Problem 1: FP (9 points)

1. [1] 7

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

   [3] 1 0000 111111

2. 1 1001 100100

3. 0 0101 011000

Problem 2: HCL (7 points)

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

2. Sol1: bool out = A ^ B || A ^ C || B ^ C

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

*Or other solutions satisfied the truth table.

Problem 3: Y86 (17 points)

1. [1] 30f40002000000000000   [2] 6300

   [3] rrmovq %rdi, %rdx   [4] mrmovq (%rdx), %rdx

   [5] 6222   [6] .pos 0x300

   [7] 0x310

2. %rax = 0x2

Problem 4: Memory Allocation (16 points)

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

↓P5 ↓P7

24/1					24/1	24/1			

↓P4 ↓P2

	24/1	24/1					24/1	16/1	

↓P6

	16/1	16/1			16/1	16/0			16/0

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

	↓P4						↓P5		
24/1					24/1	24/1			
			↓P6						↓P2
	24/1	16/1			16/1	8/0	8/0	16/1	
			↓P7						
	16/1	24/1					24/1	8/0	8/0

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)

	↓P4								↓P6
24/1					24/1	8/0	8/0	16/1	
			↓P5						↓P2
	16/1	24/1					24/1	16/1	
			↓P7						
	16/1	24/1					24/1	8/0	8/0

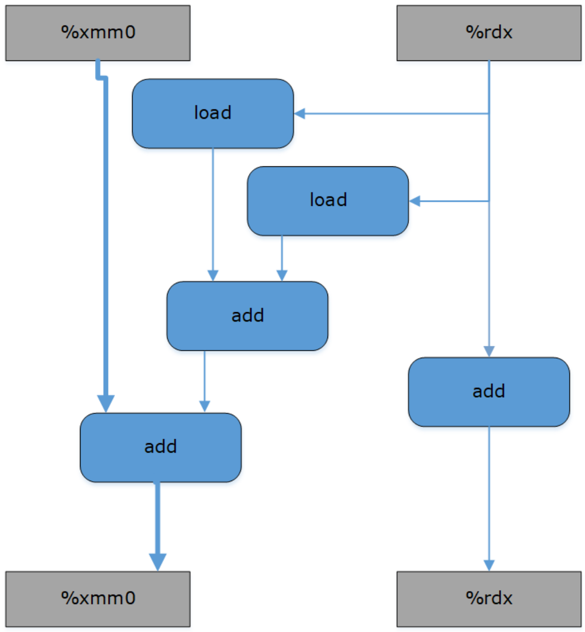
Problem 5: Optimization (16 points)

1

```
void copy_array(array_t *new, array_t *old) {
  // Eliminating loop inefficiency
  long len = get_length(old);
  // reduce function calls
  long cap = get_capacity(new);
  // remove unneeded memory references
  float *old_data = old->data;
  float *new_data = new->data;
  long niters = (len < cap) ? len : cap;
  // loop unrolling and enhancing parallelism
  for (long i = 0; i + 1 < niters; i += 2) {
    new_data[i] = old_data[i];
    new_data[i + 1] = old_data[i + 1];
  }
  if (i < niters)
```

```
    new_data[i] = old_data[i];
  new->length = niters;
}
```

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

3. [1] E_icode == ICKILLXX && 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 ckill3xx 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 == ICKILL3XX && M_Cnd)
E_bubble = ... || (M_icode == ICKILL3XX && M_Cnd)
M_bubble = ... || (M_icode == ICKILL3XX && M_Cnd)
set_cc = ... && !(M_icode == ICKILL3XX && M_Cnd)
```

   Solution 2:

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

```
D_bubble = ... || (E_icode == ICKILL3XX && e_Cnd) ||
             (M_icode == ICKILL3XX && M_Cnd)
E_bubble = ... || (E_icode == ICKILL3XX && e_Cnd)
```

6. Pros:
   - Instruction is shorter
   - No label is needed

   Cons:
   - One more bubble