上 海 交 通 大 学 试 卷（ A 卷）

（ 2018 至 2019 学年 第 1 学期 ）

班级号_______________________ 学号______________ 姓名 ____________

课程名称___计算机系统基础（1）____________________成绩 _____________

Problem 1: HCL (10points)

1.

2.

Problem 2: Y86 (18 points)

1. [1] [2]

   [3] [4]

   [5] [6]

   [7]

2.

Problem 3: Memory Allocation (16 points)

1.

我承诺，我将严格遵守考试纪律。

承诺人：________

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

2.

3.

4.

Problem 4: Optimization (22 points)

1.

2.

3.

4.

Problem 5: Processor (34 points)

1.

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

   [5] [6] [7] [8]

3.

4. a.

   b. [1] [2] [3] [4]

5.

6.

Problem 1: HCL (10 points)

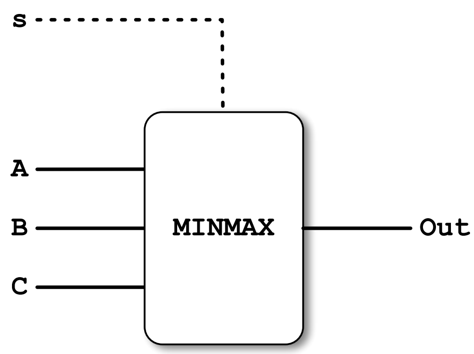
Please write down the HCL expressions for the following signals (HINT: you can refer to the **Section 4.2.2** in the CSAPP book).

EXAMPLE: a signal EQ which shows if two input a and b are equal:
`bool EQ = (a&&b) || (!a && !b);`

1. The HCL expression for a signal **IMPLIES**. "**a IMPLIES b**" for **bool a** and **bool b** means that if **a** is true, then **b** must also be true. The truth table is given below. (4')

a	b	a IMPLIES b
T	T	T
T	F	F
F	T	T
F	F	T

2. The HCL expression for a word-level signal **MINMAX** which takes three word (**word A, word B, word C**) and a Boolean selector (**bool s**) as inputs. When **s** is true, **MINMAX** outputs the minimum value of the given three words. When **s** is false, **MINMAX** outputs the maximum value of these words. The diagram of **MINMAX** is showed below. (6')



Problem 2: Y86 (18 points)

```
0x0000:                            | .pos 0
0x0000:                            | init:
0x0000: 30f50001000000000000       |     irmovq Stack, %rbp
0x000a: 30f40001000000000000       |     irmovq Stack, %rsp
0x0014: 704901000000000000         |     jmp Main
0x001d:                            | .pos 0x100
0x0100:                            | Stack:
                                   |
0x0100:                            | .pos 0x104
0x0104:                            | My_Operation:
0x0104: a05f                       |     pushq %rbp
0x0106: 2045                       |     ____[1]____
0x0108: a03f                       |     pushq %rbx
0x010a: 30f3ffffffffffffffff       |     ____[2]____
0x0114: 50151000000000000000       |     mrmovq 16(%rbp), %rcx
0x011e: 50251800000000000000       |     mrmovq 24(%rbp), %rdx
0x0128: 6300                       |     xorq %rax, %rax
0x012a: ____[3]____                |     andq %rdx, %rdx
0x012c: 714201000000000000         |     jle End
0x0135:                            | Loop:
0x0135: 6010                       |     addq %rcx, %rax
0x0137: 6032                       |     ____[4]____
0x0139: ____[5]____                |     jne Loop
0x0142:                            | End:
0x0142: b03f                       |     popq %rbx
0x0144: 2054                       |     rrmovq %rbp, %rsp
0x0146: b05f                       |     popq %rbp
0x0148: 90                         |     ret
                                   |
0x0149:                            | Main:
0x0149: 30f00300000000000000       |     ____[6]____
0x0153: 30f30400000000000000       |     irmovq $4, %rbx
0x015d: a03f                       |     pushq %rbx
0x015f: a00f                       |     pushq %rax
0x0161: ____[7]____                |     call My_Operation
0x016a: 2054                       |     rrmovq %rbp, %rsp
0x016c: 00                         |     halt
```

1. Please fill in the blanks within above Y86 binary and assembly code. (2’*7=14’)
2. What the instruction “**`0x012a: andq %rdx, %rdx`**” and “**`0x012c: jle End`**” want to protect? (2’) What problems that function **`My_Operation`** may cause without such two instruction? (2’)

Problem 3: Memory Allocation (16 points)

There are some **unallocated memory** and **heap**, as shown below. heap is organized as a sequence of **contiguous** allocated and free blocks. **Allocated** blocks are shaded, and **free** blocks are blank (each block represents 1 word = 4 bytes). **Headers** and **footers** are labeled with the number of bytes and allocated bit. The allocator maintains **double-word** alignment. You are given the execution sequence of memory allocation operations (i.e., **`malloc()`** or **`free()`**) from 1 to 5.

			↓P1						
8/0	8/0	24/1					24/1	16/0	

			↓P2						
	16/0	16/1			16/1	16/0			16/0

	↓P3				↓brk				
16/1			16/1						

```
1.P4 = malloc(5)
2.free(P1)
3.free(P2)
4.P5 = malloc(6)
5.P6 = malloc(7)
```

Please answer the questions below. Assume that **immediate coalescing** strategy and **splitting free blocks** are employed.

1. Assume **first-fit** algorithm is used to find free blocks. Please draw the **final** status of memory and mark with block size in headers and footers after the operation sequence (1~5) is executed (4').

2. Assume **next-fit** algorithm is used to find free blocks. Please draw the **final** status of memory and mark with block size in headers and footers after the operation sequence (1~5) is executed (4').(note: For next-fit algorithm, you can refer to the **Section 9.9.7.** in the CSAPP book)

3. **Instead of** executing operation sequence (1~5), we want to execute an operation 6: **`P7 = malloc(13)`**. Please explain what malloc operation may do (1') and draw the **final** status of memory and mark with block size in headers and footers. (2')

4. Please calculate internal fragmentations caused by operation 1 ,4 and 5(P4,P5,P6) (3'). As we can see, there are many fragmentations in the heap. Please provide at least one optimization to decrease the internal fragmentations. (2') (Hint: you can consider whether the headers and footers are necessary in allocated block)

Problem 4: Optimization (22 points)

```
typedef struct {
   long len;              /* number of element in this array */
   double *data;          /* raw data */
} fp_arr;                 /* floating point array */

long get_len(fp_arr arr) {
   return arr.len;
}

double * ele_at(fp_arr arr, long idx) {
   return &(arr.data[idx]);
}

void do_arr(fp_arr arr1, fp_arr arr2, fp_arr arr3, double *res) {
   for (long i = 0; i < get_len(arr1); i++) {
      *ele_at(arr3, i) = *ele_at(arr1, i) * *ele_at(arr2, i);
      *res += *ele_at(arr1, i) * *ele_at(arr2, i);
   }
}
```

```
void do_arr_v2(double *arr1, double *arr2, double *arr3, long len) {
   for (long i = 0; i < len; i++)
      arr3[i] = arr1[i] * arr2[i];
}
```

```
   movl    $0, %eax
.L3:
   movsd   (%rdi,%rax,8), %xmm0
   mulsd   (%rsi,%rax,8), %xmm0
   movsd   %xmm0, (%rdx,%rax,8)
   addq    $1, %rax
   cmpq    %rax, %rcx
   jne     .L3
```

1. Billy writes a function **do_arr** as shown in above. Please help him improve the performance by rewriting the function **do_arr** with a combination of **at least 4 different optimizations** you have learned in class. Comment briefly on the optimizations. (2'*4=8')
   NOTE: your optimizations cannot change the functionality of **do_arr**.

2. Suppose the optimizations you proposed in the first question can speedup the performance of **do_arr** to **5X** and **do_arr** takes 40% of the total time of some

program **A**. Please compute the speedup of program **A** after the optimization. (**HINT**: you may refer to 1.9.1 Amdahl's Law) (3')

3. Billy decides to simplify the `do_arr` function to `do_arr_v2` (see **line 20-23**). The assembly of `do_arr_v2` is given in **line 24-31**. Please abstract the operations as a data-flow graph. **Drawing rules**:
   a) All registers (except `%xmm0`), operations that appear in line 26-29 and the data flow between them are required to be drawn. (5')
   b) Conditional data dependency when load and store address match should also be considered and drawn in dashed line. (2')

   You may refer to Figure 5.36 (a) in the CSAPP book.

4. Suppose `arr1` and `arr2` are `double` arrays of length **10000**, and their memory range do **NOT** overlap. The latency and issue time characteristics of reference machine operations are provided below. What **CPE** of calling `do_arr_v2(arr1, arr2, arr1+1, 9999)` would you expect? Please explain your answer. (4')

operation	integer		double-precision	
	latency	issue	latency	issue
addition	1	1	3	1
multiplication	3	1	5	1
load/store	3	1	3	1

Problem 5: Processor (34 points)

Here is a program and part of its assembly code written in Y86:

`void calc(int* array, int size){` `    int sum = 0;` `    int* a = array;` `    for (; size>0 ; size--) {` `        if (*a > 0) {` `            sum += 1;` `        }` `        sum += 1;` `        a++;` `    }` `}`	`Loop:` `   mrmovq  (%rdi), %r10 // %r10 = *a` `   andq    %r10, %r10` `   jle     L2          // if (%r10 > 0)` `   iaddq   $1, %rax    //    sum++` `L2:` `   iaddq   $1, %rax    // sum++` `   iaddq   $4, %rdi    // a++` `   iaddq   $-1, %rdx   // size--` `   jg     Loop` `   halt`

1. Now run the program on unmodified **PIPE** implementation with input **{-1, 2, 3, 4, 5}** stored in `array` and `size = 5`.
    a. Show **all hazards** in the **given assembly code** and **how many bubbles** will be inserted for **each** kind of hazard. (6')
    b. Calculate the **CPI** of the **given assembly code**. **NOTE:** calculate the total cycles from the **first stage** of the first instruction to the **last stage** of the last instruction, and the total number of instructions is the **number of instructions that are actually executed**. (2')

Here we will implement a new instruction to reduce the overhead of **misprediction**: conditional redo, `credoxx`, with following encoding:

	Byte	0	
`credoxx`		`E`	`Fn`

Name	Value (hex)	Meaning
`ICREDOXX`	`E`	`Code for credoxx instruction`

This instruction (for example, `credole`) will **repeat the next instruction if the condition is satisfied**, which causes the **next instruction** be executed **twice**.

2. Please list the **detection conditions** like Figure 4.64 and **new control action** like Figure 4.66 in the following situations. **NOTE: DO NOT** consider the **combinations** of hazards here. (1'*8)

a. What if the **credoxx** is **NOT-TAKEN**? (**NOTE**: **NOT_TAKEN** means the next instruction would not be repeated.)

Condition	Trigger				
credoxx not-taken	E_icode == ICREDOXX && [1]				
	Pipeline register				
	F	D	E	M	W
	[2]	[3]	[4]	--	--

b. What if the **credoxx** is **TAKEN**? (**HINT:** Recall the usage of stall)

Condition	Trigger				
credoxx taken	E_icode == ICREDOXX && [5]				
	Pipeline register				
	F	D	E	M	W
	[6]	[7]	[8]	--	--

3. Please rewrite the **given part** of assembly code using **credoxx**. Using the given input, **how many cycles do you save** compared to question 1? (4′+2′)

4. There will be **new combination of pipeline hazards** due to **credoxx** **TAKEN**.
   a. Please draw the combination like **Figure 4.67**. (2′)
   b. Show how to handle it correctly. (4′)

Pipeline register				
F	D	E	M	W
[1]	[2]	[3]	[4]	normal

5. Please provide the **increased or modified HCL** code in **PIPE** implementation of **F_stall**, **D_bubble** and **D_stall** logic caused by **credoxx** instruction. (3′)

For example:

```
bool instr_valid = f_icode in { ICREDOXX };
```

6. According to Figure 4-57, when F stage is stalled, the **f_pc** may still receive value from **M_valA** or **W_valM**, which may change the repeated instruction to another one. Can **credoxx** work correctly in this situation? Why? (**HINT:** think about what kind of instructions will generate **M_valA** and **W_valM** and pass them to f_pc.) (3′)

草稿纸