Homework 4

Problem 1:

Consider the following functions:

```
int min( int x, int y ) { return x < y ? x : y; }
int max( int x, int y ) { return x > y ? x : y; }
void incre( int *p, int v ) { *p += v; }
int square( int x ) { return x*x; }
```

The following three code fragments call these functions:

A.
```
int i = min( x, y );
while( i < max( x, y ) )
{
    t += square(i);
    incre(i, 1);
}
```

B.
```
int i = max( x, y );
while( i >= 0 )
{
    t += square(i);
    incre(i, -2);
}
```

C.
```
for( i = min( x, y ); i < max( x, y ); incre(i,1) )
    i += square(i);
```

Assume x equals 1 and y equals 9. Fill in the following table indicating the number of times each of the four functions is called in code feagment A-C:

Code	min	max	incre	square
A				
B				
C				

Problem 2:

The assembly code generated for the compiled loop of *combine3* is shown below:

```
      combine3: data_t = float, OP = *
      i in %rdx, data in %rax, dest in %rbp
1     .L498:                                  loop:
2        movss     (%rbp), %xmm0               Read product from dest
3        mulss     (%rax, %rdx, 4), %xmm0      Multiply product by data[i]
4        movss     %xmm0, (%rbp)               Store product at dest
5        addq      $1, %rdx                    Increment i
6        cmpq%rdx, %r12                        Compare i: limit
7        jg        .L498                       If >, goto loop
```

Illustrate the code above with data-flow graph like figure 5.14(a) or (b) in textbook. You can use “store” to identify operation in line 4.

Problem 3:

Rewrite the code for prefixSum 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.

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

Problem 4:

Suppose we wish to write a function to evaluate a polynomial, where a polynomial of degree n is defined to have a set of coefficients a_0, a_1 ,a_2, ...,a_n. For a value x, we evaluate the polynomial by computing

$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

This evaluation can be implemented by the following function, having as arguments an array of coefficients a, a value x, and the polynomial degree (the value n). In this function, we compute both the successive terms of the equation and the successive powers of x within a single loop:

```
1       double poly(double a[], double x, int degree)
2       {
3           long int i;
4           double result = a[0];
5           double xpwr = x; /* Equals x^i at start of loop */
6           for (i = 1; i <= degree; i++)
7           {
8               result += a[i] * xpwr;
9               xpwr = x * xpwr;
10          }
11          return result
12      }
```

On our reference machine, with arithmetic operations having the latencies shown in Figure 5.12, we measure the CPE for this function to be 5.00. Roughly explain how this CPE arises based on the data dependencies formed between iterations due to the operations implementing lines 8–9 of the function.