Intel SIMD architecture

Overview
- SIMD
- MMX architectures
- MMX instructions
- examples
- SSE/SSE2

SIMD instructions are probably the best place to use assembly since compilers usually do not do a good job on using these instructions.

Performance boost
- Increasing clock rate is not fast enough for boosting performance

In his 1965 paper, Intel co-founder Gordon Moore observed that “the number of transistors per square inch had doubled every 18 months.”

Performance boost
- Architecture improvements (such as pipeline/cache/SIMD) are more significant
- Intel analyzed multimedia applications and found they share the following characteristics:
  - Small native data types (8-bit pixel, 16-bit audio)
  - Recurring operations
  - Inherent parallelism
SIMD

- SIMD (single instruction multiple data) architecture performs the same operation on multiple data elements in parallel

  **PADDW MM0, MM1**

<table>
<thead>
<tr>
<th>Source 1</th>
<th>X3</th>
<th>X2</th>
<th>X1</th>
<th>X0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source 2</td>
<td>Y3</td>
<td>Y2</td>
<td>Y1</td>
<td>Y0</td>
</tr>
</tbody>
</table>

  * OP

  **Destination**

  X3 OP Y3 | X2 OP Y2 | X1 OP Y1 | X0 OP Y0

- IA-32 SIMD development

  - MMX (Multimedia Extension) was introduced in 1996 (Pentium with MMX and Pentium II).
  - SSE (Streaming SIMD Extension) was introduced with Pentium III.
  - SSE2 was introduced with Pentium 4.
  - SSE3 was introduced with Pentium 4 supporting hyper-threading technology. SSE3 adds 13 more instructions.

SISD/SIMD/Streaming

- Typical elements are small, 8 bits for pixels, 16 bits for audio, 32 bits for graphics and general computing.

MMX

- After analyzing a lot of existing applications such as graphics, MPEG, music, speech recognition, game, image processing, they found that many multimedia algorithms execute the same instructions on many pieces of data in a large data set.

- Typical elements are small, 8 bits for pixels, 16 bits for audio, 32 bits for graphics and general computing.

- New data type: 64-bit packed data type. Why 64 bits?
  - Good enough
  - Practical
**MMX data types**

- Packed Byte: 8 bytes packed into 64 bits
  
  63 8 7 0

- Packed Word: 4 words packed into 64 bits
  
  63 16 15 0

- Packed Doubleword: 2 doublewords packed into 64 bits
  
  63 32 31 0

- Packed Quadword: One 64-bit quantity
  
  63 0

**MMX integration into IA**

NaN or infinity as real because bits 79-64 are ones.

Even if MMX registers are 64-bit, they don’t extend Pentium to a 64-bit CPU since only logic instructions are provided for 64-bit data.

**Compatibility**

- To be fully compatible with existing IA, no new mode or state was created. Hence, for context switching, no extra state needs to be saved.
- To reach the goal, MMX is hidden behind FPU. When floating-point state is saved or restored, MMX is saved or restored.
- It allows existing OS to perform context switching on the processes executing MMX instruction without be aware of MMX.
- However, it means MMX and FPU can not be used at the same time. Big overhead to switch.

**Compatibility**

- Although Intel defenses their decision on aliasing MMX to FPU for compatibility. It is actually a bad decision. OS can just provide a service pack or get updated.
- It is why Intel introduced SSE later without any aliasing.
**MMX instructions**

- 57 MMX instructions are defined to perform the parallel operations on multiple data elements packed into 64-bit data types.
- These include add, subtract, multiply, compare, and shift, data conversion, 64-bit data move, 64-bit logical operation and multiply-add for multiply-accumulate operations.
- All instructions except for data move use MMX registers as operands.
- Most complete support for 16-bit operations.

**Saturation arithmetic**

- Useful in graphics applications.
- When an operation overflows or underflows, the result becomes the largest or smallest possible representable number.
- Two types: signed and unsigned saturation

```
<table>
<thead>
<tr>
<th>Category</th>
<th>Wraparound</th>
<th>Signed Saturation</th>
<th>Unsigned Saturation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>Addition</td>
<td>PADD, PADDW, PADDQ</td>
<td>PADDUB, PADDUSB, PADDUSW</td>
</tr>
<tr>
<td></td>
<td>Subtraction</td>
<td>PSUBB, PSUBW, PSUBLB</td>
<td>PSUBUSB, PSUBUSB, PSUBUSW</td>
</tr>
<tr>
<td></td>
<td>Multiplication</td>
<td>PMULL, PMULLH, PMADD</td>
<td></td>
</tr>
<tr>
<td>Comparison</td>
<td>Compare for Equal</td>
<td>PCMPEQB, PCMPEQW, PCMPEPD, PCMPEPBD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Compare for Greater Than</td>
<td>PCMPEGTB, PCMPEGTVP, PCMPEGTWD</td>
<td></td>
</tr>
<tr>
<td>Conversion</td>
<td>Pack</td>
<td>PACKSSWB, PACKSSDW</td>
<td>PACKUSWB</td>
</tr>
<tr>
<td>Unpack</td>
<td>Unpack High</td>
<td>PUNPCKHDQW, PUNPCKHWD</td>
<td>PUNPCKHDQW, PUNPCKHWD</td>
</tr>
<tr>
<td></td>
<td>Unpack Low</td>
<td>PUNPCKLQW, PUNPCKLWD</td>
<td>PUNPCKLQW, PUNPCKLWD</td>
</tr>
</tbody>
</table>
```

---

**MMX instructions**

Call it before you switch to FPU from MMX; Expensive operation
Arithmetic

• **PADDB/PADDW/PADDD**: add two packed numbers, no EFLAGS is set, ensure overflow never occurs by yourself

• Multiplication: two steps

• **PMULLW**: multiplies four words and stores the four lo words of the four double word results

• **PMULHW/PMULHUW**: multiplies four words and stores the four hi words of the four double word results. **PMULHUW** for unsigned.

Detect MMX/SSE

```
mov eax, 1 ; request version info
cpuid ; supported since Pentium
test edx, 00800000h ;bit 23
    ; 02000000h (bit 25) SSE
    ; 04000000h (bit 26) SSE2
jnz HasMMX
```

cpuid

<table>
<thead>
<tr>
<th>Initial EAX Value</th>
<th>Information Provided about the Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>Basic CPUID Information</td>
</tr>
<tr>
<td>01H</td>
<td>Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6)</td>
</tr>
<tr>
<td>02H</td>
<td>Cache and TLB Information (see Table 3-17)</td>
</tr>
</tbody>
</table>
Example: add a constant to a vector

```c
char d[]={5, 5, 5, 5, 5, 5, 5, 5};
char clr[]={65, 66, 68, 87, 88}; // 24 bytes
__asm{
    movq mm1, d
    movq mm0, d
    mov cx, 3
    mov esi, 0
    L1: movq mm0, clr[esi]
        paddb mm0, mm1
        movq clr[esi], mm0
        add esi, 8
    loop L1
    emms
}
```

Comparison

- No CFLAGS, how many flags will you need?
- Results are stored in destination.
- EQ/GT, no LT

Change data types

- Pack: converts a larger data type to the next smaller data type.
- Unpack: takes two operands and interleave them. It can be used for expand data type for immediate calculation.
Unpack low portion

Unpack high portion

Keys to SIMD programming
- Efficient data layout
- Elimination of branches

Application: frame difference
Application: frame difference

\[(A-B) \text{ or } (B-A)\]

Example: image fade-in-fade-out

\[A^\alpha + B^{1-\alpha} = B + \alpha(A-B)\]

\[\alpha = 0.75\]
**Example: image fade-in-fade-out**

- Two formats: planar and chunky
- In Chunky format, 16 bits of 64 bits are wasted
- So, we use planar in the following example

---

**Example: image fade-in-fade-out**

1. Unpack byte R pixel components from image A & B
   - \( \alpha = 0.5 \)

2. Subtract image B from image A
   - \( \alpha = 0.25 \)

3. Multiply subtract result by fade value
   - \( \alpha = 0.25 \)

4. Add image B pixels
   - \( \alpha = 0.25 \)

5. Pack new composite pixels back to bytes
   - \( \alpha = 0.25 \)

---
Example: image fade-in-fade-out

```assembly
MOVQ mm0, alpha // 16-b zero-padding
MOVQ mm1, A // move 4 pixels of image A
MOVQ mm2, B // move 4 pixels of image B
PXOR mm3, mm3 // clear mm3 to all zeroes
// unpack 4 pixels to 4 words
PUNPCKLBW mm1, mm3 // Because B-A could be negative, need 16 bits
PSUBW mm1, mm2 // (B-A)
PMULHW mm1, mm0 // (B-A)*fade/256
PADDW mm1, mm2 // (B-A)*fade + B
// pack four words back to four bytes
PACKUSWB mm1, mm3
```

Data-independent computation

- Each operation can execute without needing to know the results of a previous operation.
- Example, sprite overlay

```
for i=1 to sprite_Size
  if sprite[i]=clr
    then out_color[i]=bg[i]
  else out_color[i]=sprite[i]
```

- How to execute data-dependent calculations on several pixels in parallel.

Application: sprite overlay

```
MOVQ mm0, sprite
MOVQ mm2, mm0
MOVQ mm4, bg
MOVQ mm1, clr
PCMPEQW mm0, mm1
PAND mm4 mm0
PAND mm4 mm0
POR mm0, mm4
```
char M1[4][8]; // matrix to be transposed
char M2[8][4]; // transposed matrix
int n=0;
for (int i=0;i<4;i++)
    for (int j=0;j<8;j++)
        { M1[i][j]=n; n++; }

__asm{
    // move the 4 rows of M1 into MMX registers
    movq mm1,M1
    movq mm2,M1+8
    movq mm3,M1+16
    movq mm4,M1+24

    // generate rows 1 to 4 of M2
    punpcklwb mm1, mm2
    punpcklwb mm3, mm4
    movq mm0, mm1
    punpcklwd mm1, mm3 // mm1 has row 2 & row 1
    punpckhwd mm0, mm3 // mm0 has row 4 & row 3
    movq M2, mm1
    movq M2+8, mm0
}

// generate rows 5 to 8 of M2
movq mm1, M1 // get row 1 of M1
movq mm3, M1+16 // get row 3 of M1
punpckhbw mm1, mm2
punpckhbw mm3, mm4
movq mm0, mm1
punpcklwd mm1, mm3 // mm1 has row 6 & row 5
punpckhwd mm0, mm3 // mm0 has row 8 & row 7
// save results to M2
movq M2+16, mm1
movq M2+24, mm0
emms
} //end
Performance boost (data from 1996)

Benchmark kernels:
- FFT, FIR, vector dot-product, IDCT, motion compensation.

65% performance gain

Lower the cost of multimedia programs by removing the need of specialized DSP chips

How to use assembly in projects

- Write the whole project in assembly
- Link with high-level languages
- Inline assembly
- Intrinsics

Link ASM and HLL programs

- Assembly is rarely used to develop the entire program.
- Use high-level language for overall project development
  - Relieves programmer from low-level details
- Use assembly language code
  - Speed up critical sections of code
  - Access nonstandard hardware devices
  - Write platform-specific code
  - Extend the HLL’s capabilities

General conventions

- Considerations when calling assembly language procedures from high-level languages:
  - Both must use the same naming convention (rules regarding the naming of variables and procedures)
  - Both must use the same memory model, with compatible segment names
  - Both must use the same calling convention
Inline assembly code

- Assembly language source code that is inserted directly into a HLL program.
- Compilers such as Microsoft Visual C++ and Borland C++ have compiler-specific directives that identify inline ASM code.
- Efficient inline code executes quickly because CALL and RET instructions are not required.
- Simple to code because there are no external names, memory models, or naming conventions involved.
- Decidedly not portable because it is written for a single platform.

__asm directive in Microsoft Visual C++

- Can be placed at the beginning of a single statement
- Or, it can mark the beginning of a block of assembly language statements

Syntax:

```c
__asm
{
    statement-1
    statement-2
    ...
    statement-n
}
```

Intrinsics

- An intrinsic is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions.
- The compiler manages things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.
- Intrinsic functions are inherently more efficient than called functions because no calling linkage is required. But, not necessarily as efficient as assembly.
- `_mm_<opcode>_suffix` ps: packed single-precision
- `_mm_<opcode>_suffix` ss: scalar single-precision

```c
#include <xmmintrin.h>

__m128 a, b, c;
c = _mm_add_ps( a, b );

float a[4], b[4], c[4];
for( int i = 0; i < 4; ++i )
    c[i] = a[i] + b[i];

// a = b * c + d / e;
__m128 a = _mm_add_ps( _mm_mul_ps( b, c ), _mm_div_ps( d, e ) );
```
SSE

- Adds eight 128-bit registers
- Allows SIMD operations on packed single-precision floating-point numbers
- Most SSE instructions require 16-aligned addresses

SSE features

- Add eight 128-bit data registers (XMM registers) in non-64-bit modes; sixteen XMM registers are available in 64-bit mode.
- 32-bit MXCSR register (control and status)
- Add a new data type: 128-bit packed single-precision floating-point (4 FP numbers.)
- Instruction to perform SIMD operations on 128-bit packed single-precision FP and additional 64-bit SIMD integer operations.
- Instructions that explicitly prefetch data, control data cacheability and ordering of store

SSE programming environment

MXCSR control and status register

Generally faster, but not compatible with IEEE 754
Exception

```c
_exception_MM_ALIGN16 float test1[4] = { 0, 0, 0, 1 };
_exception_MM_ALIGN16 float test2[4] = { 1, 2, 3, 0};
_exception_MM_ALIGN16 float out[4];
_exception_MM_SET_EXCEPTION_MASK(0);//enable exception
__try {
    __m128 a = _mm_load_ps(test1);
    __m128 b = _mm_load_ps(test2);
    a = _mm_div_ps(a, b);
    _mm_store_ps(out, a);
} __except(EXCEPTION_EXECUTE_HANDLER) {
    if(_mm_getcsr() & _MM_EXCEPT_DIV_ZERO)
        cout << "Divide by zero" << endl;
    return;
}
```

SSE packed FP operation

- **ADDPS/SUBPS**: packed single-precision FP

SSE scalar FP operation

- **ADDSS/SUBSS**: scalar single-precision FP

SSE2

- Provides ability to perform SIMD operations on double-precision FP, allowing advanced graphics such as ray tracing
- Provides greater throughput by operating on 128-bit packed integers, useful for RSA and RC5
**SSE2 features**

- Add data types and instructions for them

  - 128-Bit Packed Double-Precision Floating-Point
  - 128-Bit Packed Byte Integers
  - 128-Bit Packed Word Integers
  - 128-Bit Packed Doubleword Integers
  - 128-Bit Packed Quadword Integers

  - Programming environment unchanged

---

**Example**

```c
void add(float *a, float *b, float *c) {
    for (int i = 0; i < 4; i++)
        c[i] = a[i] + b[i];
}
```

```asm
movaps: move aligned packed single-precision FP
mov eax, a
addps edx, b
mov ecx, c
movaps xmm0, XMMWORD PTR [eax]
addps xmm0, XMMWORD PTR [edx]
movaps XMMWORD PTR [ecx], xmm0
}
```

---

**SSE Shuffle (SHUFPS)**

**SHUFPS xmm1, xmm2, imm8**

Select[1..0] decides which DW of DEST to be copied to the 1st DW of DEST

---

**SSE Shuffle (SHUFPS)**

CASE (SELECT[1:0]) OF
- 0: DEST[31:0] ← DEST[31:0];
- 1: DEST[31:0] ← DEST[63:32];
- 2: DEST[31:0] ← DEST[95:64];
- 3: DEST[31:0] ← DEST[127:96];

CASE (SELECT[5:4]) OF
- 0: DEST[95:64] ← SRC[31:0];
- 1: DEST[95:64] ← SRC[63:32];
- 2: DEST[95:64] ← SRC[95:64];
- 3: DEST[95:64] ← SRC[127:96];

CASE (SELECT[3:2]) OF
- 0: DEST[63:32] ← DEST[31:0];
- 1: DEST[63:32] ← DEST[63:32];
- 2: DEST[63:32] ← DEST[95:64];
- 3: DEST[63:32] ← DEST[127:96];

CASE (SELECT[7:6]) OF
- 0: DEST[127:96] ← SRC[31:0];
- 1: DEST[127:96] ← SRC[63:32];
- 2: DEST[127:96] ← SRC[95:64];
- 3: DEST[127:96] ← SRC[127:96];

ESAC;
Example (cross product)

Vector cross(const Vector& a, const Vector& b) {
    return Vector(
        (a[2] * b[0] - a[0] * b[2]),
        (a[0] * b[1] - a[1] * b[0]));
}

Example (cross product)

/* cross */
__m128 _mm_cross_ps(__m128 a, __m128 b) {
    __m128 ea, eb;
    // set to a[1][2][0][3], b[2][0][1][3]
    ea = _mm_shuffle_ps(a, a, _MM_SHUFFLE(3,0,2,1));
    eb = _mm_shuffle_ps(b, b, _MM_SHUFFLE(3,1,0,2));
    // multiply
    __m128 xa = _mm_mul_ps(ea, eb);
    // set to a[2][1][0][3], b[1][2][0][3]
    a = _mm_shuffle_ps(a, a, _MM_SHUFFLE(3,1,0,2));
    b = _mm_shuffle_ps(b, b, _MM_SHUFFLE(3,0,2,1));
    // multiply
    __m128 xb = _mm_mul_ps(a, b);
    // subtract
    return _mm_sub_ps(xa, xb);
}

Example: dot product

- Given a set of vectors \(\{v_1,v_2,...,v_n\} = \{(x_1,y_1,z_1), (x_2,y_2,z_2), ..., (x_n,y_n,z_n)\}\) and a vector \(v_c=(x_c,y_c,z_c)\), calculate \(\{v_c \cdot v_i\}\)
- Two options for memory layout
- Array of structure (AoS)

typedef struct {
    float dc, x, y, z;
} Vertex;

Vertex v[n];

- Structure of array (SoA)

typedef struct {
    float x[n], y[n], z[n];
} VerticesList;

VerticesList v;

Example: dot product (AoS)

movaps xmm0, v ; xmm0 = DC, x0, y0, z0
movaps xmm1, vc ; xmm1 = DC, xc, yc, zc
mulps xmm0, xmm1 ; xmm0=x0*c, y0*yc, z0*zc
movhpls xmm1, xmm0 ; xmm1= DC, DC, DC, x0*xc
addps xmm1, xmm0 ; xmm1 = DC, DC, DC,
                   ; x0*xc+z0*zc
movaps xmm2, xmm0
shufps xmm2, xmm2, 55h ; xmm2=DC,DC,DC,y0*yc
addps xmm1, xmm2 ; xmm1 = DC, DC, DC,
                   ; x0*xc+y0*yc+z0*zc

movhpls:DEST[63..0] := SRC[127..64]
**Example: dot product (SoA)**

\[
\begin{align*}
X &= x_1, x_2, \ldots, x_3 \\
Y &= y_1, y_2, \ldots, y_3 \\
Z &= z_1, z_2, \ldots, z_3 \\
A &= x_c, x_c, x_c, x_c \\
B &= y_c, y_c, y_c, y_c \\
C &= z_c, z_c, z_c, z_c \\
\end{align*}
\]

movaps xmm0, X ; xmm0 = x1, x2, x3, x4
movaps xmm1, Y ; xmm1 = y1, y2, y3, y4
movaps xmm2, Z ; xmm2 = z1, z2, z3, z4
mulps xmm0, A ; xmm0 = x1*x_c, x2*x_c, x3*x_c, x4*x_c
mulps xmm1, B ; xmm1 = y1*y_c, y2*y_c, y3*y_c, y4*y_c
mulps xmm2, C ; xmm2 = z1*z_c, z2*z_c, z3*z_c, z4*z_c
addps xmm0, xmm1
addps xmm0, xmm2 ; xmm0 = (x0*x_c+y0*y_c+z0*z_c) \ldots

---

**Other SIMD architectures**

- **Graphics Processing Unit (GPU):** nVidia 7800, 24 pipelines (8 vector/16 fragment)

---

**NVidia GeForce 8800, 2006**

- Each GeForce 8800 GPU stream processor is a fully generalized, fully decoupled, scalar, processor that supports IEEE 754 floating point precision.
- Up to 128 stream processors

---

**Cell processor**

- **Cell Processor (IBM/Toshiba/Sony):** 1 PPE (Power Processing Unit) +8 SPEs (Synergistic Processing Unit)
- An SPE is a RISC processor with 128-bit SIMD for single/double precision instructions, 128 128-bit registers, 256K local cache
- used in PS3.
Cell processor

Different programming paradigms

Computing $y = ax + y$ with a serial loop:

```c
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing $y = ax + y$ in parallel using CUDA:

```c
__global__ void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n ) y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

GPUs keep track to Moore’s law better

Table 1. Tale of the tape: Throughput architectures.

<table>
<thead>
<tr>
<th>Type</th>
<th>Processor</th>
<th>Cores/Core</th>
<th>ALUs/Core</th>
<th>SIMD width</th>
<th>Max T</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUs</td>
<td>AMD Radeon HD 4870</td>
<td>10</td>
<td>80</td>
<td>64</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>NVIDIA GeForce GTX 280</td>
<td>30</td>
<td>8</td>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>CPUs</td>
<td>Intel Core 2 Quad</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>GTI Cell IPE</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Sun UltraSparc T2</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

1. SSE processing only, does not account for traditional FPU
2. Streaming Processing (SPE) cores only, does not account for PPU cores.
3. 32-bit floating point operations
4. Max T is defined as the maximum ratio of hardware-managed thread execution contexts to simultaneously executable threads (not an absolute count of hardware-managed execution contexts). This ratio is a measure of a processor’s ability to automatically hide thread stalls using hardware multithreading.

References

- Intel MMX for Multimedia PCs, CACM, Jan. 1997
- Chapter 11 *The MMX Instruction Set*, The Art of Assembly
- Chap. 9, 10, 11 of IA-32 Intel Architecture Software Developer’s Manual: Volume 1: Basic Architecture
- [http://www.csie.ntu.edu.tw/~r89004/hive/sse/page_1.html](http://www.csie.ntu.edu.tw/~r89004/hive/sse/page_1.html)