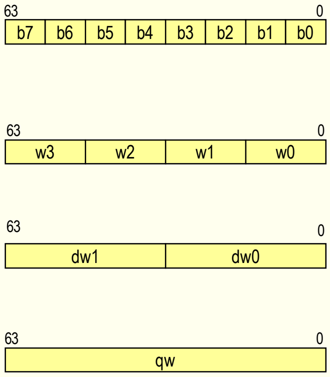
MMX and SSE

- Extensions to the instruction set for parallel SIMD operations on packed data
  - SIMD – Single Instruction stream Multiple Data stream
- MMX – Multimedia Extensions
- SSE – Streaming SIMD Extension
- SSE2 – Streaming SIMD Extension 2
- Designed to speed up multimedia and communication applications
  - graphics and image processing
  - video and audio processing
  - speech compression and recognition

MMX data types

- MMX instructions operate on 8, 16, 32 or 64-bit integer values, packed into a 64-bit field
- 4 MMX data types
  - packed byte
    8 bytes packed into a 64-bit quantity
  - packed word
    4 16-bit words packed into a 64-bit quantity
  - packed doubleword
    2 32-bit doublewords packed into a 64-bit quantity
  - quadword
    one 64-bit quantity
- Operates on integer values only



63 0
b7 b6 b5 b4 b3 b2 b1 b0

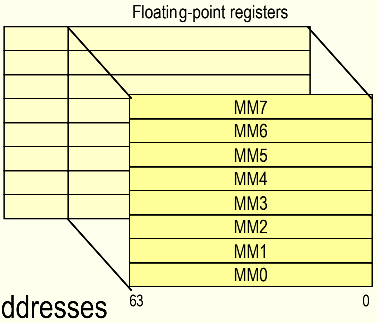
63 0
w3 w2 w1 w0

63 0
dw1 dw0

63 0
qw

MMX registers

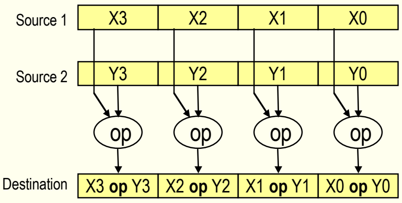
- 8 64-bit MMX registers
  - aliased to the x87 floating-point registers
  - no stack-organization
- The 32-bit general-purouse registers (EAX, EBX, ...) can also be used for operands and adresses
  - MMX registers can not hold memory addresses



- MMX registers have two access modes
  - 64-bit access
    - 64-bit memory access, transfer between MMX registers, most MMX operations
  - 32-bit access
    - 32-bit memory access, transfer between MMX and general-purpose registers, some unpack operations

MMX operation

- SIMD execution
  - performs the same operation in parallel on 2, 4 or 8 values
- MMX instructions perform arithmetic and logical operations in parallel on the bytes, words or doublewords packed in a 64-bit MMX register
- Most MMX instructions have two operands
  - op dest source
  - destination is a MMX register
  - source is a MMX register or a memory location



MMX instructions

- MMX instructions have names composed of four fields
  - a prefix P – stands for packed
  - the operation, for example ADD, SUB or MUL
  - 1-2 characters specifying unsigned or signed saturated arithmetic
    - US – Unsigned Saturation
    - S – Signed Saturation
  - a suffix describing the data type
    - B – Packed Byte, 8 bytes
    - W – Packed Word, 4 16-bit words
    - D – Packed Doubleword, 2 32-bit double words
    - Q – Quadword, one single 64-bit quadword
- Example:
  - PADDB – Add Packed Byte
  - PADDSB – Add Packed Signed Byte Integers with Signed Saturation

Saturation and wraparound arithmetic

- Operations may produce results that are out of range
  - the result can not be represented in the format of the destination
- Example:
  - add two packed unsigned byte integers 154+205=359
  - the result can not be represented in 8 bits

```
 10011010
+11001101
---------
101100111
```

- Wraparound arithmetic
  - the result is truncated to the *N* least significant bits
  - carry or overflow bits are ignored
- Saturation arithmetic
  - out of range results are limited to the smallest/largest value that can be represented
  - can have both signed and unsigned saturation

Data ranges for saturation

- Results smaller than the lower limit is saturated to the lower limit
- Results larger than the upper limit is saturated to the upper limit
- Natural way of handling under/over-flow in many applications
  - *Example*: color calculations, if a pixel becomes black, it remains black
- MMX instructions do not generate over/underflow exceptions or set over/underflow bits in the EFLAGS status register

Data type	Bits	Lower limit	Upper limit
Signed byte	8	-128	127
Unsigned byte	8	0	255
Signed word	16	-32768	32767
Unsigned word	16	0	65535

MMX instructions

- MMX instructions can be grouped into the following categories:
  - data transfer
  - arithmetic
  - comparison
  - conversion
  - unpacking
  - logical
  - shift
  - empty MMX state instruction (EMMS)

Data transfer instructions

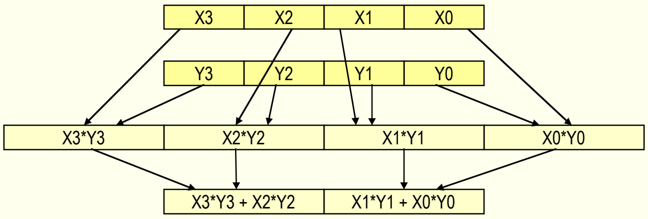
- MOVD – Move Doubleword
  - copies 32 bits of packed data
    - from memory to a MMX register (and vice versa), or
    - from a general-purpose register to a MMX register (and vice versa)
  - operates on the lower doubleword of a MMX register (bits 0-31)
- MOVQ – Move Quadword
  - copies 64 bits of packed data
    - from meory to a MMX register (and vice versa), or
    - between two MMX registers
- MOVD/MOVQ implements
  - register-to-register transfer
  - load from memory
  - store to memory

Arithmetic instructions

- Addition
  - PADDB, PADDW, PADDD – Add Packed Integers with Wraparound Arithmetic
  - PADDSB, PADDSW – Add Packed Signed Integers with Signed Saturation
  - PADDUSB, PADDUSW – Add Packed Unsigned Integers with Unsigned Saturation
- Subtraction
  - PSUBB, PSUBW, PSUBD – Wraparound arithmetic
  - PSUBSB, PSUBSW – Signed saturation
  - PSUBUSB, PSUBUSW – Unsigned saturation
- Multiplication
  - PMULHW – Multiply Packed Signed Integers and Store High Result
  - PMULLW – Multiply Packed Signed Integers and Store Low Result

Arithmetic instructions (cont.)

- Multiply and add
  - PMADDWD – Multiply And Add Packed Integers
  - multiplies the signed word operands (16 bits)
  - produces 4 intermediate 32-bit products
  - the intermediate products are summed pairwise and produce two 32-bit doubleword results

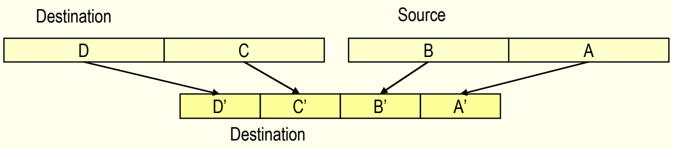


Comparison instructions

- Compare Packed Data for Equal
  - PCMPEQB, PCMPEQW, PCMPEQD
- Compare Packed Signed Integers for Greater Than
  - PCMPGTPB, PCMPGTPW, PCMPGTPD
- Compare the corresponding packed values
  - sets corresponding destination element to a mask of all ones (if comparison matches) or zeroes (if comparison does not match)
- Does not affect EFLAGS register

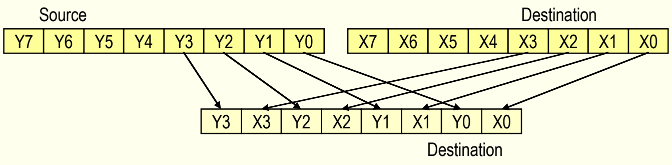
Conversion instruction

- PACKSSWB, PACKSSDW – Pack with Signed Saturation
- PACKUSWB – Pack with Unsigned Saturation
  - converts words (16 bits) to bytes (8 bits) with saturation
  - converts doublewords (32 bits) to words (16 bits) with saturation



Unpacking instructions

- PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ – Unpack and Interleave High Order Data
- PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ – Unpack and Interleave Low Order Data



Logical instructions

- PAND – Bitwise AND
- PANDN – AND NOT
- POR – OR
- PXOR – Exclusive OR
  - operate on a 64-bit quadword

Shift instructions

- PSLLW, PSLLD, PSLLQ - Shift Packed Data Left Logical
- PSRLW, PSRLD, PSRLQ – Shift Packed Data Right Logical
- PSRAW, PSRAD – Shift Packed Data Right Arithmetic
  - shifts the destination elements the number of bits specified in the count operand

EMMS instruction

- Empty MMX State
  - sets all tags in the x87 FPU tag word to indicate empty registers
- Must be executed at the end of a MMX computation before floating-point operations
- Not needed when mixing MMX and SSE/SSE2 instructions

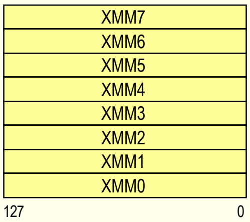
SSE

- Streaming SIMD Extension
  - introduced with the Pentium III processor
  - designed to speed up performance of advanced 2D and 3D graphics, motion video, videoconferencing, image processing, speech recognition, ...
- Parallel operations on packed single precision floating-point values
  - 128-bit packed single precision floating point data type
  - four IEEE 32-bit floating point values packed into a 128-bit field
  - must be aligned in memory on 16-byte boundaries

127			0
s3	s2	s1	s0

XMM registers

- The MMX technology introduces 8 new 128-bit registers XMM0 – XMM7
  - not aliased to other registers
  - independent of general purpose and FPU/MMX registers
  - can mix MMX and SSE instructions
- XMM registers can be accessed in 32-bit, 64-bit or 128-bit mode
  - only for operations on data, not addresses
- MXCSR control and status register, 32 bit
  - flag and mask bits for floating-point exceptions
  - rounding control bits
  - flush-to-zero bit
  - denormals-are-zero bit



XMM7
XMM6
XMM5
XMM4
XMM3
XMM2
XMM1
XMM0
127 0

SSE instructions

- Adds 70 new instructions to the instruction set
  - 50 for SIMD floating-point operations
  - 12 for SIMD integer operations
  - 8 for cache control
- Packed and scalar single precision floating-point instructions
  - operations on packed 32-bit floating-point values
    - packed instructions have the suffix PS
  - operations on a scalar 32-bit floating-point value (the 32 LSB)
    - scalar instructions have the suffix SS
- 64-bit SIMD integer instructions
  - extension to MMX
  - operations on packed integer values stored in MMX registers

SSE instructions (cont)

- State manegement intructions
  - load and save state of the MXCSR control register
- Cache control, prefetch and memory ordering instructions
  - instructions to control stores to / loads from memory
  - support for streaming data to/from memory without storing it in cache
- Temporal data
  - will be reused in the program execution
  - should be accessed through the cache
- Non-temporal data
  - will not be reused in the program execution
  - evicts temporal data if accessed through the cache (cache pollution)
  - can be accessed directly from memory using prefetching and write-combining

SSE2

- Streaming SIMD Extension 2
  - introduced in the Pentium 4 processor
  - designed to speed up performance of advanced 3D graphics, video encoding/decodeing, speech recognition, E-commerce and Internet, scientific and engineering applications
- Extends MMX and SSE with support for
  - packed double precision floating point-values
  - packed integer values
  - adds over 70 new instructions to the instruction set
- Operates on 128-bit entities
  - must be aligned on 16-bit boundaries when stored in memory

SSE2 data types

- 128-bit packed double precision floating point
  - 2 IEEE double precision floating-point values
- 128-bit packed byte integer
  - 16 byte integers (8 bits)
- 128-bit packed word integer
  - 8 word integers (16 bits)
- 128-bit packed doubleword integer
  - 4 doubleword integers (32 bits)
- 128-bit packed quadword integer
  - 2 quadword integers (64 bits)
- Same registers for SIMD operations as in SSE
  - eight 128-bit registers, XMM0 – XMM7

Compatibility with SSE and MMX operation

- The SSE2 extension is an enhancement of the SSE extension
  - no new registers or processor state
  - new instructions which operate on a wider variety of packed floating-point and integer data
- SSE2 instructions can be intermixed with SSE and MMX/FPU instructions
  - same registers for SSE and SSE2 execution
  - separate set of registers for FPU/MMX instructions

SSE2 instructions

- Operations on packed double-precision data has the suffix PD
  - *examples*: MOVAPD, ADDPD, MULPD, MAXPD, ANDPD, CPPPD
- Operations on scalar double-precision data has the suffix SD
  - *examples*: MOVSD, ADDSD, MULSD, MINSD
- Conversion instructions
  - between double precision and single precision floating-point
  - between double precision floating-point and doubleword integer
  - between single precision floating-point and doubleword integer
- Integer SIMD operations
  - both 64-bit and 128-bit packed integer data
  - 64-bit packed data uses the MMX register
  - 128-bit data uses the XMM registers
  - instructions to move data between MMX and XMM registers

Programming with MMX and SSE

- Assembly language
  - inline assembly language code
  - very good possibilities to arrange instructions for efficient execution
  - difficult to program, requires detailed knowledge of MMX/SSE operation
- Compiler intrisincs or MMX/SSE macro library
  - functions that provide access to the MMX/SSE instructions from a high-level language
  - also requires a detailed knowledge of MMX/SSE operation
- Classes
  - C++ classes that define an abstraction for the MMX/SSE datatypes
  - easy to program, does not require in-depth konwledge of MMX/SSE
- Automatic vectorization
  - easy to program, but requires a vectorizing compiler

Assembly language

- Use inline assembly code
  - for instance in a C program
- Example:
  - multiply two arrays A and B of 400 single precision floating-point values
- Can arrange instructions to avoid stalls
  - MOVAPS
    latency 6, throughput 1
  - MULPS
    latency 6, throughput 2
  - ADD/SUB
    latency 0.5, throughput 0.5
  - the branch will be correctly predicted, except the last time

```
asm {
   push esi
   push edi
                  ; Set up for loop
   mov   edi, A        ; Address of A
   mov   esi, B        ; Address of B
   mov   edx, C        ; Address of C
   mov   ecx, #100     ; Counter
L1:
   movaps xmm0, [edi]  ; Load from A
   movaps xmm1, [esi]  ; Load from B
   mulps  xmm0, xmm1   ; Multiply
   add edi, #16        ; Incr. ptr to A
   add esi, #16        ; Incr. ptr to B
   movaps [edx], xmm0  ; Store into C
   add edx, #16        ; Incr. ptr to C
   sub ecx, #1         ; Decr. counter
   jnz L1              ; Loop if not done

   pop edi
   pop esi
}
```

C compiler intrisincs or MMX/SSE macro library

- Macros containing inline assembly code for MMX/SSE operations
  - allows the programmer to use C function calls and variables
- Defines a C function for each MMX/SSE instruction
  - there are also intrisinc functions composed of several MMX/SSE instructions
- New data types to represent packed integer and floating-point values
  - __m64 represents the contents of a 64-bit MMX register
    (8, 16 or 32 bit packed integers)
  - __m128 represents 4 packed single precision floating-point values
  - __m128d represents 2 packed double precision floating-point values
  - __m128i represents packed integer values (8, 16, 32 or 64-bit)

C intrisincs

- Example:
  - multiply two arrays A and B of 400 single precision floating-point values
- Register allocation and instruction scheduling is left to the compiler

```
#define SIZE 400

float A[SIZE], B[SIZE], C[SIZE];
__m128 m1, m2, m3;

for (int i=0; i<SIZE; i+=4) {
   m1 = _mm_load_ps (A+i);
   m2 = _mm_load_ps (B+i);
   m3 = _mm_mul_ps (m1,m2);
   _mm_store_ps (C+i,m3);
}
```

- Varialbles of intrisinc data types have to be aligned to 16-bit boundaries
  - may also need to access the individual values in the packed data
  - can be done by using a union structure

```
union mmdata {
    __mm128 m;
    float f[4];
};
```

C++ classes

- C++ class defining abstractions for MMX and SSE data types
- Oveloads the arithmetic operations +, -, *, /
  - implemented using the intrisincs

```
#include fvec.h
#define SIZE 400

F32Vec4 f1, f2, f3;

for (int i=0; i<SIZE; i+=4) {
   loadu(f1, A+i);
   loadu(f2, B+i);
   f3 = f1 * f2;
   storeu(C+i, f3);
}
```

- Also possible to use automatic vectorization
  - the compiler analyzes the code and concerts simple loops to SSE instructions
  - the user can assist the compiler by inserting directives in the code