```c
/* Compute prefix sum of vector a */
void psum1(float a[], float p[], long n)
{
    long i;
    p[0] = a[0];
    for (i = 1; i < n; i++)
        p[i] = p[i-1] + a[i];
}

void psum2(float a[], float p[], long n)
{
    long i;
    p[0] = a[0];
    for (i = 1; i < n-1; i+=2) {
        float mid_val = p[i-1] + a[i];
        p[i]    = mid_val;
        p[i+1]  = mid_val + a[i+1];
    }
    /* For even n, finish remaining element */
    if (i < n)
        p[i] = p[i-1] + a[i];
}
```

Practice Problem 5.12 (solution page 613)

Rewrite the code for `psum1` (Figure 5.1) so that it does not need to repeatedly retrieve the value of `p[i]` from memory. You do not need to use loop unrolling. We measured the resulting code to have a CPE of 3.00, limited by the latency of floating-point addition.

Solution to Problem 5.12 (page 597)

Here is a revised version of the function:

```
void psum1a(float a[], float p[], long n)
{
    long i;
    /* last_val holds p[i-1]; val holds p[i] */
    float last_val, val;
    last_val = p[0] = a[0];
    for (i = 1; i < n; i++) {
        val  = last_val + a[i];
        p[i] = val;
        last_val = val;
    }
}
```

We introduce a local variable `last_val`. At the start of iteration `i`, it holds the value of `p[i-1]`. We then compute `val` to be the value of `p[i]` and to be the new value for `last_val`.

This version compiles to the following assembly code:

```
Inner loop of psum1a
a in %rdi, i in %rax, cnt in %rdx, last_val in %xmm0
.L16:                                      loop:
  vaddss   (%rdi,%rax,4), %xmm0, %xmm0       last_val = val = last_val + a[i]
  vmovss   %xmm0, (%rsi,%rax,4)              Store val in p[i]
  addq     $1, %rax                          Increment i
  cmpq     %rdx, %rax                        Compare i:cnt
  jne      .L16                              If !=, goto loop
```

This code holds `last_val` in `%xmm0`, avoiding the need to read `p[i-1]` from memory and thus eliminating the write/read dependency seen in `psum1`.

5.19 ◆◆◆

In Problem 5.12, we were able to reduce the CPE for the prefix-sum computation to 3.00, limited by the latency of floating-point addition on this machine. Simple loop unrolling does not improve things.

Using a combination of loop unrolling and reassociation, write code for a prefix sum that achieves a CPE less than the latency of floating-point addition on your machine. Doing this requires actually increasing the number of additions performed. For example, our version with two-way unrolling requires three additions per iteration, while our version with four-way unrolling requires five. Our best implementation achieves a CPE of 1.67 on our reference machine.

Determine how the throughput and latency limits of your machine limit the minimum CPE you can achieve for the prefix-sum operation.