How to Write Fast Code
SIMD Vectorization
18-645, spring 2008
13th and 14th Lecture

Instructor: Markus Püschel
Guest Instructor: Franz Franchetti
TAs: Srinivas Chellappa (Vas) and Frédéric de Mesmay (Fred)
Organization

- **Overview**
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly

- **Writing code for Intel’s SSE**
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks

- **Selected topics**
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU

- **Conclusion: How to write good vector code**
Organization

- **Overview**
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly

- **Writing code for Intel’s SSE**
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks

- **Selected topics**
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU

- **Conclusion: How to write good vector code**
SIMD (Signal Instruction Multiple Data) vector instructions in a nutshell

What are these instructions?
- Extension of the ISA. Data types and instructions for parallel computation on short (2-16) vectors of integers and floats

Why are they here?
- Useful: Many applications (e.g., multimedia) feature the required fine grain parallelism – code potentially faster
- Doable: Chip designers have enough transistors available, easy to implement
Evolution of Intel Vector Instructions

- **MMX (1996, Pentium)**
  - CPU-based MPEG decoding
  - Integers only, 64-bit divided into 2 x 32 to 8 x 8
  - Phased out with SSE4

- **SSE (1999, Pentium III)**
  - CPU-based 3D graphics
  - 4-way float operations, single precision
  - 8 new 128 bit Register, 100+ instructions

- **SSE2 (2001, Pentium 4)**
  - High-performance computing
  - Adds 2-way float ops, double-precision; same registers as 4-way single-precision
  - Integer SSE instructions make MMX obsolete

- **SSE3 (2004, Pentium 4E Prescott)**
  - Scientific computing
  - New 2-way and 4-way vector instructions for complex arithmetic

- **SSSE3 (2006, Core Duo)**
  - Minor advancement over SSE3

- **SSE4 (2007, Core2 Duo Penryn)**
  - Modern codecs, cryptography
  - New integer instructions
  - Better support for unaligned data, super shuffle engine

## Overview Floating-Point Vector ISAs

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Name</th>
<th>(v)-way</th>
<th>Precision</th>
<th>Introduced with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>SSE</td>
<td>4-way</td>
<td>single</td>
<td>Pentium III</td>
</tr>
<tr>
<td></td>
<td>SSE2</td>
<td>+</td>
<td>2-way</td>
<td>Pentium 4</td>
</tr>
<tr>
<td></td>
<td>SSE3</td>
<td></td>
<td></td>
<td>Pentium 4 (Prescott)</td>
</tr>
<tr>
<td></td>
<td>SSSE3</td>
<td></td>
<td></td>
<td>Core Duo</td>
</tr>
<tr>
<td></td>
<td>SSE4</td>
<td></td>
<td></td>
<td>Core2 Extreme (Penryn)</td>
</tr>
<tr>
<td>Intel</td>
<td>IPF</td>
<td>2-way</td>
<td>single</td>
<td>Itanium</td>
</tr>
<tr>
<td>AMD</td>
<td>3DNow!</td>
<td>2-way</td>
<td>single</td>
<td>K6</td>
</tr>
<tr>
<td></td>
<td>Enhanced 3DNow!</td>
<td></td>
<td></td>
<td>K7</td>
</tr>
<tr>
<td></td>
<td>3DNow! Professional</td>
<td>+</td>
<td>4-way</td>
<td>Athlon XP</td>
</tr>
<tr>
<td></td>
<td>AMD64</td>
<td>+</td>
<td>2-way</td>
<td>Opteron</td>
</tr>
<tr>
<td>Motorola</td>
<td>AltiVec</td>
<td>4-way</td>
<td>single</td>
<td>MPC 7400 G4</td>
</tr>
<tr>
<td>IBM</td>
<td>VMX</td>
<td>4-way</td>
<td>single</td>
<td>PowerPC 970 G5</td>
</tr>
<tr>
<td></td>
<td>SPU</td>
<td>+</td>
<td>2-way</td>
<td>Cell BE</td>
</tr>
<tr>
<td>IBM</td>
<td>Double FPU</td>
<td>2-way</td>
<td>double</td>
<td>PowerPC 440 FP2</td>
</tr>
</tbody>
</table>

Within a extension family, newer generations add features to older ones
Convergence: 3DNow! Professional = 3DNow! + SSE; VMX = AltiVec; SPU\(\frac{3}{4}\) VMX
Related Technologies

- **Original SIMD machines (CM-2, …)**
  - Don’t really have anything in common with SIMD vector extension

- **Vector Computers (NEC SX6, Earth simulator)**
  - Vector lengths of up to 128
  - High bandwidth memory, no memory hierarchy
  - Pipelined vector operations
  - Support strided memory access

- **Very long instruction word (VLIW) architectures (Itanium, …)**
  - Explicit parallelism
  - More flexible
  - No data reorganization necessary

- **Superscalar processors (x86, …)**
  - No explicit parallelism
  - Memory hierarchy

SIMD vector extensions borrow multiple concepts
How to use SIMD Vector Extensions?

- **Prerequisite:** fine grain parallelism
- **Helpful:** regular algorithm structure
- **Easiest way:** use existing libraries
  Intel MKL and IPP, Apple vDSP, AMD ACML, Atlas, FFTW, Spiral

- **Do it yourself:**
  - Use compiler vectorization: write vectorizable code
  - Use language extensions to explicitly issue the instructions
    Vector data types and intrinsic/builtin functions
    Intel C++ compiler, GNU C compiler, IBM VisualAge for BG/L,…
  - Implement kernels using assembly (inline or coding of full modules)
Characterization of Available Methods

- **Interface used**
  - Portable high-level language (possibly with pragmas)
  - Proprietary language extension (builtin functions and data types)
  - C++ Class interface
  - Assembly language

- **Who vectorizes**
  - Programmer or code generator expresses parallelism
  - Vectorizing compiler extracts parallelism

- **Structures vectorized**
  - Vectorization of independent loops
  - Instruction-level parallelism extraction

- **Generality of approach**
  - General purpose (e.g., for complex code or for loops)
  - Problem specific (for FFTs or for matrix products)
- limitations of compiler vectorization
- `C99_Complex` and `#pragma` help, but still slower than hand-vectorized code
Problems

- Correct data alignment paramount
- Reordering data kills runtime
- Algorithms must be adapted to suit machine needs
- Adaptation and optimization is machine/extension dependent
- Thorough understanding of ISA + Micro architecture required

One can easily slow down a program by vectorizing it
Organization

- **Overview**
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly

- **Writing code for Intel’s SSE**
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks

- **Selected topics**
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU

- **Conclusion: How to write good vector code**
Intel Streaming SIMD Extension (SSE)

- **Instruction classes**
  - Memory access (explicit and implicit)
  - Basic arithmetic (+, -, *)
  - Expensive arithmetic (1/x, sqrt(x), min, max, /, 1/sqrt)
  - Logic (and, or, xor, nand)
  - Comparison (+, <, >, …)
  - Data reorder (shuffling)

- **Data types**
  - float: __m128 (SSE)
  - double: __m128d (SSE2)
  - Integer: __m128i (8 bit – 128 bit)

- **Intel C++ Compiler Manual**
Intel C++ Compiler: Automatic Vectorization

- Example program: pointwise vector multiplication

```c
void func(float *c, float *a, float *b, int n) {
    for (int i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```

- Compiler invocation

```
C:\>iclvars > NUL
C:\>C>icl /Qc99 /Qrestrict /O3 /QxW /Qvec-report3 /FAs /c
  test.c

Intel(R) C++ Compiler for 32-bit applications, Version 9.1
Copyright (C) 1985-2006 Intel Corporation. All rights
reserved.

test.c

test.c(2) : (col. 5) remark: LOOP WAS VECTORIZED.
```
Language Support and Directives

This topic addresses language features that better help to vectorize code. The __declspec(alignment) declaration enables you to overcome hardware alignment constraints. The restrict qualifier and the pragma address the stylistic issues due to lexical scope, data dependency, and ambiguity resolution.

Language Support

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__declspec(alignment)</td>
<td>Directs the compiler to align the variable to an n-byte boundary. Address of the variable is address mod n=0.</td>
</tr>
<tr>
<td>__declspec(alignment, off)</td>
<td>Directs the compiler to align the variable to an n-byte boundary with offset off within each n-byte boundary. Address of the variable is address mod n=off.</td>
</tr>
<tr>
<td>restrict</td>
<td>Permits the disambiguator flexibility in alias assumptions, which enables more vectorization.</td>
</tr>
<tr>
<td>__assume_aligned(a, n)</td>
<td>Instructs the compiler to assume that array a is aligned on an n-byte boundary, used in cases where the compiler has failed to obtain alignment information.</td>
</tr>
<tr>
<td>#pragma ivdep</td>
<td>Instructs the compiler to ignore assumed vector dependencies.</td>
</tr>
<tr>
<td>#pragma vector</td>
<td>Specifies how to vectorize the loop and indicates that efficiency heuristics should be ignored.</td>
</tr>
<tr>
<td>#pragma nonvector</td>
<td>Specifies that the loop should never be vectorized.</td>
</tr>
</tbody>
</table>
Intel C++ Compiler: Options and Output

- On Windows Intel C++ compiler requires VisualStudio
- On command line `iclvars.cmd` initializes the environment

**Compiler Options**

- C99 syntax: `/Qc99 /Qrestrict`
- Full optimization: `/O3`
- Vectorization target: SSE2 `/QxW`
  other targets: `/QxK (SSE) , /QxP (SSE3), /QxT (SSSE), /QxS (SSE4)`
- Vectorization report: `/Qvec-report3`
- Assembly output (source + assembly): `/FAs`

**Check vectorization quality: Checking output assembly**

```
$B1$17:                         ; Preds $B1$17 $B1$16
  movups xmm0, XMMWORD PTR [ecx+edi*4] ;3.16
  mulps  xmm0, XMMWORD PTR [edx+edi*4] ;3.23
  movaps XMMWORD PTR [esi+edi*4], xmm0 ;3.9
  movups xmm1, XMMWORD PTR [ecx+edi*4+16] ;3.16
  mulps  xmm1, XMMWORD PTR [edx+edi*4+16] ;3.23
  movaps XMMWORD PTR [esi+edi*4+16], xmm1 ;3.9
  add    edi, 8                   ;2.5
```
Intel C++ Compiler: Language Extension

- **Language extension**
  - C99 “restrict” keyword
  - Aligned C library functions: \_mm\_malloc(), \_mm\_free()
  - \_assume\_aligned()
  - \_declspec(__align())
  - Pragmas
    - #pragma vector aligned | unaligned | always
    - #pragma ivdep
    - #pragma novector

- **Example using language extension**
  ```cpp
  void func(float *restrict c, float *restrict a, 
             float *restrict b, int n) {
    #pragma vector always
    for (int i=0; i<n; i++)
      c[i] = a[i] * b[i];
  }
  ```
Intel SSE Intrinsics Interface

- **Data types**
  - `__m128 f; // ={float f3, f2, f1, f0}`
  - `__m128d d; // ={double d1, d0}`

- **Intrinsics**
  - Native instructions: `_mm_add_ps()`, `_mm_mul_ps()`,...
  - Multi-instruction: `_mm_setr_ps()`, `_mm_set1_ps`, ...

- **Macros**
  - Transpose: `_MM_TRANSPOSE4_PS()`,...
  - Helper: `_MM_SHUFFLE()`
Load Operations for Streaming SIMD Extensions

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the _mmintrin.h header file.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with R0-R3. RC, R1, R2 and R3 each represent one of the 4 32-bit pieces of the result register.

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_loadh_pi</td>
<td>Load high</td>
<td>MOVHPS reg, mem</td>
</tr>
<tr>
<td>_mm_loadl_pi</td>
<td>Load low</td>
<td>MOVLP3 reg, mem</td>
</tr>
<tr>
<td>_mm_load8_pi</td>
<td>Load the low value and clear the three high values</td>
<td>MOVSS</td>
</tr>
<tr>
<td>_mm_load4_ps</td>
<td>Load one value into all four words</td>
<td>MOVSS + Shuffling</td>
</tr>
<tr>
<td>_mm_load4_pi</td>
<td>Load four values, address aligned</td>
<td>MOVUPS</td>
</tr>
<tr>
<td>_mm_loadu4_ps</td>
<td>Load four values, address unaligned</td>
<td>MOVUPS</td>
</tr>
<tr>
<td>_mm_loadr_ps</td>
<td>Load four values in reverse</td>
<td>MOVAPS + Shuffling</td>
</tr>
</tbody>
</table>

__m128_mm_loadh_pi(__m128 a, __m64 const *p)

Sets the upper two SP FP values with 64 bits of data loaded from the address p.
# Intel SSE: Vector Arithmetic

## Arithmetic Operations for Streaming SIMD Extensions

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the `xmmintrin.h` header file.

The results of each intrinsic operation are placed in a register. This register is illustrated for each intrinsic with RO-R3, R0, R1, R2 and R3 each represent one of the 4 32-bit pieces of the result register.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_add_ss</td>
<td>Addition</td>
<td>ADDSS</td>
</tr>
<tr>
<td>_mm_add_ps</td>
<td>Addition</td>
<td>ADDP5</td>
</tr>
<tr>
<td>_mm_sub_ss</td>
<td>Subtraction</td>
<td>SUBSS</td>
</tr>
<tr>
<td>_mm_sub_ps</td>
<td>Subtraction</td>
<td>SUBPS</td>
</tr>
<tr>
<td>_mm_mul_ss</td>
<td>Multiplication</td>
<td>MULSS</td>
</tr>
<tr>
<td>_mm_mul_ps</td>
<td>Multiplication</td>
<td>MULPS</td>
</tr>
<tr>
<td>_mm_div_ss</td>
<td>Division</td>
<td>DIVSS</td>
</tr>
<tr>
<td>_mm_div_ps</td>
<td>Division</td>
<td>DIVPS</td>
</tr>
<tr>
<td>_mm_sqrt_ss</td>
<td>Squared Root</td>
<td>SQRTSS</td>
</tr>
<tr>
<td>_mm_sqrt_ps</td>
<td>Squared Root</td>
<td>SQRTPS</td>
</tr>
<tr>
<td>_mm_rcp_ss</td>
<td>Reciprocal</td>
<td>RCPSS</td>
</tr>
<tr>
<td>_mm_rcp_ps</td>
<td>Reciprocal</td>
<td>RCPPS</td>
</tr>
<tr>
<td>_mm_rsqrt_ss</td>
<td>Reciprocal Squared Root</td>
<td>RSQRTSS</td>
</tr>
<tr>
<td>_mm_rsqrt_ps</td>
<td>Reciprocal Squared Root</td>
<td>RSQRTPS</td>
</tr>
<tr>
<td>_mm_min_ss</td>
<td>Computes Minimum</td>
<td>MINSS</td>
</tr>
</tbody>
</table>
Single-precision Floating-point Vector Intrinsics for Streaming SIMD Extensions 3

The single-precision floating-point vector intrinsics listed here are designed for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3).

The results of each intrinsic operation are placed in the registers R0, R1, R2, and R3.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

The prototypes for these intrinsics are in the pmmintrin.h header file.

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE3 Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_addsub_ps</td>
<td>Subtract and add</td>
<td>ADDSUBPS</td>
</tr>
<tr>
<td>_mm_hadd_ps</td>
<td>Add</td>
<td>HADDPS</td>
</tr>
<tr>
<td>_mm_hsub_ps</td>
<td>Subtracts</td>
<td>HESUBPS</td>
</tr>
<tr>
<td>_mm_moveldup_ps</td>
<td>Duplicates</td>
<td>MOV3SIDUP</td>
</tr>
<tr>
<td>_mm_moveldup_ps</td>
<td>Duplicates</td>
<td>MOV3SIDUP</td>
</tr>
</tbody>
</table>

extern _m128 _mm_addsub_ps(_m128 a, _m128 b);  
Subtracts even vector elements while adding odd vector elements.

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0 - b0;</td>
<td>a1 + b1;</td>
<td>a2 - b2;</td>
<td>a3 + b3;</td>
</tr>
</tbody>
</table>

extern _m128 _mm_hadd_ps(_m128 a, _m128 b);  
Adds adjacent vector elements.

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0 + a1;</td>
<td>a2 + a3;</td>
<td>b0 + b1;</td>
<td>b2 + b3;</td>
</tr>
</tbody>
</table>
Intel SSE: Reorder Instructions

Miscellaneous Intrinsics Using Streaming SIMD Extensions

The prototypes for Streaming SIMD Extensions (SSE) intrinsics are in the intrin.h header file.

The results of each intrinsic operation are placed in registers. The information about what is placed in each register appears in the tables below, in the detailed explanation of each intrinsic. R0, R1, R2 and R3 represent the registers in which results are placed.

To see detailed information about an intrinsic, click on that intrinsic name in the following table.

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_shuffle_ps</td>
<td>Shuffle</td>
<td>SHUFFPS</td>
</tr>
<tr>
<td>_mm_unpackhi_ps</td>
<td>Unpack High</td>
<td>UNPKHIHPS</td>
</tr>
<tr>
<td>_mm_unpacklo_ps</td>
<td>Unpack Low</td>
<td>UNPKHLPS</td>
</tr>
<tr>
<td>_mm_move_si</td>
<td>Set low word, pass in three high values</td>
<td>MCVSS</td>
</tr>
<tr>
<td>_mm_movehl_ps</td>
<td>Move High to Low</td>
<td>MOVEHLPS</td>
</tr>
<tr>
<td>_mm_movehl_ps</td>
<td>Move Low to High</td>
<td>MOVLHPS</td>
</tr>
<tr>
<td>_mm_movemask_ps</td>
<td>Create four-bit mask</td>
<td>MOVEMASKPS</td>
</tr>
</tbody>
</table>

__m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)___
Selects four specific FP values from a and b, based on the mask imm8. The mask must be an immediate. See Macro Function for Shuffle Using Streaming SIMD Extensions for a description of the shuffle semantics.

__m128 _mm_unpackhi_ps(__m128 a, __m128 b)___
Organization

- **Overview**
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly

- **Writing code for Intel’s SSE**
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks

- **Selected topics**
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU

- **Conclusion: How to write good vector code**
Intel SSE: Transpose Macro

Macro Function for Matrix Transposition

The Streaming SIMD Extensions (SSE) provide the following macro function to transpose a 4 by 4 matrix of single precision floating point values.

```
_MM_TRANSPOSE4_PS(row0, row1, row2, row3)
```

The arguments row0, row1, row2, and row3 are __m128 values whose elements form the corresponding rows of a 4 by 4 matrix. The matrix transposition is returned in arguments row0, row1, row2, and row3 where row0 now holds column 0 of the original matrix, row1 now holds column 1 of the original matrix, and so on.

The transposition function of this macro is illustrated in the "Matrix Transposition Using the _MM_TRANSPOSE4_PS" figure.
Example: Complex Multiplication SSE3

\[(a + ib)(c + id) = (ac - bd) + i(ad + bc)\]

Memory

Result:
4 load/stores
3 arithm. ops.
1 reorder op

Not available in SSE2

Memory
Looking at the Assembly

**SSE3:**

```
movapd  xmm0, XMMWORD PTR A
movddup xmm2, QWORD PTR B
mulpd  xmm2, xmm0
movddup xmm1, QWORD PTR B+8
shufpd xmm0, xmm0, 1
mulpd  xmm1, xmm0
addsubpd xmm2, xmm1
movapd  XMMWORD PTR C, xmm2
```

**SSE2:**

```
movsd  xmm3, QWORD PTR A
movapd  xmm4, xmm3
movsd  xmm5, QWORD PTR A+8
movapd  xmm0, xmm5
movsd  xmm1, QWORD PTR B
mulpd  xmm4, xmm1
mulpd  xmm5, xmm1
mulsd  xmm0, xmm2
mulsd  xmm3, xmm2
mulsd  xmm4, xmm0
subsd  QWORD PTR C, xmm4
addsd  QWORD PTR C, xmm5
movsd  QWORD PTR C, xmm5
```

In SSE2 Intel C++ generates *scalar* code (better?)
Organization

- **Overview**
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly

- **Writing code for Intel’s SSE**
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks

- **Selected topics**
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU

- **Conclusion: How to write good vector code**
**Intel SSE: Integer Modes**

Register Insertion/Extraction Intrinsics for Streaming SIMD Extensions 4

These intrinsics enable data insertion and extraction between general purpose registers and XMM registers.

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Operation</th>
<th>Corresponding SSE-4 Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>__mm_insert_ps</td>
<td>Insert single precision float into packed single precision array element selected by index</td>
<td>INSERTPS</td>
</tr>
<tr>
<td>__mm_extract_ps</td>
<td>Extract single precision float from packed single precision array element selected by index</td>
<td>EXTRACTPS</td>
</tr>
<tr>
<td>__mm_extract_epi8</td>
<td>Extract integer byte from packed integer array element selected by index</td>
<td>PEXTPB</td>
</tr>
<tr>
<td>__mm_extract_epi32</td>
<td>Extract integer double word from packed integer array element selected by index</td>
<td>PEXTPD</td>
</tr>
<tr>
<td>__mm_extract_epi64</td>
<td>Extract integer quad word from packed integer array element selected by index</td>
<td>PEXTPQ</td>
</tr>
<tr>
<td>__mm_extract_epi16</td>
<td>Extract integer word from packed integer array element selected by index</td>
<td>PEXTRU</td>
</tr>
<tr>
<td>__mm_insert_epi8</td>
<td>Insert integer byte into packed integer array element selected by index</td>
<td>PINSRB</td>
</tr>
<tr>
<td>__mm_insert_epi32</td>
<td>Insert integer double word into packed integer array element selected by index</td>
<td>PINSRD</td>
</tr>
<tr>
<td>__mm_insert_epi64</td>
<td>Insert integer quad word into packed integer array element selected by index</td>
<td>PINSRPQ</td>
</tr>
</tbody>
</table>
SSE Integer Modes (1)

- **SSE generations**
  - Introduced with SSE2
  - Functionality extended drastically with SSSE3 and SSE4

- **Modes**
  - 1x128 bit, 2x64 bit, 4x32 bit 8x 16 bit, 16x8 bit
  - Signed and unsigned
  - Saturating and non-saturating

- **Operations**
  - Arithmetic, logic, and shift, mullo/hi
  - Compare, test; min, max, and average
  - Conversion from/to floating-point, across precisions
  - Load/store/set
  - Shuffle, insert, extract, blend
SSE Integer Modes (2)

- **Interoperability**
  - Integer operations can be used with floating-point data
  - Typecast support

- **Problems**
  - Only subset of operations available in each mode
  - Sometimes need to “build” operation yourself
  - Gathers and scatters even more expensive (8- and 16-way)

```c
// right-shift for signed __int8 16-way
__forceinline __m128i _mm_srlv_epi8(__m128i x, int sh) {
  __m128i signs = _mm_and_si128(x, _mm_set1_epi32(0x80808080));
  __m128i z = _mm_srlv_epi16(x, 1);
  z = _mm_and_si128(z, _mm_set1_epi32(0x7f7f7f7f));
  return _mm_or_si128(z, signs);
}
```
Extending Floating-Point Functionality

- **Sign change**
  - No sign-change instruction for vector elements exist
  - Integer exclusive-or helps

    ```
    // sign-change of second vector element
    __forceinline __m128 _mm_chsgn2_ps(__m128 f) {
        return _castsi128_ps(_mm_xor_si128(
            _mm_castps_si128(f),
            _mm_castps_si128(_mm_set_ps(0.0,0.0,-0.0,0.0))));
    }
    ```

- **Align instruction**
  - `alignr` only exists for signed 8-bit integer

    ```
    // alignr 4-way float variant
    __forceinline __m128 _mm_alignr_ps(__m128 f1, __m128 f2, int sh) {
        return _castsi128_ps(_mm_alignr_epi8(
            _mm_castps_si128(f1), _mm_castps_si128(f2), sh));
    }
    ```
Organization

- **Overview**
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly

- **Writing code for Intel’s SSE**
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks

- **Selected topics**
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU

- **Conclusion: How to write good vector code**
AltiVec, VMX, Cell BE PPU and SPU,…

- **AltiVec**: 4-way float, 4-, 8-, and 16-way integer
  - Introduced with Motorola MPC 7400 G4 (direct competitor to Intel SSE and Pentium III)
  - Gave big boost to Apple multimedia applications
  - Still available in Freescale PowerPC processors
  - Supported by GNU C builtin functions (2.95, 3.X)

- **AltiVec became IBM VMX**
  - PowerPC 970 G5 (G4 successor) and POWER6
  - Cell BE PPU (PowerPC)
  - VMX128 version for Xbox 360 (Xenon processor)

- **Cell SPU**: closely aligned with VMX
  - Double-precision instructions (very slow at this point)
AltiVec vs. SSE

- **AltiVec**: PowerPC is 3-operand RISC
  - Fused multiply-add
  - Powerful general shuffle instruction
  - More registers (32 – 128)

- **Problem: non-vector memory access**
  - Unaligned load/store
  - Subvector load/store

- **AltiVec/VMX is not changing as quickly as SSE**
  - Variants: AltiVec/VMX, VMX128, SPU
  - AltiVec important in embedded computing
  - SSE is closer to the consumer market, permanently updated
Organization

- **Overview**
  - Idea, benefits, reasons, restrictions
  - History and state-of-the-art floating-point SIMD extensions
  - How to use it: compiler vectorization, class library, intrinsics, inline assembly

- **Writing code for Intel’s SSE**
  - Compiler vectorization
  - Intrinsics: instructions
  - Intrinsics: common building blocks

- **Selected topics**
  - SSE integer instructions
  - Other SIMD extensions: AltiVec/VMX, Cell SPU

- **Conclusion: How to write good vector code**
How to Write Good Vector Code?

- Take the “right” algorithm and the “right” data structures
  - Fine grain parallelism
  - Correct alignment in memory
  - Contiguous arrays

- Use a good compiler (e.g., vendor compiler)

- First: Try compiler vectorization
  - Right options, pragmas and dynamic memory functions
    (Inform compiler about data alignment, loop independence,…)
  - Check generated assembly code and runtime

- If necessary: Write vector code yourself
  - Most expensive subroutine first
  - Use intrinsics, no (inline) assembly
  - Important: Understand the ISA
Remaining time: Discussion