Exercise 8

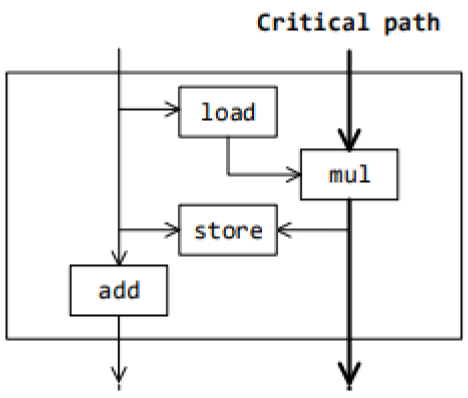
Problem 1

```
1   .L2:
2   mulsd a(,%rax,8), %xmm0  // a is the address of an array.
3   movsd %xmm0, a(,%rax,8)
4   addq  $1, %rax
5   cmpq  %rdx, %rax
6   jl .L2
```

Assume that there is only ONE double-precision multiplication unit in the processor. All other CPU resources are UNLIMITED. The latency and issue time of the units are given in the below table.

operation	Integer		Double-precision	
	latency	Issue	latency	issue
Addition	1	1	2	1
Multiplication	3	1	5	1
Load/Store	3	1	3	1

1. Draw the data flow graph and mark the critical path.



2. Please calculate the CPE on current CPU. If we have UNLIMITED number of multiplication units, how much is CPE?

5 5. There are dependencies between xmm0 so multiple units cannot accelerate it.

3. Now we swap the instruction at line 3 and line 4, please give out the CPE with original LIMITED number of multiplication units and explain your answer.

11. Now the critical path is load-mul-store.

Problem 2

Usually we use the following representation of polynomials in math:

$$f(x) = a_n * x^n + a_{n-1} * x^{n-1} + \cdots + a_1 * x + a_0$$

But this form is not suitable for computation in computer. Instead, we use the following representation:

$$f(x) = a_0 + x(a_1 + x\big(a_2 + \cdots + x(a_{n-1} + xa_n)\big))$$

1. Please explain why the latter representation is faster. (**HINT:** Consider the number of computation primitive used)

Calculate *xi* is expensive. Even we calculate it iteratively, there is still one more multiplication to get *ai∗xi* in each iteration.

2. We have the following code to evaluate the polynomial on a given x, but it's very slow. Please optimize it using machine-independent optimization.

```
struct coefficient {
    int a;
    struct coefficient *next;
}

// the coefficients are given
in reverse linked list
// e.g. alist->a = $a_n$
// alist->next->a = $a_{n-1}$
// ...

int get_n(struct coefficient
*alist) {
    int n = 0;
    while (alist) {
        n++;
        alist = alist->next;
    }
    return n;
}
```

```
int get_ai(struct
coefficient *alist, int i) {
  int current = get_n(alist);
  while (current != i) {
    alist = alist->next;
    current--;
  }
  return alist->a;
}

int calculate(struct
coefficient *alist, int x) {
    int result = get_ai(alist,
n);
    for (int i = get_n(alist)
- 1; i >= 0; i--)
       result = result * x +
       get_ai(alist, i);
    return result;
}
```

```
int calculate(struct coefficient *alist, int x) {
    int result = alist->a;
    alist = alist->next;
    while (alist) {
        result = result * x + alist->a;
        alist = alist->next;
    }
    return result;
}
```

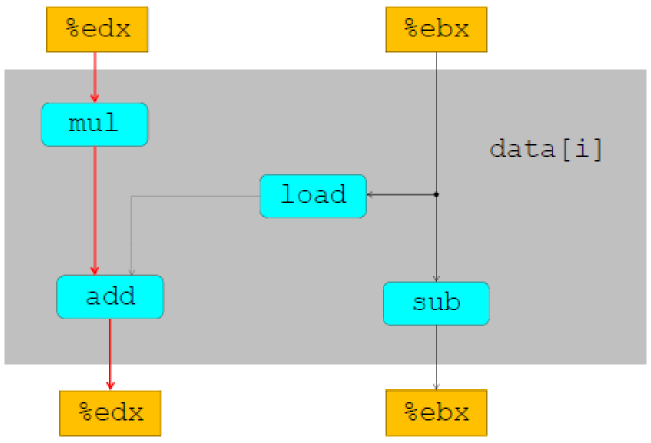
3. Here is the array version of the function:

```
int calculate(int *a, int n, int x) {
    int result = a[n];
    for (int i = n - 1; i >= 0; i--)
        result = result * x + a[i];
}
```

And the loop code looks like this:

```
loop:
testl %ebx, %ebx
jge done
imull %r13d, %edx
movl (%r14, %ebx, 4), %eax
addl %eax, %edx
subl $1, %ebx
jmp loop
done:
…
```

a. Draw the data-flow graph and show the critical path.



b. Can you use multiple accumulators to optimize this program? How or Why?

No. Each iteration must use the result generated in the last iteration.

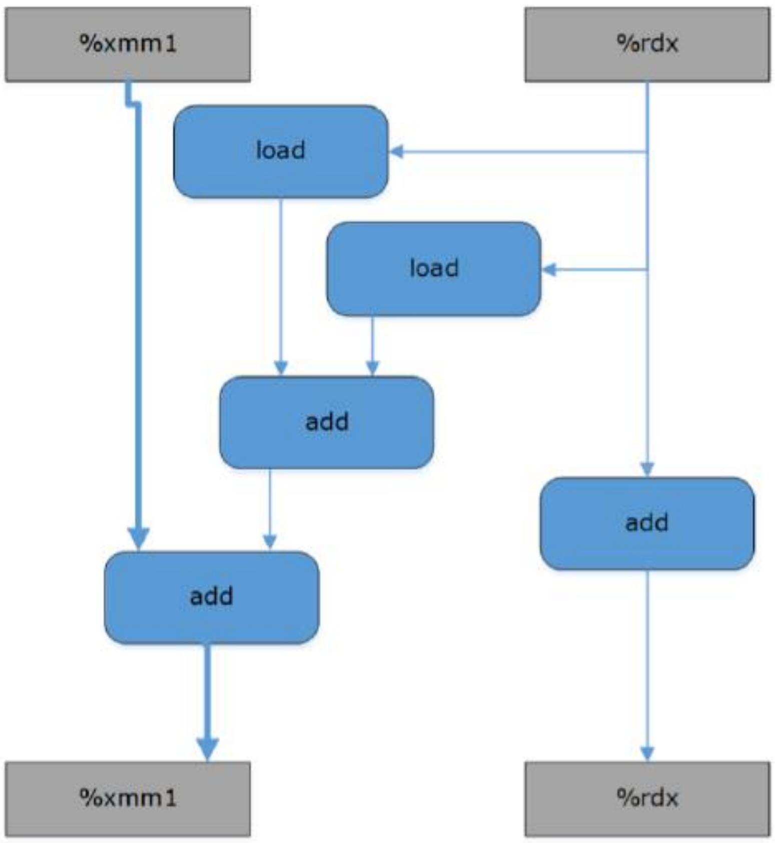
Problem 3

Following is the code of a loop and the assembly code of the loop. This loop wants to calculate the sum of a float array stored in **`arr`**.

```
float* arr;
 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];
```

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

1. Draw the data flow graph and mark the critical path(s).



2. What’s the CPE of this loop? Why?

1.5. The loop calculate 2 elements in each iteration. The critical path takes a float addition and its latency is 3 cycles.

3. Now we modify the statement in the loop to the following one. 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.