

In an attempt to squeeze every last bit of precision from a floating-point operation, the standard allows some numbers to be represented in unnormalized form. Rather than having a gap between 0 and the smallest normalized number, IEEE allows *denormalized numbers* (also known as *denorms* or *subnormals*). They have the same exponent as zero but a nonzero fraction. They allow a number to degrade in significance until it becomes 0, called *gradual underflow*. For example, the smallest positive single precision normalized number is

$$1.0000\ 0000\ 0000\ 0000\ 0000\ 000\text{two} \times 2^{-126}$$

but the smallest single precision denormalized number is

$$0.0000\ 0000\ 0000\ 0000\ 0000\ 001\text{two} \times 2^{-126}, \text{ or } 1.0\text{two} \times 2^{-149}$$

For double precision, the denorm gap goes from  $1.0 \times 2^{-1022}$  to  $1.0 \times 2^{-1074}$ .

The possibility of an occasional unnormalized operand has given headaches to floating-point designers who are trying to build fast floating-point units. Hence, many computers cause an exception if an operand is denormalized, letting software complete the operation. Although software implementations are perfectly valid, their lower performance has lessened the popularity of denorms in portable floating-point software. Moreover, if programmers do not expect denorms, their programs may surprise them.

## 3.6

### Parallelism and Computer Arithmetic: Subword Parallelism

Since every desktop microprocessor by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations.

Many graphics systems originally used 8 bits to represent each of the three primary colors plus 8 bits for a location of a pixel. The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well. Audio samples need more than 8 bits of precision, but 16 bits are sufficient.

Every microprocessor has special support so that bytes and halfwords take up less space when stored in memory (see Section 2.9), but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there was little support beyond data transfers. Architects recognized that many graphics and audio applications would perform the same operation on vectors of this data. By partitioning the carry chains within a 128-bit adder, a processor could use **parallelism** to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands. The cost of such partitioned adders was small.

Given that the parallelism occurs within a wide word, the extensions are classified as *subword parallelism*. It is also classified under the more general name of *data level parallelism*. They have been also called vector or SIMD, for single instruction, multiple data (see Section 6.6). The rising popularity of multimedia



applications led to arithmetic instructions that support narrower operations that can easily operate in parallel.

For example, ARM added more than 100 instructions in the NEON multimedia instruction extension to support subword parallelism, which can be used either with ARMv7 or ARMv8. It added 256 bytes of new registers for NEON that can be viewed as 32 registers 8 bytes wide or 16 registers 16 bytes wide. NEON supports all the subword data types you can imagine *except* 64-bit floating point numbers:

- 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers
- 32-bit floating point numbers

Figure 3.19 gives a summary of the basic NEON instructions.

Data transfer	Arithmetic	Logical/Compare
VLDR.F32	VADD.F32, VADD{L,W}{S8,U8,S16,U16,S32,U32}	VAND.64, VAND.128
VSTR.F32	VSUB.F32, VSUB{L,W}{S8,U8,S16,U16,S32,U32}	VORR.64, VORR.128
VLD{1,2,3,4}.{I8,I16,I32}	VMUL.F32, VMULL{S8,U8,S16,U16,S32,U32}	VEOR.64, VEO.128
VST{1,2,3,4}.{I8,I16,I32}	VMLA.F32, VMLAL{S8,U8,S16,U16,S32,U32}	VBIC.64, VBIC.128
VMOV.{I8,I16,I32,F32}, #imm	VMLS.F32, VMLSL{S8,U8,S16,U16,S32,U32}	VORN.64, VORN.128
VMVN.{I8,I16,I32,F32}, #imm	VMAX.{S8,U8,S16,U16,S32,U32,F32}	VCEQ.{I8,I16,I32,F32}
VMOV.{I64,I128}	VMIN.{S8,U8,S16,U16,S32,U32,F32}	VCGE.{S8,U8,S16,U16,S32,U32,F32}
VMVN.{I64,I128}	VABS.{S8,S16,S32,F32}	VCGT.{S8,U8,S16,U16,S32,U32,F32}
	VNEG.{S8,S16,S32,F32}	VCLE.{S8,U8,S16,U16,S32,U32,F32}
	VSHL.{S8,U8,S16,U16,S32,S64,U64}	VCLT.{S8,U8,S16,U16,S32,U32,F32}
	VSHR.{S8,U8,S16,U16,S32,S64,U64}	VTST.{I8,I16,I32}

**FIGURE 3.19 Summary of ARM NEON instructions for subword parallelism.** We use the curly brackets {} to show optional variations of the basic operations: {S8,U8,8} stand for signed and unsigned 8-bit integers or 8-bit data where type doesn't matter, of which 16 fit in a 128-bit register; {S16,U16,16} stand for signed and unsigned 16-bit integers or 16-bit type-less data, of which 8 fit in a 128-bit register; {S32,U32,32} stand for signed and unsigned 32-bit integers or 32-bit type-less data, of which 4 fit in a 128-bit register; {S64,U64,64} stand for signed and unsigned 64-bit integers or type-less 64-bit data, of which 2 fit in a 128-bit register; {F32} stand for signed and unsigned 32-bit floating point numbers, of which 4 fit in a 128-bit register. Vector Load reads one n-element structure from memory into 1, 2, 3, or 4 NEON registers. It loads a single n-element structure to one lane (See Section 6.6), and elements of the register that are not loaded are unchanged. Vector Store writes one n-element structure into memory from 1, 2, 3, or 4 NEON registers.

**Elaboration:** In addition to signed and unsigned integers, ARM includes “fixed-point” format of four sizes called I8, I16, I32, and I64, of which 16, 8, 4, and 2 fit in a 128-bit register, respectively. A portion of the fixed point is for the fraction (to the right of the binary point) and the rest of the data is the integer portion (to the left of the binary point). The location of the binary point is up to the software. Many ARM processors do not have floating point hardware and thus floating point operations must be performed by library routines. Fixed point arithmetic can be significantly faster than software floating point routines, but more work for the programmer.

## 3.7

## Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86

The original MMX (*MultiMedia eXtension*) and SSE (*Streaming SIMD Extension*) instructions for the x86 included similar operations to those found in ARM NEON. Chapter 2 notes that in 2001 Intel added 144 instructions to its architecture as part of SSE2, including double precision floating-point registers and operations. It includes eight 64-bit registers that can be used for floating-point operands. AMD expanded the number to 16 registers, called XMM, as part of AMD64, which Intel relabeled EM64T for its use. Figure 3.20 summarizes the SSE and SSE2 instructions.

In addition to holding a single precision or double precision number in a register, Intel allows multiple floating-point operands to be packed into a single 128-bit SSE2 register: four single precision or two double precision. Thus, the 16 floating-point registers for SSE2 are actually 128 bits wide. If the operands can be arranged in memory as 128-bit aligned data, then 128-bit data transfers can load and store multiple operands per instruction. This packed floating-point format is supported by arithmetic operations that can operate simultaneously on four singles (PS) or two doubles (PD).

Data transfer	Arithmetic	Compare
MOV {A/U} {SS/PS/SD/PD} xmm, mem/xmm	ADD {SS/PS/SD/PD} xmm, mem/xmm	CMP {SS/PS/SD/PD}
	SUB {SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL {SS/PS/SD/PD} xmm, mem/xmm	
	DIV {SS/PS/SD/PD} xmm, mem/xmm	
	SQRT {SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN {SS/PS/SD/PD} mem/xmm	

**FIGURE 3.20 The SSE/SSE2 floating-point instructions of the x86.** xmm means one operand is a 128-bit SSE2 register, and mem/xmm means the other operand is either in memory or it is an SSE2 register. We use the curly brackets {} to show optional variations of the basic operations: {SS} stands for *Scalar Single* precision floating point, or one 32-bit operand in a 128-bit register; {PS} stands for *Packed Single* precision floating point, or four 32-bit operands in a 128-bit register; {SD} stands for *Scalar Double* precision floating point, or one 64-bit operand in a 128-bit register; {PD} stands for *Packed Double* precision floating point, or two 64-bit operands in a 128-bit register; {A} means the 128-bit operand is aligned in memory; {U} means the 128-bit operand is unaligned in memory; {H} means move the high half of the 128-bit operand; and {L} means move the low half of the 128-bit operand.

In 2011 Intel doubled the width of the registers again, now called YMM, with *Advanced Vector Extensions (AVX)*. Thus, a single operation can now specify eight 32-bit floating-point operations or four 64-bit floating-point operations. The legacy SSE and SSE2 instructions now operate on the lower 128 bits of the YMM registers. Thus, to go from 128-bit and 256-bit operations, you prepend the letter “v” (for vector) in front of the SSE2 assembly language operations and then use the YMM register names instead of the XMM register name. For example, the SSE2 instruction to perform two 64-bit floating-point multiplies

```
addpd %xmm0, %xmm4
```

It becomes

```
vaddpd %ymm0, %ymm4
```

which now produces four 64-bit floating-point multiplies.

**Elaboration:** AVX also added three address instructions to x86. For example, `vaddpd` can now specify

```
vaddpd %ymm0, %ymm1, %ymm4 # %ymm4 = %ymm1 + %ymm2
```

instead of the standard two address version

```
addpd %xmm0, %xmm4 # %xmm4 = %xmm4 + %xmm0
```

(Unlike MIPS, the destination is on the right in x86.) Three addresses can reduce the number of registers and instructions needed for a computation.

## 3.8

### Going Faster: Subword Parallelism and Matrix Multiply

To demonstrate the performance impact of subword parallelism, we’ll run the same code on the Intel Core i7 first without AVX and then with it. [Figure 3.21](#) shows an unoptimized version of a matrix-matrix multiply written in C. As we saw in Section 3.5, this program is commonly called *DGEMM*, which stands for Double precision GEneral Matrix Multiply. Starting with this edition, we have added a new section entitled “Going Faster” to demonstrate the performance benefit of adapting software to the underlying hardware, in this case the Sandy Bridge version of the Intel Core i7 microprocessor. This new section in Chapters 3, 4, 5, and 6 will incrementally improve DGEMM performance using the ideas that each chapter introduces.

[Figure 3.22](#) shows the x86 assembly language output for the inner loop of [Figure 3.21](#). The five floating point-instructions start with a `v` like the AVX instructions, but note that they use the XMM registers instead of YMM, and they include `sd` in the name, which stands for scalar double precision. We’ll define the subword parallel instructions shortly.

```

1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.     for (int i = 0; i < n; ++i)
4.         for (int j = 0; j < n; ++j)
5.             {
6.                 double cij = C[i+j*n]; /* cij = C[i][j] */
7.                 for( int k = 0; k < n; k++ )
8.                     cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.                 C[i+j*n] = cij; /* C[i][j] = cij */
10.            }
11. }

```

**FIGURE 3.21 Unoptimized C version of a double precision matrix multiply, widely known as DGEMM for Double-precision GEneral Matrix Multiply (GEMM).** Because we are passing the matrix dimension as the parameter *n*, this version of DGEMM uses single dimensional versions of matrices *C*, *A*, and *B* and address arithmetic to get better performance instead of using the more intuitive two-dimensional arrays that we saw in Section 3.5. The comments remind us of this more intuitive notation.

```

1. vmovsd (%r10),%xmm0           # Load 1 element of C into %xmm0
2. mov     %rsi,%rcx             # register %rcx = %rsi
3. xor     %eax,%eax             # register %eax = 0
4. vmovsd (%rcx),%xmm1           # Load 1 element of B into %xmm1
5. add     %r9,%rcx              # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
7. add     $0x1,%rax             # register %rax = %rax + 1
8. cmp     %eax,%edi             # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0      # Add %xmm1, %xmm0
10. jg      30 <dgemm+0x30>       # jump if %eax > %edi
11. add     $0x1,%r11d            # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)          # Store %xmm0 into C element

```

**FIGURE 3.22 The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in Figure 3.21.** Although it is dealing with just 64-bits of data, the compiler uses the AVX version of the instructions instead of SSE2 presumably so that it can use three address per instruction instead of two (see the Elaboration in Section 3.7).

```

1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.     for ( int i = 0; i < n; i+=4 )
5.         for ( int j = 0; j < n; j++ ) {
6.             __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.             for( int k = 0; k < n; k++ )
8.                 c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                                     _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                                                    _mm256_broadcast_sd(B+k+j*n)));
11.             _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.         }
13. }

```

**FIGURE 3.23 Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86.** Figure 3.24 shows the assembly language produced by the compiler for the inner loop.

While compiler writers may eventually be able to routinely produce high-quality code that uses the AVX instructions of the x86, for now we must “cheat” by using C intrinsics that more or less tell the compiler exactly how to produce good code. Figure 3.23 shows the enhanced version of Figure 3.21 for which the Gnu C compiler produces AVX code. Figure 3.24 shows annotated x86 code that is the output of compiling using gcc with the -O3 level of optimization.

The declaration on line 6 of Figure 3.23 uses the `__m256d` data type, which tells the compiler the variable will hold 4 double-precision floating-point values. The intrinsic `_mm256_load_pd()` also on line 6 uses AVX instructions to load 4 double-precision floating-point numbers in parallel (`_pd`) from the matrix `C` into `c0`. The address calculation `C+i+j*n` on line 6 represents element `C[i+j*n]`. Symmetrically, the final step on line 11 uses the intrinsic `_mm256_store_pd()` to store 4 double-precision floating-point numbers from `c0` into the matrix `C`. As we’re going through 4 elements each iteration, the outer *for* loop on line 4 increments `i` by 4 instead of by 1 as on line 3 of Figure 3.21.

Inside the loops, on line 9 we first load 4 elements of `A` again using `_mm256_load_pd()`. To multiply these elements by one element of `B`, on line 10 we first use the intrinsic `_mm256_broadcast_sd()`, which makes 4 identical copies of the scalar double precision number—in this case an element of `B`—in one of the YMM registers. We then use `_mm256_mul_pd()` on line 9 to multiply the four double-precision results in parallel. Finally, `_mm256_add_pd()` on line 8 adds the 4 products to the 4 sums in `c0`.

Figure 3.24 shows resulting x86 code for the body of the inner loops produced by the compiler. You can see the five AVX instructions—they all start with `v` and

```

1. vmovapd (%r11),%ymm0          # Load 4 elements of C into %ymm0
2. mov     %rbx,%rcx             # register %rcx = %rbx
3. xor     %eax,%eax             # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add     $0x8,%rax             # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1     # Parallel mul %ymm1, 4 A elements
7. add     %r9,%rcx              # register %rcx = %rcx + %r9
8. cmp     %r10,%rax             # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0      # Parallel add %ymm1, %ymm0
10. jne    50 <dgemm+0x50>       # jump if not %r10 != %rax
11. add     $0x1,%esi            # register %esi = %esi + 1
12. vmovapd %ymm0,(%r11)         # Store %ymm0 into 4 C elements

```

**FIGURE 3.24** The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in [Figure 3.23](#). Note the similarities to [Figure 3.22](#), with the primary difference being that the five floating-point operations are now using YMM registers and using the pd versions of the instructions for parallel double precision instead of the sd version for scalar double precision.

four of the five use pd for parallel double precision—that correspond to the C intrinsics mentioned above. The code is very similar to that in [Figure 3.22](#) above: both use 12 instructions, the integer instructions are nearly identical (but different registers), and the floating-point instruction differences are generally just going from *scalar double* (sd) using XMM registers to *parallel double* (pd) with YMM registers. The one exception is line 4 of [Figure 3.24](#). Every element of A must be multiplied by one element of B. One solution is to place four identical copies of the 64-bit B element side-by-side into the 256-bit YMM register, which is just what the instruction `vbroadcastsd` does.

For matrices of dimensions of 32 by 32, the unoptimized DGEMM in [Figure 3.21](#) runs at 1.7 GigaFLOPS (FLOating point Operations Per Second) on one core of a 2.6 GHz Intel Core i7 (Sandy Bridge). The optimized code in [Figure 3.23](#) performs at 6.4 GigaFLOPS. The AVX version is 3.85 times as fast, which is very close to the factor of 4.0 increase that you might hope for from performing 4 times as many operations at a time by using **subword parallelism**.



**Elaboration:** As mentioned in the Elaboration in Section 1.6, Intel offers Turbo mode that temporarily runs at a higher clock rate until the chip gets too hot. This Intel Core i7 (Sandy Bridge) can increase from 2.6 GHz to 3.3 GHz in Turbo mode. The results above are with Turbo mode turned off. If we turn it on, we improve all the results by the increase in the clock rate of  $3.3/2.6 = 1.27$  to 2.1 GFLOPS for unoptimized DGEMM and 8.1 GFLOPS with AVX. Turbo mode works particularly well when using only a single core of an eight-core chip, as in this case, as it lets that single core use much more than its fair share of power since the other cores are idle.