Problem 1: Pipeline Performance

The following program will calculate %rdx = |*%rax| + |*%rbx| + |*%rcx|

```
f:
        irmovq      $0, %rdx
        mrmovq      (%rax), %rax
        andq        %rax, %rax
        jle         sub_rax
        addq        %rax, %rdx
        jmp         rax_done
sub_rax:
        subq        %rax, %rdx
rax_done:

        mrmovq      (%rbx), %rbx
        andq        %rbx, %rbx
        jle         sub_rbx
        addq        %rbx, %rdx
        jmp         rbx_done
sub_rbx:
        subq        %rbx, %rdx
```

```
rbx_done:

        mrmovq      (%rcx), %rcx
        andq        %rcx, %rcx
        jle         sub_rcx
        addq        %rcx, %rdx
        jmp         rcx_done
sub_rcx:
        subq        %rcx, %rdx
rcx_done:
```

We run this program on the following three pipeline implementation:

1. No prediction
2. Always taken prediction
3. Always not-taken prediction

Assume *%rax=1, *%rbx=2, *%rcx=-1, please show the CPI of the three implementations.

$$\text{A}: 1+\frac{1+2+1+2+1+2}{15}=\frac{24}{15}=1.6$$

$$\text{B}: 1+\frac{1+2+1+2+1}{15}=\frac{22}{15}=1.47$$

$$\text{C}: 1+\frac{1+1+1+2}{15}=\frac{20}{15}=1.33$$

Problem 2: Optimization

In US presidential election, each state votes separately. state_result records how many people vote for Trump and Clinton in a state. According to a state_result array ra, function stat computes the total votes among all states, and the winner and the gap between two candidates for each state.

```
typedef struct {
  int trump;
  int clinton;
  int winner;   // 0 for Trump, 1 for Clinton
  int gap;
} state_result;
```

```
7.
8.  typedef struct {
9.    int length;
10.   state_result *data;
11. } rst;
12.
13. int get_length(rst *ra) { return ra->length; }
14. int get_t(rst *ra, int i) {return (ra->data)[i].trump;}
15. int get_c(rst *ra, int i) {return (ra->data)[i].clinton;}
16.
17. void stat(rst *ra, int *total) {
18.   for (int i = 0; i < get_length(ra); i++)
19.     *total = *total + get_t(ra, i) + get_c(ra, i);
20.
21.   state_result *states = ra->data;
22.   for (int i = 0; i < get_length(ra); i++)
23.     if (states[i].trump > states[i].clinton) {
24.       states[i].winner = 0;
25.       states[i].gap = states[i].trump - states[i].clinton;
26.     } else {
27.       // assume Clinton wins if she gets equal or more
28.       states[i].winner = 1;
29.       states[i].gap = states[i].clinton - states[i].trump;
30.     }
31. }
32.
33. int total_trump(state_result *r, int len) {
34.   if (len <= 0) return 0;
35.   return r->trump + total_trump(r + 1, len - 1);
36. }
```

Note: your optimizations cannot change the functionality of code above.

1. Please rewrite the loop in line 18-19 with what you have learned in the class. Comment briefly on the optimization.

```
int tmp1 = 0, tmp2 = 0, i;
// reduce loop overhead
int len = get_length(ra);
// reduce function call
state_result *sa = ra->data;
for (i = 0; i < len - 1; i += 2) {
  // two way loop unrolling + reassociation
  tmp1 = tmp1 + (sa[i].trump + sa[i].clinton);
  // two way loop unrolling + two accumulators
  tmp2 = tmp2 + (sa[i+1].trump + sa[i+1].clinton);
}
for (; i < len; i++)
  tmp1 += sa[i].trump + sa[i].clinton;
// reduce memory access
*total = tmp1 + tmp2;
```

2. For an array of length L, the recursion function total_trump will recur L times. Please rewrite the function body in line 34-35 to reduce the depth to about L/2, with an optimization similar to loop unrolling. But you cannot use loop in your solution. NOTE that you can show how to invoke your optimized function if it helps simplify your solution.

```
// optimized code
int total_trump(state_result *r, int len) {
    if (len <= 0) return 0;
return r[0].trump + r[1].trump + total_trump(r + 2, len - 2);
}

// invocation example
extern int length;
extern state_result *array;
int res;
if (length % 2 == 0)
  res = total_trump(array, length);
else {
  res = array[0].trump;
  res += total_trump(array + 1, length - 1);
}
```

Problem 3: Optimization

```
typedef struct {
    float *data; /* points to an array */
    long capacity; /* the maximum length of the array */
    long length; /* number of elements in the array */
} array_t;

long get_length (array_t *arr) {return arr->length;}

long get_capacity (array_t *arr) {return arr->capacity;}

void copy_array(array_t *dst, array_t *src) {
    for (long i = 0; i < get_length(src); i++) {
        if (i >= get_capacity(dst))
            break;
        dst->data[i] = src->data[i];
    }
    dst->length = min(get_length(src), get_capacity(dst));
}
void sum_array(float *arr, long n, long *sum) {
    float ans = 0;
    for (long i = 0; (i+1) < n; i += 2)
        ans = ans + (arr[i] + arr[i + 1]);
    if (i < n)
        ans += arr[i];
    *sum = ans;
}
```

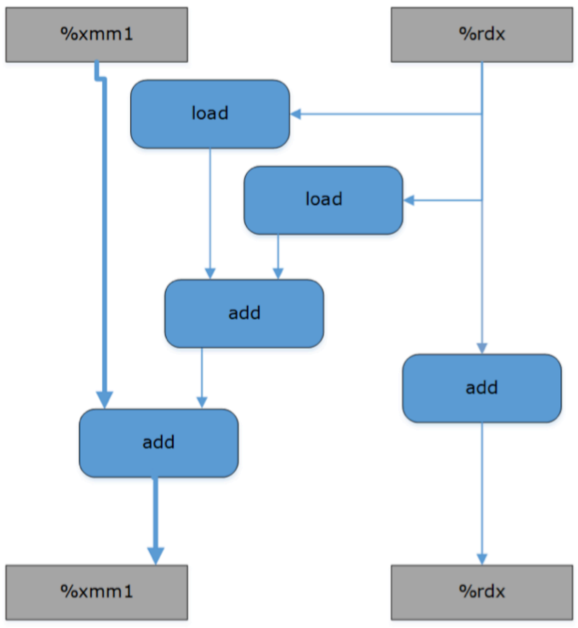
```
.Loop:
  movss (%rax, %rdx, 8), %xmm0
  addss 8(%rax, %rdx, 8), %xmm0
  addss %xmm0, %xmm1
  addq $2, %rdx
  cmpq $rdx, %rbp
  jg .Loop
```

1. Please rewrite the function copy_array with what you have learned in the class. Comment briefly on the optimization. NOTE: your optimizations cannot change the functionality of code above.

```
1.  void copy_array(array_t *dst, array_t *src) {
2.    // Eliminating loop inefficiency
3.    long len = get_length(src);
4.    // reduce function calls
5.    long cap = get_capacity(dst);
6.    // remove unneeded memory references
7.    float *src_data = src->data;
8.    float *dst_data = dst->data;
9.    long niters = (len < cap) ? len : cap;
10.   // loop unrolling and enhancing parallelism
11.   for (long i = 0; i + 1 < niters; i += 2) {
12.     dst_data[i] = src_data[i];
13.     dst_data[i + 1] = src_data[i + 1];
14.   }
```

```
15.   if (i < niters)
16.     dst_data[i] = src_data[i];
17.   dst->length = niters;
18. }
```

2. The translation of code in line 18-19 is presented in line 24-30. Please abstract the operations as a data-flow graph and draw the graph. Please also mark the critical path(s) in the graph.



3. The code in line 19 is modified as the following code in the table. After the modification, the CPE measurement increases from X to 2X. Please point out why the CPE measurement increases.

```
ans = (ans + arr[i]) + arr[i + 1];
```

We have two **load** and two **add** operations. **After the modification,** both **add** operations form a dependency chain between loop registers. While **before the modification**, only one of the **add** operations forms a data-dependency chain between loop registers. The first addition within each iteration can be performed without waiting for the accumulated value from the previous iteration. Thus, we reduce the minimum possible CPE by a factor of around 2.