Computer Systems
A Programmer’s Perspective
Practice Problem 3.50 (solution page 383)
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:

\[
\begin{align*}
\text{double fcvt2}(\text{int } &\text{ip, float } \text{fp, double } \text{dp, long } \text{l}) \\
\text{ip in } &\%\text{rdi, fp in } \%\text{rsi, dp in } \%\text{rdx, l in } \%\text{rcx} \\
\text{Result returned in } &\%\text{xmm0}
\end{align*}
\]

1. \text{fcvt2:}
2. \text{movl } (\%\text{rdi}), \%\text{eax}
3. \text{vmovss } (\%\text{rsi}), \%\text{xmm0}
4. \text{vcvttsd2si } (\%\text{rdx}), \%\text{r8d}
5. \text{movl } \%\text{r8d}, (\%\text{rdi})
6. \text{vcvtsi2ss } \%\text{eax}, \%\text{xmm1}, \%\text{xmm1}
7. \text{vmovss } \%\text{xmm1}, (\%\text{rsi})
8. \text{vcvtsi2sdq } \%\text{rcx}, \%\text{xmm1}, \%\text{xmm1}
9. \text{vmovsd } \%\text{xmm1}, (\%\text{rdx})
10. \text{vunpcklps } \%\text{xmm0}, \%\text{xmm0}, \%\text{xmm0}
11. \text{vcvtpps2pd } \%\text{xmm0}, \%\text{xmm0}
12. \text{ret}

Practice Problem 3.51 (solution page 384)
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 that argument x is either in \%xmm0 or in the appropriately named portion of register \%rdi (i.e., \%rdi or \%edi). One or two instructions are 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). Show the instruction(s), including the source and destination registers.

<table>
<thead>
<tr>
<th>(T_x)</th>
<th>(T_y)</th>
<th>Instruction(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>double</td>
<td>vcvtsi2sdq %rdi, %xmm0</td>
</tr>
<tr>
<td>double</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>long</td>
<td></td>
</tr>
</tbody>
</table>

### 3.11.2 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. As is illustrated in Figure 3.45, 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:

```c
double f1(int x, double y, long z);
```
This function would have \(x\) in %edi, \(y\) in %xmm0, and \(z\) in %rsi.

```c
double f2(double y, int x, long z);
```
This function would have the same register assignment as function \(f1\).

```c
double f1(float x, double *y, long *z);
```
This function would have \(x\) in %xmm0, \(y\) in %rdi, and \(z\) in %rsi.

### Practice Problem 3.52 (solution page 384)

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);
3.11.3 Floating-Point Arithmetic Operations

Figure 3.49 documents a set of scalar AVX2 floating-point instructions that perform arithmetic operations. Each has either one \((S_1)\) or two \((S_1, S_2)\) source operands and a destination operand \(D\). The first source operand \(S_1\) can be either an XMM register or a memory location. The second source operand and the destination operands must be XMM registers. 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:

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

The x86-64 code is as follows:

```c
double funct(double a, float x, double b, int i)
{ a in %xmm0, x in %xmm1, b in %xmm2, i in %edi

1 funct:
    The following two instructions convert x to double
2  vunpcklps %xmm1, %xmm1, %xmm1
3  vcvt-ps2pd %xmm1, %xmm1
4  vmulsd %xmm0, %xmm1, %xmm1        Multiply a by x
5  vcvt-si2sd %edi, %xmm1, %xmm1    Convert i to double
6  vdivsd %xmm1, %xmm2, %xmm2       Compute b/i
```

<table>
<thead>
<tr>
<th>Single</th>
<th>Double</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vaddss</td>
<td>vaddsd</td>
<td>(D \leftarrow S_2 + S_1)</td>
<td>Floating-point add</td>
</tr>
<tr>
<td>vsubss</td>
<td>vsubsd</td>
<td>(D \leftarrow S_2 - S_1)</td>
<td>Floating-point subtract</td>
</tr>
<tr>
<td>vmulss</td>
<td>vmulsd</td>
<td>(D \leftarrow S_2 \times S_1)</td>
<td>Floating-point multiply</td>
</tr>
<tr>
<td>vdivss</td>
<td>vdivsd</td>
<td>(D \leftarrow S_2/S_1)</td>
<td>Floating-point divide</td>
</tr>
<tr>
<td>vmaxss</td>
<td>vmaxsd</td>
<td>(D \leftarrow \max(S_2, S_1))</td>
<td>Floating-point maximum</td>
</tr>
<tr>
<td>vminss</td>
<td>vminsd</td>
<td>(D \leftarrow \min(S_2, S_1))</td>
<td>Floating-point minimum</td>
</tr>
<tr>
<td>sqrtss</td>
<td>sqrtsd</td>
<td>(D \leftarrow \sqrt{S_1})</td>
<td>Floating-point square root</td>
</tr>
</tbody>
</table>

**Figure 3.49** Scalar floating-point arithmetic operations. These have either one or two source operands and a destination operand.
Floating-Point Code

7 vsubsd %xmm2, %xmm0, %xmm0 Subtract from a*x
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. The standard two-instruction sequence is used to convert argument x to double (lines 2–3). Another conversion instruction is required to convert argument i to double (line 5). The function value is returned in register %xmm0.

Practice Problem 3.53 (solution page 384)

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, gcc generates the following code:

double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
1 funct1:
2    vcvtsi2ssq %rsi, %xmm2, %xmm2
3    vaddss %xmm0, %xmm2, %xmm0
4    vcvtsi2ss %edi, %xmm2, %xmm2
5    vdivss %xmm0, %xmm2, %xmm0
6    vunpcklps %xmm0, %xmm0, %xmm0
7    vcvtss2pd %xmm0, %xmm0
8    vsubsd %xmm1, %xmm0, %xmm0
9    ret

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

Practice Problem 3.54 (solution page 385)

Function funct2 has the following prototype:

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

gcc generates the following code for the function:

double funct2(double w, int x, float y, long z)
    w in %xmm0, x in %edi, y in %xmm1, z in %rsi
1 funct2:
2    vcvtsi2ss %edi, %xmm2, %xmm2
3    vmulss %xmm1, %xmm2, %xmm1
3.11.4 Defining and Using Floating-Point Constants

Unlike integer arithmetic operations, AVX 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:

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

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

```assembly
cel2fahr:
    vmulsd .LC2(%rip), %xmm0, %xmm0  ; Multiply by 1.8
    vaddsd .LC3(%rip), %xmm0, %xmm0  ; Add 32.0
    ret

.LC2:
    .long 3435973837  ; Low-order 4 bytes of 1.8
    .long 1073532108  ; High-order 4 bytes of 1.8

.LC3:
    .long 0            ; Low-order 4 bytes of 32.0
    .long 1077936128   ; High-order 4 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 0xcccccccccccc, 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.
Section 3.11 Floating-Point Code

### 3.11.5 Using Bitwise Operations in Floating-Point Code

At times, we find `gcc` generating code that performs bitwise operations on XMM registers to implement useful floating-point results. Figure 3.50 shows some relevant instructions, similar to their counterparts for operating on general-purpose registers. These operations all act on packed data, meaning that they update the entire destination XMM register, applying the bitwise operation to all the data in the two source registers. Once again, our only interest for scalar data is the effect these instructions have on the low-order 4 or 8 bytes of the destination. These operations are often simple and convenient ways to manipulate floating-point values, as is explored in the following problem.

**Practice Problem 3.56 (solution page 386)**

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

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

Below, we show the AVX2 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.**

1. `vmovsd .LC1(%rip), %xmm1`
2. `vandpd %xmm1, %xmm0, %xmm0`
3. `.LC1:`
4. `.long 4294967295`
5. `.long 2147483647`
6. `.long 0`
7. `.long 0`

**B.**

1. `vxorpd %xmm0, %xmm0, %xmm0`