复旦大学计算机科学技术学院

《计算机原理》期中考试试卷

A 卷　共 12 页

课程代码：COMP130007.0_　　考试形式：☑开卷　□闭卷　　2013 年 5 月

（本试卷答卷时间为 120 分钟，答案必须写在试卷上，做在草稿纸上无效）

专业_____学号_____姓名_____成绩_____

题号	一	二	三	四	五	六	七	八	九	总分
得分										

Problem 1　(10 points):

We are running programs on a machine with the following characteristics:

·Values of type int are 32 bits. They are represented in two's complement, and they are right shifted arithmetically. Values of type unsigned are 32 bits.

·Values of type float are represented using the 32-bit IEEE floating point format, while values of type double use the 64-bit IEEE floating point format.

We generate arbitrary values x, y, and z, and convert them to other forms as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
/* Convert to other forms */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
double dx = (double) x;
double dy = (double) y;
double dz = (double) z;
```

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If so, circle "Y". If not, circle "N".

Expression	Always True?
(x<y) == (-x>-y)	Y　N
((x+y)<<4) + y-x == 17*y+15*x	Y　N

~x+~y+1 == ~(x+y)	Y N
ux-uy == -(y-x)	Y N
(x >= 0) || (x < ux)	Y N
((x >> 1) << 1) <= x	Y N
(double)(float) x == (double) x	Y N
dx + dy == (double) (y+x)	Y N
dx + dy + dz == dz + dy + dx	Y N
dx * dy * dz == dz * dy * dx	Y N

Answer:

Expression	Always True?
1. (x<y) == (-x>-y)	N: Let x = Tmin, y = 0
2. ((x+y)<<4) + y-x == 17*y+15*x	Y: Associative, commutative, distributes
3. ~x+~y+1 == ~(x+y)	Y: (-x-1)+(-y-1)+1 == -(x+y)-1
4. ux-uy == -(y-x)	Y
5. (x >= 0) || (x < ux)	N: x = -1. Comparison x < ux is never true.
6. ((x >> 1) << 1) <= x	Y: x>>1 rounds toward minus infinity.
7. (double)(float) x == (double) x	N: Try x = Tmax.
8. dx + dy == (double) (y+x)	N: Try x=y=Tmin.
9. dx + dy + dz == dz + dy + dx	Y: Within range of exact representation by double's.
10. dx * dy * dz == dz * dy * dx	N: Try x= 1110474423,y=1613900859,z=362894370

Problem 2: (9*1=9pts)

Consider a 8-bit floating-point representation based on the IEEE floating point format, with one sign bit, 3 exponent bits (k=3), and 4 fraction bits (n=4). The exponent bias is $2^{k-1}-1 = 3$ and $V = (-1)^s \times M \times 2^E$.

Fill the blank in the table below. (You need not fill in entries marked with "X".)

Description	**Binary**	**M**	**E**	**Value**
X	01000001	17/16(1.0625)	4-3=1	17/8(2.125)
Largest normalized (positive)	01101111	X	X	31/2(15.5)
Smallest denormalized (negative)	10001111	15/16(0.9375)	1-3 = -2	-15/64(0.234375)

Note:

Binary: The 8 bit binary representation. **M:** The value of the significand.

E: The integer value of the exponent. **Value:** The numeric value represented.

Problem 3：(18pts)

Implement the following functions as you did in Lab1(Data.Lab)

Part 1 (4*2=8pts)

```
/*
  * bitCount - returns count of number of 1's in word
  *      Examples: bitCount(5) = 2, bitCount(7) = 3
  *      Legal ops: ! ~ & ^ | + << >>
  *      Max ops: 40
  *      Rating: 4
  */
int bitCount(int x) {
        int base1 =   0x55   ;
        int mask1 = (base1<<24) + (base1<<16) + (base1<<8) + base1;
        int base2 = 0x33;
        int mask2 =   (base2<<24) + (base2<<16) + (base2<<8) + base2   ;
        int mask3 = (0xf<<24) + (0xf<<16) + (0xf<<8) + 0xf;
        int mask4 = (0xff<<16) + 0xff;
        int mask5 = (0xff<<8) + 0xff;
        x = (x & mask1) +  ((x>>1) & mask1)   ;
        x = (x & mask2) +  ((x>>2) & mask2)   ;
        x = (x + (x>>4)) & mask3;
        x = (x + (x>>8)) & mask4;
        x = (x + (x>>16)) & mask5;
        return x;
}
```

Part 3 (5*2=10pts)

```
/*
  * float_i2f - Return bit-level equivalent of expression (float) x
  *      Result is returned as unsigned int, but it is to be interpreted as the bit-level
  *      representation of a single-precision floating point values.
  *      Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
  *      Max ops: 30
  *      Rating: 4
  */
unsigned float_i2f(int x) {
        int sign, i, frac, temp, rounding;
        if (x == 0) return 0;
        if (x < 0){
                sign = 0x80000000;
                x = -x;
        } else {
                sign = 0;
        }
        i = 0;
```

```
    frac = x;
    while (1){
        temp = frac;
        frac =  frac << 1  ;
        ++i;
        if (  temp & 0x80000000  ) break;
    }
    if ((frac & 0x1ff) > 0x100)
        rounding = 1;
    else if (  (frac & 0x3ff) == 0x300  )
        rounding = 1 ;
    else
        rounding = 0 ;
    return  sign + ((159 - i) << 23) + (frac >> 9) + rounding  ;
}
```

Problem 4: (8 points)

You come across the following code in a large C program.

```
typedef struct {
    int left;
    a_struct a[CNT];
    int right;
} b_struct;

void test(int i, b_struct *bp) {
    int n = bp->left + bp->right;
    a_struct *ap = &bp->a[i];
    ap->x[ap->idx] = n;
}
```

Unfortunately, the '.h' file defining the compile-time constant **CNT** and the structure **a_struct** are in files for which you do not have access privileges. Fortunately, you have access to a '.o' version of the function "test", which you are able to disassemble with the objdump program, yielding the disassembly code:

```
00000000 <test>:
 0:   push %ebp
 1:   mov %esp,%ebp
 3:   push %ebx
 4:   mov 0x8(%ebp),%eax
 7:   mov 0xc(%ebp),%ecx
 a:   lea (%eax,%eax,4),%eax
 d:   lea 0x4(%ecx,%eax,4),%eax
```

```
11:   mov (%eax),%edx
13:   shl $0x2,%edx
16:   mov 0xb8(%ecx),%ebx
1c:   add (%ecx),%ebx
1e:   mov %ebx,0x4(%edx,%eax,1)
22:   pop %ebx
23:   mov %ebp,%esp
25:   pop %ebp
26:   ret
```

Using your reverse engineering skills, deduce the following:

Part 1. (4pts)

The value of CNT.is 9 .

Part 2. (4pts)

Assume that the only fields in **a_struct** are **idx** and **x**. Complete the declaration of structure a_struct.

```
typedef struct {
    int idx;
    int x[4];
} a_struct;
```

Problem 5: (6*2=12points)

Here is the C code of function **foo** and corresponding assemble code. Fill in the blank to complete the C code.

C code:

```
int foo (int x, int y, int n) {
    do {
        switch(x) {
            case 30:
                y = y << 3 ;
                break;
            case 31:
                y = y << 2 ;
                break;
            case 33:
                y = y - 6;
            case 34:
                y = y + 6 ;
                break;
            default:
                break;
```

```
        }
        x = y;
        n--;
    } while (n > 0);
    return  y  ;
}
```

Assemble code:

```
NOTE: Initially x, y, and n are at offsets 8, 12
and 16 from %ebp.
foo:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
.L2:
    movl    8(%ebp), %eax
    leal    -30(%eax), %edx
    cmpl    $4, %edx
    ja      .L3
    jmp     *.L8(,%edx, 4)
.L4:
    sall    $1, 12(%ebp)
.L5:
    sall    $2, 12(%ebp)
.L6:
    jmp     .L3
.L7:
    addl    $6, 12(%ebp)
.L3:
    movl    12(%ebp), %eax
    movl    %eax, 8(%ebp)
    subl    $1, 16(%ebp)
    cmpl    $0, 16(%ebp)
    jg      .L2
    movl    12(%ebp), %eax
    leave
    ret
```

```
.section .rodata        #Jump Table
.align 4
.L8:
        .long   .L4
        .long   .L5
        .long   .L3
        .long   .L6
        .long   .L7
```

Problem 6：(18pts)

Consider the following assembly code from Lab2(Bomb.Lab), Phase4.

```
8048ca0<func4>:
8048ca0:    push    %ebp
8048ca1:    mov     %esp, %ebp
8048ca3:    sub     $0x10, %esp
```

```
8048ca6:    push    %esi
8048ca7:    push    %ebx
8048ca8:    mov     0x8(%ebp), %ebx
8048cab:    cmp     $0x1, %ebx
8048cae:    jle     8048cd0 <func4+0x30>
8048cb0:    add     $0xfffffff4, %esp
8048cb3:    lea     -0x1(%ebx), %eax
8048cb6:    push    %eax
8048cb7:    call    8048ca0 <func4>
8048cbc:    mov     %eax,%esi
8048cbe:    add     $0xfffffff4, %esp
8048cc1:    lea     -0x2(%ebx), %eax
8048cc4:    push    %eax
8048cc5:    call    8048ca0 <func4>
8048cca:    add     %esi, %eax
8048ccc:    jmp     8048cd5 <func4+0x35>
8048cce:    mov     %esi, %esi
8048cd0:    mov     $0x1,%eax
8048cd5:    lea     -0x18(%ebp), %esp
8048cd8:    pop     %ebx
8048cd9:    pop     %esi
8048cda:    mov     %ebp,%esp
8048cdc:    pop     %ebp
8048cdd:    ret

08048ce0 <phase_4>:
......
8048d15:    call    8048ca0 <func4>
8048d1a:    add     $0x10, %esp
8048d1d:    cmp     $0x37,%eax
8048d20:    je      8048d27 <phase_4+0x47>
8048d22:    call    80494fc <explode_bomb>
 ......
```

Here is corresponding C program

```
int func4(int n) {
    if (n <= 1)
        return 1;
    return func4(n - 1) + func4(n - 2);
}
```

Part 1(5*2=10pts)

Fill the blanks in the assembly code and C code of func4.

Part 2(8*1=8pts)

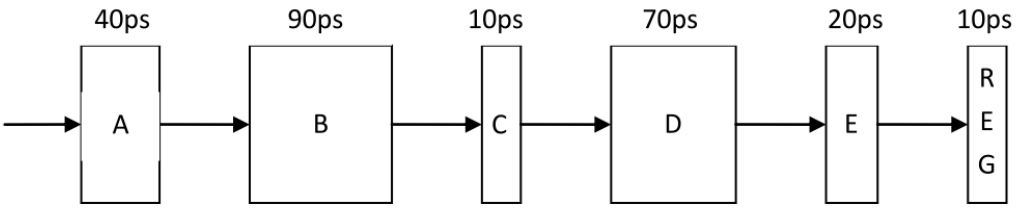
Suppose the value of register **%esp** is **0xffffd2c0** before the program enter the function "func4" first time(0x8048d15 in <phase_4>) with parameter **0x03**. Consider the diagram of **stack frame** before we return from the first call to function "func4" with parameter **0x01**.

Fill in the following blanks with the corresponding address and/or value. ("…" represents some wasted space on the stack). If there is not enough information to determine the value, fill in the blank with "**UND**". You need not care about the part of the stack that is beyond the address of the following table.

Address	**Value**
0xffffd2c0	0x03 (parameter to call func4)
0xffffd2bc	0x08048d1a (return address)
0xffffd2b8	0xffffd2e8 (old %ebp)
…	…
0xffffd2a4	UND (saved %esi)
0xffffd2a0	UND (saved %ebx)
…	…
0xffffd290	0x02 (parameter to call func4)
0xffffd28c	0x08048cbc (return address)
0xffffd288	0xffff2b8 (old %ebp)
…	…
0xffffd274	UND (saved %esi)
0xffffd270	0x03 (saved %ebx)
…	…
0xffffd260	0x01 (parameter)
0xffffd25c	0x08048cbc(return address)
0xffffd258	0xffff288(old %ebp)
…	…

Problem 7：(2*4=8pts)

Suppose we analyze the combinational logic of the figure below and determine that it can be separated into a sequence of five blocks, named A to E, having delays of 40, 90, 10, 70, 20ps, respectively:



To create pipelined versions, we need to insert pipeline registers between pairs of these blocks. Assume that a pipeline register has a delay of **10 ps**.

(1) Inserting one single register gives a two-stage pipeline. To get the maximize throughput, where should we insert this register? And what would be the latency and throughput?

The best partition would be to have blocks A and B in the first stage, while C, D and E in the second. We therefore have a latency of 280 ps and a throughput of 7.14 GOPS.

(2) Where should three registers be inserted to maximize the throughput of a four-stage pipeline? What would be the latency and throughput?

The best partition would be to have block A in the first stage, block B in the second, block C and D in the third while block E in the fourth. Thus the latency is 400 ps and the throughput is 10 GOPS.

Problem 8：(15pts)

Part 1(1*5=5pts)

Translate the following x86 instructions into y86 instructions and encoding these instructions.

X86	Y86	Encoding
andl %ebx, %edi	andl %ebx, %edi	62 37
movl $8001000, %eax	irmovl $8001000, %eax	30 80 e8 15 7a 00 或 30 f0 e8157a00
movl -12(%esp),%edx	mrmovl -12(%esp), %edx	50 24 f4 ff ff ff
pushl %eax	pushl %eax	a0 08 或 a0 0f
jl $0x80010f0	jl $0x80010f0	72 f0 10 00 08

Part 2(+6=10pts)

Consider the following block of code. Suppose that it will be executed on a PIPE with stalling and .

```
1   irmovl   $10, %ebx
2   rrmovl   $ebx, %ecx
3   mrmovl   0(%edx), %eax
4   subl     %ecx, %eax
5   rmmovl   %eax, 20(%edx)
6   halt
```

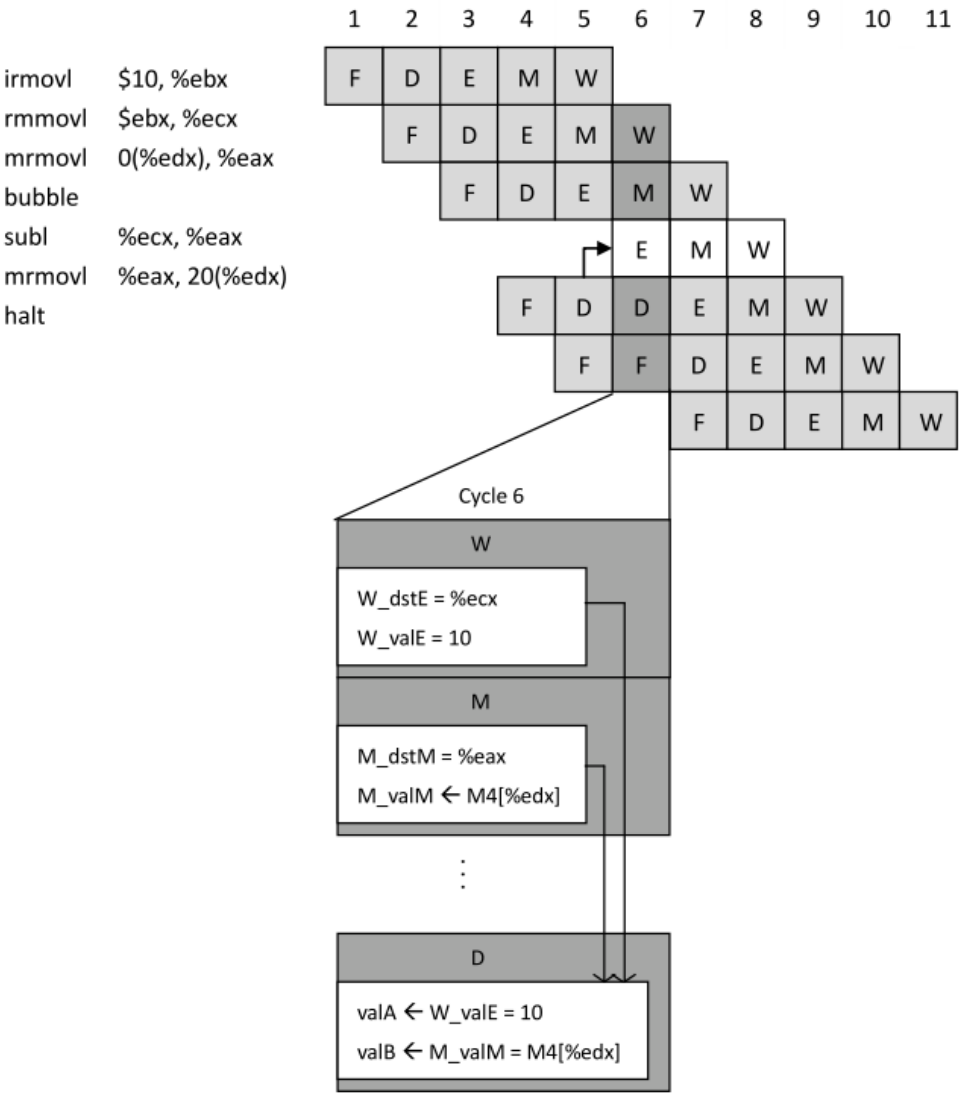
(1) Identify all of the data hazards in the above code.

There is a load/use hazards between step 3 (mrmovl 0(%edx), %eax) and step 4 (subl %ecx, %eax).

Data hazard between 1&2, 2&4, 4&5 can be solved by forwarding.

Anybody who writes down these must point out that they are not real hazards.

2) Draw the pipelined execution diagram (using forwarding) in the form showed on page 336(figure 4.55) of your ICS book.



Problem 9：(7pt, Extra Credit)

Read the following material, explain it using what you have learned in this course.

登鼓山与老和尚喝茶。我说：“我放不下一些事，放不下一些人。”他说：“没有什么东西是放不下的”。我说：“可我偏偏放不下。”他说，你不是喜欢喝茶吗，就递我一个茶杯然后往里面倒热水，一直倒到热水溢出来。我连忙松手并大喊：“***烫烫烫烫烫烫烫***”，和尚说：你的栈内存没初始化吧！

Note: "烫" is encoded into 0xCCCC in GBK encoding.

在 Windows 系统下使用 VC 编译器，在 Debug 模式下生成的汇编代码会将未初始化的栈空间字节字符设为 0xCC（中断 int 3 的指令编码）。由于 0xCC 超过了 ASCII 码 0-127 这个范围，因此这个“字符串”被系统当成了宽字符组成的字符串，即两个字节数据组成一个字符，而 0xCCCC 表示的宽字符正好是 GBK 编码中的“烫”字，因此向控制台输出时就显示为“烫”。

例如，下面的程序在 VisualStudio2010，Debug 模式下，向控制台输出“***烫烫烫烫烫烫烫***”。

```
#include<stdio.h>
```

```
int main (int argc, char** argv) {
    char a[15];
    a[14] = '\0';
    printf("%s\n", a);
    return 0;
}
```