Homework 5 Solution

Problem 1

The following problem illustrates the way memory aliasing can cause unexpected program behavior. Consider the following procedure to swap two values:

```
1       /* Swap value x at xp with value y at yp */
2       void swap(int *xp, int *yp)
3       {
4           *xp = *xp + *yp;  /* x + y       */
5           *yp = *xp - *yp;  /* x + y - y = x  */
6           *xp = *xp - *yp;  /* x + y - x = y  */
7       }
```

If this procedure is called with `xp` equal to `yp`, what effect will it have?

The value at xp and yp will be set to zero.

Problem 2

Consider the following functions:

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

The following three code fragments call these functions:

A.
```
for (i = min(x, y); i < max(x, y); incr(&i, 1))
    t += square(i);
```

B.
```
for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
    t += square(i);
```

C.
```
int low = min(x, y); int high = max(x, y);
for (i = low; i < high; incr(&i, 1))
    t += square(i);
```

Assume `x` equals **1** and `y` equals **10**. Fill in the following table indicating the number of times each of the four functions is called in code fragments A – C:

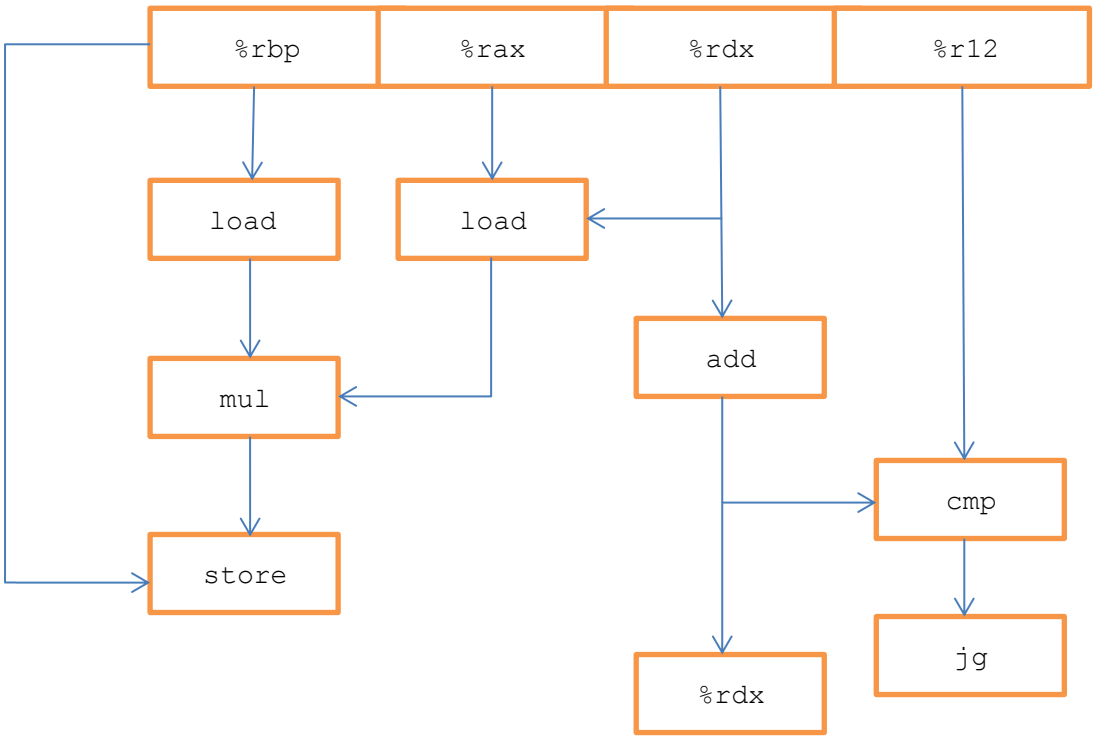
Code	`min`	`max`	`incr`	`square`
A.	1	10	9	9
B.	10	1	9	9
C.	1	1	9	9

Problem 3

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 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

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

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             result += a[i] * xpwr;
8             xpwr = x * xpwr;
9         }
10        return result
11    }
```

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

B. 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 7–8 of the function.

A. The function performs $2n$ multiplications and n additions.

B. We can see that the performance limiting computation here is the repeated computation of the expression

```
xpwr = x * xpwr
```

This requires a double-precision, floating-point multiplication (5 clock cycles), and the computation for one iteration cannot begin until the one for the previous iteration has completed. The updating of result only requires a floating-point addition (3 clock cycles) between successive iterations.