CS:APP2e Web Aside ASM:SSE: SSE-Based Support for Floating Point[*]

Randal E. Bryant
David R. O'Hallaron

August 5, 2014

Notice

1 Introduction

The *floating-point architecture* for a processor consists of the different aspects that affect how programs operating on floating-point data are mapped onto the machine, including:

- How floating-point values are stored and accessed. This is typically via some form of registers.
- The instructions that operate on floating-point data.
- The conventions used for passing floating-point values as arguments to functions, and for returning them as results.

In this document, we will describe the floating-point architecture for x86 processors known as *SSE*.

Since the introduction of the Pentium MMX in 1997, both Intel and AMD have incorporated successive generations of *media* instructions to support graphics and image processing. Starting with the Pentium III in 1999, these instructions have been known as *SSE*, for "Streaming SIMD Extensions." In its original form, SSE did not support double-precision floating-point arithmetic, but since the introduction of SSE2 with the

[*]Copyright © 2010, R. E. Bryant, D. R. O'Hallaron. All rights reserved.

Pentium 4 (2000), SSE provides a viable mechanism for implementing both single and double-precision floating-point arithmetic. We will use the term "SSE2+" to denote the floating-point support provided by SSE versions 2 and higher.

All processors capable of executing x86-64 code support SSE2 or higher, and hence x86-64 floating-point is based on SSE, including conventions for passing procedure arguments and return values [3]. For IA32, GCC must be explicitly commanded to generate SSE code using both command-line parameters '`-mfpmath=sse`' and '`-msse2`' (or '`-msse3`' or higher if the machine supports more recent versions of SSE.) Even then, the code remains compatible with IA32 conventions for passing function arguments and return values.

The media instructions originally focused on allowing multiple operations to be performed in a parallel mode known as *single instruction, multiple data* or *SIMD* (pronounced SIM-DEE). In this mode the same operation is performed on a number of different data values in parallel. The media instructions implement SIMD operations by having a set of registers that can hold multiple data values in *packed* format. SSE2+ provides either eight (with IA32) or sixteen (with x86-64) *XMM* registers of 128 bits each, named `%xmm0`, `%xmm1`, and so on, up to either 7 or 15. Each one of these registers can hold a vector of K elements of N bits each, such that $K \times N = 128$. For integers, N can be 8, 16, 32, or 64 bits, while for floating-point numbers, N can be 32 or 64. For example, a single SSE instruction can add two byte vectors of eight elements each, while another can multiply two vectors, each containing four single-precision floating point numbers. The floating-point formats match the IEEE standard formats for single and double-precision values. The major use of these media instructions are in library routines for graphics and image processing. These routines can be written in assembly code, or by using special extensions to C supported by GCC, as is covered in Web Aside OPT:SIMD. There has been considerable effort to enable compilers to extract parallelism from sequential programs, including the *Autovectorization Project* in GCC [4], but so far their capabilities have proved limited.

With SSE2 came the opportunity to completely change the way floating-point code is compiled for x86 processors. As described in Web Aside ASM:X87, floating point was historically implemented in IA32 based on a floating-point architecture dating back to the 8087, a floating-point coprocessor for the Intel 8086. With this architecture, often referred to as "x87," floating-point data are held in a shallow stack of registers, and the floating-point instructions push and pop stack values. This is a difficult target for optimizing compilers. The architecture also has many quirks due to a nonstandard 80-bit floating-point format, as described in Web Aside DATA:IA32-FP.

The SSE2+ instructions include a set of instructions to operate on *scalar* floating-point data, using single values in the low-order 32 or 64 bits of XMM registers, This scalar mode provides a set of registers and instructions that are more typical of the way other processors support floating point. For compilation on x86-64 and for suitably configured IA32 machines, GCC now maps the floating-point data and operations of a source program into SSE code.

This section describes the implementation of floating point based on SSE2+. We mostly use x86-64 code in our examples but also illustrate how code generated for IA32 can make use of SSE. Readers may wish to refer to the Intel documentation for the individual instructions [1, 2]. As with integer operations, note that the ATT format we use in our presentation differs from the Intel format used in these documents.

2 Floating-Point Movement and Conversion Operations

Instruction	Source	Destination	Description
`movss`	M_{32}/X	X	Move single precision
`movss`	X	M_{32}	Move single precision
`movsd`	M_{64}/X	X	Move double precision
`movsd`	X	M_{64}	Move double precision
`cvtss2sd`	M_{32}/X	X	Convert single to double precision
`cvtsd2ss`	M_{64}/X	X	Convert double to single precision
`cvtsi2ss`	M_{32}/R_{32}	X	Convert integer to single precision
`cvtsi2sd`	M_{32}/R_{32}	X	Convert integer to double precision
`cvtsi2ssq`	M_{64}/R_{64}	X	Convert quadword integer to single precision
`cvtsi2sdq`	M_{64}/R_{64}	X	Convert quadword integer to double precision
`cvttss2si`	X/M_{32}	R_{32}	Convert with truncation single precision to integer
`cvttsd2si`	X/M_{64}	R_{32}	Convert with truncation double precision to integer
`cvttss2siq`	X/M_{32}	R_{64}	Convert with truncation single precision to quadword integer
`cvttsd2siq`	X/M_{64}	R_{64}	Convert with truncation double precision to quadword integer

X: XMM register (e.g., `%xmm3`)
R_{32}: 32-bit general-purpose register (e.g., `%eax`)
R_{64}: 64-bit general-purpose register (e.g., `%rax`)
M_{32}: 32-bit memory range
M_{64}: 64-bit memory range

Figure 1: **Scalar floating-point movement and conversion operations.** These operations transfer values between memory and registers, possibly converting between data types.

Figure 1 shows a set of instructions for transferring data and for performing conversions between floating-point and integer data types. These are all *scalar* instructions, meaning that they operate on individual, rather than packed, data values. Floating-point data are held either in memory (indicated in the table as M_{32} and M_{64}) or in XMM registers (shown in the table as X). Integer data are held either in memory (indicated in the table as M_{32} or M_{64}) or in general-purpose registers (shown in the table as R_{32} and R_{64}). These instructions will work correctly regardless of the alignment of data, although the code optimization guidelines recommend that 32-bit memory data satisfy a 4-byte alignment, and that 64-bit data satisfy an 8-byte alignment.

The floating-point movement operations can transfer data from register to register, from memory to register and from register to memory. As is true for the integer case, a single floating-point instruction cannot move data from memory to memory. The floating-point conversion operations have either memory or a register as source and a register as destination, where the registers are general-purpose registers for integer data and XMM registers for floating-point data. The instructions, such as `cvttss2si`, for converting floating-point values to integers use truncation, always rounding values toward zero, as is required by C and most other programming languages.

As an example of the different floating-point move and conversion operations, consider the following C function:

```
double fcvt(int i, float *fp, double *dp, long *lp)
{
    float f = *fp; double d = *dp; long l = *lp;
    *lp = (long)    d;
    *fp = (float)   i;
    *dp = (double)  l;
    return (double) f;
}
```

and its associated x86-64 assembly code

```
    x86-64 implementation of fcvt
    Arguments:
         i       %edi    int
         fp      %rsi    float *
         dp      %rdx    double *
         lp      %rcx    long *
fcvt:
  movss   (%rsi), %xmm0              Get f = *fp
  movq    (%rcx), %r8                Get l = *lp
  cvttsd2siq      (%rdx), %rax       Get d = *dp and convert to long
  movq    %rax, (%rcx)               Store at lp
  cvtsi2ss        %edi, %xmm1        Convert i to float
  movss   %xmm1, (%rsi)              Store at fp
  cvtsi2sdq       %r8, %xmm1         Convert l to double
  movsd   %xmm1, (%rdx)              Store at dp
  cvtss2sd        %xmm0, %xmm0       Convert f to double
  ret                                Return f
```

All of the arguments to `fcvt` are passed through the general-purpose registers, since they are either integers or pointers. The return value is returned in register `%xmm0`, the designated return register for `float` or `double` values. In this code, we see a number of the movement and conversion instructions of Figure 1.

By comparison, the following is the IA32 code for body of function `fcvt`:

```
    IA32+SSE implementation of fcvt
    Arguments:
         i       %edp+8  int
         fp      %ebp+12 float *
         dp      %ebp+16 double *
         lp      %ebp+20 long *
  movl    12(%ebp), %ebx             Get fp
  movl    16(%ebp), %esi             Get dp
  movl    20(%ebp), %edx             Get lp
  movss   (%ebx), %xmm1              Get f = *fp
  movl    (%edx), %ecx               Get l = *lp
  cvttsd2si       (%esi), %eax       Get d = *dp and convert to long
  movl    %eax, (%edx)               Store at dp
  cvtsi2ss        8(%ebp), %xmm0     Get i and convert to float
  movss   %xmm0, (%ebx)              Store at *fp
```

```
10   cvtsi2sd        %ecx, %xmm0         Convert l to double
11   movsd   %xmm0, (%esi)               Store at dp
12   cvtss2sd        %xmm1, %xmm1        Convert f to double
13   movsd   %xmm1, -16(%ebp)            Store in memory
14   fldl    -16(%ebp)                   Read from memory and push onto x87 stack
```

The main difference with the IA32 code is that all arguments are passed on the stack. The function must first load the arguments into registers before it can access the function data. Note also the use of the `cvttsd2si` instruction (line 6) to convert the double-precision to data type `long`, whereas the x86-64 code used a `cvttsd2siq` instruction (line 4). For IA32, both `int` and `long` are four bytes long. A final difference is how floating-point values are returned from functions, as implemented by instructions 13–14. The x87 floating-point architecture includes a set of eight floating-point registers organized as a shallow stack (see ASM:X87). Any floating-point value returned from a function should be at the top of this stack, as implemented by the `fldl` instruction (for double-precision) or the `flds` instruction (for single-precision.) The only way to transfer data from an XMM register to an x87 register is to first store it to memory with an SSE instruction (line 13) and then retrieve it from memory and push it onto the x87 stack with an x87 instruction (line 14.)

Practice Problem 1:

For the following C code, the expressions `val1`–`val4` all map to the program values `i`, `f`, `d`, and `l`:

```
double fcvt2(int *ip, float *fp, double *dp, long l)
{
    int i = *ip; float f = *fp; double d = *dp;
    *ip = (int)     val1;
    *fp = (float)   val2;
    *dp = (double)  val3;
    return (double) val4;
}
```

Determine the mapping, based on the following x86-64 code for the function:

```
    x86-64 implementation of fcvt2
    Arguments:
         ip      %rdi    int *
         fp      %rsi    float *
         dp      %rdx    double *
         l       %rcx    long
fcvt2:
  movl    (%rdi), %r8d
  movss   (%rsi), %xmm0
  cvttsd2si       (%rdx), %eax
  movl    %eax, (%rdi)
  cvtsi2ss        %r8d, %xmm1
  movss   %xmm1, (%rsi)
  cvtsi2sdq       %rcx, %xmm1
  movsd   %xmm1, (%rdx)
  cvtss2sd        %xmm0, %xmm0
  ret
```

Practice Problem 2:

The following C function converts an argument of type `src_t` to a return value of type `dst_t`, where these two types are defined using `typedef`.

```
dest_t cvt(src_t x)
{
    dest_t y = (dest_t) x;
    return y;
}
```

For execution on x86-64, assume argument `x` is either in `%xmm0` or in the appropriately named portion of register `%rdi` (i.e., `%rdi` or `%edi`), and that one of the conversion instructions is to be used to perform the type conversion and to copy the value to the appropriately named portion of register `%rax` (integer result) or `%xmm0` (floating-point result). Fill in the following table indicating the instruction, the source register, and the destination register for the following combinations of source and destination type:

$T_{\mathtt{x}}$	$T_{\mathtt{y}}$	Instruction	S	D
`long`	`double`	`cvtsi2sdq`	`%rdi`	`%xmm0`
`double`	`int`			
`float`	`double`			
`long`	`float`			
`float`	`long`			

3 Floating-Point Code in Procedures

With x86-64, the XMM registers are used for passing floating-point arguments to functions and for returning floating-point values from them. Specifically, the following conventions are observed:

- Up to eight floating point arguments can be passed in XMM registers `%xmm0`–`%xmm7`. These registers are used in the order the arguments are listed. Additional floating-point arguments can be passed on the stack.
- A function that returns a floating-point value does so in register `%xmm0`.
- All XMM registers are caller saved, The callee may overwrite any of these registers without first saving it.

When a function contains a combination of pointer, integer, and floating-point arguments, the pointers and integers are passed in general-purpose registers, while the floating-point values are passed in XMM registers. This means that the mapping of arguments to registers depends on both their types and their ordering. Here are several examples:

```
double f1(int x, double y, long z);
```

This function would have `x` in `%edi`, `y` in `%xmm0`, and `z` in `%rsi`.

```
double f2(double y, int x, long z);
```

This function would have the same register assignment as function `f1`.

```
double f1(float x, double *y, long *z);
```

This function would have `x` in `%xmm0`, `y` in `%rdi`, and `z` in `%rsi`.

Practice Problem 3:

For each of the following function declarations, determine the register assignments for the arguments:

A. `double g1(double a, long b, float c, int d);`

B. `double g2(int a, double *b, float *c, long d);`

C. `double g3(double *a, double b, int c, float d);`

D. `double g4(float a, int *b, float c, double d);`

4 Floating-Point Arithmetic operations

Single	Double	Effect	Description
`addss`	`addsd`	$D \leftarrow D + S$	Floating-point add
`subss`	`subsd`	$D \leftarrow D - S$	Floating-point subtract
`mulss`	`mulsd`	$D \leftarrow D \times S$	Floating-point multiply
`divss`	`divsd`	$D \leftarrow D \,/\, S$	Floating-point divide
`maxss`	`maxsd`	$D \leftarrow \max(D,\ S)$	Floating-point maximum
`minss`	`minsd`	$D \leftarrow \min(D,\ S)$	Floating-point minimum
`sqrtss`	`sqrtsd`	$D \leftarrow \sqrt{S}$	Floating-point square root

Figure 2: **Scalar floating-point arithmetic operations.** All have source S and destination D operands.

Figure 2 documents a set of scalar SSE2+ floating-point instructions that perform arithmetic operations. Each has two operands: a source S, which can be either an XMM register or a memory location, and a destination D, which must be an XMM register. Each operation has an instruction for single-precision and an instruction for double precision. The result is stored in the destination register.

As an example, consider the following floating-point function:

```
double funct(double a, float x, double b, int i)
{
    return a*x - b/i;
}
```

The x86-64 code is as follows:

```
    x86-64 implementation of funct
    Arguments:
          a       %xmm0   double
          x       %xmm1   float
          b       %xmm2   double
          i       %edi    int
1  funct:
2    cvtss2sd        %xmm1, %xmm1       Convert x to double
3    mulsd   %xmm0, %xmm1               Multiply x by a
4    cvtsi2sd        %edi, %xmm0        Convert i to double
5    divsd   %xmm0, %xmm2               Compute b/i
6    movapd  %xmm1, %xmm0               Copy x*a
7    subsd   %xmm2, %xmm0               Subtract b/i
8    ret                                Return
```

The three floating point arguments `a`, `x`, and `b` are passed in XMM registers `%xmm0`–`%xmm2`, while integer argument `i` is passed in register `%edi`. Conversion instructions are required to convert arguments `x` and `i` to double (lines 2 and 4.) We will describe the `movapd` instruction (line 6) in Section 6. Suffice it to say that, in this case, it copies source register `%xmm1` to destination register `%xmm0`. The function value is returned in register `%xmm0`.

By comparison, refer to the x87 code for this function in Section 6 of Web Aside ASM:X7. Whereas the x87 code involves operating on the floating-point register stack, the SSE code uses registers as individually addressable storage locations, much as does code operating on integer data.

Practice Problem 4:

For the following C function, the types of the four arguments are defined by `typedef`:

```
double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
{
    return p/(q+r) - s;
}
```

When compiled for x86-64, GCC generates the following code:

```
    x86-64 implementation of funct1
funct1:
  cvtsi2ss        %edi, %xmm3
  cvtsi2ssq       %rsi, %xmm2
  addss   %xmm0, %xmm2
  divss   %xmm2, %xmm3
  cvtss2sd        %xmm3, %xmm3
  movapd  %xmm3, %xmm0
  subsd   %xmm1, %xmm0
  ret
```

Determine the possible combinations of types of the four arguments (there may be more than one.)

Practice Problem 5:

Function `funct2` has the following prototype

```
double funct2(double w, int x, float y, long z);
```

When the function is compiled for x86-64, GCC generates the following code:

```
   x86-64 implementation of funct2
funct2:
  cvtsi2ss        %edi, %xmm2
  mulss   %xmm1, %xmm2
  cvtss2sd        %xmm2, %xmm2
  cvtsi2sdq       %rsi, %xmm1
  divsd   %xmm1, %xmm0
  subsd   %xmm0, %xmm2
  movapd  %xmm2, %xmm0
  ret
```

Write a C version of `funct2`.

Unlike integer arithmetic operations, the SSE floating-point operations cannot have immediate values as operands. Instead, the compiler must allocate and initialize storage for any constant values. The code then reads the values from memory. This is illustrated by the following Celsius to Fahrenheit conversion function:

```
double cel2fahr(double temp)
{
    return 1.8 * temp + 32.0;
}
```

The relevant parts of the x86-64 assembly code are as follows:

```
    Code

    x86-64 implementation of cel2fahr
    Argument temp in %xmm0
cel2fahr:
  mulsd   .LC2(%rip), %xmm0         Multiply by 1.8
  addsd   .LC3(%rip), %xmm0         Add 32.0
  ret
.LC2:
  .long   3435973837        Low order four bytes of 1.8
  .long   1073532108        High order four bytes of 1.8
.LC3:
  .long   0                 Low order four bytes of 32.0
  .long   1077936128        High order four bytes of 32.0
```

We see that the function reads the value 1.8 from the memory location labeled `.LC2`, and the value 32.0 from the memory location labeled `.LC3`. Looking at the values associated with these labels, we see that each is specified by a pair of `.long` declarations with the values given in decimal. How should these be interpreted as floating-point values? Looking at the declaration labeled `.LC2`, we see that the two values are 3435973837 (`0xcccccccd`) and 1073532108 (`0x3ffccccc`). Since the machine uses little-endian byte ordering, the first value gives the low-order 4 bytes, while the second gives the high-order 4 bytes. From the high-order bytes, we can extract an exponent field of `0x3ff` (1023), from which we subtract a bias of 1023 to get an exponent of 0. Concatenating the fraction bits of the two values, we get a fraction field of `0xccccccccccccd`, which can be shown to be the fractional binary representation of 0.8, to which we add the implied leading one to get 1.8.

Practice Problem 6:

Show how the numbers declared at label `.LC3` encode the number 32.0.

5 Floating-Point Comparison Operations

SSE2+ provides two instructions for comparing floating-point values:

Instruction	Based on	Description
`ucomiss` S_2, S_1	S_1 `-` S_2	Compare single precision
`ucomisd` S_2, S_1	S_1 `-` S_2	Compare double precision

These instructions are similar to the `cmpl` and `cmpq` instructions (see CS:APP2e Section 3.6), in that they compare operands S_1 and S_2 and set the condition codes to indicate their relative values. As with `cmpq`, they follow the ATT-format convention of listing the operands in reverse order. Argument S_2 must be in an XMM register, while S_1 can either be in an XMM register or in memory.

The floating-point comparison instructions set three condition codes: the zero flag `ZF`, the carry flag `CF`, and the parity flag `PF`. We did not document the parity flag in CS:APP2e Chapter 3, because it is not used in GCC-generated x86 code. For integer operations, this flag is set when the most recent arithmetic or logical operation yielded a value where the least significant byte has even parity (i.e., an even number of 1's in the byte). For floating-point comparisons, however, the flag is set when either operand is *NaN*. By convention, any comparison in C is considered to fail when one of the arguments is a *NaN*, and this flag is used to detect such a condition. For example, even the comparison `x == x` yields 0 when `x` is a *NaN*.

The condition codes are set as follows:

Ordering	`CF`	`ZF`	`PF`
Unordered	1	1	1
`<`	1	0	0
`=`	0	1	0
`>`	0	0	0

The *Unordered* case occurs when either of the operands is *NaN*. This can be detected from the parity flag. Commonly, the `jp` (for "jump on parity") instruction is used to conditionally jump when a floating-point comparison yields an unordered result. Except for this case, the values of the carry and zero flags are the same as those for an unsigned comparison: `ZF` is set when the two operands are equal, and `CF` is set when $S_1 < S_2$. Instructions such as `ja` and `jb` are used to conditionally jump on various combinations of these flags.

As an example of floating-point comparisons, the following C function classifies argument `x` according to its relation to 0.0, returning an enumerated type as result.

```
typedef enum {NEG, ZERO, POS, OTHER} range_t;

range_t find_range(float x)
{
    int result;
    if (x < 0)
        result = NEG;
    else if (x == 0)
        result = ZERO;
    else if (x > 0)
        result = POS;
    else
        result = OTHER;
    return result;
}
```

Enumerated types in C are encoded as integers, and so the possible function values are: 0 (`NEG`), 1 (`ZERO`), 2 (`POS`), and 3 (`OTHER`). This final outcome occurs when the value of `x` is *NaN*.

GCC generates the following x86-64 code for `find_range`:

```
    x86-64 implementation of find_range
    Argument x in %xmm0 (single precision)
find_range:
  movl    $0, %eax                      Set result = 0
  ucomiss .LC0(%rip), %xmm0             Compare x:0
  jp      .L4                           If NaN, goto nonneg
  jb      .L7                           If <, goto done
.L4:                                  nonneg:
  movl    $1, %eax                      Set result = 1
  ucomiss .LC0(%rip), %xmm0             Compare x:0
  jp      .L8                           if NaN, goto  posornan
  je      .L7                           If ==, goto done
.L8:                                  posornan:
  ucomiss .LC0(%rip), %xmm0             Compare x:0
  setbe   %al                           Set result = NaN ? 1 : 0
  movzbl  %al, %eax                     Zero extend
  addl    $2, %eax                      result += 2 (2 for >0, 3 for NaN)
.L7:                                  done:
  rep ; ret                             Return result
```

The code is somewhat arcane—it compares `x` to 0.0 three times, even though the required information could be obtained with a single comparison. Let us trace the flow of the function for the four possible comparison results.

`x` < 0.0**:** The `jb` instruction on line 5 will be taken, jumping to the end with a return value of 0.

`x` $= 0.0$**:** The `je` instruction (line 10) will be taken, jumping to the end with a return value of 1 (set on line 7.)

`x` > 0.0**:** No branches will be taken. The `setbe` (line 13) will yield 0, and this will be incremented by the `addl` instruction (line 15) to give a return value of 2.

`x` $= NaN$**:** Both `jp` branches (lines 4 and 9) will be taken. Then the `setbe` instruction (line 13) will change the return value from 0 to 1, and this value is then incremented from 1 to 3 (line 15.)

Compared to the awkward procedure required to extract and test the floating-point status word with x87 (Web Aside ASM:X87, Section 7), the SSE instructions to conditionally compare and test values is very similar to their counterparts for comparing and testing integers.

Practice Problem 7:

Function `funct3` has the following prototype

```
double funct3(int *ap, double b, long c, float *dp);
```

When the function is compiled for x86-64, GCC generates the following code:

x86-64 implementation of funct3

```
funct3:
  movss   (%rdx), %xmm2
  cvtsi2sd        (%rdi), %xmm1
  ucomisd %xmm1, %xmm0
  jbe     .L6
  cvtsi2ssq       %rsi, %xmm0
  mulss   %xmm2, %xmm0
  cvtss2sd        %xmm0, %xmm0
  ret
.L6:
  cvtsi2ssq       %rsi, %xmm0
  movaps  %xmm2, %xmm1
  addss   %xmm2, %xmm1
  addss   %xmm1, %xmm0
  cvtss2sd        %xmm0, %xmm0
  ret
```

Write a C version of `funct3`.

6 Performing Common Floating-Point Operations in Uncommon Ways

At times, GCC makes surprising choices of instructions for performing common tasks. As examples, we've seen how the `leal` instruction is often used to perform integer arithmetic (CS:APP2e Section 3.5), and the `xorl` instruction is used to set registers to 0 (CS:APP2e Problem 3.10).

There are far more instructions for performing floating-point operations than we have documented here, and some of these appear in unexpected places. We document a few such cases here.

6.1 Using Instructions for Manipulating Packed Data

Instruction	Source	Destination	Description
`movaps`	X	X	Move aligned, packed single precision
`movapd`	X	X	Move aligned, packed double precision
`cvtps2pd`	X	X	Convert packed single to packed double precision
`cvtpd2ps`	X	X	Convert packed double to packed single precision
`unpcklps`	X	X	Unpack and interleave low packed single precision
`unpcklpd`	X	X	Unpack and interleave low packed double precision

Figure 3: **Some packed format floating-point movement and conversion operations.** These instructions are often found in scalar code.

Figure 3 shows a number of instructions for manipulating XMM registers containing *packed* floating-point data, where a single XMM register holds either two double-precision or four single-precision values. We find these instructions being used in code that operates only on *scalar* data, low-order value in an XMM register.

The `movapd` and `movaps` instructions copy the entire contents of one XMM register to another. (They can also copy XMM register contents to and from memory, but we will not consider these cases here.) We have already seen instances of the `movapd` instruction being used to copy from one XMM register to another. For these cases, whether the program copies the entire register or just the low-order value affects neither the program functionality nor the execution speed, and so using this instruction rather than the more natural `movsd` makes no real difference.

6.2 Converting between Single and Double Precision

Some versions of GCC generate code that uses an idiosyncratic means of converting between single and double precision values. For example, suppose the low-order four bytes of `%xmm0` hold a single-precision value, then the instruction `cvtss2sd %xmm0, %xmm0` would convert it to double precision and store it in the lower eight bytes of `%xmm0`. Instead, we find the following code generated by GCC:

```
    Conversion from single to double precision
unpcklps       %xmm0, %xmm0     Replicate first vector element
cvtps2pd       %xmm0, %xmm0     Convert two vector elements to double
```

The instruction `unpcklps` instruction is normally used to interleave the values in two XMM registers. That is, if the source register contains words $[s_3, s_2, s_1, s_0]$, and the destination register contains words $[d_3, d_2, d_1, d_0]$, then the resulting value of the destination register would be $[s_1, d_1, s_0, d_0]$. In the code above, we see that same register being used as source and destination, and so if the original register held values $[x_3, x_2, x_1, x_0]$, then the instruction would update the register to hold values $[x_1, x_1, x_0, x_0]$. The `cvtps2pd` instruction expands the two low-order single-precision values in the source XMM register to be the two double-precision values in the destination XMM register. Applying this to the result of the preceding `unpcklps` instruction would give values $[dx_0, dx_0]$, where dx_0 is the result of converting x to double precision. That is, the net effect of the two instructions is to convert the original single-precision value in the low-order 4 bytes of `%xmm0` to double precision and store two copies of it in `%xmm0`. It is unclear why GCC generates this code. There is no benefit or need to have the value duplicated within the XMM register.

`Gcc` generates similar code for converting from double to single precision:

```
    Conversion from double to single precision
   unpcklpd       %xmm0, %xmm0      Replicate first vector element
   cvtpd2ps       %xmm0, %xmm0      Convert two vector elements to double
```

Suppose these instructions start with register `%xmm0` holding two double-precision values $[x_1, x_0]$. Then the `unpcklpd` instruction will set it to $[x_0, x_0]$. The `cvtpd2ps` will convert these values to single precision, pack them into the low-order half or the register, and set the upper half to 0, yielding a result $[0.0, 0.0, x_0, x_0]$ (recall that floating-point value 0.0 is represented by a bit pattern of all 0s.) Again, there is no clear value in computing the conversion from one precision to another this way, rather than by using the single instruction `cvtsd2ss %xmm0, %xmm0`. The fact that this code is generated only by some versions of GCC and not by others seems to indicate that it has no particular benefit.

6.3 Using Bit-Wise Operations

Single	Double	Effect	Description
`xorps`	`xorpd`	$D \leftarrow D$ ^ S	Bit-wise Exclusive-Or
`andps`	`andpd`	$D \leftarrow D$ & S	Bit-wise And

Figure 4: **Bit-wise operations on packed data.** These instructions perform Boolean operations on all 128 bits in an XMM register

Figure 4 show that we can perform bitwise operations on XMM registers, much as we can for the general-purpose registers. In the code generated by GCC, we often see these operations being applied to an entire XMM register, rather than just the low-order 4 or 8 bytes. These operations are often simple and convenient ways to manipulate floating-point values, as is explored in the following problem.

Practice Problem 8:

Consider the following C function, where `EXPR` is a macro defined with `#define`:

```
double simplefun(double x) {
    return EXPR(x);
}
```

Below we show the SSE code generated for different definitions of `EXPR`, where value `x` is held in `%xmm0`. All of them correspond to some useful operation on floating-point values. Identify what the operations are. Your answers will require you to understand the bit patterns of the constant words being retrieved from memory.

A.
```
  andpd   .LC1(%rip), %xmm0
.LC1:
  .long   4294967295
  .long   2147483647
  .long   0
  .long   0
  .align 16
```

B.
```
  xorpd   %xmm0, %xmm0
```

C.
```
  xorpd   .LC2(%rip), %xmm0
.LC2:
  .long   0
  .long   -2147483648
  .long   0
  .long   0
```

7 Final Observations

We see that the general style of machine code generated for operating on floating-point data with SSE is similar to what we have seen for operating on integer data. Both use a collection of registers to hold and operate on values. For x86-64 code, we also use these registers for passing function arguments.

Of course, there are many complexities in dealing with the different data types and the rules for evaluating expressions containing a mixture of data types, but fundamentally, SSE code is more straightforward than the x87 code historically used to implement floating-point operations on x86 machines. In addition, the SSE code generally runs faster, since there is less need to move data back and forth between registers and memory.

SSE also has the potential to make computations run faster by performing parallel operations on packed data. Compiler developers are working on automating the conversion of scalar code to parallel code, but currently the most reliable way to achieve higher performance through parallelism is to use the extensions to the C language supported by GCC for manipulating vectors of data. See Web Aside OPT:SIMD to see how this can be done.

Solutions to Practice Problems

Problem 1 Solution: [Pg. 5]

This exercise requires that you step through the code, paying careful attention to which conversion and data movement instructions are used. We can see the values being retrieved and converted as follows:

- The value at `dp` is retrieved, converted to an `int` (line 4), and then stored at `ip`. We can therefore infer that `val1` is `d`.
- The value at `ip` is retrieved, converted to a `float` (line 6), and then stored at `fp`. We can therefore infer that `val2` is `i`.
- The value of `l` is converted to a `double` (line 8) and stored at `dp`. We can therefore infer that `val3` is `l`
- The value at `fp` is retrieved, converted to a `double` (line 10) and left in register `%xmm0` as the return value. We can therefore infer that `val4` is `f`.

Problem 2 Solution: [Pg. 6]

These cases can be handled by selecting the appropriate entry from the table in Figure 1.

T_x	T_y	Instruction	S	D
`long`	`double`	`cvtsi2sdq`	`%rdi`	`%xmm0`
`double`	`int`	`cvttsd2si`	`%xmm0`	`%eax`
`float`	`double`	`cvtss2sd`	`%xmm0`	`%xmm0`
`long`	`float`	`cvtsi2ssq`	`%rdi`	`%xmm0`
`float`	`long`	`cvtss2siq`	`%xmm0`	`%rax`

Problem 3 Solution: [Pg. 7]

The basic rules for mapping arguments to registers are fairly simple (although they become much more complex with more and other types of arguments [3]).

A. `double g1(double a, long b, float c, int d);`

   Registers: `a` in `%xmm0`, `b` in `%rdi` `c` in `%xmm1`, `d` in `%esi`.

B. `double g2(int a, double *b, float *c, long d);`

   Registers: `a` in `%edi`, `b` in `%rsi`, `c` in `%rdx`, `d` in `%rcx`.

C. `double g3(double *a, double b, int c, float d);`

   Registers: `a` in `%rdi`0, `b` in `%xmm0`, `c` in `%esi`, `d` in `%xmm1`.

D. `double g4(float a, int *b, float c, double d);`

   Registers: `a` in `%xmm0`, `b` in `%rdi`, `c` in `%xmm1`, `d` in `%xmm2`.

Problem 4 Solution: [Pg. 8]

This problem demonstrates the challenge of matching arguments to registers when we do not know the argument data types. One strategy is to work backwards. We see that the subtraction instruction on line 8 computes the difference of registers `%xmm0` and `%xmm1`, where the former contains the result from the instruction on the preceding line (which must be the result of computing the expression `p/(q+r)`, and the latter contains argument `s`. Tracing back with `%xmm1`, we can see that this it must hold the second floating-point argument, and we can infer that `s` is of data type `double`.

The result of computing the expression `p/(q+r)` gets converted from single to double precision by the instruction of line 6, and so we can conclude that among arguments `p`, `q`, and `r` there must be at least one with data type `float`, and none with data type `double`.

We see that the first two integer arguments are converted to data type `float` on lines 2 (from `int`) and 3 (from `long`). The former is used in the numerator in the division instruction of line 5, and hence must be `p`, while the second is added to the first floating-point argument.

Fundamentally, however, there is no way to determine the relative orderings of the first floating-point argument and the second integer argument, since the addition operation of line 4 is commutative. One must be argument `p` and the other must be argument q, but there is no way to distinguish between the two.

In fact, GCC generates the exact came code when the arguments of `funct1` are defined according to either of the following two prototypes:

```
double funct1a(int p, float q, long r, double s);
double funct1b(int p, long q, float r, double s);
```

Problem 5 Solution: [Pg. 9]

This problem can readily be solved by stepping through the assembly code and determining what is computed on each step, as shown with the annotations below:

```
funct2:
 x86-64 implementation of funct2
 Arguments:
       w      %xmm0   double
       x      %edi    int
       y      %xmm1   float
       z      %rsi    long
  cvtsi2ss        %edi, %xmm2        Convert x to float
  mulss   %xmm1, %xmm2               Multiply by y
  cvtss2sd        %xmm2, %xmm2       Convert x*y to double
  cvtsi2sdq       %rsi, %xmm1        Convert z to double
  divsd   %xmm1, %xmm0               Compute w/z
  subsd   %xmm0, %xmm2               Compute x*y-w/z
  movapd  %xmm2, %xmm0
  ret
```

We can conclude from this analysis that the function computes `y*x - w/z`.

Problem 6 Solution: [Pg. 10]

This problem involves the same reasoning as was required to see that numbers declared at label `.LC2` encode 1.8, but with a simpler example.

We see that the two values are 0 and 1077936128 (`0x40400000`). From the high-order bytes, we can extract an exponent field of `0x404` (1028), from which we subtract a bias of 1023 to get an exponent of 5. Concatenating the fraction bits of the two values, we get a fraction field of 0, but with the implied leading value giving value 1.0. The constant is therefore $1.0 \times 2^5 = 32.0$.

Problem 7 Solution: [Pg. 12]

Again, we annotate the code, including dealing with the conditional branch:

```
    x86-64 implementation of funct3
    Arguments:
          ap      %rdi    int *
          b       %xmm0   double
          c       %rsi    long
          dp      %rdx    double *
funct3:
  movss    (%rdx), %xmm2                   Get d = *dp
  cvtsi2sd         (%rdi), %xmm1           Get a = *ap and convert to double
  ucomisd %xmm1, %xmm0                     Compare b:a
  jbe      .L6                             If b <= a, goto lesseq
  cvtsi2ssq        %rsi, %xmm0             Convert c to float
  mulss    %xmm2, %xmm0                    Multiply by d
  cvtss2sd         %xmm0, %xmm0            Convert c*d to double
  ret                                      Return
.L6:                                     lesseq:
  cvtsi2ssq        %rsi, %xmm0             Convert c to float
  movaps   %xmm2, %xmm1                    Copy d
  addss    %xmm2, %xmm1                    Compute d+d = 2*d
  addss    %xmm1, %xmm0                    Compute c + 2*d
  cvtss2sd         %xmm0, %xmm0            Convert to double
  ret                                      Return
```

From this, we can write the following code for `funct3`:

```
double funct3(int *ap, double b, long c, float *dp) {
    int a = *ap;
    float d = *dp;
    if (a < b)
        return c*d;
    else
        return c+2*d;
}
```

Problem 8 Solution: [Pg. 14]

A. We see here that the 16 bytes starting at address `.LC1` form a mask, where the low-order 8 bytes contain all 1's, except for most significant bit, which is the sign bit of a double-precision value. When we compute the AND of this mask with `%xmm0`, it will clear the sign bit of `x`, yielding the absolute value. In fact, we generated this code by defining `EXPR(x)` to be `fabs(x)`, where `fabs` is defined in `<math.h>`.

B. We see that the `xorpd` instruction sets the entire register to 0, and so this is a way to generate floating-point constant 0.0.

C. We see that the 16 bytes starting at address `.LC2` form a mask with a single one bit, at the position of the sign bit for the low-order value in the XMM register. When we compute the EXCLUSIVE-OR of this mask with `%xmm0`, we change the sign of `x`, computing the expression `-x`.

References

[1] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference A–M*, 2009. Order Number 253667.

[2] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference N–Z*, 2009. Order Number 253668.

[3] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V application binary interface AMD64 architecture processor supplement. Technical report, AMD64.org, 2009. Available at `http://www.x86-64.org/`.

[4] D. Naishlos. Autovectorization in GCC. In *GCC Developers Summit*, 2006.

Index

NaN, floating-point not-a-number, 10
CS:APP2e , 1
`%xmm0`, Return floating-point value register, 4
`%xmm0`, return floating-point value, 6
8086, Intel 16-bit microprocessor, 2
8087, floating-point coprocessor, 2

`andpd` [x86-64] And packed double precision, 14
`andps` [x86-64] And packed single precision, 14
architecture
    floating-point, *1*
ATT format assembly code, 2
ATT format, argument listing, 10

`cvtpd2ps` [x86-64] Convert packed double to packed single precision, 13
`cvtps2pd` [x86-64] Convert packed single to packed double precision, 13, 14
`cvtsd2ss` [x86-64] Convert single to double precision, 3
`cvtsi2sd` [x86-64] Convert integer to double precision, 3
`cvtsi2sdq` [x86-64] Convert quadword integer to double precision, 3
`cvtsi2ss` [x86-64] Convert integer to single precision, 3
`cvtsi2ssq` [x86-64] Convert quadword integer to single precision, 3
`cvtss2sd` [x86-64] Convert single to double precision, 3
`cvttsd2si` [x86-64] Convert double precision to integer, 3
`cvttsd2siq` [x86-64] Convert double precision to quadword integer, 3
`cvttss2si` [x86-64] Convert single precision to integer, 3
`cvttss2siq` [x86-64] Convert single precision to quadword integer, 3

`fldd` [IA32/x86-64] x87 FP load double precision, 5
`flds` [IA32/x86-64] x87 FP load single precision, 5
floating-point architecture, *1*

Intel format assembly code, 2

`jp` [x86-64] Jump when parity flag set, 10

MMX, Intel media extension, 1
`movapd` [x86-64] Move aligned, packed double precision, 13
`movaps` [x86-64] Move aligned, packed single precision, 13
`movsd` [x86-64] Move double precision, 3
`movss` [x86-64] Move single precision, 3

packed data format, 2
packed format data, 13

scalar format data, *2*, 13
SIMD, Single-Instruction Multiple Data execution, 2
SSE, Streaming SIMD Extensions, *1*
SSE2+, SSE versions 2 and higher, 2

`ucomisd` [x86-64] compare double precision, 10
`ucomiss` [x86-64] compare single precision, 10
Unordered, floating-point comparison outcome, 10
`unpcklpd` [x86-64] Unpack and interleave low packed double precision, 13, 14
`unpcklps` [x86-64] Unpack and interleave low packed single precision, 13, 14

x87, floating-point instructions for 8087, 2
XMM, SSE2+ registers, 2
`xorpd` [x86-64] Exclusive-or packed double precision, 14
`xorps` [x86-64] Exclusive-or packed single precision, 14