 2,4 Millionen | 2,1 Millionen | 2,4 Millionen |
| Transistoren f. L1 | 0,9 Millionen | 1,0 Millionen | 1,9 Millionen | 0,9 Millionen | 0,9 Millionen |
| Transistoren f. L2 | – | 15 Millionen | – | – | – |
| Alle Transistoren | 3,3 Millionen | 20,5 Mill. | 4,3 Millionen | 3,0 Millionen | 3,3 Millionen |
| Fassungstyp | CPGA | CPGA | CPGA | CPGA | CPGA |
| Anzahl Pins | 296 Pins | 387 Pins | 296 Pins | 296 Pins | 463 Pins |
| Die Size | 163 $mm^2$ | 306 $mm^2$ 202 $mm^2$ | 225 $mm^2$ | 394 $mm^2$ | 196 $mm^2$ |
| Herstellungskosten | 120 $ | 350 $ [1] | 170 $ | 340 $ | 200 $ |
| Leistungsaufnahme | 10 Watt | 20 Watt 2 | 12 Watt | 10 Watt | 16 Watt |
| Verfügbarkeit | 2Q94 | 3Q95 | 3Q95 | 3Q95 | 3Q94 |
| SPECint92 | 113 | 200 | 130 | 120 | 110 |
| SPECfp92 | 82 | 200 | 75 | 70 | – |

Quelle: Microprozessor Report

[1] Ohne L2-Cache

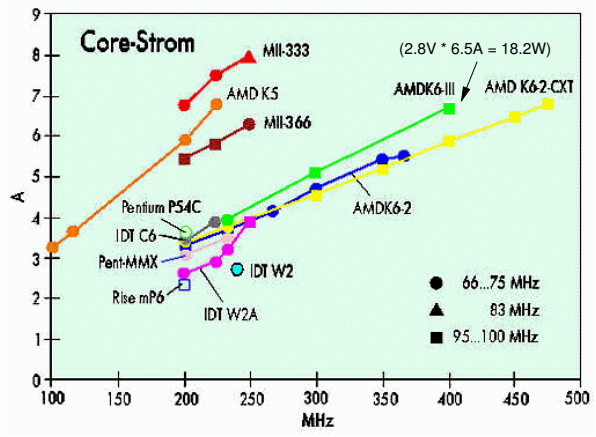
[2] Inclusive L2-Cache

- fünf Designs, vier Firmen
- alle superskalar
- dispatch 2-5X
- execute 3-7X
- Herstellungskosten (!)

[c't 05/95 122]



## x86: Pentium-Klasse: Verlustleistung . . .

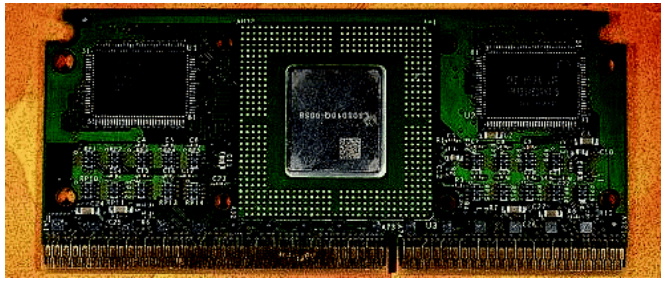



- CMOS-Technologie: Leistung ~ (f/Hz) * (U/Volt) ²
- Kühltechnologie begrenzt auf < 50 W

[c't 10/99 176]



## x86: Pentium-II/400 Package



Intel Verpackungstechnologie Q1/1999:

- CPU/FPU mit 16KB/16KB I+D Cache im Plastikgehäuse
- zwei externe SRAM-Chips für 512KB L2-Cache
- "Slot-1" Einsteckkarte (Busprotokoll patentiert)



Leerseite

## x86: Performance 1999 . . .

**Leistungsfähigkeit der Prozessor-Familien**

| Prozessor | BAPCo SYSmark'98 | CPU 3DMark99[1] | PovRay 3.0 [s] | Unreal [fps][2] |
|---|---|---|---|---|
| | besser ▶ | besser ▶ | ◀ besser | besser ▶ |
| *bei 400 MHz, 64 MByte, Riva-TNT Grafik* | | | | |
| AMD K6-2 | 131 | 5588 | 57 | 25,2 |
| AMD K6-III | 151 | 6309 | 44 | 26,3 |
| Intel Celeron | 147 | 3681 | 42 | 27,7 |
| Intel Pentium II | 162 | 3903 | 44 | 30,6 |
| *bei 450 MHz, 64 MByte, Riva-TNT Grafik* | | | | |
| AMD K6-2 | 137 | 5899 | 52 | 24,7 |
| AMD K6-III | 164 | 7090 | 39 | 26,6 |
| *bei 500 MHz, 128 MByte, Riva-TNT2 Grafik* | | | | |
| Intel Celeron | 178 | 4479 | 34 | 37,9 |
| Intel Pentium III | 192 | 7599 | 38 | 42,3 |
| AMD Athlon | 212 | 9343 | 27 | 44,1 |
| *bei 600 MHz, 128 MByte, Riva-TNT2 Grafik*[3] | | | | |
| Intel Pentium III | 221 | 9060 | 32 | 45,7 |
| AMD Athlon | 238 | 10915 | 22 | 47,3 |

[1] Futuremarks 3DMark99 Max; davon der CPU-Test, der von der Grafikkarte weitgehend unabhängig ist.
[2] Unreal 2.20, 800 x 600 Punkte, 16 Bit Farbtiefe
[3] 700-MHz-Werte stehen auf Seite 132

- Performance ~ Taktfrequenz, Architekturdifferenzen irrelevant (10%)
- K6-2 ohne L2-Cache, Celeron ohne ISSE/3Dnow!

[c't 10/99 176]



## x86: Performance 2000 . . .

**Leistungsdaten aktueller AMD- und Intel-Systeme**

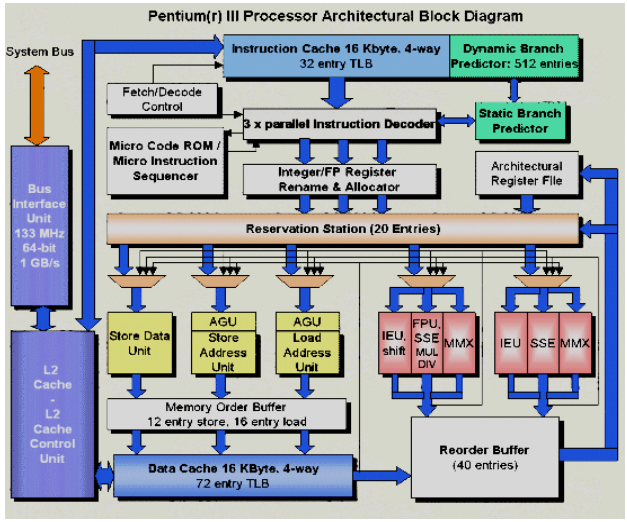
| | Prozessor | FSB [MHz] | Speicher | Board | Windows 98 SE – BAPCo SYSmark2000 SYSmark | Internet Content Creation | Office Productivity | PovRay 3.1g PPS | Linux-Kernel sec |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | besser ▶ | besser ▶ | besser ▶ | besser ▶ | ◀ besser |
| *Fliegengewicht* | | | | | | | | | |
| AMD | K6-2/550 | 100 | PC100-222 | P5A | 78 | 69 | 85 | 240 | 246 |
| Intel | P II 450 MHz | 100 | PC100-222 | P3B-F | 93 | 86 | 98 | 265 | 252 |
| AMD | K6-III/450 | 100 | PC100-222 | P5A | 88 | 75 | 98 | 257 | 212 |
| Intel | Celeron 500 | 66 | PC66-222 | P3B-F | 94 | 89 | 97 | 313 | 245 |
| *Mittelgewicht* | | | | | | | | | |
| Intel | FC-PGA-Celeron 600 | 66 | PC66-222 | P3B-F | 112 | 114 | 111 | 385 | 206 |
| Intel | Pentium III 600 (Katmai) | 100 | PC100-222 | P3B-F | 124 | 124 | 124 | 353 | 202 |
| Intel | FC-PGA-Celeron 700 | 66 | PC100-222 | CUV4X | 123 | 126 | 120 | 433 | 184 |
| AMD | Athlon-600 | 100 | PC133-333 | K7V | 129 | 128 | 130 | 468 | 173 |
| AMD | Duron-650 | 100 | PC133-333 | KT133 | 132 | 134 | 131 | 515 | 174 |
| AMD | Duron-700 | 100 | PC133-333 | KT133 | 139 | 141 | 137 | 556 | 166 |
| *Schwergewicht* | | | | | | | | | |
| Intel | Pentium III 800 | 133 | PC133-333 | D1184 | 167 | 167 | 167 | 556 | 122 |
| AMD | Athlon-800 | 100 | PC133-222 | K7V | 155 | 159 | 152 | 614 | 138 |
| Intel | Pentium III 1000 | 133 | PC133-333 | CUV4X | 185 | 189 | 182 | 698 | 102 |
| AMD | Athlon-1000 (Thunderbird) | 100 | PC133-333 | K7V | 186 | 187 | 186 | 800 | 103 |
| Intel | Pentium III 1000 (Rambus) | 133 | PC800-45 | VCB20 | 197 | 198 | 197 | 698 | 101 |

- alle Prozessoren mit integriertem L2-Cache (außer K6-2 und Athlon)
- Performance weitgehend proportional zum Takt
- keine signifikanten Vorteile für Intel oder AMD

[c't 14/00 098]

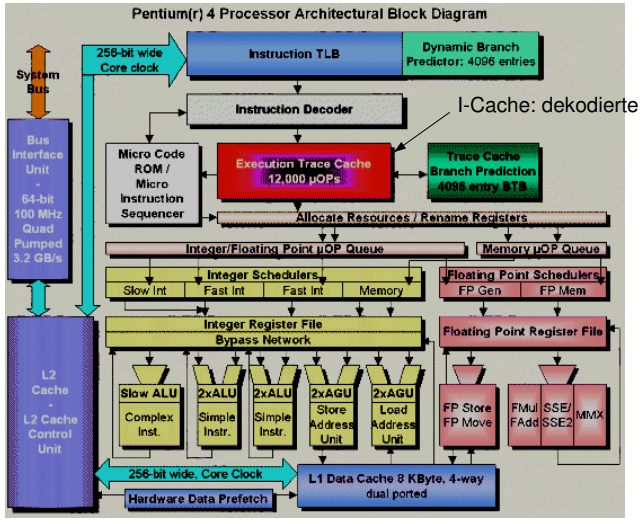


## Pentium III

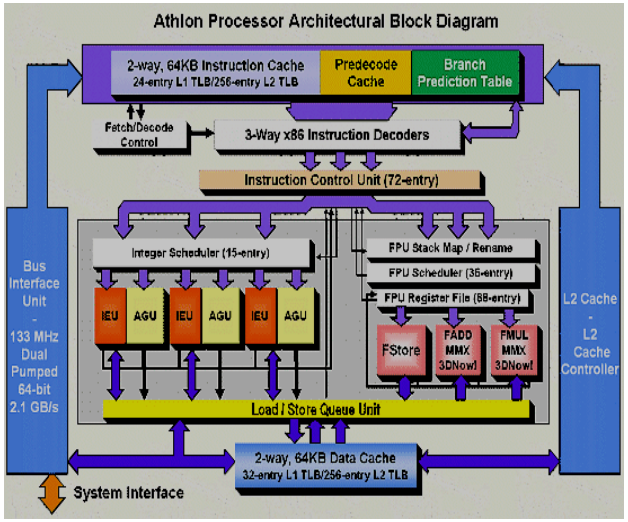





## Pentium IV

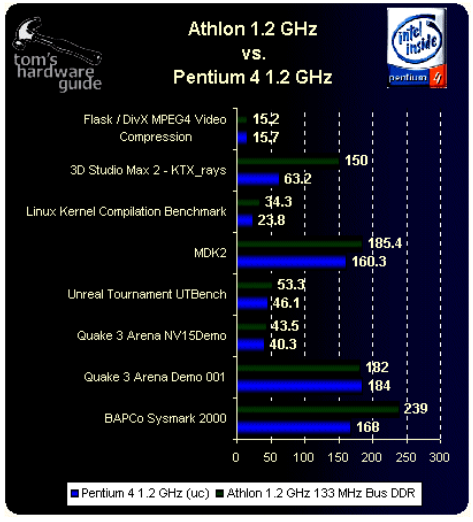


## Athlon (Thunderbird)



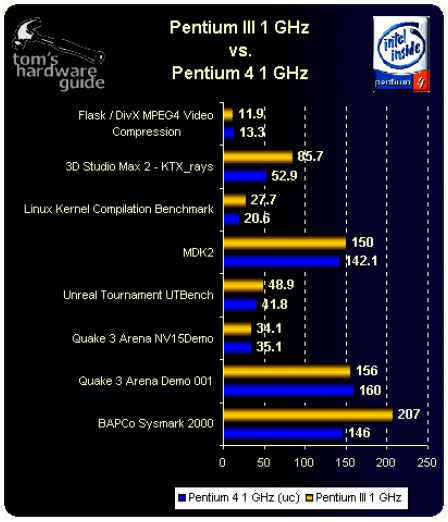



## Benchmarks: Pentium IV vs. Athlon



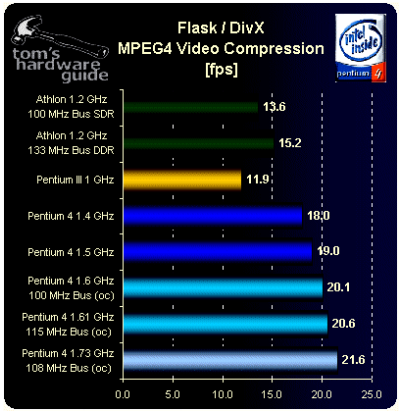


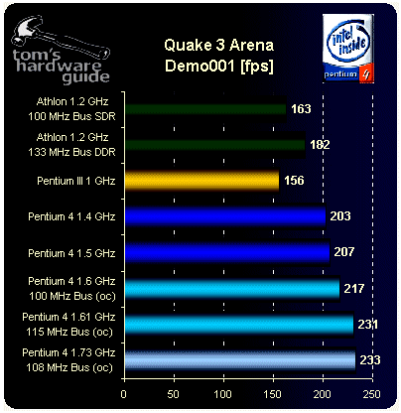

## Benchmarks: Pentium IV vs. Pentium III






## Benchmarks: DivX / Quake



# x86: Probleme der x86-Architektur

*"Insgesamt betrachtet, läßt sich die Lage der IA-32 mit dem Zustand der Himmelsmechanik kurz vor Kopernikus vergleichen. Die damalige Theorie, die Erde stünde fest verankert und bewegungslos im Raum, während die Planeten in Epizyklen um sie kreisen, beherrschte die Astronomie. Als jedoch die Beobachtungen immer besser wurden, kamen immer mehr Epizyklen dazu, bis das ganze Modell wegen seiner internen Komplexität in sich zusammenstürzte.*

*Intel befindet sich heute in einer ähnlichen Klemme..."*
*[Tanenbaum 99]*

Zukunft der x86-Architektur?!

=> noch eine Erweiterung: AMD x86-64 Architektur
=> sauberer Neubeginn: Intel IA-64 Itanium



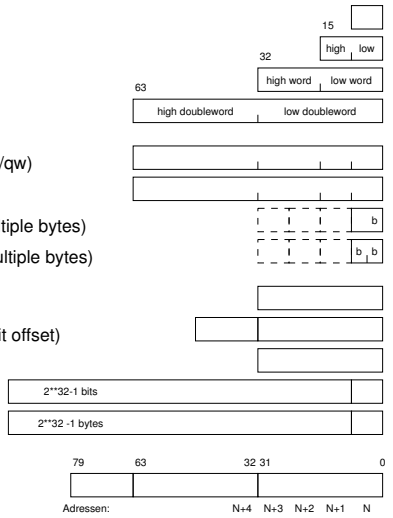
# x86: Datentypen: CISC . . .

bytes
word
doubleword
quadword

integer (2-complement b/w/dw/qw)
ordinal (unsigned b/w/dw/qw)
BCD (one digit per byte, multiple bytes)
packed BCD (two digits per byte, multiple bytes)

near pointer (32 bit offset)
far pointer (16 bit segment + 32 bit offset)
bit field
bit string
byte string

float / double / extended






# x86: Befehlssatz

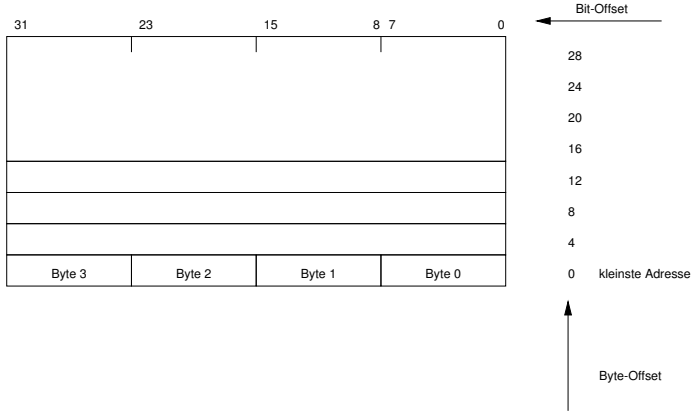
- Datenzugriff — mov, xchg
- Stack-Befehle — push, pusha, pop, popa
- Typumwandlung — cwd, cdq, cbw (byte->word), movsx, . . .
- Binärarithmetik — add, adc, inc, sub, sbb, dec, cmp, neg, . . . mul, imul, div, idiv,
- Dezimalarithmetik — packed / unpacked BCD: daa, das, aaa, aas, . . .
- Logikoperationen — and, or, xor, not, sal, shr, shr, . . .
- Sprungbefehle — jmp, call, ret, int, iret, loop, loopne, . . .

- String-Operationen — movs, cmps, scas, load, stos, . . .
- "high-level" — enter (create stack frame), . . .
- diverses — lahf (load AH from flags), . . .
- Segment-Register — far call, far ret, lds (load data pointer)

=> CISC — zusätzlich diverse Ausnahmen/Spezialfälle



# x86: Byteorder




- "little endian": LSB eines Wortes bei der kleinsten Adresse

# x86: Byteorder

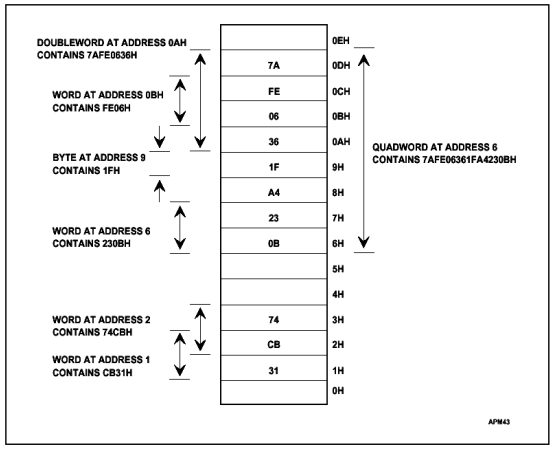



**Figure 3-3. Bytes, Words, Doublewords and Quadwords in Memory**

- Speicher ist voll byte-adressierbar



# x86: Befehlsformate: CISC . . .

außergewöhnlich komplexes Befehlsformat:

| | | |
|---|---|---|
| 1) | prefix | (repeat / segment override / etc.) |
| 2) | opcode | (eigentlicher Befehl) |
| 3) | register specifier | (Ziel / Quellregister) |
| 4) | address mode specifier | (diverse Varianten) |
| 5) | scale-index-base | (Speicheradressierung) |
| 6) | displacement | (Offset) |
| 7) | immediate operand | |

- ausser dem Opcode alle Bestandteile optional
- unterschiedliche Länge der Befehle, von 1 .. 37 Byte

=> extrem aufwendige Dekodierung



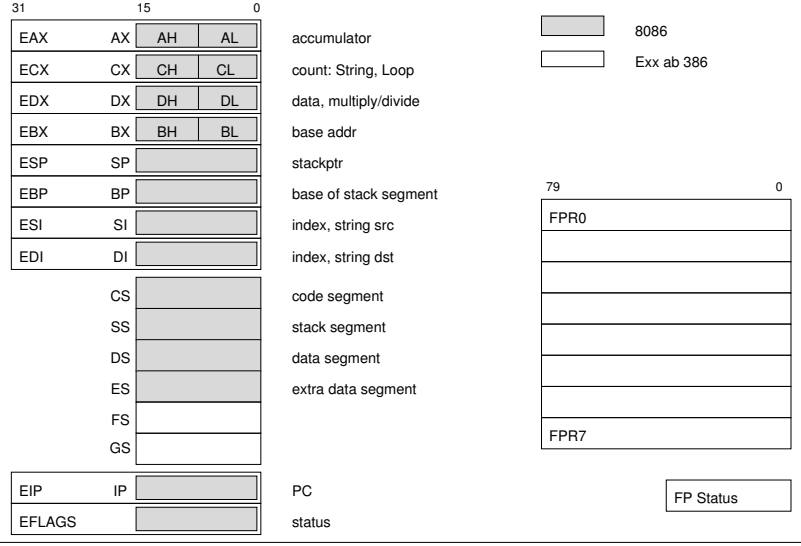
# x86: Modifier

alle Befehle können mit "Modifiern" ergänzt werden:

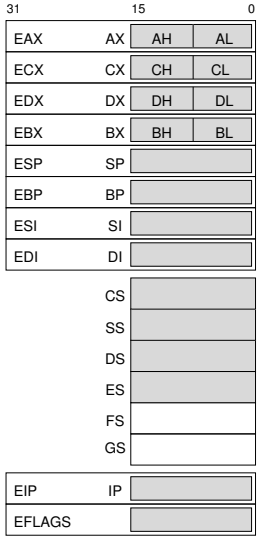
| | |
|---|---|
| • segment override | Addr. aus angewähltem Segmentregister |
| • address size | Umschaltung 16/32-bit |
| • operand size | Umschaltung 16/32-bit |
| • repeat | für Stringoperationen<br>Operation auf allen Elementen ausführen |
| • lock | Speicherschutz für Multiprozessoren |



# x86: Register



# x86: Register




- sehr wenig Register
- alle Register haben Spezialaufgaben
- aber EAX .. EDI auch als GP Register

- viele Speicherzugriffe
- komplexe Segmentadressierung

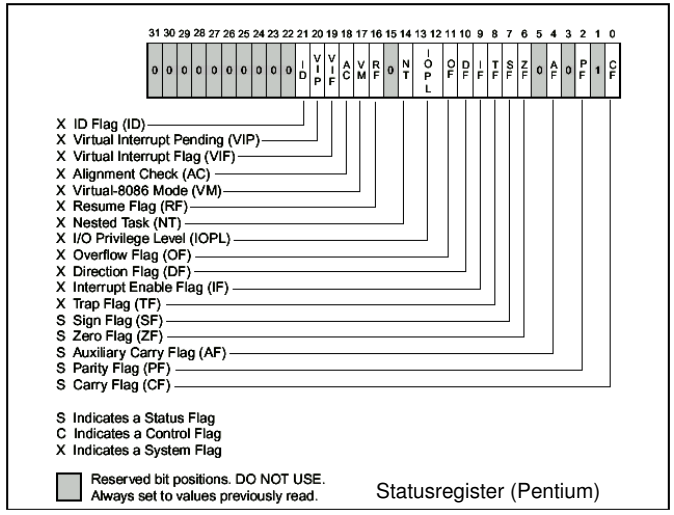
- FP-Register als Stack organisiert
- schwer optimierbar



Leerseite



# x86: EFLAGS Register




Statusregister (Pentium)



Leerseite

# x86: CISC: Vergleichsbefehle

**Table 4-3. Conditional Jump Instructions**

| Mnemonic | Flag States | Description |
|---|---|---|
| **Unsigned Conditional Jumps** | | |
| JA/JNBE | (CF or ZF)=0 | Above/not below nor equal |
| JAE/JNB | CF=0 | Above or equal/not below |
| JB/JNAE | CF=1 | Below/not above nor equal |
| JBE/JNA | (CF or ZF)=1 | Below or equal/not above |
| JC | CF=1 | Carry |
| JE/JZ | ZF=1 | Equal/zero |
| JNC | CF=0 | Not carry |
| JNE/JNZ | ZF=0 | Not equal/not zero |
| JNP/JPO | PF=0 | Not parity/parity odd |
| JP/JPE | PF=1 | Parity/parity even |
| **Signed Conditional Jumps** | | |
| JG/JNLE | ((SF xor OF) or ZF) =0 | Greater/not less nor equal |
| JGE/JNL | (SF xor OF)=0 | Greater or equal/not less |
| JL/JNGE | (SF xor OF)=1 | Less/not greater nor equal |
| JLE/JNG | ((SF xor OF) or ZF)=1 | Less or equal/not greater |
| JNO | OF=0 | Not overflow |
| JNS | SF=0 | Not sign (non-negative) |
| JO | OF=1 | Overflow |
| JS | SF=1 | Sign (negative) |



# x86: CISC: "enter" instruction

The ENTER instruction can be used in two ways: nested and non-nested. If the lexical level is 0, the non-nested form is used. The non-nested form pushes the contents of the EBP register on the stack, copies the contents of the ESP register into the EBP register, and subtracts the first operand from the contents of the ESP register to allocate dynamic storage. The non-nested form differs from the nested form in that no stack frame pointers are copied. The nested form of the ENTER instruction occurs when the second parameter (lexical level) is not zero.

The following pseudo code shows the formal definition of the ENTER instruction. STORAGE is the number of bytes of dynamic storage to allocate for local variables, and LEVEL is the lexical nesting level.

```
PUSH EBP;
FRAME_PTR ← ESP;
IF LEVEL > 0
    THEN
        DO (LEVEL – 1) times
            EBP ← EBP – 4;
            PUSH Pointer(EBP); (* doubleword pointed to by EBP *)
        OD;
    PUSH FRAME_PTR;
FI;
EBP ← FRAME_PTR;
ESP ← ESP – STORAGE;
```

The main procedure (in which all other procedures are nested) operates at the highest lexical level, level 1. The first procedure it calls operates at the next deeper lexical level, level 2. A level 2 procedure can access the variables of the main program, which are at fixed locations specified by the compiler. In the case of level 1, the ENTER instruction allocates only the requested dynamic storage on the stack because there is no previous display to copy.

- volle Stackverwaltung für geschachtelte Funktionsaufrufe :-)



# x86: Assembler-Beispiel

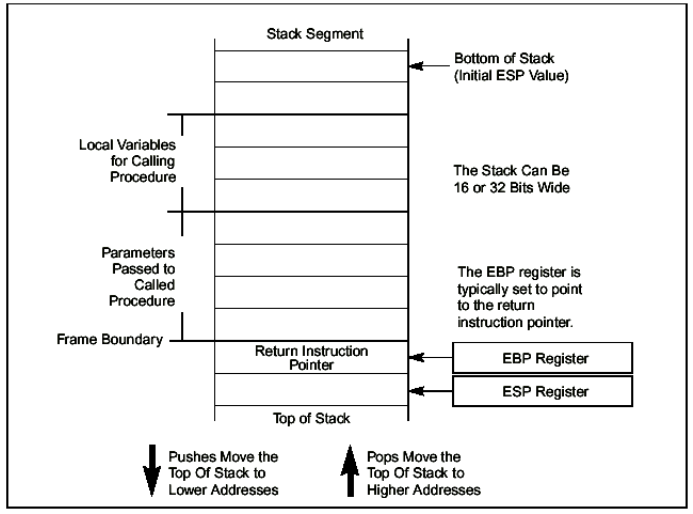
```
addr opcode    assembler                c quellcode
-------------------------------------------------------------------------------
               .file     "hello.c"
               .text
0000 48656C6C  .string  "Hello x86!\n"
     6F207838
     36210A00
               .text
               print:
0000 55           pushl %ebp            | void print( char* s ) {
0001 89E5         movl %esp,%ebp
0003 53           pushl %ebx
0004 8B5D08       movl 8(%ebp),%ebx
0007 803B00       cmpb $0,(%ebx)        |   while( *s != 0 ) {
000a 7418         je .L18
                  .align 4
               .L19:
000c A100000000   movl stdout,%eax      |     putc( *s, stdout );
0011 50           pushl %eax
0012 0FBE03       movsbl (%ebx),%eax
0015 50           pushl %eax
0016 E8FCFFFF     call _IO_putc
     FF
001b 43           incl %ebx             |     s++;
001c 83C408       addl $8,%esp          |   }
001f 803B00       cmpb $0,(%ebx)
0022 75E8         jne .L19
               .L18:
0024 8B5DFC       movl -4(%ebp),%ebx    | }
0027 89EC         movl %ebp,%esp
0029 5D           popl %ebp
002a C3           ret
```

# x86: Assembler-Beispiel (2)

```
addr opcode    assembler                c quellcode
-------------------------------------------------------------------------------
               .Lfe1:
               .Lscope0:
002b 908D7426  .align 16
     00
               main:
0030 55           pushl %ebp            | int main( int argc, char** argv ) {
0031 89E5         movl %esp,%ebp
0033 53           pushl %ebx

0034 BB00000000   movl $.LC0,%ebx       |   print( "Hello x86!\n" );
0039 803D0000     cmpb $0,.LC0
     000000
0040 741A         je .L26
0042 89F6         .align 4
               .L24:
0044 A100000000   movl stdout,%eax
0049 50           pushl %eax
004a 0FBE03       movsbl (%ebx),%eax
004d 50           pushl %eax
004e E8FCFFFFFF   call _IO_putc
0053 43           incl %ebx
0054 83C408       addl $8,%esp
0057 803B00       cmpb $0,(%ebx)
005a 75E8         jne .L24
               .L26:
005c 31C0         xorl %eax,%eax        |   return 0;
005e 8B5DFC       movl -4(%ebp),%ebx    | }
0061 89EC         movl %ebp,%esp
0063 5D           popl %ebp
0064 C3           ret
```

# x86: Stack-Layout






# x86: Stack-Verwaltung bei Interrupts

If no stack switch occurs, the processor does the following when calling an interrupt or exception handler (refer to Figure 4-5):

1. Pushes the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack.
2. Pushes an error code (if appropriate) on the stack.
3. Loads the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers, respectively.
4. If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
5. Begins execution of the handler procedure at the new privilege level.

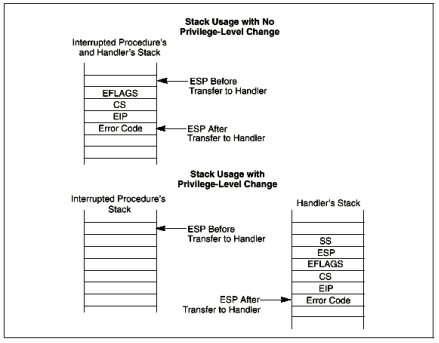
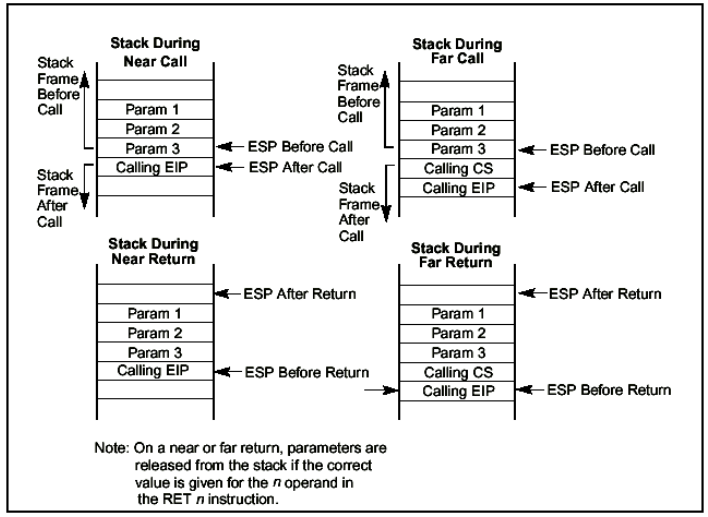



**Figure 4-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines**



# x86: Stack: near/far Calls






# x86: Adressierungsarten

displacement
base
base + displacement
(index*scale) + displacement
base + index + displacement
base + (index*scale) + displacement
immediate

| Segment | + | Base | + | (Index*Scale) | + | Displacement |
|---|---|---|---|---|---|---|
| CS | | EAX | | 1 | | none |
| SS | | ECX | | 2 | | 8-bit offset |
| DS | + | EDX | + | 4 | + | 32-bit offset |
| ES | | . . . | | 8 | | |
| FS | | ESP | | | | |
| GS | | EDI | | | | |

# x86: Modi

real mode: 8086+

- segmentierte Adressierung, kein Speicherschutz
- direkte Hardwarezugriffe, z.B. Interrupt-Vektoren

protected mode: 80286+

- Segmentdeskriptoren, Speicherschutz: Ring 0 .. 3
- Hardwareunterstützung für Multitasking, Call Gates, ...

enhanced mode: 80386+

- 32-bit Register und Operanden
- Segmentierung und Paging, MMU, ...

viritual 8086 mode: 80386+

- Adressierung / Zugriff wie 8086, aber anschliessend Paging



# x86: protected mode

protected mode := Speicherzugriff mit Gültigkeitsprüfung

- Segment-Adressierung (ab 286)
- vier (sechs) Segmentregister
- Adresse = Segment-Basisadresse + Offset im Segment
- Überprüfung von Segmentgrenzen und - rechten

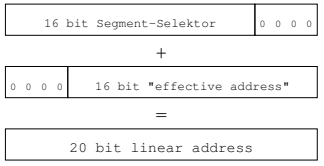
- Paging (ab 386)
- extrem flexibles Konzept für virtuellen Speicher
- Paging ist mit Segmentierung kombinierbar



# x86: real mode

real mode := Speicherkonzept des 8086 Prozessors

- 20-bit Adressen, aber nur 16-bit Register




Adressüberlauf möglich, wenn Segmentselektor zu groß:

- nur 20 Adressleitungen am 8086: wrap around
- ab 80286 Problem mit Adressen 100000h - 10FFFEh (A20 Gate)



# x86: Segment-Adressierung

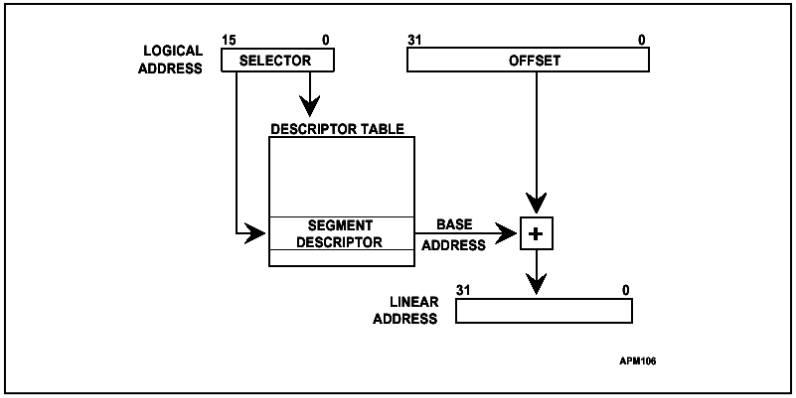



**Figure 11-5. Segment Translation**

- "far pointer": 16 bit Segment-Selektor, 32 bit Offset
- Deskriptortabelle enthält Basisadresse und Zugriffsrechte

# x86: Segment-Adressierung

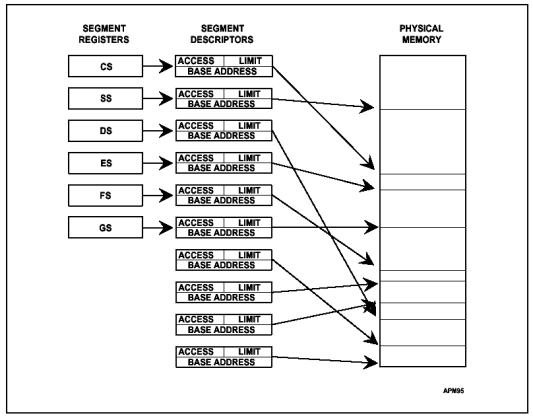



Figure 11-3. Multisegment Model

- sechs Segmentregister, bis zu 16383 Segmente a 4 GByte
- individuell einstellbare Zugriffsrechte pro Segment



# x86: Segmentdeskriptor

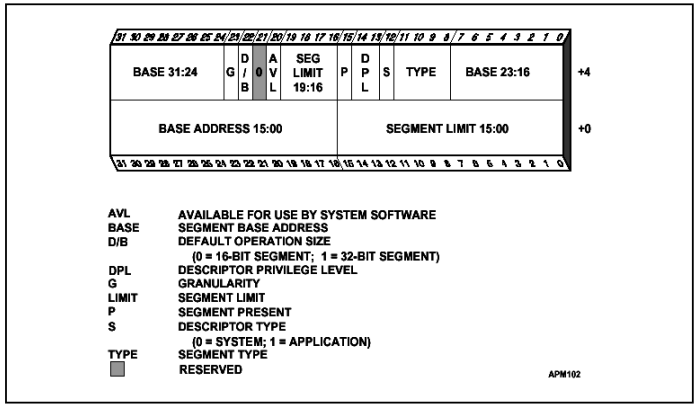



Figure 11-8. Segment Descriptors

- 32 bit Basisadresse und 20 bit Segmentlänge
- diverse Flags für Zugriffsrechte usw.



# x86: "flat addressing"

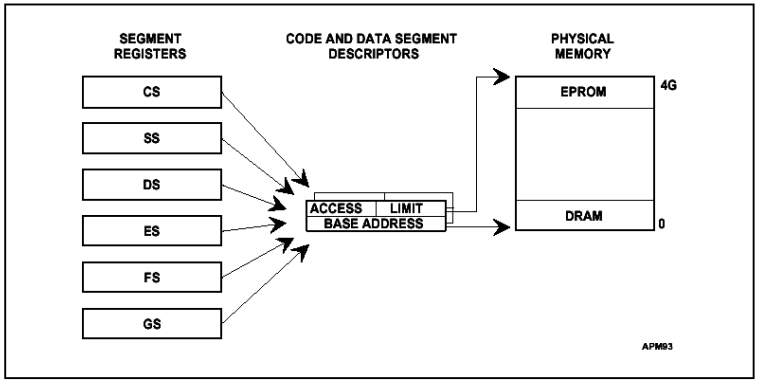



Figure 11-1. Flat Model

- alle Segmentregister enthalten dieselben Werte
- flacher 32-bit Adressraum (mit Range-Checks)



# x86: "protected flat addressing"

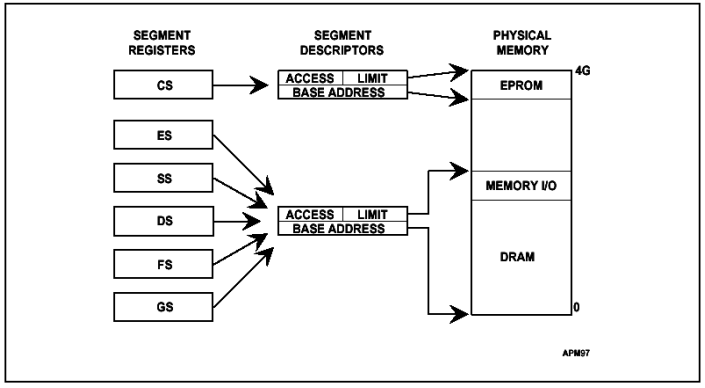



Figure 11-2. Protected Flat Model

- separates Code-Segment (evtl. mit Wraparound)
- kein Zusatzaufwand für Überprüfung von Speicher/Stackzugriffen

## x86: 32 vs. 16 bit Code ...

8086 aufwärts: "16-bit Code":

- segmentierte Adressierung (real- oder protected mode)
- 16-bit Arithmetik, kein Zugriff auf die erweiterten Register
- Segmentgröße maximal 64 KB
- daher Probleme mit Code/Daten größer 64 KB, insbesondere Arrays
- ständiges Neuladen der Segmentregister

ab 80386: "32-bit Code":

- Zugriff auf die vollen 32-bit EAX .. EDI Register
- EAX .. EDI auch als 8 Universalregister nutzbar
- 32-bit Arithmetik



## x86: 32 vs 16 bit Code: Addition

```
                          ; 32 bit add in 16 bit Code
mov ax, word ptr a        ; niederwertiger Teil von a
mov dx, word ptr a+2      ; höherwertiger Teil von a
add ax, word ptr b        ; addiere b, setzt Carry
adc dx, word ptr b+2      ; addiere mit Carry,
mov word ptr c, ax        ; niederwertiger Teil der Summe
mov word ptr c+2, dx      ; höherwertiger Teil der Summe


mov eax, a                ; 32 bit add in 32 bit Code
add eax, b
mov c, eax
```



## x86: 32 vs. 16 bit Code: Arrayzugriff

```
LONG countBlack( BYTE __huge *lpBits, LONG nbits ) {
   LONG result = 0; LONG i;
   for( i=0; i < nbits; ++i ) {
      if (!lpBits[i]) ++result;
   }
   return result;
}
```

```
   ; 16 bit Code für lpBits[i]
   mov ax, WORD PTR i
   mov dx, WORD PTR i+2
   mov cx, WORD PTR lpBits
   mov bx, WORD PTR lpBits+2
   add ax, cx
   adc dx, 0
   mov cx, OFFSET  __AHSHIFT ; Wert 3: (i/65536)*8
   shl dx, cl                ; Siehe Oney, Win95 Prog., S.99
   add dx, bx
   mov bx, ax
   mov es, dx                ; lädt Segmentregister ...
   mov al, BYTE PTR es:[bx]
```

```
; 32 bit Code für lpBits[i]:
mov eax, DWORD PTR i
mov ecx, DWORD PTR lpBits
xor edx, edx
mov dl, BYTE PTR [eax+ecx]
```



Leerseite

# x86: Segmente und Pages

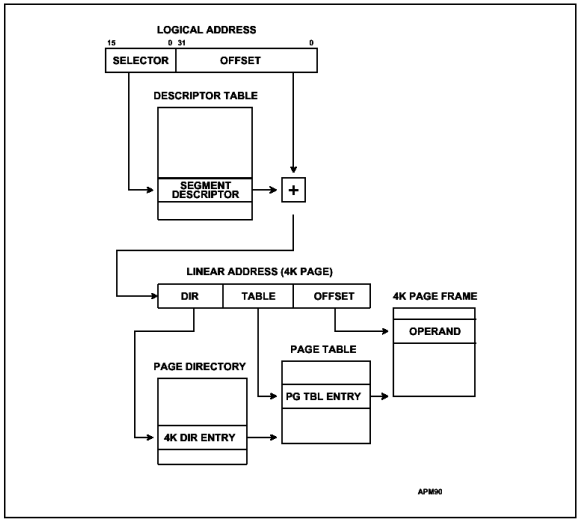



**Figure 11-16. Combined Segment and Page Address Translation**



# x86: Page Tables

### 11.3.3. Page Tables

A page table is an array of 32-bit entries. A page table is itself a page, and contains 4096 bytes of data or at most 1K 32-bit entries. Four kilobyte pages, including page directories and page tables, are aligned to 4K-byte boundaries. Two levels of tables are used to address a page of memory. At the highest level is a page directory. A page directory holds up to 1K entries that address page tables of the second level. A page table of the second level addresses up to 1K pages in physical memory. All the tables addressed by one page directory, therefore, can address 1M ($2^{20}$) four-Kbyte pages. If each page contains 4K ($2^{12}$) bytes, the tables of one page directory can span a linear address space of four gigabytes ($2^{20} \times 2^{12} = 2^{32}$). For information on support of page sizes larger than 4K, see Appendix H.

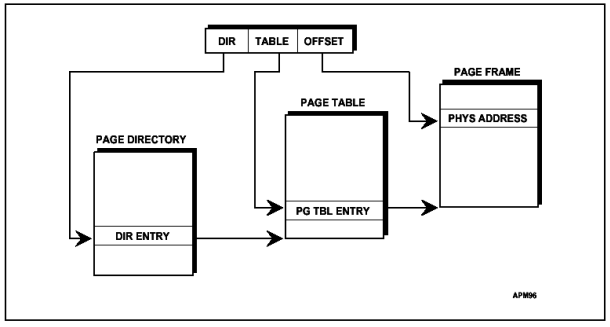
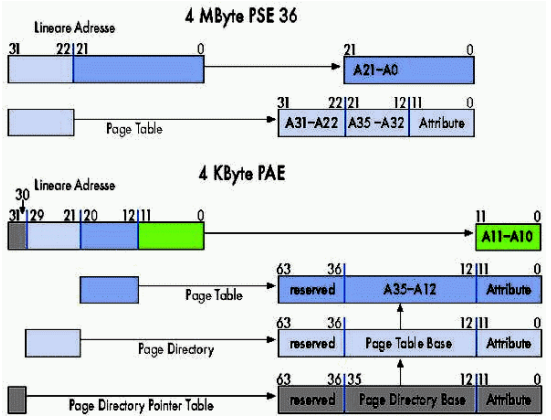



**Figure 11-13. Page Translation**



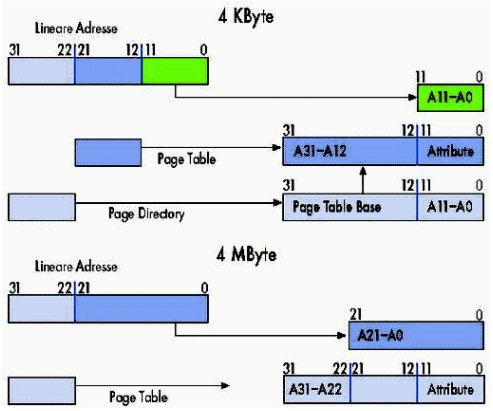
# x86: erweiterter Adressraum




- Erweiterung auf 64 bit physikalische Adressen
- weiterhin < 4 GB pro Task

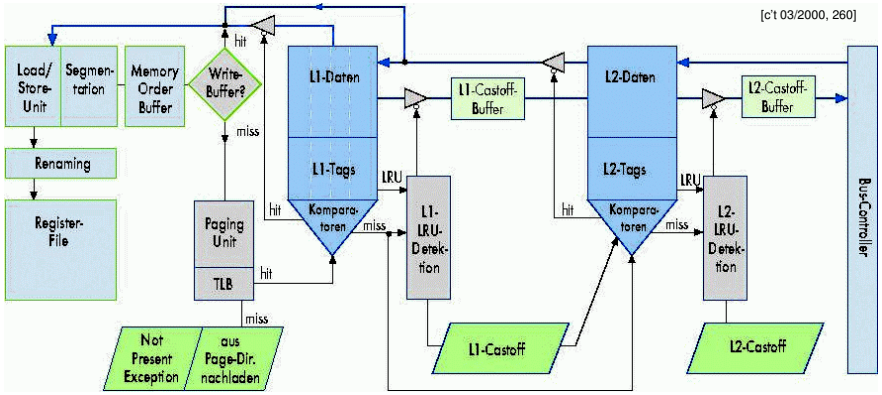


# x86: Pagesizes: 4 KB vs. 4 MB




- kleine Pages erlauben feine Granularität
- aber evtl. viele TLB-Misses
- große Pages günstig für Betriebssystem / Framebuffer

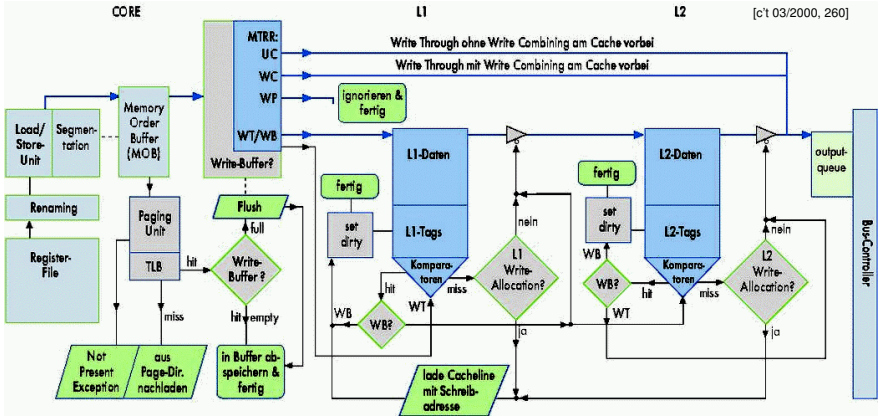
## x86: Pentium II Lesezugriff . . .




- Cachezugriffe: L1 typ. 1..2 Takte, L2 typ. 2..10 Takte
- Speicherzugriffe: ca. 100 Takte



## x86: Pentium II Schreibzugriff . . .




- MemoryTypeRangeRegister: schnelle I/O, z.B. Graphikkarte
- weitere Stufen (z.B. AGP GART) im Chipsatz ...



## x86: Deskriptor-Tabellen

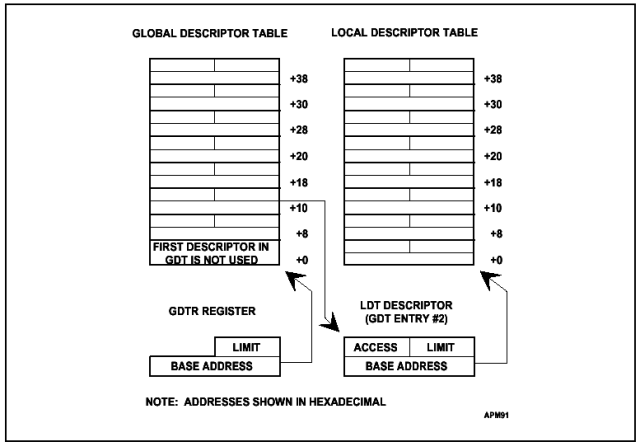
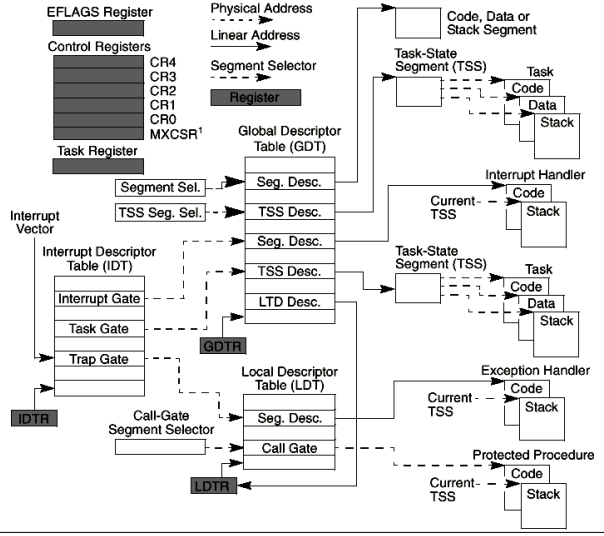



**Figure 11-10. Descriptor Tables**

- Prüfung von Zugriffsrechten und Adressgrenzen



## x86: Pentium Datenstrukturen



# x86: Protection Rings

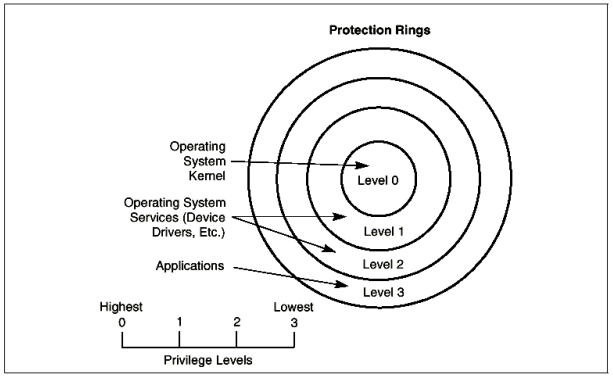



**Figure 4-3. Protection Rings**

- x86 unterstützt vier (!) getrennte Prioritätsstufen
- sehr feine Steuerung von Zugriffsrechten möglich



Leerseite



# x86: I/O permission mask

### 10.5.2. I/O Permission Bit Map

The I/O permission bit map is a device for permitting limited access to I/O ports by less privileged programs or tasks and for tasks operating in virtual-8086 mode. The I/O permission bit map is located in the TSS (refer to Figure 10-2) for the currently running task or program. The address of the first byte of the I/O permission bit map is given in the I/O map base address field of the TSS. The size of the I/O permission bit map and its location in the TSS are variable.

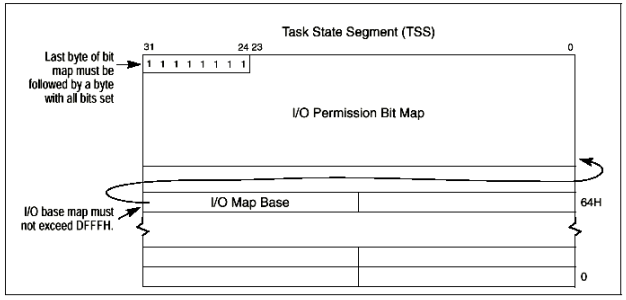



**Figure 10-2. I/O Permission Bit Map**

- Kontrolle der Zugriffsrechte für jede einzelne I/O-Adresse



Leerseite

# x86: INTn INT3 BOUND

### 4.4.4. INT *n*, INTO, INT 3, and BOUND Instructions

The INT *n*, INTO, INT 3, and BOUND instructions allow a program or task to explicitly call an interrupt or exception handler. The INT *n* instruction uses an interrupt vector as an argument, which allows a program to call any interrupt handler.

The INTO instruction explicitly calls the overflow exception (#OF) handler if the overflow flag (OF) in the EFLAGS register is set. The OF flag indicates overflow on arithmetic instructions, but it does not automatically raise an overflow exception. An overflow exception can only be raised explicitly in either of the following ways:

- Execute the INTO instruction.
- Test the OF flag and execute the INT *n* instruction with an argument of 4 (the vector number of the overflow exception) if the flag is set.

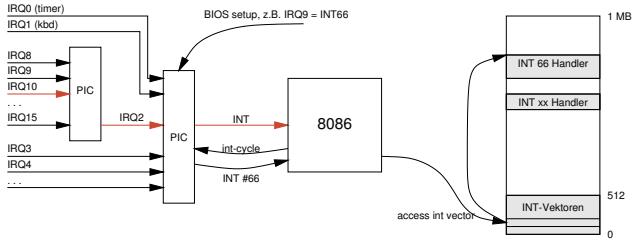
Both the methods of dealing with overflow conditions allow a program to test for overflow at specific places in the instruction stream.

The INT 3 instruction explicitly calls the breakpoint exception (#BP) handler.

- wichtigster Mechanismus zum Aufruf von OS-Funktionen



# x86: Interrupts im real-mode




```
void INT_66_handler() {
  save_registers_to_stack();
  read_master_PIC();
  if (master_PIC_active) { // hardware interrupt
    read_slave_PIC();      // but which one?
    switch( slave_PIC ) {
      case slave_IRQ8: // handle_RTC_interrupt

      case slave_IRQ9: // handle s/w int a0h

      case slave_IRQ10: // free

    }
    reset_slave_PIC();
    reset_maser_PIC()
   }
  else { // software interrupt 66

  }
  restore_registers;
  IRET;
}
```

- BIOS programmiert den PIC 8259
- Umsetzung IRQ auf INT-Nummern
- INT-Vektoren ab Adresse 0



# x86: Interrupt/exception vectors

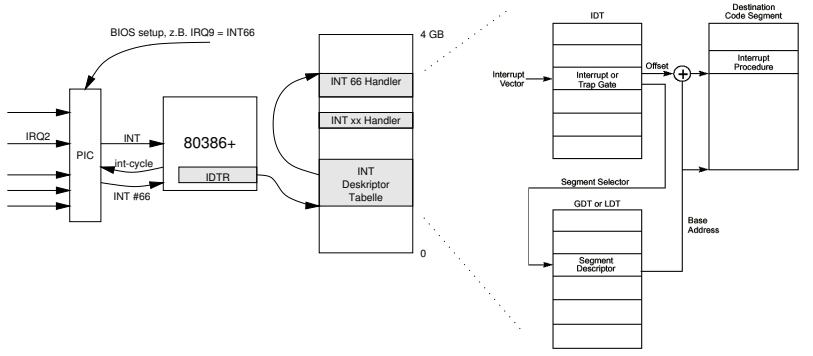
**Table 4-1. Exceptions and Interrupts**

| Vector No. | Mnemonic | Description | Source |
|---|---|---|---|
| 0 | #DE | Divide Error | DIV and IDIV instructions. |
| 1 | #DB | Debug | Any code or data reference. |
| 2 | | NMI Interrupt | Non-maskable external interrupt. |
| 3 | #BP | Breakpoint | INT 3 instruction. |
| 4 | #OF | Overflow | INTO instruction. |
| 5 | #BR | BOUND Range Exceeded | BOUND instruction. |
| 6 | #UD | Invalid Opcode (UnDefined Opcode) | UD2 instruction or reserved opcode.[1] |
| 7 | #NM | Device Not Available (No Math Coprocessor) | Floating-point or WAIT/FWAIT instruction. |
| 8 | #DF | Double Fault | Any instruction that can generate an exception, an NMI, or an INTR. |
| 9 | | CoProcessor Segment Overrun (reserved) | Floating-point instruction.[2] |
| 10 | #TS | Invalid TSS | Task switch or TSS access. |
| 11 | #NP | Segment Not Present | Loading segment registers or accessing system segments. |
| 12 | #SS | Stack Segment Fault | Stack operations and SS register loads. |
| 13 | #GP | General Protection | Any memory reference and other protection checks. |
| 14 | #PF | Page Fault | Any memory reference. |
| 15 | | (Intel reserved. Do not use.) | |
| 16 | #MF | Floating-Point Error (Math Fault) | Floating-point or WAIT/FWAIT instruction. |
| 17 | #AC | Alignment Check | Any data reference in memory.[3] |
| 18 | #MC | Machine Check | Error codes (if any) and source are model dependent.[4] |
| 19 | #XF | Streaming SIMD Extensions | SIMD floating-point numeric exceptions.[5] |
| 20-31 | | (Intel reserved. Do not use.) | |
| 32-255 | | Maskable Interrupts | External interrupt from INTR pin or INT *n* instruction. |

1. The UD2 instruction was introduced in the Pentium® Pro processor.
2. IA processors after the Intel386™ processor do not generate this exception.
3. This exception was introduced in the Intel486™ processor.
4. This exception was introduced in the Pentium® processor and enhanced in the Pentium® Pro processor.
5. This exception was introduced in the Pentium® III processor.



# x86: Interrupts im enhanced mode




- Register IDTR: Basisadresse + Größe der Deskriptortabelle
- spezielle Befehle LIDT / SIDT
- Interrupt Deskriptortabelle irgendwo im Hauptspeicher
- mehrere Tabellen möglich: Umladen von IDTR

## DOS: Tastaturzugriff

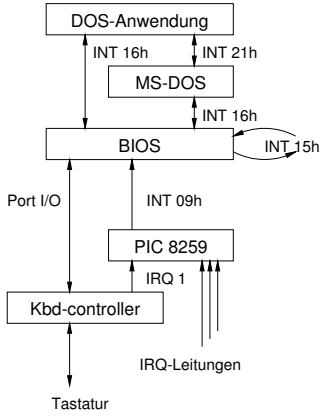
direkter Zugriff auf alle Geräte:

- real-mode Adressierung
- kein Speicherschutz

- direkte I/O Portzugriffe
- Interruptcontroller 8259
- DMA-Controller, ...

direkte BIOS-Aufrufe

=> Multitasking sehr problematisch
=> "Virtual 8086 Mode"






## Win9X: "virtuelle Hardware"

virtuelle Maschinen für

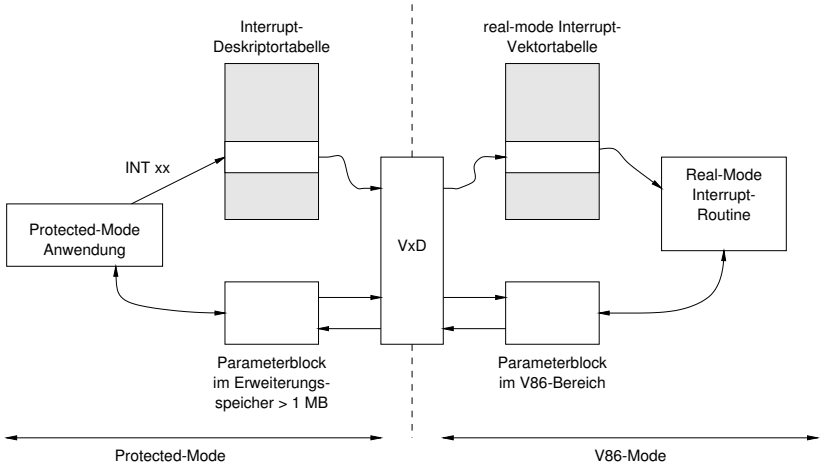
- DOS-Anwendungen
- (veraltete) real-mode Treiber

- Abfangen aller direkten I/O-Hardwarezugriffe
- Überprüfung, ob Zugriff zulässig
- wenn ja, Zugriff ausführen

- nutzt 386+ Virtual 8086 Mode
- real-mode Adressierung (20 bit) plus Paging (32 bit)



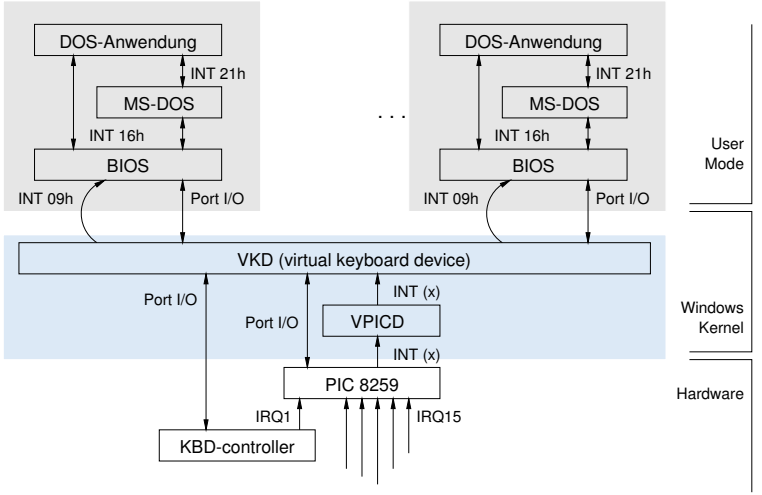
## Win9x: "Software-Virtualisierung"



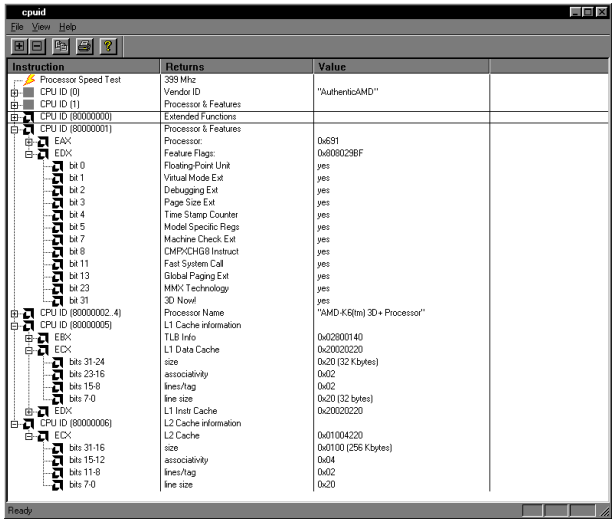

- Interrupt abfangen, Register anpassen, Mode umschalten, ...



## Win9X: virtueller Tastaturzugriff



# x86: CPUID, Beispiel AMD K6-III)






# x86: Appendix H

## APPENDIX H
## ADVANCED FEATURES

Some non-essential information regarding the Pentium processor are considered Intel confidential and proprietary and have not been documented in this publication. This information is provided in the *Supplement to the Pentium® Processor Developer's Manual* and is available with the appropriate non-disclosure agreements in place. Please contact Intel Corporation for details.

The *Supplement to the Pentium® Processor Developer's Manual* contains Intel confidential information on architecture extensions to the Pentium processor which are non-essential for standard applications. This includes low-level registers that provide access to such features as page size extensions, virtual mode extensions, testing and performance monitoring.

This information is specifically targeted at writers of the following types of software:

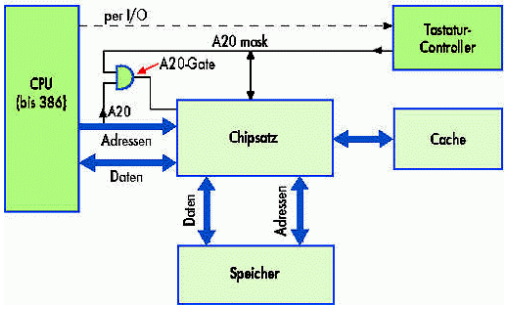
- Operating system kernels
- Virtual memory managers
- BIOS software

If you are writing software that does not fall into one of these categories, this information is non-essential and all required programming details are contained in the publicly available *Pentium® Processor Developer's Manual,* three-volume set.

- Details zum Pentium: nur per non-disclosure agreement (NDA)



# x86: A20-Gate . . .




- 8086 Bug: Überlauf der real-mode Adressen möglich, z.B.:
  0ffffh + ffff0h = 10fffeh, aber nur 20 Adressleitungen: x0fffeh
- von Microsoft für DOS-Funktionen benutzt :-(
- Spezialbehandlung des Adressbereichs in Chipsatz oder Prozessor
- Umschaltung mühsam über Tastaturcontroller-Interrupt. . .



Leerseite

## RISC vs. CISC: Motivation

- Rechner möglichst schnell, klein, sparsam, ..., aber billig
- sehr vielfältige Lösungen möglich
- Modeerscheinungen - z.B. "high-level instruction sets"

=> Rechnerarchitektur ist eine "Kunst"
=> gute Lösungen abhängig von HW/SW-Technologie

Ausgangsbasis für CISC: VAX, x86, 68000, ...

- Assemblerprogrammierung, schlechte Compiler
- Microcode schneller als Hauptspeicher
- Hardware für Rechenwerke vergleichsweise teuer

=> viele spezielle Maschinenbefehle
=> einmal eingeführte Befehle müssen später mitgeschleppt werden



## RISC: die IBM 801

John Cocke et.al., IBM, 1975:

- warum CISC? Cache-Zugriffe genauso schnell wie Microcode...

=> Compiler-geeignete Rechnerarchitektur
=> ausschließlich Hochsprache "PL.8": auch für OS und Treiber

=> nur wenige, reguläre Maschinenbefehle
=> aber diese schnell: Pipeline, CPI ~= 1
=> separate I/D-Caches
=> 32 Universal-Register

- 801 vs. S/370: 801 in allen Aspekten besser
- sehr guter Compiler
- wenig publiziert



## 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<br>• Pipeline<br>• viele Register, weniger Speicherzugriffe<br>• gut optimierende Compiler<br>• Caches, insbesondere I-Cache | auch für CISC möglich! |



## 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)

## *Loop: Instruction Scheduling*

HW/SW-Interaktion auf Prozessoren mit Pipeline:

- Daten/Kontrollabhängigkeiten
- Wartezyklen (stalls), bis Vorgängerstufen fertig

=> sinnvolle Anordnung der Befehle notwendig
=> große Bedeutung optimierender Compiler

- Beispiel-Latenzen: DLX single-issue RISC aus [H&P]

| instruction producing | instruction using result | latency [clocks] |
|---|---|---|
| FP ALU op. | FP ALU op. | 3 |
| FP ALU op. | FP STORE | 2 |
| FP LOAD | FP ALU op. | 1 |
| FP LOAD | FP STORE | 0 |

```
ld R3, R2(0)
; wait 1
add R4, R4, R3
; wait 3
add R5, R5, R4
```



## *Loop: ohne Scheduling*

```
Loop:  LD      F0,0(R1)     ; F0 = array element
       ADDD    F4,F0,F2     ; add scalar in F2
       SD      0(R1),F4     ; store result
       SUBI    R1,R1,#8     ; decrement pointer
       BNEZ    R1, Loop     ; branch R1!=zero
```

- Ausführung auf der Pipeline:

```
Loop:  LD      F0, 0(R1)    ; 1  (F0 laden)
       stall               ; 2
       ADDD    F4,F0,F2     ; 3  (F0 geladen)
       stall               ; 4
       stall               ; 5
       SD      0(R1),F4     ; 6  (F4 fertig)
       SUBI    R1,R1,#8     ; 7
       BNEZ    R1, Loop     ; 8
       stall               ; 9
```

- 9 Takte / Iteration

```
int i; double s, x[];
...
for( i=1; i<=1000; i++) {
  x[i] = x[i] + s;
}
...
```



## *Loop: Vektor = Vektor + Skalar*

- typisches Programmbeispiel: Vektor = Vektor + Skalar

```
int i; double s, x[];
...
for( i=1; i<=1000; i++) {
  x[i] = x[i] + s;
}
...
```

- nicht optimierter Code (am Beispiel DLX):

```
Loop:  LD      F0, 0(R1)    ; F0 = array element
       ADDD    F4,F0,F2     ; add scalar in F2
       SD      0(R1),F4     ; store result
       SUBI    R1,R1,8      ; decrement pointer
       BNEZ    R1, Loop     ; branch R1!=zero
```



## *Loop: mit Scheduling*

```
Loop:  LD      F0, 0(R1)    ; 1
       stall               ; 2
       ADDD    F4,F0,F2     ; 3
       stall               ; 4
       stall               ; 5
       SD      0(R1),F4     ; 6
       SUBI    R1,R1,#8     ; 7
       BNEZ    R1, Loop     ; 8
       stall               ; 9
```

- Ausnutzen des "branch delay slot": 6 Takte / Iteration

```
Loop:  LD      F0, 0(R1)    ; 1
       stall               ; 2
       ADDD    F4,F0,F2     ; 3
       SUBI    R1,R1,#8     ; 4
       BNEZ    R1, Loop     ; 5
       SD      8(R1),F4     ; 6
```

offset geändert!

```
int i; double s, x[];
...
for( i=1; i<=1000; i++) {
  x[i] = x[i] + s;
}
...
```

## Loop: Unrolling

```
; solange R1 >= 3:
;
Loop:  LD      F0, 0(R1)     ; element 0
       ADDD    F4,F0,F2      ;
       SD      0(R1),F4      ;

       LD      F6,-8(R1)     ; element 1
       ADDD    F8,F6,F2
       SD      -8(R1),F8

       LD      F10,-16(R1)   ; element 2
       ADDD    F12,F10,F2
       SD      -16(R1),F12

       LD      F14,-24(R1)   ; element 3
       ADD     F16,F14,F2
       SD      -24(R1),F16

       SUBI    R1,R1,#32     ;
       BNEZ    R1, Loop      ;
```

```
int i; double s, x[];
...
for( i=1; i<=1000; i++) {
  x[i] = x[i] + s;
}
...
```

- noch kein Scheduling
- 6.8 Takte / Iteration



## Loop: Diskussion

- optimierte Loop 3X schneller
- guter Compiler essentiell

aber:

- Optimierungen/Compiler nicht trivial
- maschinenspezifisch wegen Latenzen/Abhängigkeiten

- Loop-Unrolling erfordert viele Register
- erst recht für superskalare Maschinen

x86 hat zu wenig Register:

=> Compiler kann nicht optimieren

=> Register-Renaming / Tomasulo's Algorithmus



## Loop: Unrolling, mit Scheduling

```
; solange R1 >= 3:
;
Loop:  LD      F0, 0(R1)     ; element 0
       LD      F6,-8(R1)     ; element 1
       LD      F10,-16(R1)   ; element 2
       LD      F14,-24(R1)   ; element 3

       ADDD    F4,F0,F2
       ADDD    F8,F6,F2
       ADDD    F12,F10,F2
       ADDD    F16,F14,F2

       SD      0(R1),F4
       SD      -8(R1),F8
       SD      -16(R1),F12
       SUBI    R1,R1,#32     ;
       BNEZ    R1, Loop      ;
       SD      8(R1),F16     ; 8-32 = -24
```

jeweils drei Takte Latenz eingehalten, aber Pipeline läuft weiter!

```
int i; double s, x[];
...
for( i=1; i<=1000; i++) {
  x[i] = x[i] + s;
}
...
```

- 3.5 Takte / Iteration
- dreimal schneller als "triviale" Version!



## Loop: Register Renaming

```
; solange R1 >= 3:
;
Loop:  LD      F0, 0(R1)     ; nur F0,F2,F4 verfügbar:
       ADDD    F4,F0,F2      ;
       SD      0(R1),F4      ; => zusätzliche Abhängigkeiten

       LD      F0,-8(R1)     ;
       ADDD    F4,F0,F2      ;
       SD      -8(R1),F4     ;

       LD      F0,-16(R1)    ;
       ADDD    F4,F0,F2      ;
       SD      -16(R1),F4    ;

       LD      F0,-24(R1)    ;
       ADDD    F4,F0,F2      ;
       SD      24(R1),F4     ;
```

- x86-Compiler hat nicht genug Register zur Auswahl
- viele zusätzliche "Name-Dependencies"

=> "Register Renaming" zur Laufzeit im Prozessor (!)

=> ~100 Register mit "Scoreboard" zur Kontrolle der Abhängigkeiten
Athlon: bis zu 72 Befehle aktiv ...

## *superskalar: Register Renaming*

```
...
R3 = R0 * R1
R4 = R0 + R2
R5 = R0 + R1
R6 = R1 + R4
R7 = R1 * R2
R1 = R0 - R2
R3 = R3 * R1
R1 = R4 + R4
...
```

```
...
R7 = R1 * R2
S1 = R0 - R2
R3 = R3 * S1
R1 = R4 + R4
...
```

```
...
R7 = R1 * R2
R1 = R4 + R4
...
```

```
S1 = R0 - R2
R3 = R3 * S1
```

- Compiler darf nur "definierte" Register verwenden
- auch für Zwischenergebnisse
- dadurch zusätzliche, unnötige RAW/WAR/WAW-Konflikte

- Auflösen der Konflikte durch Einsatz "interner" Register
- Verwaltung automatisch durch den Prozessor: Scoreboard



## *superskalare Prozessoren: Scoreboard*

| | | | | | Gelesene Register | | | | | | | | Beschriebene Register | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Zy | # | Dekodiert | Iss | Ret | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 1 | R3=R0 * R1 | 1 | | 1 | 1 | | | | | | | | | | 1 | | | | |
| | 2 | R4=R0 + R2 | 2 | | 2 | 1 | 1 | | | | | | | | | 1 | 1 | | | |
| 2 | 3 | R5=R0 + R1 | 3 | | 3 | 2 | 1 | | | | | | | | | 1 | 1 | 1 | | |
| | 4 | R6=R1 + R4 | – | | 3 | 2 | 1 | | | | | | | | | 1 | 1 | 1 | | |
| 3 | 5 | R7=R1 * R2 | 5 | | 3 | 3 | 2 | | | | | | | | | 1 | 1 | 1 | | 1 |
| | 6 | S1=R0 – R2 | 6 | | 4 | 3 | 3 | | | | | | | | | 1 | 1 | 1 | | 1 |
| | | | | 2 | 3 | 3 | 2 | | | | | | | | | 1 | | 1 | | 1 |
| 4 | | | 4 | | 3 | 4 | 2 | | 1 | | | | | | | 1 | | 1 | 1 | 1 |
| | 7 | R3=R3 * S1 | – | | 3 | 4 | 2 | | 1 | | | | | | | 1 | | 1 | 1 | 1 |
| | 8 | S2=R4 + R4 | 8 | | 3 | 4 | 2 | | 3 | | | | | | | 1 | | 1 | 1 | 1 |
| | | | | 1 | 2 | 3 | 2 | | 3 | | | | | | | | | 1 | 1 | 1 |
| | | | | 3 | 1 | 2 | 2 | | 3 | | | | | | | | | | 1 | 1 |
| 5 | | | | 6 | | 2 | 1 | | 3 | | | | | 1 | | | | | 1 | 1 |
| 6 | | | 7 | | | 2 | 1 | 1 | 3 | | | | | 1 | | 1 | | | 1 | 1 |
| | | | | 4 | | 1 | 1 | 1 | 2 | | | | | 1 | | 1 | | | | 1 |
| | | | | 5 | | | | 1 | 2 | | | | | 1 | | 1 | | | | |
| | | | | 8 | | | | 1 | | | | | | | | 1 | | | | |
| 7 | | | | | | | | 1 | | | | | | | | 1 | | | | |
| 8 | | | | | | | | 1 | | | | | | | | 1 | | | | |
| 9 | | | | 7 | | | | | | | | | | | | | | | | |

*Abb. 4.44: Operation einer superskalaren CPU mit Ausgabe und Fertigstellung von Instruktionen außer der Reihe*

[Tanenbaum 99]

Befehlsausführung
superskalar,
out-of-order execution
(9 Takte)



## *superskalare Prozessoren: Scoreboard*

| | | | | | Gelesene Register | | | | | | | | Beschriebene Register | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Zy | # | Dekodiert | Iss | Ret | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 1 | R3=R0 * R1 | 1 | | 1 | 1 | | | | | | | | | | 1 | | | | |
| | 2 | R4=R0 + R2 | 2 | | 2 | 1 | 1 | | | | | | | | | 1 | 1 | | | |
| 2 | 3 | R5=R0 + R1 | 3 | | 3 | 2 | 1 | | | | | | | | | 1 | 1 | 1 | | |
| | 4 | R6=R1 + R4 | | | 3 | 2 | 1 | | | | | | | | | 1 | 1 | 1 | | |
| 3 | | | | | 3 | 2 | 1 | | | | | | | | | 1 | 1 | 1 | | |
| 4 | | | | 1 | 2 | 1 | 1 | | | | | | | | | | | 1 | 1 | |
| | | | | 2 | 1 | 1 | | | | | | | | | | | | | 1 | |
| | | | | 3 | | | | | | | | | | | | | | | | |
| 5 | | | 4 | | | 1 | | | 1 | | | | | | | | | | 1 | 1 |
| | 5 | R7=R1 * R2 | 5 | | | 2 | 1 | | 1 | | | | | | | | | | 1 | 1 |
| 6 | 6 | R1=R0 – R2 | – | | | 2 | 1 | | 1 | | | | | | | | | | 1 | 1 |
| 7 | | | | 4 | | 1 | 1 | | | | | | | | | | | | | 1 |
| 8 | | | | 5 | | | | | | | | | | | | | | | | |
| 9 | | | 6 | | 1 | | 1 | | | | | | | 1 | | | | | | |
| | 7 | R3=R3 * R1 | 7 | | 1 | 1 | 1 | 1 | | | | | | 1 | | 1 | | | | |
| 10 | | | | | 1 | 1 | 1 | 1 | | | | | | 1 | | 1 | | | | |
| 11 | | | | 6 | | 1 | | 1 | | | | | | | | 1 | | | | |
| 12 | | | | 7 | | | | | | | | | | | | | | | | |
| 13 | 8 | R1=R4 + R4 | 8 | | | | | | 2 | | | | | 1 | | | | | | |
| 14 | | | | | | | | | 2 | | | | | 1 | | | | | | |
| 15 | | | | 8 | | | | | | | | | | | | | | | | |

*Abb. 4.43: Operation einer superskalaren CPU mit Ausgabe und Fertigstellung von Instruktionen entsprechend ihrer Reihenfolge*

[Tanenbaum 99]

Befehlsausführung
superskalar,
in-order execution
(15 Takte)



## *superskalare Prozessoren*

VLSI-Technologie erlaubt immer mehr Transistoren/Chip

- größere Caches?
- komplexere Prozessoren?

| | | CPI |
|---|---|---|
| • klassischer CISC: | serieller Befehlszyklus | 5 .. 15 |
| • einfacher RISC: | Pipeline, 1 Befehl/Takt | ~ 1 |
| • superskalar: | mehrere Befehle/Takt | < 1 |

=> I-Cache muss mehrere Befehle pro Takt liefern
=> Daten/Kontrollabhängigkeiten berücksichtigen
=> Ressourcen-Konflikte, Scoreboarding
=> extreme Komplexität

=> Speicherzugriffe sind das Nadelöhr:
1 GHz, 4 Befehle/Takt, 100 ns Latenz: 400 Befehle idle

## *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 | • | • |

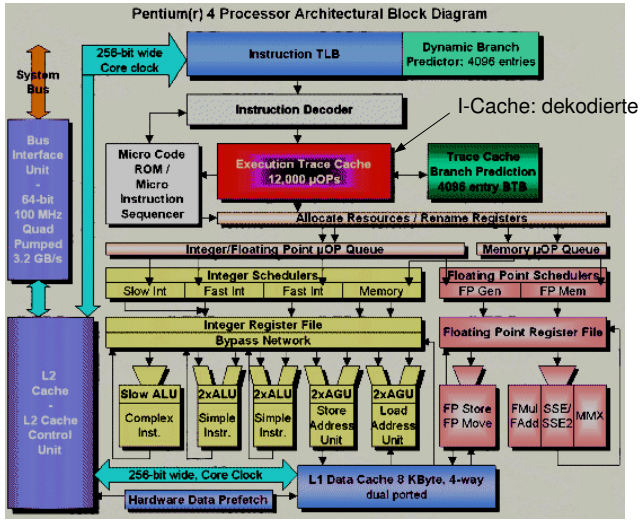
=> extreme Komplexität für RISC und CISC

- 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"

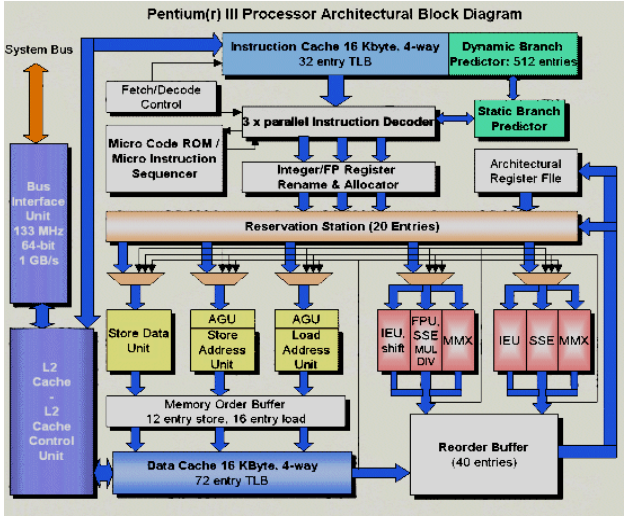


## *Pentium IV*



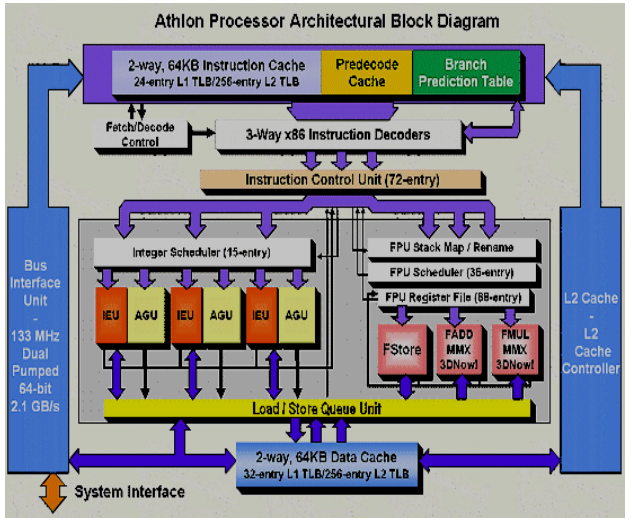



## *Pentium III*






## *Athlon (Thunderbird)*

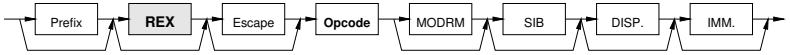


## x86: AMD x86-64

- 64-bit Erweiterung der IA-32 [www.amd.com]
- voll abwärtskompatibel
- gute Performance für 32-bit Applikationen

- 64-bit Register und Programmzähler
- flacher 64-bit Adressraum
- 8 zusätzliche Universalregister
- 8 zusätzliche ISSE-Register, ISSE2 Funktionen

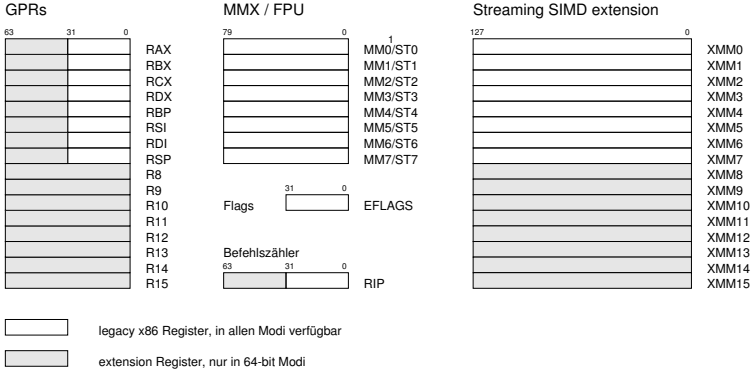
- diverse Betriebsmodi (64-bit / compatibility-64 / legacy-32)
- Trick: neuer Befehlsprefix "REX" für die 64-bit Befehle






## x86: AMD x86-64

- fast doppelte Anzahl der Register gegenüber IA-32
- erstmals wirkliche Universalregister






## x86: AMD x86-64: Modi

- 64-bit mode: alle 64-bit Erweiterungen
- compatibility mode: für 16/32-bit Applikationen unter 64-bit OS
- legacy mode: Pentium-kompatibel

| Mode | | OS required | App. rcompile | #adress bits | operand size | register extensions | GPR width |
|---|---|---|---|---|---|---|---|
| long mode | 64-bit mode | new 64-bit OS | yes | 64 | 32 | yes | 64 |
| | compatibiliy mode | | no | 32 | 32 | no | 32 |
| | | | | 16 | | | |
| legacy mode | | legacy 32/16-bit OS | no | 32 | 32 | no | 32 |
| | | | | 16 | 16 | | |



## x86: Intel IA-64

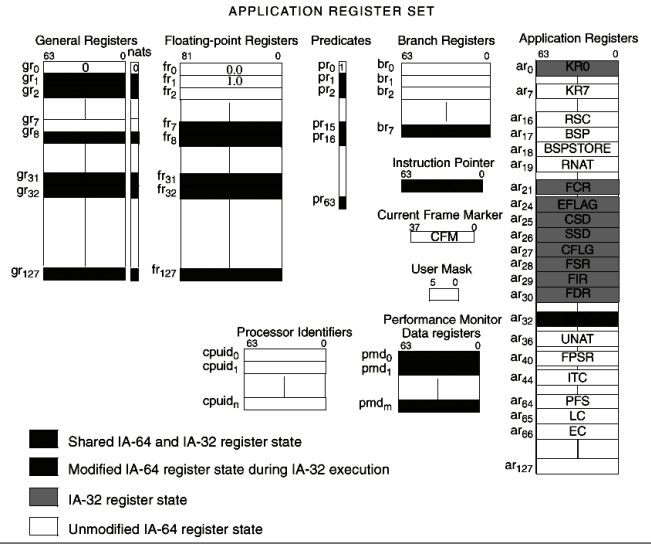
- völlig neue 64-bit Architektur
- basiert auf 64-bit RISC, vor allem HP PA-RISC

- Load/Store Architektur, 64-bit Register, 64-bit Adressen
- ein Befehlsformat: 41 bit mit Opcode und 3-Registeradressen
- viele parallele Funktionseinheiten
- sehr viele Register (mehr als 300 im Merced)

- Bündelung: je drei Befehle in zwei Speicherworten, "Maske"
- parallele Ausführung der Befehlsbündel (sofern möglich)
- Compiler verantwortlich für effiziente Bündelung
- "explicitly parallel instruction computing", EPIC
- Prädikation

- zusätzlicher Emulations-Modus für x86-Programme

# IA64: Registersatz (und IA32-Modus)






# x86: IA-64 load

- Speicherzugriffe sind langsam
- erst recht bei Cache-Misses oder Multiprozessorsystemen
- precise exceptions für Speicherzugriffe sind problematisch

"spekulatives Laden":

- Compiler verschiebt Leseoperationen möglichst nach vorne
- Zielregister der Ladeoperation wird als "dirty" markiert
- Speicher/Cachezugriffe werden "auf Probe" durchgeführt
- Resultat steht (hoffentlich) rechtzeitig zur Verfügung

wenn nicht:

- Compiler erzeugt CHECK-Befehle vor Lesen des Zielregisters
- Wartezyklen / Exception nur dann, wenn "dirty" noch gesetzt



# x86: IA-64 "predication"

- bedingte Sprünge behindern das Pipelining
- effiziente Sprungvorhersage notwendig (BTB, BTC, ...)
- oft genügt "bedingte Ausführung" statt eines Sprunges:

```
if (R1 == 0) {              CMP R1, 0              CMOVZ R2, R3, R1
  R2 = R3;                  BNE L1
}                           NOV R2, R3             ; conditional move
                      L1: ...
```

- allgemein: Bedingung setzt Flag in einem Register
- then-Zweig arbeitet mit CMOV, else-Zweig mit CMOVN
- keine Sprungbefehle, Pipeline wird nicht behindert

```
if (R1 == 0) {              CMP R1, 0              CMOVZ R2,R3,R1
  R2 = R3;                  BNE L1                 CMOVZ R4,R5,R1
  R4 = R5;                  MOV R2, R3             CMOVN R6,R7,R1
} else {                    MOV R4, R5             CMOVN R8,R9,R1
  R6 = R7;                  BR L2
  R8 = R9;              L1: MOV R6, R7
}                           MOV R7, R8
                      L2:
```



# x86: IA-64 vs. AMD x86-64

Marktbedeutung der IA-64 kaum abschätzbar:

- Unterstützung durch alle grossen Firmen angekündigt
- erste Versionen von Compiler und Tools verfügbar
- aber Hardware ("Merced") verspätet und zu langsam

- Rolle der AMD x86-x64 noch unsicherer:
- Notlösung, falls IA-64 "durchfällt" ?!

- erstmal neue IA-32 Prozessoren: Pentium-IV, K7-Ultra, ...

- siehe Intel IA-64 Präsentation (EPIC-Konzept, Merced, Tools)
- siehe AMD Roadmap und x86-64 Präsentation