Using SSE registers to accelerate cryptography

Most CPUs these days have registers capable of processing multiple pieces of data in parallel.

- SSE (Streaming SIMD Extensions) introduced by Intel in 1999 is 16 bytes wide.
- AVX (Advances Vector Extensions) introduced by Intel in 2011 is 32 bytes wide.
- AVX-512 introduced by Intel in 2013 is 64 bytes wide.
- NEON introduced by ARM in 2004 is 16 bytes wide.

This note will focus on SSE as an introductory vehicle.

The easiest way to use these registers is by using compiler *intrinsics*. An intrinsic is a function call in the compiler's library that compiles to a single assembly instruction. These intrinsics, along with a datatype `__m128i` that represents an SSE register, allow the manipulation of 16 bytes of data per instruction.

Making the intrinsics available and documentation

`#include <immintrin.h>`

Your compiler may need to be told to compile for these instructions, so if you get errors related to undefined intrinsics, try adding `-march=native` to your GCC or Clang command line. This tells the compiler to allow all intrinsics available on your computer.

You can find more intrinsics for manipulating vectors by going to Intel's online Intrinsics Guide. Check the "SSE2" checkbox to filter out most of the intrinsics we won't use.

To load and store a register:

```
__m128i _mm_loadu_si128 (__m128i const* mem_addr)
void _mm_storeu_si128 (__m128i* mem_addr, __m128i a)
```

These instructions load and store 16 bytes little-endian, so if memory has `{00,01,02,03,04,05,06,07,08,09,0A,0B,0C,0D,0E,0F}` then the register has `[0F,0E,0D,0C,0B,0A,09,08,07,06,05,04,03,02,01,00]` and vice-versa.

When you treat an SSE register as if it has four uint32_t, the above register of bytes is interpreted as the following: `[0F0E0D0C,0B0A0908,07060504,03020100]`. So, the elements are each loaded little-endian, but they also appear in reverse order in register as they do in memory. Here's another example:

```
uint32_t buf[4] = {1,2,3,4};
// Compiler puts {01,00,00,00,02,00,00,00,03,00,00,00,04,00,00,00} in memory for you
__m128i x = _mm_loadu_si128 ((__m128i *)buf);
// After load register x has [00,00,00,04,00,00,00,03,00,00,00,02,00,00,00,01]
```

Operations on a vector of values:

Here are a few operations that act on a vector register as if it held four 32-bit integers.

```
__m128i _mm_add_epi32 (__m128i a, __m128i b)
__m128i _mm_slli_epi32 (__m128i a, int imm8)  // Shift each element left
__m128i _mm_srli_epi32 (__m128i a, int imm8)  // Shift each element right
etc.
```

and a couple that operate on the whole register at once (because they are bitwise operations)

```
__m128i _mm_or_si128 (__m128i a, __m128i b)
__m128i _mm_xor_si128 (__m128i a, __m128i b)
etc.
```

Setting vector values:

You can set a vector register's values by putting the values in memory and loading them, or you can use any of the following.

```
__m128i _mm_set_epi32 (int e3, int e2, int e1, int e0)  \\ Set register to [e3,e2,e1,e0]
__m128i _mm_set1_epi32 (int a)                          \\ Set register to [a,a,a,a]
__m128i _mm_setzero_si128 ()
```

Shuffling vector elements:

Some algorithms need to move the vector elements around in the vector. Here's a version that allows you to rearrange a vector of four ints.

```
__m128i _mm_shuffle_epi32 (__m128i a, int imm8)
```

The `imm8` parameter controls where each int from `a` goes. Think of `imm8` as four two-bit numbers concatenated together. The first two-bits specifies which of the four source ints is the leftmost in the destination register. The next two bits specifies the next int in the destination, etc. The indices run from 00 is the right most int to 11 indicating the leftmost int.

So, if `[0F0E0D0C,0B0A0908,07060504,03020100]` was your source register then your destination register would be

`imm8`	dest reg
0b00011011	`[03020100,07060504,0B0A0908,0F0E0D0C]`
0b00000000	`[03020100,03020100,03020100,03020100]`

Sample:

Here's some sample code that verifies the above loads and shuffles.

```
#include <stdio.h>
#include <immintrin.h>

void pbuf16(void *p) {          // Print 16 bytes from memory starting at address p
    for (int i=0; i<16; i++) {
        printf("%02X", ((unsigned char *)p)[i]);
    }
    printf("\n");
}

void psse_int(__m128i x) {      // Print an SSE register as if it holds four ints
    int *p = (int *)&x;
    for (int i=0; i<4; i++) {
        printf("%08X ", p[3-i]);
    }
    printf("\n");
}

int main(int argc, char *argv[]) {
    unsigned char mem[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
    pbuf16(mem);
    __m128i x = _mm_loadu_si128 ((__m128i *)mem);
    psse_int(x);
    __m128i y = _mm_shuffle_epi32(x,0b00011011);
    psse_int(y);
    __m128i z = _mm_shuffle_epi32(x,0b00000000);
    psse_int(z);
    return 0;
}
```

AVX:

On newer Intel computers with AVX2 or AVX-512 registers, all of the above can be extended to registers holding vectors of even more ints. At the intrinsics guide, turn on AVX2 and search for loadu, add, xor, etc, and read about it if you're interested.