上 海 交 通 大 学 试 卷（A 卷）

（ 2020 至 2021 学年 第 2 学期 ）

班级号_____ 学号_____ 姓名 _____

课程名称 计算机系统基础（组成） 成绩 _____

Problem 1

1. [1] [2]

   [3] [4]

   [5] [6]

2. [1] [2]

   [3] [4]

   [5] [6]

Problem 2

1.

2.

Problem 3

1.

2. [1] [2]

   [3]

我承诺，我将严格遵守考试纪律。

题号	1	2	3	4	5				
得分									
批阅人(流水阅卷教师签名处)									

3. [1] [2]

[3] [4]

Problem 4

1.

2.

3.1

3.2

Problem 5

1. [1] [2]
2. [3] [4]
   [5]

3.

4. [6] [7]
   [8]

5.

Problem 1: Y86-Assembly(24 Points)

The following assembly code is used to get the sum of the array.

0x0000:	.pos 0
0x0000:	init:
0x0000: ____[1]____	irmovq stack, %rsp
0x000a: 2045	rrmovq %rsp, %rbp
0x000c: 801600000000000000	call main
0x0015: 00	halt
0x0016:	main:
0x0016: a05f	pushq %rbp
0x0018: 2045	rrmovq %rsp, %rbp
0x001a: 30f77800000000000000	irmovq Array, %rdi
0x0024: 30f60300000000000000	irmovq $0x3, %rsi
0x002e: 30f20000000000000000	irmovq $0, %rdx
0x0038:	loop:
0x0038: 5007000000000000000	____[2]____
0x0042: ____[3]____	addq %rax, %rdx
0x0044: 30f00800000000000000	irmovq $0x8, %rax
0x004e: 6007	addq %rax, %rdi
0x0050: 30f00100000000000000	irmovq $0x1, %rax
0x005a: 6106	subq %rax, %rsi
0x005c:	test:
0x005c: 6266	andq %rsi, %rsi
0x005e: 743800000000000000	jne loop
0x0067: 30f00000000000000000	irmovq $0x0, %rax
0x0071: 2054	rrmovq %rbp, %rsp
0x0073: b05f	popq %rbp
0x0075: 90	ret
	.align 8
____[4]____: 1111000000000000	Array:
0x0080: ____[5]____	.quad 0x1111
0x0088: 2233000000000000	.quad 0x133
0x0300:	.quad 0x3322
0x0300:	____[6]____
	stack:

1. Please fill in the blanks above with Y86 binary and assembly code(2'*6=12')

2. Please fill the value of registers listed in following blank after the program **HALT**(2'*6=12').

%rsi	**[1]**
%rdx	**[2]**
%rbp	**[3]**
CC.ZF	**[4]**
CC.SF	**[5]**
CC.OF	**[6]**

Problem 2 Y86-HCL(8 Points)

Please write down the HCL expressions for the following signals. You can use Boolean expressions or case expressions.
Example: Show if the two input signals a and b are equal:

```
bool eq = (a&&b) || (!a && !b);
```

1. The HCL expression for a three-way signal called **NAE**, which returns true if and only if all three inputs are not equal. The three input signals (**a**, **b,** and **c**) are one-bit wise. For example, {true, true, false} returns true; {false, false, false} return false(4').

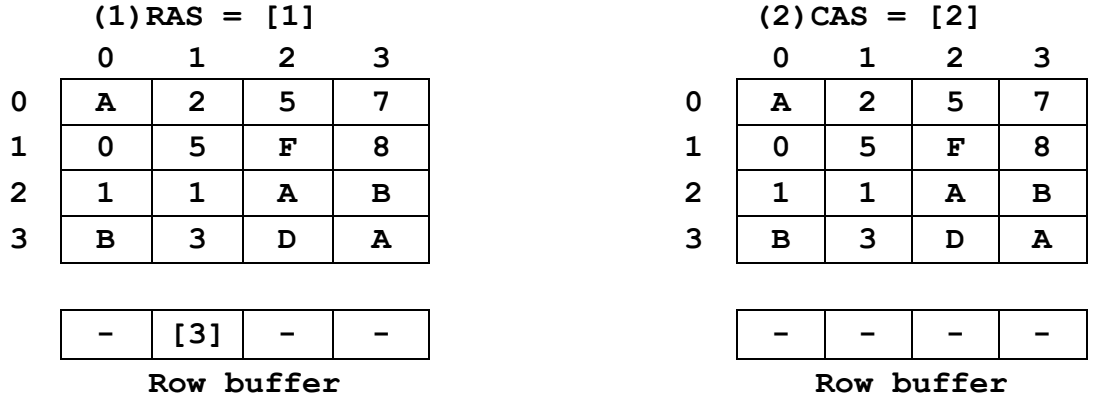
2. The HCL expression for the **range** (极差) of three inputs **a**, **b** and **c**. For example, in {4, 9, 6} the lowest value is 4, and the highest is 9, thus the range is 5(4').

Problem 3 Storage Hierarchy (18 Points)

From Figure 6.1 in CSAPP book, we have learned three typical storage devices: **DRAM, SSD** and **Hard disk**, in which DRAM is volatile, and SSD and Hard disk are non-volatile (persistent).

1. We know that **random write** is slower than **sequential write** in both **Hard disk** and **SSD**. Are they due to the same reason? If yes, explain the reason. If no, explain the specific two reasons(4').

2. Suppose there is a 16x8 DRAM as figure shows and memory controller sends a request to DRAM to get one byte value.(3'*2=6')



Suppose we finally get value **0xF**, please fill the blank.

3. Recently with appearance of new storage technologies such as PCM(Phase-change memory) and 3D XPoint, a new kind of storage device called **PM**(**P**ersistent **M**emory) is invented.

It has three main features: **1) Persistence** (持久性): data in it will not be lost after power failure ; **2) Byte-addressability**: it can be accessed like DRAM instead of block devices (SSD, HDD) which can only be r/w at block granularity; **3)** Comparable **performance** to DRAM: it has less than 10x latency and 1/6-1/3 bandwidth compared to DRAM.

Please choose the most suitable storage device for each application (use **DRAM**, **PM**, **SSD** and **Hard disk** to fill the blank). **Note:** each device occurs only once.(4'*2=8')

Fronted cache for backend database	**[1]**
Video player	**[2]**
Long-loading games	**[3]**
Persistent data structures	**[4]**

Problem 4 Optimization (20 points)

```
1.  typedef struct {
2.    double* item_prices; // the price of each item
3.    double total;        // the total price of the order
4.  } order_t;
5.  void populate_order(order_t* order, double* prices) {
6.    double sum = 0;
7.    for (int i = 0; i < number_of_items(order); i++) {
8.      if (order->item_prices == NULL)
9.        order->item_prices = new double[number_of_items(order)];
10.     order->item_price[i] = prices[i];
11.     sum += order->item_price[i];
12.   }
13.   order->total = sum;
14. }
15. void do_discount(int* discount1, int* discount2, int n) {
16.   int penalty = 0;
17.   for (int i = 0; i < n; i++) {
18.     *discount1 = penalty;
19.     penalty = i + (*discount1);
20.   }
21. }
```

```
22.   .Loop:
23.     movq %rax, (%rdi)  // write penalty to discount1
24.     movq (%rdi), rax   // write to discount1
25.     addq %rcx, %rax    // add i
26.     addq $1, %rcx      // if i < n, loop again
27.     cmpq $rcx, %rbp
28.     jg .Loop
```

1. Please rewrite the function **`populate_order`** with the **optimizations** you have learned in the class, including a **2 x 2** loop unrolling. Comment briefly on each optimization. NOTE: your optimizations cannot change the functionality of code above(10').

2. The translation of code in line 17-19 is presented in line 22-28. Please abstract the operations as a data-flow graph and draw the graph. Please also mark the critical path(s) in the graph(5').

3. Suppose we add one line **`*discount2 = penalty`** right after 18, and change line 19 to **`penalty = (*discount1)+(*discount2)`**. Assume that all store units share only **ONE** available entry in the store buffer (the capacity of store buffer is one, and shared across all store units), and other resource are UNLIMITED. As a consequence of the single store buffer entry, we further assume that for any two consequent store instructions, if they both write to the **SAME** address in the store buffer, then the later store instruction only needs a latency of **2** cycles to complete. The following table shows the latency and issue time of different instructions in our CPU(5').

Operation	Integer	
	Latency	Issue
Addition	1	1
Load/Store	3	1
Store (Same address)	2	1

3.1 Given **`discount1 == discount2`**, please estimate the CPE.

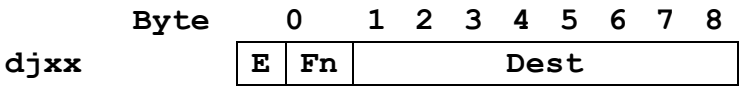
3.2 Given **`discount1 != discount2`**, please estimate the CPE.

Problem 5 Pipeline (30 points)

Suppose Jack adds a new instruction, delayed jump, **`djxx`**, to the original Y86-64 instruction set. Jack implements **`djxx`** based on the final pipeline implementation from Figure 4.52 in CSAPP book. This instruction **conditionally "delays" jumping to the target until the next instruction is executed**. Thus, the instruction after **`djxx`** is executed whether or not the branch is taken. The next instruction is referred to be in the **delayed slot (Instr$_d$)**. For example, consider the following instruction sequence:

```
        xorq %rax, %rax        // xor
        dje target             // delayed jump
        iaddq $0xA, %rbx       // iadd1
        iaddq $0xB, %rcx       // iadd2
target:
        iaddq $0xC, %rdx       // iadd3
```

The execution sequence will appear as 1. **`xor`** 2. **`iadd1`** 3. jump to **`target`** 4. **`iadd3`**. The **$Instr_d$** is **`iaddq $0xA, %rbx`** in the **delayed slot**. Suppose **`djxx`** uses **TAKEN** prediction strategies and has the following encoding:



Name	Value (hex)	Meaning
IDJXX	E	Code for djxx instruction

NOTE: ①For Question **1** to **3**, we assume that delayed slots contain **NO** control transfer instructions (**`jxx`**, **`djxx`**, **`call`**, **`ret`**). ②Only **increased** HCL is shown.

1. We need to modify the **PC prediction logic** in the F-stage for **`djxx`**. Fill the blanks to complete the implementation (2'+2'=4').

```
word f_predPC = [
    f_icode in { IJXX, ICALL} : f_valC;
    __[1]__ ==  IDJXX : __[2]__;
    1: f_valP;
]
```

2. Similar to mispredicted **`jxx`**, there will be some hazards due to **`djxx misprediction`**. Please list new detection conditions like Figure 4.64 and new control action like Figure 4.66. (3*2' = 6') Use "--" for no additional control action("normal").

Condition	Trigger				
djxx misprediction	[3]				
	Pipeline register				
	F	D	E	M	W
	--	[4]	[5]	--	--

3. Similar to mispredicted **`jxx`**, when an mispredicted **`djxx`** enters the **M-stage**, we need select the correct PC value using the PC selection logic in the **F-stage**. Jack modifies the HCL below but finds that **`M_valA`** is **NOT** the value expected. Explain why it is wrong (3') and figure out a way to correct (3'). (**Hint**: you are allowed to change the PIPE hardware)

```
word f_pc = [ M_icode == IDJXX && !M_Cnd : M_valA ];
```

4. Now consider that the three control transfer instructions (**`jxx`**, **`call`**, **`ret`**) can occur in the **delayed slots**. We decide that: If both **`djxx`** and **$Instr_d$** are both **TAKEN**, jump to the target of **$Instr_d$**; If one is **TAKEN** and the other is **NOT TAKEN**, jump to the target of the **TAKEN** one.

   NOTE: **`call`** and **`ret`** are always **TAKEN** instructions.
   a) Draw the new combination of hazard for **`djxx`** and **`ret`** like **Figure 4.67.** (3')
   b) Show how to handle it correctly. (2'*3 = 6')

Pipeline register				
F	**D**	**E**	**M**	**W**
[6]	[7]	[8]	--	--

5. Jack further extend **`djxx`** to **`djxx2`**. which will conditionally **delay** jumping to the target until **the next 2** instructions are executed. In lab6 you were asked to implement a **ncopy** program, which will copy and count the number of positive integers. Here is the core loop in the initial version. How can you rewrite this loop using **`djxx2`** to get better **CPE**?(3') Explain the reason.(2')

```
1. Loop:
2.     mrmovq (%rdi), %r10       // %r10 = *src
3.     rmmovq %r10, (%rsi)       // *dst = %r10
4.     andq %r10, %r10
5.     jle Npos                  // if (%r10 > 0)
6.     iaddq $1, %rax            //    %rax++
7. Nops:
8.     iaddq $8, %rdi            // src++
9.     iaddq $8, %rsi            // dst++
10.    iaddq $-1, %rdx           // size--
11.    jg Loop                   // if (size > 0) continue
12.    halt                      // else, halt
```