上 海 交 通 大 学 试 卷（ B 卷）

（ 2013 至 2014 学年 第 2 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: Memory Allocation (16points)

1.

2.

3.

Problem 2: Linking (18points)

1. [1] [2] [3] [4]
   [5] [6] [7] [8]
   [9] [10]

2.

3. 1)

   2)

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)									

Problem 3: Linking (23points)

1. [1] [2]

   [3] [4] [5]

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

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

3. [1] [2]

   [3] [4]

Problem 4: Processor (45points)

1.

Field	rret rB	rcall rB valC
Fetch		
Decode		
Execute		
Memory		
Write Back		
PC update		

2.

3.

4.

5.

6. [1] [2] [3] [4]

7.

Problem 1: Memory Allocation (16 points)

The figure simulates the **initial** status of memory at a certain time. 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. Given the execution sequence of memory allocation operations (**`malloc()`** or **`free()`**) from 1 to 7. Please answer the following questions. Assume that **immediate coalescing** strategy and **splitting free blocks** are employed. (NOTE: you don't need consider P1, P2 and P3 when calculating internal fragments)


```
1.  P4 = malloc(7)
2.  free(P2)
3.  P5 = malloc(1)
4.  P6 = malloc(3)
5.  free(P1)
6.  P7 = malloc(5)
```

1. Assume **first-fit** algorithm is used to find free blocks. Please draw the status of memory and mark with variables after the operation sequence is executed (3′). Please also identify the **total bytes** of the **internal fragments**. (2′)

2. Assume **best-fit** algorithm is used to find free blocks. Please draw the status of memory and mark with variables after the operation sequence is executed (3′). Please also identify the **total bytes** of the **internal fragments**. (2′).

3. Suppose that we use the entire free block instead of splitting when doing allocation. Identify that whether the operation sequence will success or not for both **first-fit** and **best-fit** respectively. If **yes**, please give the **total bytes** of the **internal fragments**. If **no**, please point out **which step** causes the failure. (6′)

Problem 2: Linking (18 points)

The following program consists of two modules: foo and bar. Their corresponding source code files are shown below.

```
/* file: foo.c */
#include <stdio.h>

void f(void);

short a = 0x1;
short b;
static short c = 0x3;

int main(void) {
   b = 0x2;
   short d = 0x4;
   f();
   printf("a=0x%x b=0x%x c=0x%x d=0x%x\n",
          a, b, c, d);
   return 0;
}
```

```
/* file: bar.c */

long long a;
int d;

void f(void) {
   a = 0x0;
   d = 0x0;
}
```

1. For each symbol that is defined and referenced in **foo.o**, please indicate whether it will have a symbol table entry in the **.symtab** section in module **foo.o**. If **Yes**, please fill the symbol type (**global**, **local** or **extern**); If **No**, fill with '--'. (10')

Symbol	.symtab entry (foo.o)	Symbol Type
a	[1]	[2]
b	[3]	[4]
c	[5]	[6]
d	[7]	[8]
f	[9]	[10]

2. Please write down the output of `foo.c`. (4')

3. Assume that the start address of **_GLOBAL_OFFSET_TABLE_** is **0x0804962c** and the partial **.PLT(Procedure Linkage Table)** after linking is:

```
080482fc <printf@plt>:
  80482fc:  ff 25 40 96 04 08  jmp    *0x8049640
  8048302:  68 10 00 00 00     push   $0x10
  8048307:  e9 c0 ff ff ff     jmp    80482cc <_init+0x30>
```

   1) What is the value stored in the address **0x08049640** before first calling the **printf()** function? (NOTE: resolved as a 32-bit hexadecimal) (2')

   2) What is the index of **printf()** in **_GLOBAL_OFFSET_TABLE_**? (NTOE: The index starts from 0) (2')

Problem 3: Linking (21 points)

The following program consists of two source files: **foo.c** and **bar.c**. The relocatable object files are also listed.

```
/* foo.c */

static int n = 2013;
int *p_n = &n;

int foo(int x) {
   if (x < n) return 1;
   return foo(x-1)*n;
}
```

```
.text:
 00000000 <foo>:
  0:  55                    push   %ebp
  1:  89 e5                 mov    %esp,%ebp
  3:  83 ec 08              sub    $0x8,%esp
  6:  a1 00 00 00 00        mov    0x0,%eax
  b:  39 45 08              cmp    %eax,0x8(%ebp)
  e:  7d 07                 jge    17 <foo+0x17>
 10:  b8 01 00 00 00        mov    $0x1,%eax
 15:  eb 1b                 jmp    32 <foo+0x32>
 17:  8b 45 08              mov    0x8(%ebp),%eax
 1a:  83 e8 01              sub    $0x1,%eax
 1d:  83 ec 0c              sub    $0xc,%esp
 20:  50                    push   %eax
 21:  e8 fc ff ff ff        call   22 <foo+0x22>
 26:  83 c4 10              add    $0x10,%esp
 29:  8b 15 00 00 00 00     mov    0x0,%edx
 2f:  0f af c2              imul   %edx,%eax
 32:  c9                    leave
 33:  c3                    ret

.data:
 00000000 <n>:
  0:  dd 07 00 00
 00000004 <p_n>:
  4:  00 00 00 00
```

```
/* bar.c */

extern int foo(int n);
extern int *p_n;

int n = 2015;
int a[2048];

void bar(void) {
   *p_n = 2014;
   a[2] = foo(n);
}
```

```
.text:
 00000000 <bar>:
  0:  55                    push   %ebp
  1:  89 e5                 mov    %esp,%ebp
  3:  83 ec 08              sub    $0x8,%esp
  6:  a1 00 00 00 00        mov    0x0,%eax
  b:  c7 00 de 07 00 00     movl   $0x7de,(%eax)
 11:  a1 00 00 00 00        mov    0x0,%eax
 16:  83 ec 0c              sub    $0xc,%esp
 19:  50                    push   %eax
 1a:  e8 fc ff ff ff        call   1b <bar+0x1b>
 1f:  83 c4 10              add    $0x10,%esp
 22:  a3 08 00 00 00        mov    %eax,0x8
 27:  c9                    leave
 28:  c3                    ret
```

	.data: 00000000 <n>: 0: df 07 00 00

1. Fill in the symbol table of **foo.o** and **bar.o** respectively. (1'*5=5')
   Hints: You should fill in the Bind field with '**GLOBAL**' or '**LOCAL**', the Section field with '**.data**', '**.text**', '**.bss**' or '**-**' if it can't be determined.

   Part of the symbol table of **foo.o**

TYPE	Bind	Section	Name
OBJECT	[1]	.data	n
OBJECT	[2]	.data	p_n
FUNC	GLOBAL	.text	foo

   Part of the symbol table of **bar.o**

TYPE	Bind	Section	Name
NOTYPE	GLOBAL	[3]	p_n
NOTYPE	GLOBAL	--	foo
OBJECT	GLOBAL	[4]	n
OBJECT	GLOBAL	[5]	a

2. Fill in the relocation entries of **foo.o** and **bar.o** respectively. (1'*8=8')

   Relocation entries of **bar.o**

Section	Offset	Type	Symbol Name
.text	00000007	[1]	p_n
.text	00000012	R_386_32	n
.text	[2]	R_386_PC_32	foo
.text	[3]	R_386_32	[4]

   Relocation entries of **foo.o**

Section	Offset	Type	Symbol Name
.data	00000004	[5]	.data
.text	00000007	R_386_32	.data
.text	[6]	[7]	[8]
.text	0000002b	R_386_32	.data

3. After relocation and the program is built, what changes will happen to the underlined 4 instructions/data according to a part of the symbol table given below? (2'*4=8')

Name	Type	Value
foo	FUNC	080483cb
bar	FUNC	080483ff
a	OBJECT	08049780
n (in foo's .data)	OBJECT	0804972c
p_n	OBJECT	08049730
n (in bar's .data)	OBJECT	08049734

```
In foo:
  00000004 <p_n>:
   4: 00 00 00 00                          After relocation:  [1]

In bar:
   6: a1 00 00 00 00   (mov 0x0,%eax)       After relocation:  [2]
  1a: e8 fc ff ff ff   (call 1b <bar+0x1b>) After relocation:  [3]
  22: a3 08 00 00 00   (mov %eax,0x8)       After relocation:  [4]
```

Problem 4: Processor (45 points)

In original architecture, the stack (memory) is used to keep the **return address**. However, in this problem, we directly use a register to keep the return address for a better performance. Suppose two new instructions, namely **`rcall`** and **`rret`**, are used to replace **`call`** and **`ret`** instruction in Y86 instruction sets, which have the following encoding. (NOTE the original call and ret instruction will never be used)

	Byte 0	1	2		6
`rcall rB valC`	**E**	**Fn**	**F**	**rb**	**valc**
`rret rB`	**E**	**Fn**	**F**	**rb**	

1. Please fill in the **generic** function of each stage for **`rcall rB valC`** and **`rret rB`** on updated **sequential** implementation like **Figure 4.21**. (12')
   (NOTE: fill all functions in each stage, and use '–' for empty stage)

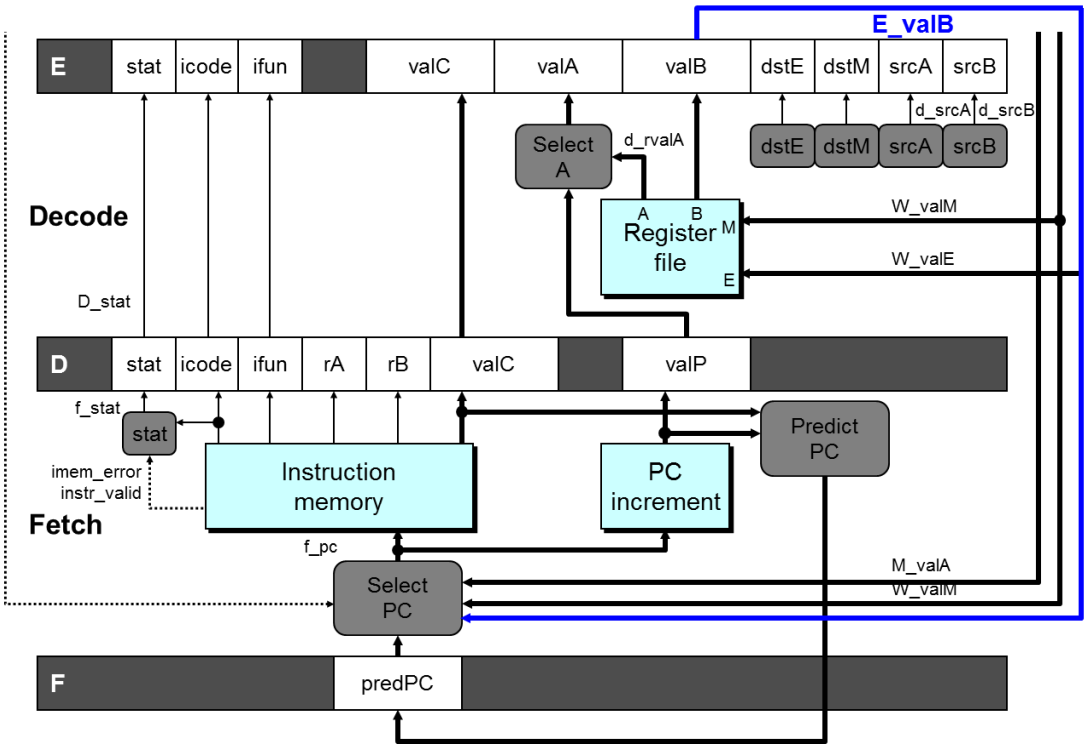
Field	rret rB	rcall rB valC
Fetch	[1]	[7]
Decode	[2]	[8]
Execute	[3]	[9]
Memory	[4]	[10]
Write Back	[5]	[11]
PC update	[6]	[12]

2. Suppose we just add a new forwarding logic from **`W_valE`** to **`f_pc`**, and the rest of pipeline hardware structure is the same to original (Figure 4.41). Please describe all possible hazards due to the two new instructs respectively. You need provide detail explanation and list **detection conditions** like Figure 4.64 and **control action** like Figure 4.66. (6')

Condition	Trigger

Condition	Pipeline register				
	F	D	E	M	W

3. As shown in the following new PIPE logic figure, we add a **return forwarding** logic from **`E_valB`** to **`f_pc`** to take back return address for the new instructions. Please describe all possible hazards again on optimized hardware structure. You still need provide detail explanation and list **detection conditions** like Figure 4.64 and **control action** like Figure 4.66. (6')



4. According to new hardware structure with **return forwarding** (mentioned in problem 4.3), please describes the modification and provides increased HCL code of **`f_pc`**, ,**`F_stall`**, **`D_stal`**l and **`D_bubble`** logic for two new instructions (**`rcall`** and **`rret`**). (NOTE: you need to provide all increased codes due to **`rcall`** and **`rret`**, even the symbol **`IRCALL`** or **`IRRET`** **don't appear in the expression directly**) (8')

   For example:

   ```
   bool instr_valid = f_icode in {IRCALL , IRRET};
   ```

5. Compared with the original instruction set and hardware structure (Figure 4.41), the two new instructions (**`rcall`** and **`rret`**) and return forwarding will cause new combinations of hazards. Please draw the **pipeline states** figure (Figure 4.67) and list **pipeline control action** (see the table of Problem 4.35 and 4.36) for new combinations about new instructions. (6')

6. Please calculate the number of **cycles** and **waste cycles** for the following codes in **original** and new architecture. The initial value of all registers are **zero** and we always use **TAKEN** branch prediction strategy for all conditional jump. (**Hint**: you need calculate the number of cycles until the last stage of the last instruction) (4')

Original	**New**

`irmovl Stack,%esp`
`irmovl $2,%eax`
`loop:`
`call foo`
`irmovl $-1,%ebx`
`addl %ebx,%eax`
`jne loop`
`halt`
`foo:`
`ret %edi`
`irmovl $1,%eax`
`Stack:`
	`irmovl Stack,%esp`
	`irmovl $2,%eax`
	`loop:`
	`rcall %edi,foo`
	`irmovl $-1,%ebx`
	`addl %ebx,%eax`
	`jne loop`
	`halt`
	`foo:`
	`rret %edi`
	`irmovl $1,%eax`
	`Stack:`

Cyclyes: [1] **Wasted cycles: [2]**	**Cyclyes: [3]** **Wasted cycles: [4]**

7. To further improve the pipeline logic for new instructions. We use the **fast forwarding** (from **d_rvalB** to **f_pc)** to replace the **return forwarding** (from **E_valB** to **f_pc**) in problem 4.3. Does the fast forwarding provide the correct result and resolve hazard? Please provide some explanation to your answer. (3')