```
    long i;
    double result = a[degree];
    for (i = degree-1; i >= 0; i--)
        result = a[i] + x*result;
    return result;
}
```

A. For degree n, how many additions and how many multiplications does this code perform?

B. On our reference machine, with the arithmetic operations having the latencies shown in Figure 5.12, we measure the CPE for this function to be 8.00. Explain how this CPE arises based on the data dependencies formed between iterations due to the operations implementing line 7 of the function.

C. Explain how the function shown in Practice Problem 5.5 can run faster, even though it requires more operations.

5.8 Loop Unrolling

Loop unrolling is a program transformation that reduces the number of iterations for a loop by increasing the number of elements computed on each iteration. We saw an example of this with the function `psum2` (Figure 5.1), where each iteration computes two elements of the prefix sum, thereby halving the total number of iterations required. Loop unrolling can improve performance in two ways. First, it reduces the number of operations that do not contribute directly to the program result, such as loop indexing and conditional branching. Second, it exposes ways in which we can further transform the code to reduce the number of operations in the critical paths of the overall computation. In this section, we will examine simple loop unrolling, without any further transformations.

Figure 5.16 shows a version of our combining code using what we will refer to as "2×1 loop unrolling." The first loop steps through the array two elements at a time. That is, the loop index `i` is incremented by 2 on each iteration, and the combining operation is applied to array elements i and $i + 1$ in a single iteration.

In general, the vector length will not be a multiple of 2. We want our code to work correctly for arbitrary vector lengths. We account for this requirement in two ways. First, we make sure the first loop does not overrun the array bounds. For a vector of length n, we set the loop limit to be $n - 1$. We are then assured that the loop will only be executed when the loop index i satisfies $i < n - 1$, and hence the maximum array index $i + 1$ will satisfy $i + 1 < (n - 1) + 1 = n$.

We can generalize this idea to unroll a loop by any factor k, yielding $k \times 1$ *loop unrolling*. To do so, we set the upper limit to be $n - k + 1$ and within the loop apply the combining operation to elements i through $i + k - 1$. Loop index `i` is incremented by k in each iteration. The maximum array index $i + k - 1$ will then be less than n. We include the second loop to step through the final few elements of the vector one at a time. The body of this loop will be executed between 0 and $k - 1$ times. For $k = 2$, we could use a simple conditional statement

```
/* 2 x 1 loop unrolling */
void combine5(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc = (acc OP data[i]) OP data[i+1];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

Figure 5.16 **Applying 2×1 loop unrolling.** This transformation can reduce the effect of loop overhead.

to optionally add a final iteration, as we did with the function `psum2` (Figure 5.1). For $k > 2$, the finishing cases are better expressed with a loop, and so we adopt this programming convention for $k = 2$ as well. We refer to this transformation as "$k \times 1$ loop unrolling," since we unroll by a factor of k but accumulate values in a single variable `acc`.

Practice Problem 5.7 (solution page 611)

Modify the code for `combine5` to unroll the loop by a factor $k = 5$.

When we measure the performance of unrolled code for unrolling factors $k = 2$ (`combine5`) and $k = 3$, we get the following results:

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine4`	551	No unrolling	1.27	3.01	3.01	5.01
`combine5`	568	2×1 unrolling	1.01	3.01	3.01	5.01
		3×1 unrolling	1.01	3.01	3.01	5.01
Latency bound			1.00	3.00	3.00	5.00
Throughput bound			0.50	1.00	1.00	0.50

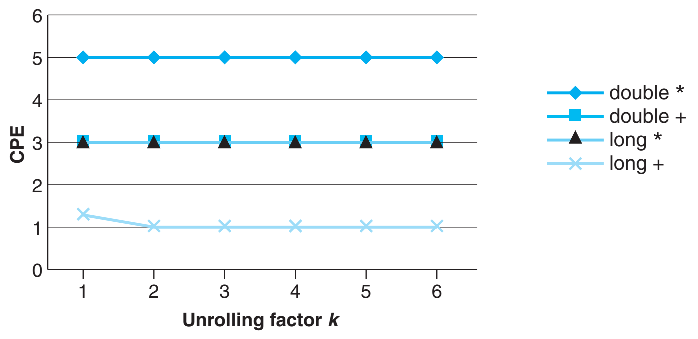


Figure 5.17
CPE performance for different degrees of $k \times 1$ loop unrolling. Only integer addition improves with this transformation.

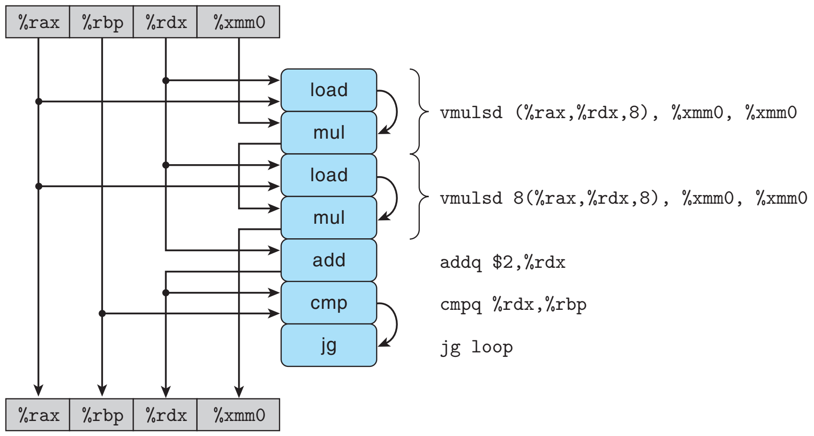
We see that the CPE for integer addition improves, achieving the latency bound of 1.00. This result can be attributed to the benefits of reducing loop overhead operations. By reducing the number of overhead operations relative to the number of additions required to compute the vector sum, we can reach the point where the 1-cycle latency of integer addition becomes the performance-limiting factor. On the other hand, none of the other cases improve—they are already at their latency bounds. Figure 5.17 shows CPE measurements when unrolling the loop by up to a factor of 10. We see that the trends we observed for unrolling by 2 and 3 continue—none go below their latency bounds.

To understand why $k \times 1$ unrolling cannot improve performance beyond the latency bound, let us examine the machine-level code for the inner loop of `combine5`, having $k = 2$. The following code gets generated when type `data_t` is `double`, and the operation is multiplication:

```
Inner loop of combine5.  data_t = double, OP = *
i in %rdx, data %rax, limit in %rbx, acc in %xmm0
.L35:                                          loop:
  vmulsd  (%rax,%rdx,8), %xmm0, %xmm0            Multiply acc by data[i]
  vmulsd  8(%rax,%rdx,8), %xmm0, %xmm0           Multiply acc by data[i+1]
  addq    $2, %rdx                               Increment i by 2
  cmpq    %rdx, %rbp                             Compare to limit:i
  jg      .L35                                   If >, goto loop
```

We can see that GCC uses a more direct translation of the array referencing seen in the C code, compared to the pointer-based code generated for `combine4`.[2] Loop index `i` is held in register `%rdx`, and the address of `data` is held in register `%rax`. As before, the accumulated value `acc` is held in vector register `%xmm0`. The loop unrolling leads to two `vmulsd` instructions—one to add `data[i]` to `acc`, and

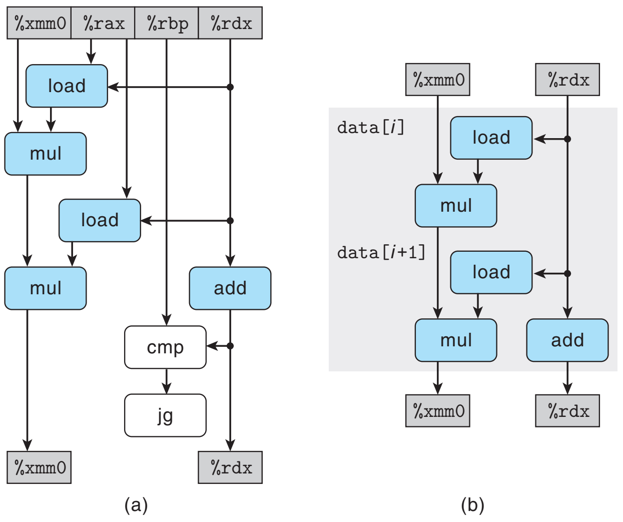
2. The GCC optimizer operates by generating multiple variants of a function and then choosing one that it predicts will yield the best performance and smallest code size. As a consequence, small changes in the source code can yield widely varying forms of machine code. We have found that the choice of pointer-based or array-based code has no impact on the performance of programs running on our reference machine.



```
vmulsd (%rax,%rdx,8), %xmm0, %xmm0
vmulsd 8(%rax,%rdx,8), %xmm0, %xmm0
addq $2,%rdx
cmpq %rdx,%rbp
jg loop
```

Figure 5.18 Graphical representation of inner-loop code for `combine5`. Each iteration has two `vmulsd` instructions, each of which is translated into a load and a mul operation.

Figure 5.19 Abstracting `combine5` **operations as a data-flow graph.** We rearrange, simplify, and abstract the representation of Figure 5.18 to show the data dependencies between successive iterations (a). We see that each iteration must perform two multiplications in sequence (b).



the second to add `data[i+1]` to `acc`. Figure 5.18 shows a graphical representation of this code. The `vmulsd` instructions each get translated into two operations: one to load an array element from memory and one to multiply this value by the accumulated value. We see here that register `%xmm0` gets read and written twice in each execution of the loop. We can rearrange, simplify, and abstract this graph, following the process shown in Figure 5.19(a), to obtain the template shown in Figure 5.19(b). We then replicate this template $n/2$ times to show the computation for a vector of length n, obtaining the data-flow representation

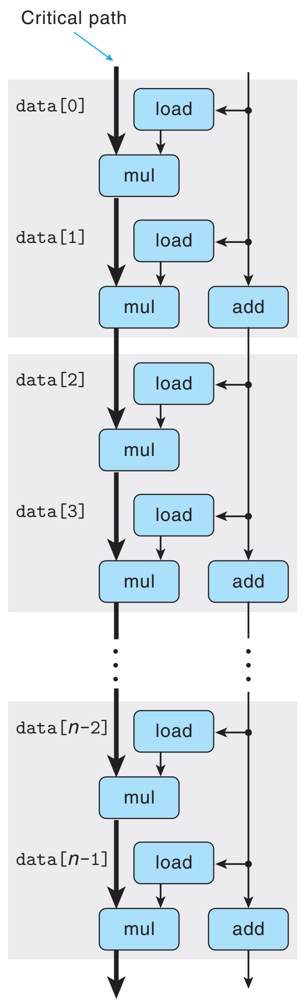


Figure 5.20
Data-flow representation of `combine5` operating on a vector of length n. Even though the loop has been unrolled by a factor of 2, there are still n mul operations along the critical path.

shown in Figure 5.20. We see here that there is still a critical path of n mul operations in this graph—there are half as many iterations, but each iteration has two multiplication operations in sequence. Since the critical path was the limiting factor for the performance of the code without loop unrolling, it remains so with $k \times 1$ loop unrolling.

Aside Getting the compiler to unroll loops

Loop unrolling can easily be performed by a compiler. Many compilers do this as part of their collection of optimizations. GCC will perform some forms of loop unrolling when invoked with optimization level 3 or higher.

5.9 Enhancing Parallelism

At this point, our functions have hit the bounds imposed by the latencies of the arithmetic units. As we have noted, however, the functional units performing addition and multiplication are all fully pipelined, meaning that they can start new operations every clock cycle, and some of the operations can be performed by multiple functional units. The hardware has the potential to perform multiplications and additions at a much higher rate, but our code cannot take advantage of this capability, even with loop unrolling, since we are accumulating the value as a single variable `acc`. We cannot compute a new value for `acc` until the preceding computation has completed. Even though the functional unit computing a new value for `acc` can start a new operation every clock cycle, it will only start one every L cycles, where L is the latency of the combining operation. We will now investigate ways to break this sequential dependency and get performance better than the latency bound.

5.9.1 Multiple Accumulators

For a combining operation that is associative and commutative, such as integer addition or multiplication, we can improve performance by splitting the set of combining operations into two or more parts and combining the results at the end. For example, let P_n denote the product of elements $a_0, a_1, \ldots, a_{n-1}$:

$$P_n = \prod_{i=0}^{n-1} a_i$$

Assuming n is even, we can also write this as $P_n = PE_n \times PO_n$, where PE_n is the product of the elements with even indices, and PO_n is the product of the elements with odd indices:

$$PE_n = \prod_{i=0}^{n/2-1} a_{2i}$$

$$PO_n = \prod_{i=0}^{n/2-1} a_{2i+1}$$

Figure 5.21 shows code that uses this method. It uses both two-way loop unrolling, to combine more elements per iteration, and two-way parallelism, accumulating elements with even indices in variable `acc0` and elements with odd indices in variable `acc1`. We therefore refer to this as "2×2 loop unrolling." As before, we include a second loop to accumulate any remaining array elements for the case where the vector length is not a multiple of 2. We then apply the combining operation to `acc0` and `acc1` to compute the final result.

Comparing loop unrolling alone to loop unrolling with two-way parallelism, we obtain the following performance: