APPENDIX I

Intrinsics

Objective

Appendix I introduces **intrinsics**, which are functions that are usually not implemented as part of a package or library for a programming language, but instead are implemented and handled by a compiler. Intrinsics work much like inline functions: intrinsics are replaced with a sequence of instructions that represent the desired action at every instance. Intrinsics are highly-optimized since they are built in to the compiler. The optimizations are usually regarding parallelization, such as SIMD operations.

The C++ compilers we have referenced throughout the book (Clang, GCC, Visual C++) implement intrinsics for x86/x86_64 SIMD instructions. Also as mentioned in **Chapter 9**, Microsoft x64 does not allow inline Assembly statements, which means to achieve similar low-level operations, programmers *must* use intrinsics. Some compilers use platform-specific intrinsics, while some compiler frameworks provide more portability across platforms. Always keep in mind that intrinsic support will vary by compiler and version.

Resources

- https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (an amazing interactive site by Intel)
- https://software.intel.com/en-us/node/523351 (Intel Intrinsics Reference)
- https://msdn.microsoft.com/en-us/library/26td21ds.aspx (Microsoft Compiler Intrinsics)
- Intel 64 and IA-32 Software Developer's Manual, Volume 2A, Chapter 3, Section 3.1.1.10 "Intel C/C++ Compiler Intrinsics Equivalents Section"
- Intel 64 and IA-32 Software Developer's Manual, Volume 2C, Appendix C "Intel C/C++ Compiler Intrinsics and Functions Equivalents"

Intrinsic Datatypes

Just as with SSE data in raw Assembly, data for SSE intrinsics must be aligned on 16-byte boundaries. Table I.1 presents MMX and SSE intrinsic datatypes.

Table I.1 MMX and SSE intrinsic datatypes

Datatype	Description
__m64	contents of an MMX register (8-bit, 16-bit, 32-bit, and 64-bit values)
__m128	contents of an XMM register used by an SSE intrinsic (one scalar or four 32-bit packed single-precision floating-point values)
__m128d	contents of an XMM register used by an SSE intrinsic (one scalar or two 64-bit packed double-precision floating-point values)
__m128i	contents of an XMM register used by an SSE intrinsic (sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit integers)

Table I.2 presents AVX and AVX2 datatypes, which must be aligned on 32-byte boundaries. Most AVX intrinsics can also use SSE datatypes since AVX and AVX-512 are backward compatible.

Table I.2 AVX Intrinsic datatypes

Datatype	Description
__m256	contents of an YMM register used by an AVX intrinsic (eight 32-bit packed single-precision floating-point values)
__m256d	contents of an YMM register used by an AVX intrinsic (four 64-bit packed double-precision floating-point values)
__m256i	contents of an YMM register used by an AVX intrinsic (thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit integers)

Table I.3 presents AVX-512 datatypes, which must be aligned on 64-byte boundaries.

Table I.3 AVX-512 Datatypes

Datatype	Description
__m512	contents of an ZMM register used by an AVX-512 intrinsic (sixteen 32-bit packed single-precision floating-point values)
__m512d	contents of an ZMM register used by an AVX-512 intrinsic (eight 64-bit packed double-precision floating-point values)
__m512i	contents of an ZMM register used by an AVX-512 intrinsic (sixty-four 8-bit, thirty-two 16-bit, sixteen 32-bit, or eight 64-bit integers)

The use of intrinsic datatypes has some restrictions that are covered in the Intel manual. To use intrinsic datatypes, you must include the header file with the datatype definitions. For example, the SSE header file is `<xmmintrin.h>` and the AVX header file is `<immintrin.h>`. With Clang and GCC, you can include `<x86intrin.h>`, which includes all extension headers. The file is `<intrin.h>` for Visual C++.

Table I.4 Intrinsic header files

Extension	Header file
MMX	`<mmintrin.h>`
SSE	`<xmmintrin.h>`
SSE2	`<emmintrin.h>`
SSE3	`<pmmintrin.h>`
SSSE3	`<tmmintrin.h>`
SSE4.1	`<smmintrin.h>`
SSE4.2	`<nmmintrin.h>`
SSE4a, XOP	`<ammintrin.h>`
AES	`<wmmintrin.h>`
AVX, FMA	`<immintrin.h>`
AVX512	`<zmmintrin.h>`

Intrinsics

Most intrinsics are **simple**, meaning only one intrinsic is needed to achieve the desired operation. Some intrinsics are **composite**, which means more than one statement is necessary for implementation.

Intrinsic template

```
__mm_<operation>_<suffix>
```

Referring to the intrinsic template, operations are usually equivalent or similar to some form of Assembly instruction, such as ADD or SUB. The suffix denotes the datatype used in the operation. The suffix begins with `p` (packed), `ep` (extended packed), or `s` (scalar). Additional letters further describe the data. For example, `ps` indicates packed single precision, while `ss` indicates scalar single precision. Table I.5 presents examples of additional suffix letters.

Table I.5 Intrinsic suffix examples

Additional suffix letters	Description
s	Single-precision floating point
d	Double-precision floating point
i8	signed 8-bit integer
u8	unsigned 8-bit integer
i16	signed 16-bit integer
u16	unsigned 16-bit integer
i32	signed 32-bit integer
u32	unsigned 32-bit integer
i64	signed 64-bit integer
u64	unsigned 64-bit integer
i128	signed 128-bit integer
i256	signed 256-bit integer
i512	signed 512-bit integer

Based on the intrinsic template, an intrinsic function in code has the following form.

Intrinsic function

```
datatype intrinsic_name(parameters);
```

The `datatype` is the return datatype, which can be `void`, `int`, or one of the supported intrinsic datatypes (e.g., `__m64`, `__m128i`). The `intrinsic_name` is the function-like operation you wish to perform, which is based on the intrinsic template. The `parameters` are the data on which the intrinsic operates.

Table I.6 presents three intrinsic examples. First is an SSE intrinsic to add packed single-precision values. Second is an AVX intrinsic to move (load) a YMM register with aligned packed double-precision values. Third is an AVX-512 intrinsic to calculate the sine value of packed single-precision floats. Notice that the AVX-512 intrinsic has no Assembly instruction counterpart.

Table I.6 Intrinsic examples

Assembly Mnemonic (ext)	Description	Intrinsic
ADDPS (SSE)	Add packed single precision	`__m128 _mm_add_ps(__m128 a, __m128 b)`
VMOVAPD (AVX)	Move aligned packed double precision	`__m256d _mm256_load_pd(double const *a)`
N/A (AVX-512)	Calculate sine value of packed 32-bit floats	`__m512 _mm512_sin_ps(__m512 v)`

Programs

Here we provide programs to illustrate the behavior and use of intrinsics. We encourage you to work through the examples presented and then explore other intrinsics. The programs include snippets of Disassembly, so this section assumes you have reviewed **Appendix C: Disassembly**. With your preferred development environment and compiler, write Program I.1 and set a breakpoint at `return 0;` so you can examine the local variables and register state.

Program I.1 *Intrinsic load packed single precision*

C++

```
#include <xmmintrin.h>
int main()
{
    float array[4] = { 1.0, 2.0, 3.0, 4.0 };
    __m128 result = _mm_load_ps(array);      // MOVAPS

   return 0;  // set breakpoint here
}
```

Program I.1 simply creates an `array` of 32-bit floats, creates a `result` that is of type `__m128`, and uses the `_mm_load_ps` intrinsic to load the array comprised of packed single-precision floating-point values into an SSE register. Note that the intrinsic parameter is the address of `array`.

Example I.1 shows the Disassembly of Program I.1, of which most is the process of setting up the stack frame for `main` and storing local variables. The first arrow (->) in the Disassembly indicates where the data pointed to by the address in *rcx* (the address of `array`) is moved to *xmm0* using the MOVAPS instruction. The second arrow is where the values are copied to `result`.

Keep in mind that the compiler will choose the register(s) to use for operations. For Program I.1, *xmm0* is likely to be chosen.

Example I.1 Disassembly of Program I.1

Xcode (Clang/LLVM) – AT&T syntax

```
cpp_testing`main:
    0x100000f10 <+0>:  pushq  %rbp
    0x100000f11 <+1>:  movq   %rsp, %rbp
    0x100000f14 <+4>:  subq   $0x40, %rsp
    0x100000f18 <+8>:  movq   0xe1(%rip), %rax
    0x100000f1f <+15>: leaq   -0x20(%rbp), %rcx
    0x100000f23 <+19>: movq   (%rax), %rdx
    0x100000f26 <+22>: movq   %rdx, -0x8(%rbp)
    0x100000f2a <+26>: movl   $0x0, -0x2c(%rbp)
    0x100000f31 <+33>: movq   0x68(%rip), %rdx
    0x100000f38 <+40>: movq   %rdx, -0x20(%rbp)
    0x100000f3c <+44>: movq   0x65(%rip), %rdx
    0x100000f43 <+51>: movq   %rdx, -0x18(%rbp)
    0x100000f47 <+55>: movq   %rcx, -0x28(%rbp)
    0x100000f4b <+59>: movq   -0x28(%rbp), %rcx
->  0x100000f4f <+63>: movaps (%rcx), %xmm0
->  0x100000f52 <+66>: movaps %xmm0, -0x40(%rbp)
    0x100000f56 <+70>: movq   (%rax), %rax
    0x100000f59 <+73>: cmpq   -0x8(%rbp), %rax
    0x100000f5d <+77>: jne    0x100000f6b
    0x100000f63 <+83>: xorl   %eax, %eax
    0x100000f65 <+85>: addq   $0x40, %rsp          ; breakpoint
    0x100000f69 <+89>: popq   %rbp
    0x100000f6a <+90>: retq
    0x100000f6b <+91>: callq  0x100000f70
```

If you set the breakpoint and examine *xmm0*, you will find the float values `1.0, 2.0, 3.0, 4.0`. Packed floats are stored in reverse order with the lowest element used in scalar operations. Consider the following additional line in Program I.2 (i.e., `float element = result[0];`).

Program I.2 *Accessing the first element in* `result`

```
#include <xmmintrin.h>
int main()
{
   float array[4] = { 1.0, 2.0, 3.0, 4.0 };
   __m128 result = _mm_load_ps(array);

   float element = result[0];

   return 0;
}
```

The float value `element` will contain `1.0` after the assign statement. So `1.0` is in `result[0]`, `2.0` is in `result[1]`, and so on. A different intrinsic could be used to set the floats in the same way they visually appear in C++ (think Little-Endian versus Big-Endian).

Example I.2 Set packed single precision

```
__m128 result = _mm_set_ps(1.0, 2.0, 3.0, 4.0);
float element = result[0];
```

If executing the code in Example I.2, `element` would hold `4.0` because we use the `_mm_set_ps` intrinsic instead of the `_mm_load_ps` intrinsic.

By having SSE registers loaded with values and intrinsic datatype variables, we can achieve low-level operations quite easily. Program I.3 shows another modification to Program I.1, an easy way to square the contents of `result`.

Program I.3 *Square elements in* `result`

```
#include <xmmintrin.h>
int main()
{
   float array[4] = { 1.0, 2.0, 3.0, 4.0 };
   __m128 result = _mm_load_ps(array);

   result = result * result;

   return 0;
}
```

The Disassembly of the `result` multiplication in Program I.3 will look something like Example I.3, which shows the Assembly instructions the compiler uses to achieve the operation.

Example I.3 Partial disassembly of Program I.3

```
0x100000f3b <+59>: movaps %xmm0, -0x40(%rbp)
0x100000f3f <+63>: mulps  -0x40(%rbp), %xmm0
0x100000f43 <+67>: movaps %xmm0, -0x40(%rbp)
```

In Example I.3, we see the values in *xmm0* are copied to memory, the elements in memory are multiplied by their counterparts in *xmm0* using the MULPS instruction with the result stored in *xmm0*, which is then copied back to memory (`result`). We could have used a multiplication intrinsic if desired, as shown in Example I.4.

Example I.4 Multiply packed single precision

```
result = _mm_mul_ps(result, result);
```

Program I.4 shows another way in which intrinsics can be used. Intrinsic datatypes can be used in aggregates such as **unions** so the elements of a vector can be manipulated easily. For example, the address of a vector and addresses of elements can be taken, and brackets `[]` can be used to access elements.

Program I.4 also illustrates the use of AVX2 intrinsics. AVX introduced the 256-bit YMM registers with floating-point instructions; and AVX2 introduced integer operations with YMM registers. If using Clang or GCC, the `-mavx2` complier flag is necessary to enable the AVX2 instruction set for a target. Specific flags can be used for each extension (e.g., `-msse`, `-msse4`, `-msse4.2`). Another approach is to enable all available extensions supported by the processor using `-march=native`. **Appendix G: Using CPUID** explains how CPUID can be used to test for extensions supported by a processor.

Program I.4 *Add thirty-two packed 8-bit integers*

C++

```
#include <immintrin.h>
// also set -mavx2 compiler flag

union U32i
{
    __m256i v;
    int8_t a[32];
};

int main()
{
    U32i testing;

    for(int i = 0; i < 32; i++){
        testing.a[i] = i;
    }

    __m256i v2 = _mm256_add_epi8(testing.v, testing.v);  // VPADDB

    _mm256_load_si256(&v2);  // VMOVDQA

    return 0;  // set breakpoint here
}
```

The `U32i` union contains an `__m256i` 256-bit integer vector (`v`), and sharing the same memory space is an array of thirty-two `int8_t` 8-bit integers (`a`). Using a *union* allows us to access all elements of the vector via the array. We use a `for` loop to fill the vector with integers in ascending order from 0 to 31. Then, the instance of the union in Program I.4, `testing`, serves as a parameter for two intrinsics.

First, we create another vector (`v2`) that stores the result of the intrinsic addition. We just add the vector to itself, effectively doubling all the values simultaneously. Note that the `_epi8` suffix indicates extended packed 8-bit integers (extended meaning AVX and YMM as opposed to SSE and XMM). Second, we load the signed integers in the 256-bit vector into a YMM register. The Disassembly of the intrinsics is shown in Example I.5.

Example I.5 Partial disassembly of Program I.4

```
0x100000f6f <+79>:  vmovaps 0x40(%rsp), %ymm0
0x100000f75 <+85>:  vmovaps %ymm0, 0xa0(%rsp)
0x100000f7e <+94>:  vmovaps %ymm0, 0x80(%rsp)
0x100000f87 <+103>: vmovaps 0xa0(%rsp), %ymm1
0x100000f90 <+112>: vpaddb %ymm0, %ymm1, %ymm0
0x100000f94 <+116>: vmovaps %ymm0, (%rsp)
```

Breaking down the Disassembly: (1) the vector values are moved to *ymm0*, (2) the vector values are copied to two memory locations (a copy of each parameter) by using the VMOVAPS instruction, (3) the vector is copied to *ymm1*, (4) the VPADDB instruction adds *ymm0* and *ymm1*, saving the result to *ymm0*, and (5) the result is saved back to memory.

Notice that even though intrinsics are in the form of functions, with return values and parameters, no actual *call* takes place, similar to inline functions. Yet, some behaviors are similar, which is why the vector is copied to memory locations in pass-by-value fashion; a local copy is made for each parameter. Also, even if an intrinsic is used that has an SSE or AVX equivalent, such as our second intrinsic in Program I.4, the compiler may not use the equivalent. A more optimal instruction or combination of instructions may be used to accomplish the task. In Program I.4, the *ymm0* register already holds the vector values after the addition, so the load intrinsic is optimized away.