SIMD: Media processing

"Media processing" mit dem PC ?!

- steigende Anforderungen für Audio, Video, Image, 3D
- grosse Datenmengen
- geringe Genauigkeit (8 bit .. 16 bit)
- x86-FPU ausgereizt

=> Trick: vorhandene ALUs/Datenpfade für SIMD verwenden

Befehlssatzerweiterungen:

• MMX	"multimedia extension"	1996
• 3Dnow!		1998
• ISSE	"internet SIMD streaming extension"	1999
• AltiVec	(PowerPC G4, Macintosh)	1999

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 :-)

SIMD: Literatur

MMX: *"The MMX technology page has been removed"*

- developer.intel.com/drg/mmx/manuals/
- developer.intel.com/drg/mmx/appnotes/
- Linux "parallel-processing-HOWTO"
- IEEE Micro 8/96 S.42, c't 01/97 S.228ff

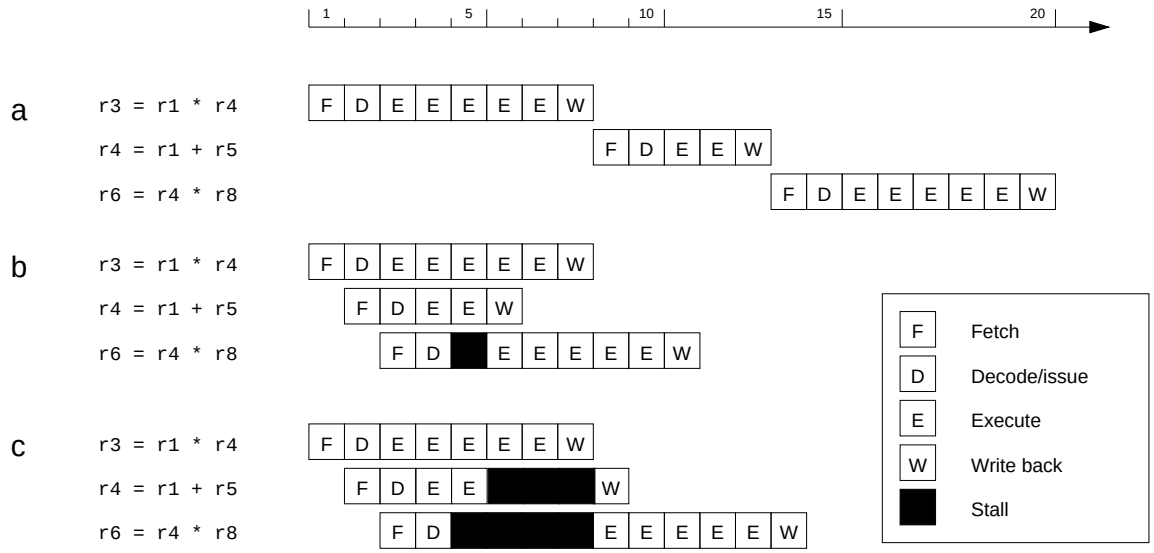
ISSE: Intel website:

- developer.intel.com/software/idap/resources/technical_collateral/pentiumiii/
- c't 04/00 S.314 (ISSE/3Dnow/Altivec)

3D Now! AMD website:

- www.amd.com/K6/K6docs/, www.amd.com/swdev/
- c't 15/98 S.186 ff
- IEEE Micro 3/4-99 S.37ff

Befehlspipeline: in order / out of order

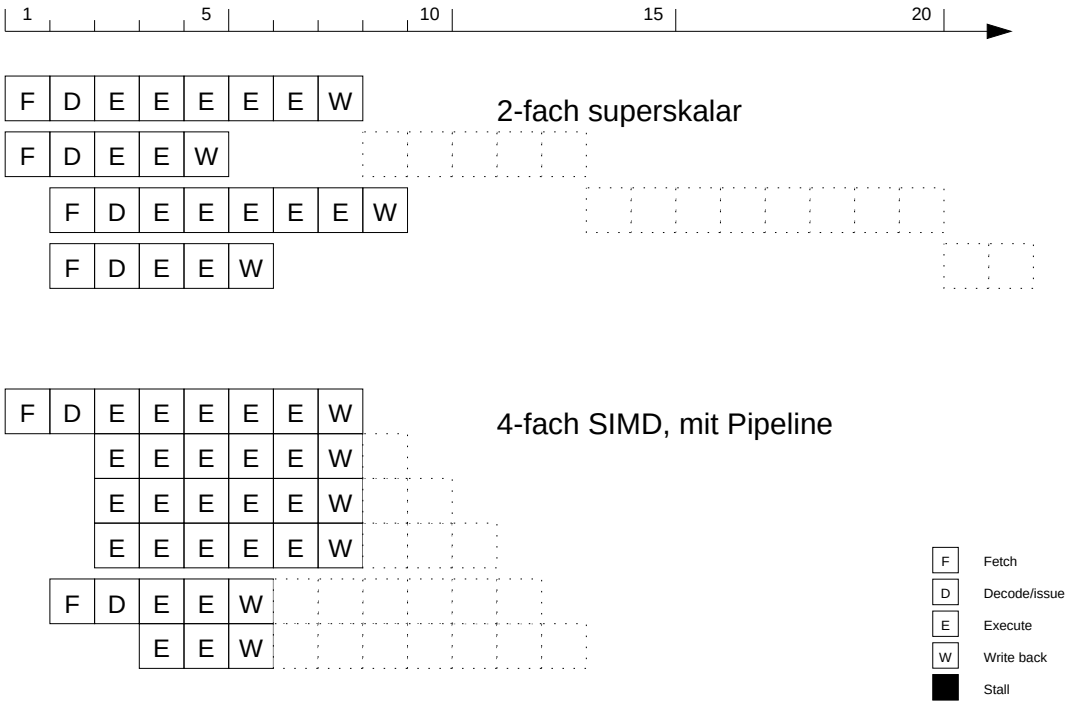


a) serielle Befehlsbearbeitung

b) pipeline, out-of-order completion

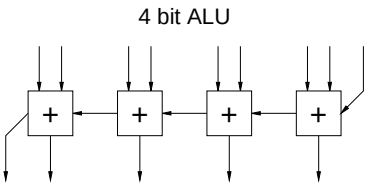
c) in-order-completion

Superskalar, SIMD



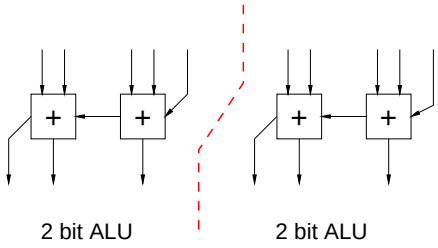
MMX: Grundidee

- 32/64-bit Datenpfade sind "overkill"
- ALUs aber leicht parallel nutzbar:
- carry-chain auftrennen



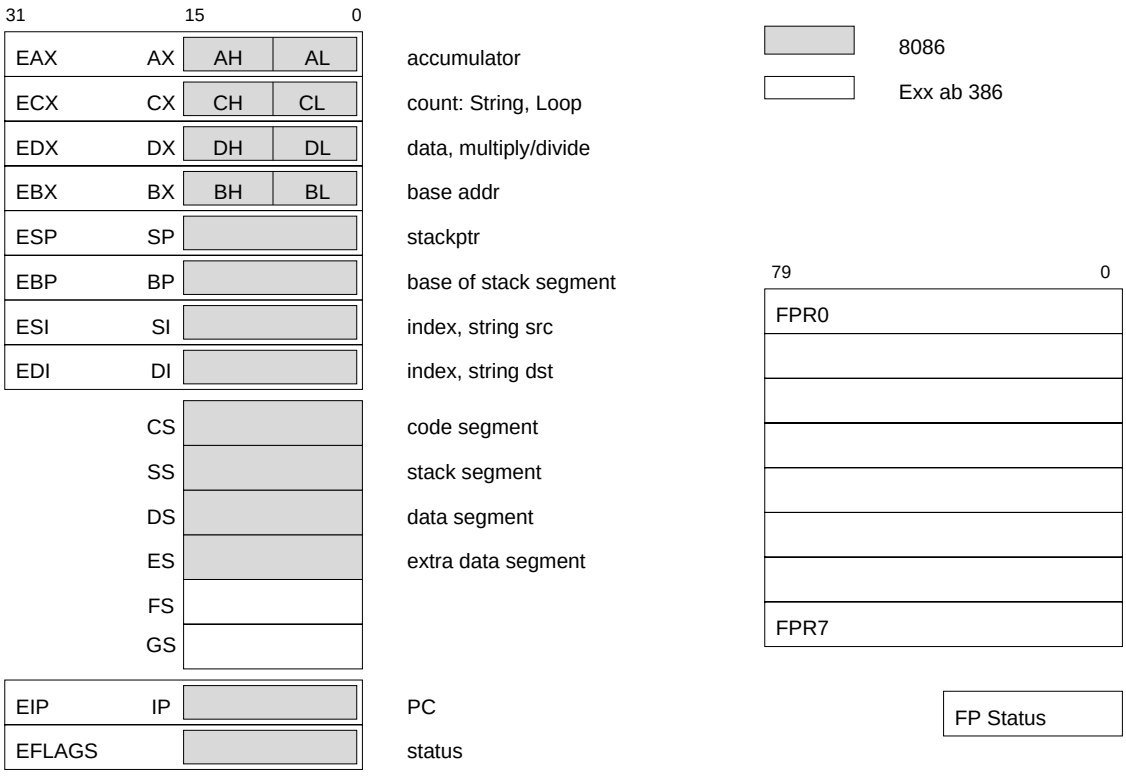
=> SIMD leicht implementierbar
~10% area on Pentium/MMX

=> Performance 2x .. 8x für MMX Ops
=> Performance 1.5x .. 2x für Apps



- MMX press release 03.05.1996
- Nutzen vs. Marketing ?!

x86: Register



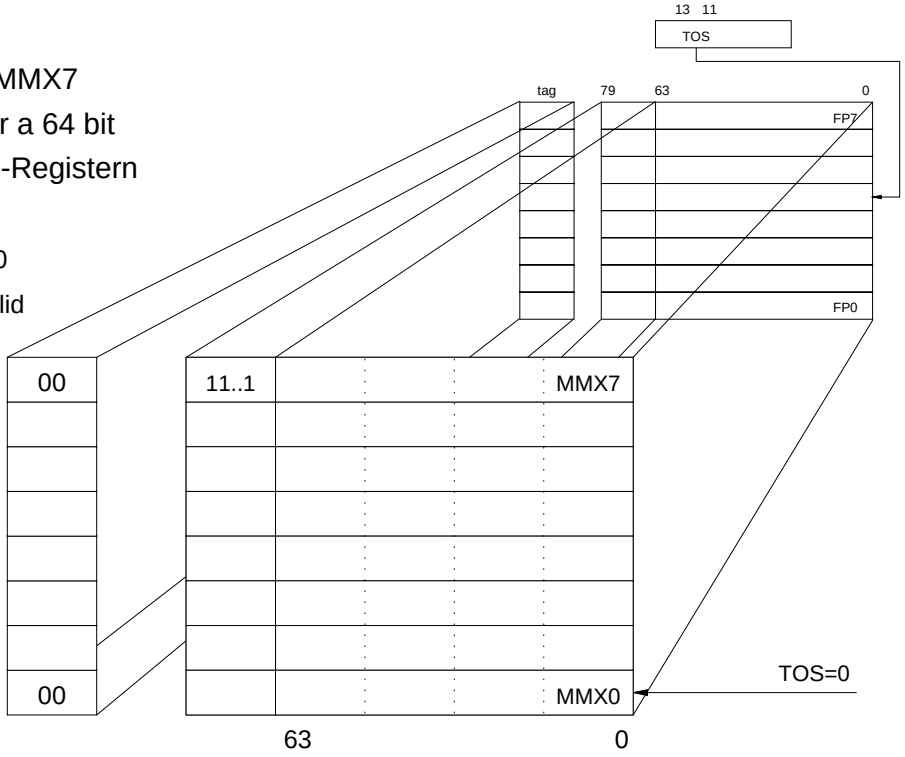
MMX: Entwurfsentscheidungen

Kompatibilität zu alten Betriebssystemen / Apps:

- keine neuen Register möglich => FP-Register nutzen
- keine neuen Exceptions => Überlauf ignorieren
  => saturation arithmetic
- bestehende Datenpfade nutzen => 64 bit
- möglichst wenig neue Opcodes
- alte Prozessoren und neue Software => Code doppelt
  => MMX DLLs
- Test-Applikationen: => 16 bit dominiert
  (audio/image/MPEG-1/3D-Graphik/...)
- keine Tools => Assembler
- optimierte Libraries verfügbar

MMX: Register

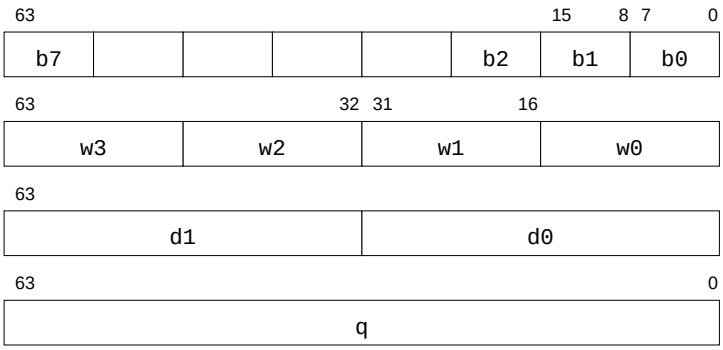
- MMX0 .. MMX7
- 8 Register a 64 bit
- in den FP-Registern
- FP NaN
- FP TOS = 0
- tag 00 = valid



MMX: Datenformate

64-bit Register, 4 Datentypen:

- packed byte *8 / packed word *4 / packed doubleword *2 / quadword
- Zugriff abhängig vom Befehl

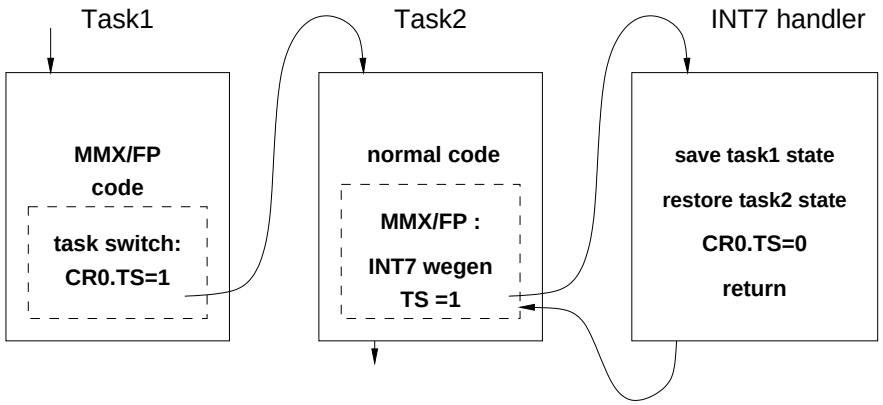


MMX: Befehlssatz

EMMS (FSAV / FRESTOR)	clear MMX state (handle FP regs)
MOVD mm1, mm2/mem32	move 32 bit data
MOVQ mm1, mm2/mem64	move 64 bit data
PACKSSWB mm1, mm2/mem64	pack 8*16 into 8*8 signed saturate
PUNPCKH mm1, mm2/mem64	fancy unpacking (see below)
PACKSSDW mm1, mm2/mem64	pack 4*32 into 4*16 signed saturate
PAND mm1, mm2/mem64	mm1 AND mm2/mem64 / auch OR/XOR/NAND
PCMPEQB mm1, mm2/mem64	8*a==b, create bit mask / auch GT
PADDB mm1, mm2/mem64	8*add 8 bit data
PSUBD mm1, mm2/mem64	2*sub 32 bit data / signed wrap
PSUBUSD mm1, mm2/mem64	2*sub 32 bit data / unsigned saturate
PSLL mm1,mm2/mem64/imm8	shift left mm1 / auch PSRA/PSRL
PMULL/HW mm1, mm2/mem64	4*mul 16*16 store low/high 16 bits
PMADDWD mm1, mm2/mem64	MAC 4*16 -> 2*32
insgesamt 57 Befehle	(Varianten B/W/D S/US)

MMX: Multitaskting . . .

Interaktion mit Betriebssystem / Taskwechsel:



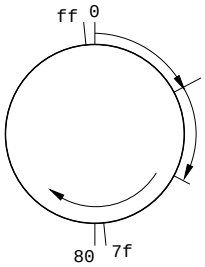
- FP-Register nur bei Bedarf sichern
- vorhandene FP INT7 Routine funktioniert auch für MMX
- keine Anpassung des Betriebssystems notwendig

MMX: "Saturation Arithmetic"

was soll bei einem Überlauf passieren?

- wrap-around

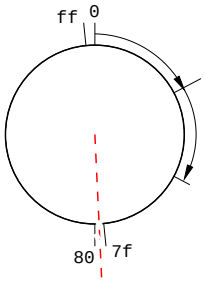
  ..., 125, 126, 127, -128, -127, ...



- saturation

  ..., 125, 126, 127, 127, 127, ...

- Zahlenkreis "aufgeschnitten"
- gut für DSP-Anwendungen



paddw (wrap around):

a3	a2	a1	7FFFh
+	+	+	+
b3	b2	b1	0004h
a3+b3	a2+b2	a1+b1	8003h

paddusw (saturating):

a3	a2	a1	7FFFh
+	+	+	+
b3	b2	b1	0003h
a3+b3	a2+b2	a1+b1	7FFFh

MMX: "packed multiply add word"

für Skalarprodukte:

pmaddwd

a3	a2	a1	a0
*	*	*	*
b3	b2	b1	b0
a3*b3+a2*b2		a1*b1+a0*b0	

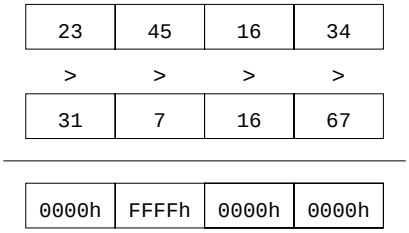
```
vector_x_matrix_4x4( MMX64* v, MMX64 *m ) {
  MMX64  v0101, v2323, t0, t1, t2, t3;

  v0101 = punpckldq( v, v );  // unpack v0/v1
  v2323 = punpckhdq( v, v );  // unpack v2/v3

  t0    = pmaddwd( v0101, m[0] );  // v0|v1 * first 2 rows
  t1    = pmaddwd( v2323, m[1] );  // v2|v3 * first 2 rows
  t2    = pmaddwd( v0101, m[2] );  // v0|v1 * last 2 rows
  t3    = pmaddwd( v2323, m[3] );  // v2|v3 * last 2 rows

  t0    = paddd(    t0, t1 );      // add
  t2    = paddd(    t2, t3 );      //
  v     = packssdw( t0, t2 );      // pack 32->16, saturate
}
```

MMX: "parallel compare"

Vergleiche / Sprungbefehle:

- schlecht parallelisierbar
- Pipeline-Abhängigkeiten

pcmpgtw:

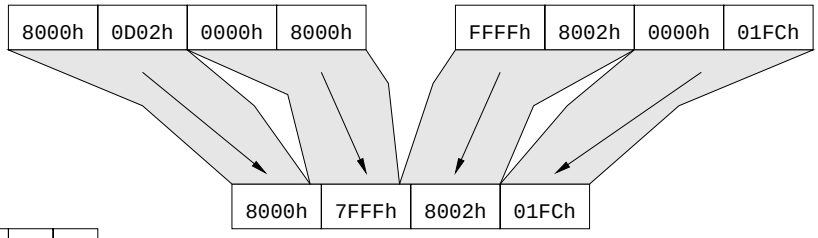


=> keine Sprungbefehle in MMX
=> compare-Operationen setzen Bitmasken

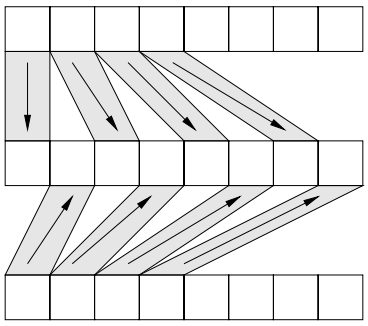
- Bitmasken für logische Ops verwendbar
- Beispiel: chroma-keying

MMX: packssdw / punpckhbw

packssdw: pack with saturation 32 -> 16 signed data:



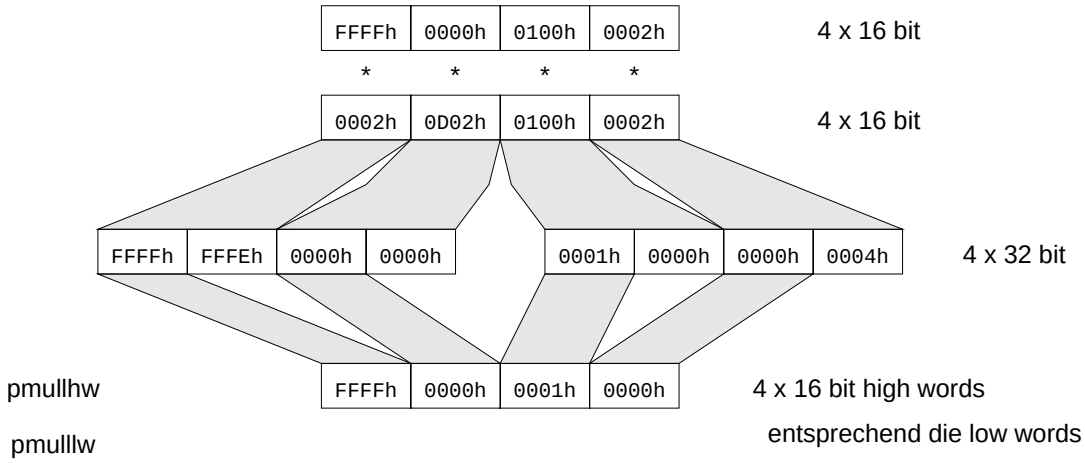
punpckhbw:



punpcklbw: lower 32 bits

MMX: pmulllw / pmullhw

pmull[lh]w: multiply 4 words, write low/high byte of results:



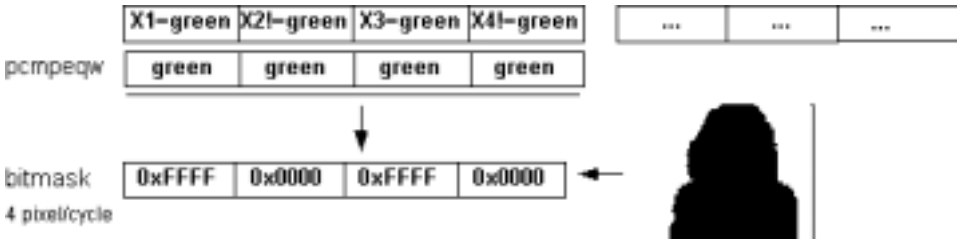
mit Packbefehlen kombinieren, wenn 32-bit Resultate gewünscht

MMX: Chroma Keying (1)

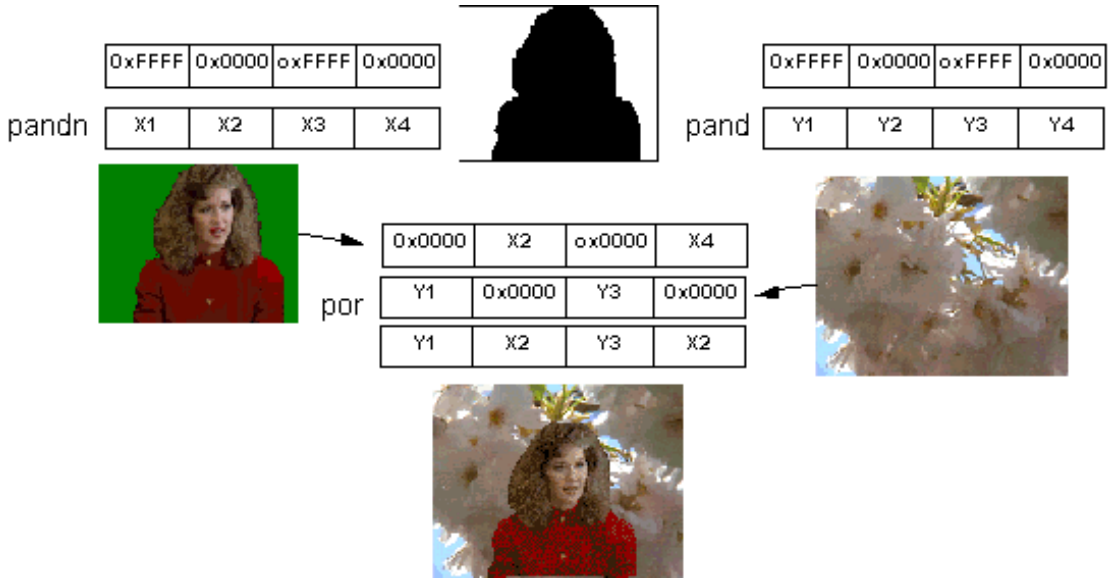
"Wetterbericht":

- MMX berechnet 4 Pixel / Takt
- keine Branch-Befehle

- Schritt 1: Maske erstellen (high-color: 16 bit/pixel)



MMX: Chroma Keying (2)



MMX: Zufallszahlen

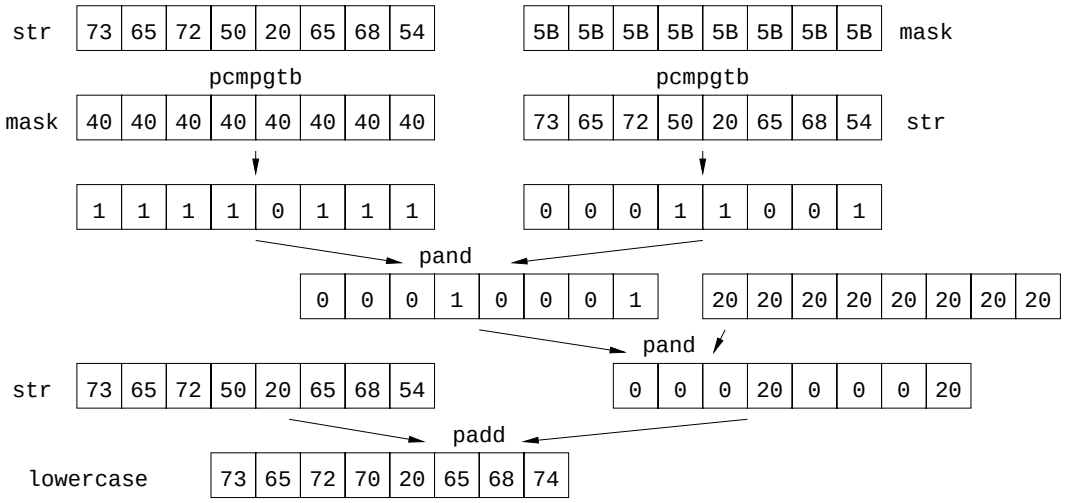
```
x(t) = (x(t-1) * 47989) & 0xFFFF;
```

```
QuadWord  DithMultVal = 0x4f314f314f314f31;
QaudWord  DithRegInit = 0x4f31994d2379bb75;

Init:
  MOVQ    mm0, DithRegInit;

Loop:                          // x(t) -> x(t+1)
  PMULLW  mm0, DithMulVal      // 3 clocks
  MOVQ    [result64], mm0      // 1 clocks
```

- PMULLW latency 3, throughput 1 (on Pentium)
- bis zu vier Zufallszahlen pro Takt (U/V pipelines genutzt)

MMX: toLowerCase()

String lower-to-upper-case conversion:



(aber Probleme mit Umlauten...)

[aus Intel MMX appnote]

3Dnow! Motivation

- stark wachsende Bedeutung von 3D-Spielen
- 32-bit Gleitkommaoperationen nötig für Geometrie-Transformationen
- FPU im AMD K6 vergleichsweise langsam
- MMX unterstützt nur Integer-Datentypen

=> SIMD-Befehle für 32-bit float Datentypen

- schnelle Add/Mult/MAC/Sqrt-Befehle
- muß ohne OS-Unterstützung nutzbar sein
- MMX-Register verwenden
- MMX zwei-Operanden Adressierung
- je zwei float-Datenwerte pro MMX-Register

=> 3Dnow! Spezifikation
(vergleiche Motorola Altivec / Intel ISSE)

3Dnow! Entscheidungen

SIMD-Befehle für 32-bit float Datentypen:

- MMX-Register verwenden, zwei Datenworte pro Register
- zwei-Adress-Befehle
- keine Status-Flags, keine Exceptions
- MMX-Befehle nutzbar (logische, Vergleiche, ...)
- belegt nur einen einzigen x86 Opcode (0F0F ... subobcode)

möglichst wenig Chipfläche:

- keine Unterstützung für NaN/INF/...
- nur round-to-nearest-even Modus, +- 1LSB
- Saturation-Arithmetik statt Überlauf
- Approximation für Division und Quadratwurzel

3Dnow! Prefetch

Speicherzugriffe in Multimedia-Applikationen:

- reguläre Speicherzugriffsmuster
- ungewöhnliche Lokalität
- viele Daten werden (pro Frame) nur einmal benötigt
- aber regelmässig (in jedem Frame)
- Performance stark von optimaler Cache-Ausnutzung abhängig

=> prefetch-Befehl

- quasi normaler Ladebefehl, aber ohne Zielregister
- gewünschte Daten werden in L1/L2-Cache geladen
- löst keine Exceptions / Page Faults aus

=> "memory streaming"
=> auch für andere Anwendung gut nutzbar (etwa Numerik)

3Dnow! Division / Quadratwurzel

- Rechenwerk für Division / Sqrt ist sehr aufwendig
- möglichst wenig Chipfläche für 3Dnow!
- teilweise nur geringe Genauigkeit benötigt
- etwa Shading/Beleuchtungsberechnung für 3D-Graphik

=> Division und Quadratwurzel per Approximation

- erster Befehl liefert 14/15 bit Approximation
- aus Lookup-Table und Interpolation
- mit vollem Takt

- zusätzliche Befehle für Newton-Iteration
- quadratische Konvergenz: zwei Iterationsschritte für volle Genauigkeit
- wenig Hardwareaufwand
- voll in Pipeline integriert, maximaler Durchsatz

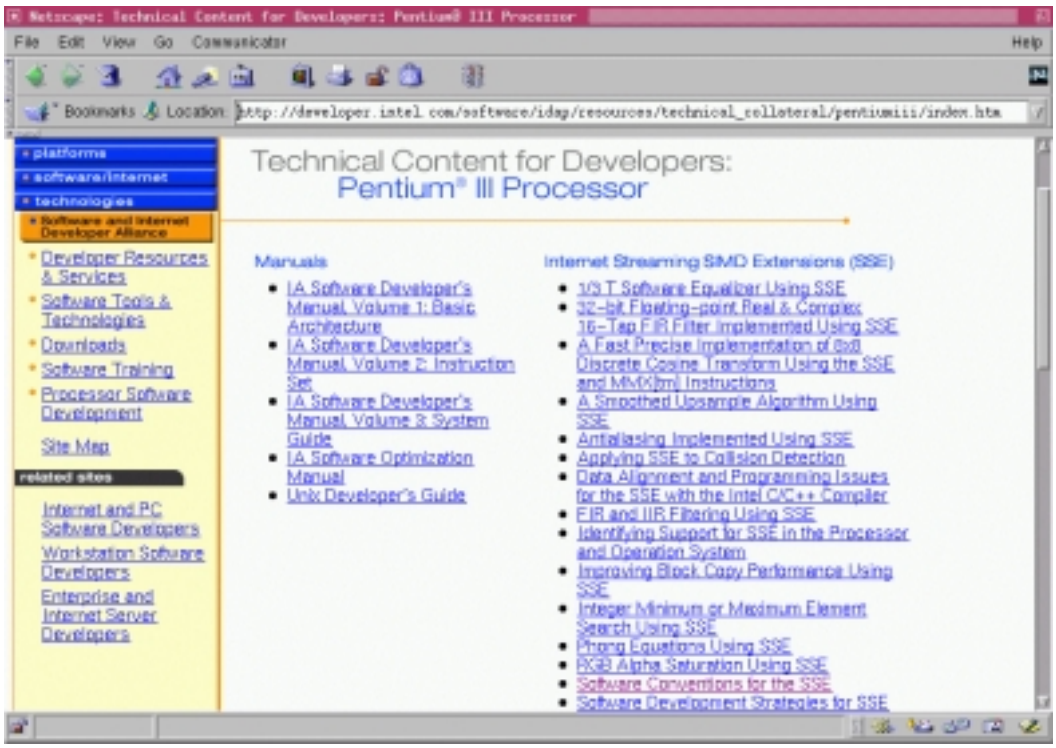
3D Now! Apfelmännchen

```
Function IterPasD
     (I,R :Double; Grenze, Tiefe :Paratyp):Paratyp;
     var A,B,C:double;
  Begin
   Count:= 0;
   A:=0; B:=0;
   Repeat
      C:= SQR(A) - SQR(B) + R;
      B:= 2*A*B + I;
      A:= C;
      INC (Count);
   Until (abs (A) >Grenze) or (Abs (B) > Grenze)
           or (Count=Tiefe);
  IterpasD:=Count;
  End;
```

3D Now! Apfelmännchen

```
;   Quadriere (A + jB)**2 = A**2 - B**2 + j 2*A*B
;   Entry   MM0                     ;A          |  B
;           MM1                     ;1          | -1
;           MM2                     ;R          |  I
;
loop:
            MOVQ    MM3,MM0         ;MM3=A      | B
            MOVQ    MM4,MM0         ; oh weh
            PSLLQ   MM3,32          ; das Vertauschen ist
            PSRLQ   MM4,32          ; sehr mühsam ...
            POR     MM3,MM4         ;MM3=B      | A

            PFMUL   MM3,MM0         ;MM3= A*B   | A*B
            PFMUL   MM0,MM0         ;MM0= A**2  | B**2
            PFMUL   MM0,MM1         ;MM0= A**2  | -B**2
            PFACC   MM0,MM3         ;MM0= A**2 - B**2     | A*B+A*B
            PFADD   MM0,MM2         ;MM0= A**2 - B**2 + R | 2*A*B+I
                                    ;   = A(n+1) |    = B(n+1)
            PF2ID   MM4,MM0         ;iA = INT(A) | iB = Int(B)
            MOVQ    iA,MM4
;  Sieh nach, ob A oder B > GRENZE ist
            ...
            dec     CX              ; iteration counter
            jnz  loop
```


ISSE: Homepage / Literatur



ISSE: Entwurfsentscheidungen

- Markt fordert 3D
- mindestens doppelte FP-Performance notwendig

2-fach oder 4-fach SIMD?

- 128-bit machbar (FP bereits 80-bit)
- bereits 2 64-bit ALUs auf dem Prozessor

=> 4-fach SIMD

"already register-starved IA32 architecture"

=> neue Register, 128-bit

=> erfordert OS-Unterstützung

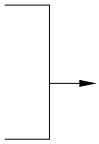
- 70 neue Befehle
- sowohl "packed" als auch "scalar ISSE instructions"

ISSE: "Streaming"

typisch für Medienverarbeitung:

- hohe Datenmenge / Datenrate
- geringe Lokalität: viele Daten (Pixel) werden nur 1x benötigt

=> Cache-"Pollution"
=> herkömmliche Cache-Strategien nutzlos
=> ALUs müssen auf die Daten warten



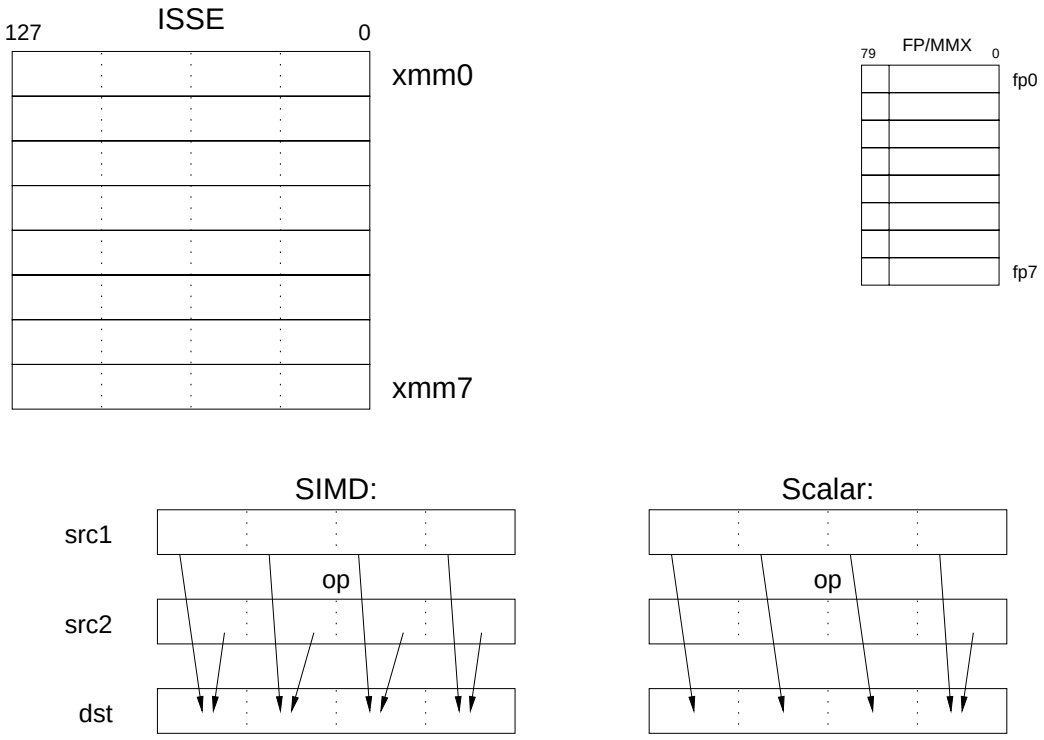
Performance leidet (extrem)

- 1GHz, 8x SIMD, 100 nsec Speicher: 800 OPs / 1 Zugriff

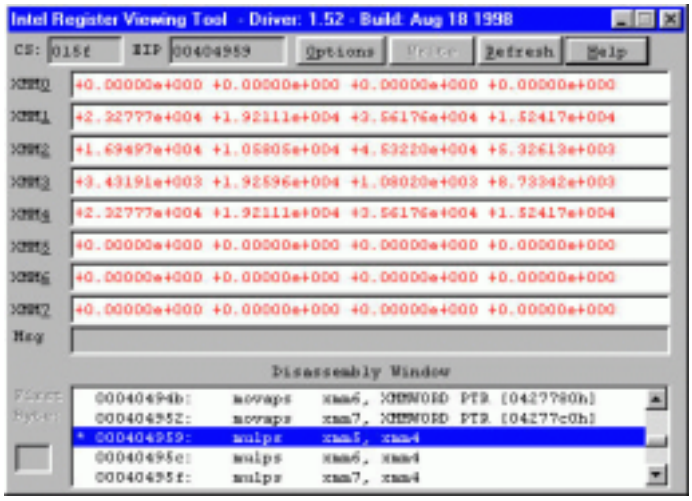
Streaming:

- Cache-Nutzung anpassen
- Prefetch: Daten rechtzeitig anfordern
- Speicherlatenz fast perfekt versteckt (für Media-Apps.)

ISSE: Register



ISSE: Register Viewing Tool



Softwareentwicklung für MMX / ISSE / 3Dnow:

- nur rudimentäre Compiler- und Tool-Unterstützung
- oft handoptimierter Assembler wg. bester Performance

ISSE: Programmierung

Intel VTune Performance Enhancement Environment:

- optimierender Compiler mit ISSE-Unterstützung:

• Intrinsics	C-Funktionen, Compiler inlining
• Vector Class Library	Klassen, inlining durch Compiler
• Vectorization	optimierender Compiler
• Intel Performance Library Suite	

- erfordert 16-Byte Alignment aller Datentypen
- umfangreiche Profiling-Tools

- sehr teuer

ISSE: Programmierung mit "Intrinsics"

```
float xa[SIZE], xb[SIZE], xc[SIZE];
float q;

void do_c_triad() {
  for( int j=0; j < SIZE; j++ ) {
    xa[j] = xb[j] + q*xc[j];
  }
}
```

ISSE-Programmierung mit "Intrinsics" und VTUNE:

```
#define VECTOR_SIZE 4
__declspec(align(16)) float xa[SIZE], xb[SIZE], xc[SIZE];
float q;

void do_intrin_triad() {
  __m128 tmp0, tmp1;

  tmp1 = _mm_set_ps1(q);
  for( int j=0; j < SIZE; j+= VECTOR_SIZE) {
    tmp0 = __mm_mul_ps( *((__m128 *) &xc[j]), tmp1 );
    *(__m128 *) &xa[j] =
                  _mm_add_ps(tmp0, *((__m128 *) &xb[j];
  }
}
```

ISSE: AoS / SoA

Array of Structures:

- Daten lokal
- Anordnung schlecht für SIMD

```
struct
{
  float A, B, C;
} AoS_data[1000];
```

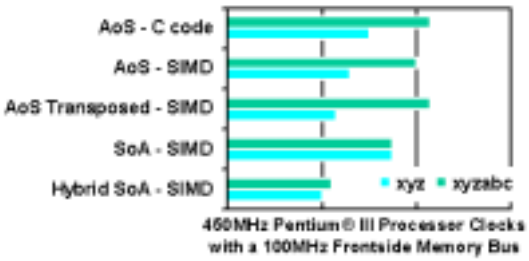
Structure of Arrays:

- Anordnung optimal für SIMD
- aber im Speicher "verstreut"

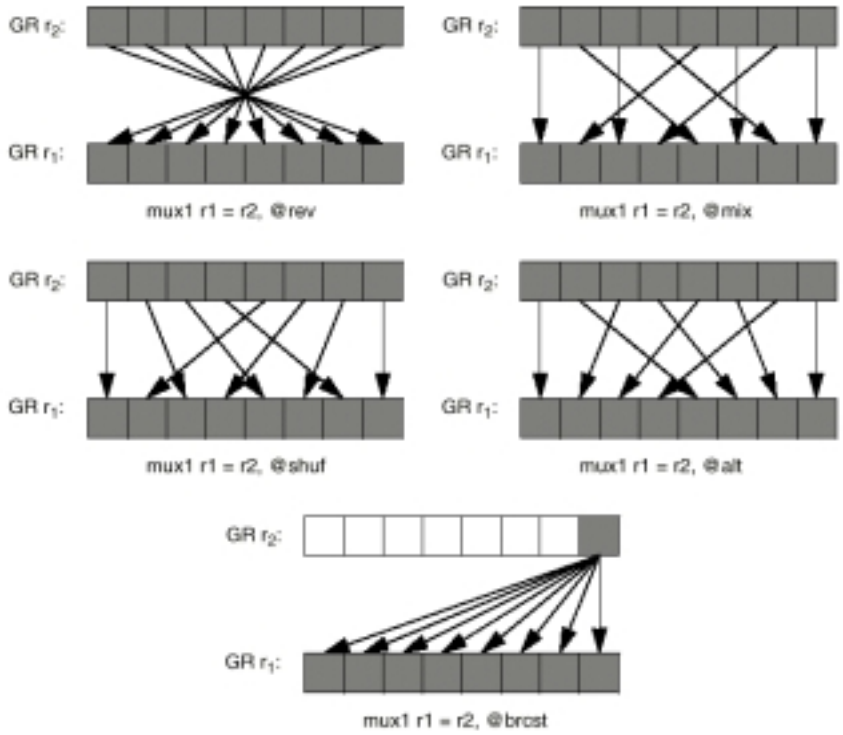
```
struct
{
  float A[1000],B[1000],C[1000];
} SoA_data;
```

=> Hybrid SoA - SIMD

```
struct
{
  float A[8],B[8],C[8];
} Hybrid_data[125];
```



ISSE2: mux1-Befehl (IA64)



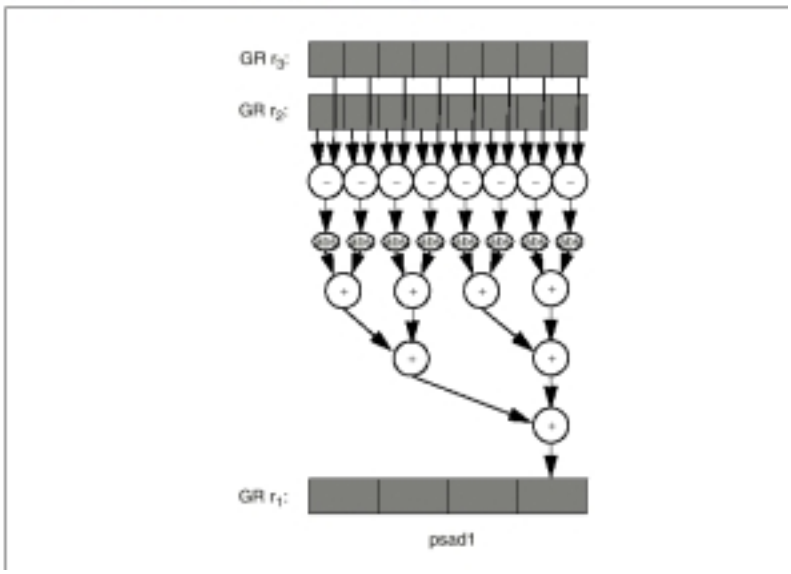
ISSE2: psad1-Befehl (IA64)

Parallel Sum of Absolute Difference

Format: (qp) psad1 $r_1 = r_2, r_3$

Description: The unsigned 8-bit elements of GR r_2 are subtracted from the unsigned 8-bit elements of GR r_3. The absolute value of each difference is accumulated across the elements and placed in GR r_1.

Figure 7-36. Parallel Sum of Absolute Difference Example



ISSE2: pavg2-Befehl (IA64)