Problem 1: FP (9 points)

1. [1] -6

   [2] x 1111 yyyyyy (x is 0 or 1, yyyyyy is none zero)

   [3] 0 0000 111111

2. 1 1001 011100

3. 1 0100 111000

Problem 2: HCL (7 points)

1. bool nand = !a || !b;

2. Sol1: bool out = (A && B && C) || (!A && !B && !C)

   Sol2:
```
bool out = [A && B && C    : 1;
            A ^ B          : 0;
            A ^ C          : 0;
            B ^ C          : 0;
            1              : 1;
           ];
```

*Or other solutions satisfied the truth table.

Problem 3: Y86 (17 points)

1. [1] 30f50002000000000000 [2] irmovq $3, %rsi

   [3] xorq %rax, %rax [4] mrmovq (%rdx), %rcx

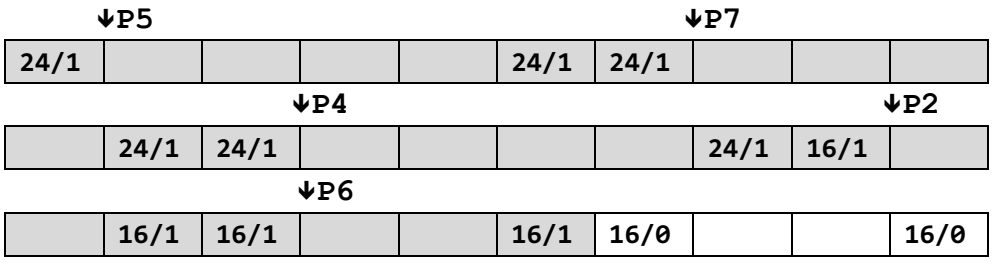
   [5] 6260 [6] .pos 0x200

   [7] 0x320

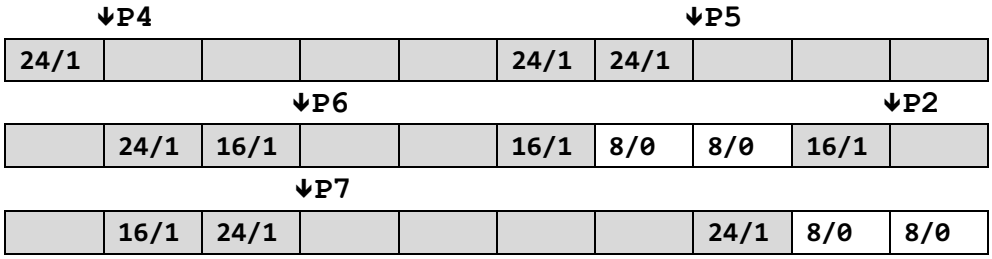
2. %rax = 0x3

Problem 4: Memory Allocation (16 points)

1. (Whether marking out pointers or not are both correct)



2.(Whether marking out pointers or not are both correct)

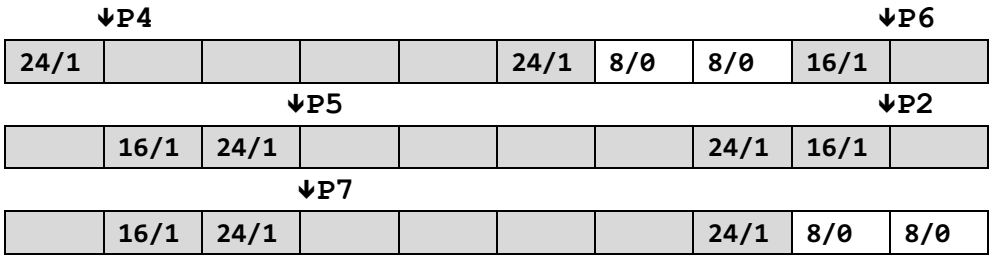


3. Consider the following operation sequence:

```
p = malloc(5);
free(p);
p = malloc(5);
free(p);
p = malloc(5);
...
```

Assume the allocated block of p is adjacent to another free block. In such situation, every time we free pointer p will cause a coalescing operation, which will dramatically influence the performance.

4. (Whether marking out pointers or not are both correct)



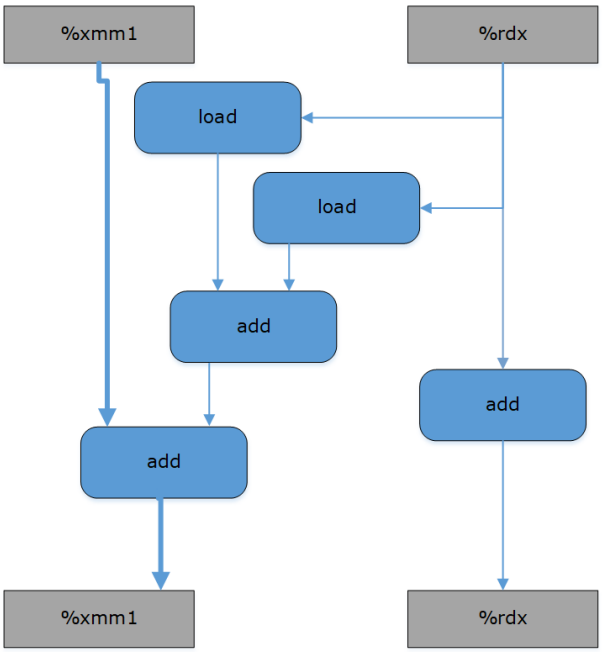
Problem 5: Optimization (16 points)

1

```
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.



3.

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.

Problem 6 : Processor (35 points)

1. Load/use hazard: 6
   misprediction of jle: 6
   misprediction of jg: 2
2. Loop: mrmovq (%rdi), %r10
   rmmovq %r10, (%rsi)
   andq %r10, %r10
   cskiple
   iaddq $1, %rax
   iaddq $8, %rdi
   iaddq $8, %rsi
   iaddq $-1, %rdx
   jg Loop
   Load/use hazard: 6
   misprediction of cskiple: 3
   misprediction of jg: 2

3. [1] E_icode == ICSKIP && e_Cnd

   [2] normal [3] normal [4] bubble

4. solution1: [1] normal [2] normal [3] bubble [4] normal

   solution2: [1] stall [2] bubble [3] bubble [4] normal

5. No. Because the third instruction is not loaded when cskip3xx is in E stage/Because you can't insert bubble in F

   Solution 1:

   Delay the handle to M stage, so we could insert 3 bubbles at once. Be careful with CC!!

```
D_bubble = ... || (M_icode == ICSKIP3XX && M_Cnd)
E_bubble = ... || (M_icode == ICSKIP3XX && M_Cnd)
M_bubble = ... || (M_icode == ICSKIP3XX && M_Cnd)
set_cc = ... && !(M_icode == ICSKIP3XX && M_Cnd)
```

   Solution 2:

   Insert 2 bubbles when detected in E, and insert one more in M.

```
D_bubble = ... || (E_icode == ICSKIP3XX && e_Cnd) ||
           (M_icode == ICSKIP3XX && M_Cnd)
E_bubble = ... || (E_icode == ICSKIP3XX && e_Cnd)
```

6. Pros:
   Instruction is shorter
   No label is needed

   Cons:
   One more bubble