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　　2014 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	总分
得分									

Problem 1: (10 points)

We would like to write C function in 32-bit machine to set the penult(倒数第二个) significant byte of x to 0 and set the least significant byte to 0xFF. Please fill the blank and make the function portable(可移植) to 64-bit machine.

```
int bis ( int x )
{
    int m = ____________; /* m is the mask word */
    x = ____________;
    x = x | ____________;
    return x
}
```

Answer:

~0xFF00

x & m

0xFF

或者

0xFF00

x & ~m

0xFF

Problem 2: (15 points)

Consider a 9-bit floating-point representation based on the IEEE floating point format, with one sign bit, 3 exponent bits (k=3), and 5 fraction bits (n=5). The exponent bias is $2^{k-1}-1 = 3$ and $V = (-1)^s \times M \times 2^E$, where M is the significand and E is the biased exponent..

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

Description	**Binary**	**M**	**E**	**Value**
X	010000001	33/32(1.03125)	4-3=1	33/16(2.0625)
Largest normalized (positive)	011011111	63/32(1.96875)	6-3=3	63/2(31.5)
Smallest denormalized (negative)	100011111	31/32(0.96875)	1-3 = -2	-31/128(0.2421875)
Infinity	011100000	X	X	+∞
X	010111010	29/16(1.8125)	5-3=2	7.25

Problem 3: (10pts)

In the C function that follows, we have omitted the body of the switch statement. In the C code, the case labels did not span a contiguous range, and some cases had multiple labels.

```
int switch2(int x) {
int result = 0;
switch (x) {
/* Body of switch statement omitted */
}
return result;
}
```

In compiling the function, GCC generates the assembly code that follows for the initial part of the procedure and for the jump table. Variable x is initially at offset 8 relative to register %ebp.

Setting up jump table access	*Jump table for switch2*
1 movl 8(%ebp),%eax *Retrieve x*	. L11 :
2 addl $4,%eax	.long .L4
3 cmpl $8,%eax	.long .L10
4 ja .L5	.long .L5
5 jmp *.L11(,%eax,4)	.long .L6
	.long .L8
	.long .L5
	.long .L9
	.long .L8
	.long .L10

Use the foregoing information to answer the following questions:

A. What were the values of the case labels in the switch statement body?

B. What cases had multiple labels in the C code?

空半页

Solution

This problem gives you a chance to reason about the control flow of a switch statement. Answering the questions requires you to combine information from several places in the assembly code:

1. Line 2 of the assembly code adds 4 to X to set the lower range of the cases to 0. That means that the minimum case label is -4.

2. Lines 3 and 4 cause the program to jump to the default case when the adjusted case value is greater than 8. This implies that the maximum case label is -4 + 8 = 4.

3. In the jump table, we see that the third and sixth entry (case label -2 and 1) has the same destination (. L5) as the jump instruction on line 4, indicating the default case behavior. Thus, case label -2 and 1 is missing in the switch statement body.

4. In the jump table, we see that the fifth and eighth entries have the same destination .L8. These correspond to case labels 0 and 3. In the same way, second and ninth entries have the same destination .L10 corresponding to case labels -3 and 4.

From this reasoning, we draw two conclusions:

A. The case labels in the switch statement body had values -4,-3,-1, 0, 2, 3, and 4.

B. The case with destination .L8 had labels 0 and 3. The case with destination .L10 had labels-3 and 4. *不答 L5,default 情况*

Problem 4: (12pts)

The following C code sets the diagonal elements of a fixed-size array to val:

```
#define N __      %when in your program, you must fill in the blank to point out the value of N
typedef int fix_matrix[N] [N];
/* Set all diagonal elements to val */
void fiX_set_diag(fix_matriX A, int val)
{
int i;
for (i = 0; i < N; i++)
A[i][i] = val;
}
```

When compiled, GCC generates the following assembly code:

```
movl 12(%ebp),%edx
movl 8(%ebp),%eax
movl $31,%ecx
addl $4092,%eax
.p2a1ign 4,,7     %Added to optimize cache performance
.L50:
movl %edx,(%eax)
addl $-132,%eax
decl %ecx
jns .L50
```

Create a C code program fiX_Set_diag_Opt that uses optimizations similar to those in the assembly code, in the same style as the code in the Figure below. Notice that in your program, you must point out the value of N by "#define N __" (fill in the blank).

```
/* Compute i,k of fixed matrix product */
int fix_prod_ele_opt(fix_matrix A, fix_matriX B, int i, int k)
{
int *Aptr = &A[i][0];
int *Bptr = &B[0][k];
int cnt = N - 1;
int result = 0;
do {
  result +=(*Aptr) * (*Bptr);
  Aptr += 1;
  Bptr += N;
  cnt --;
  } while (cnt >= 0);

  return result;
  }
```

Solution

~~*This exercise requires you to study assembly code to understand how it has been optimized. This is an important skill for improving program performance. By adjusting your source code, you can have an effect on the efficiency of the generated machine code.*~~

The following is an optimized version of the C code:

```
#define N 32
typedef int fix_matrix[N] [N];

/* Set all diagonal elements to val */
void fix_set_diag_opt(fix_matrix A, int val)
{
   int *Aptr = &A[0][0] + 1023;
   int cnt = N - 1;
   do {
      *Aptr = val;
      Aptr -= (N+1);
      cnt--;
   } while (cnt >= O);
}
```

Problem 5: (11pts)

Complete the following blanks according to what you learned from Lab2 (Bomb Lab).

Note: the bomb is generated in an **AMD64** Linux machine.

C code

```
int func4(int a, int b, int c)
{
    int d;

    d = b + (c - b) /   2  ;

    if (d > a)
    return func4(a, b, d-1) <<  1 ;
    else if (d < a)
    return (func4(a, d+1, c) <<  1 ) + 1;
    else
    return 0;
}

void phase_4(char *input) {
    int user_val, user_path, result, target_path, numScanned;

    numScanned = sscanf(input, "%d %d", &user_val, &user_path);
    if ((numScanned != 2) || user_val < 0 || user_val > 14) {
    explode_bomb();      % Program terminate
    }

    target_path = 3;
    result = func4(user_val, 0, 14);

    if (result != target_path || user_path != target_path) {
    explode_bomb();
    }
}
```

Assembly Code

```
00000000004010f4 <func4>:
  4010f4:  ** ** ** **          sub    $0x8,%rsp
  4010f8:  89 d0                mov    %edx,%eax
  4010fa:  29 f0                sub    %esi,%eax
  4010fc:  89 c1                mov    %eax,%ecx
  4010fe:  c1 e9 1f             shr    $0x1f,%ecx
  401101:  01 c8                add    %ecx,%eax
  401103:  d1 f8                sar    %eax
  401105: 8d 0c 30              lea    (%rax,%rsi,1),%ecx
  401108: 39 f9                 cmp    %edi,%ecx
  40110a: ** **                 jle    401118 <func4+0x24>
  40110c: 8d 51 ff              lea    -0x1(%rcx),%edx
  40110f: e8 e0 ff ff ff        callq  4010f4 <func4>
  401114: 01 c0                 add    %eax,%eax
  401116: eb 15                 jmp    40112d <func4+0x39>
  401118: b8 00 00 00 00        mov    $0x0,%eax
  40111d: 39 f9                 cmp    %edi,%ecx
  40111f: ** 0c                 jge    40112d <func4+0x39>
  401121: 8d 71 01              lea    0x1(%rcx),%esi
  401124: e8 cb ff ff ff        callq  4010f4 <func4>
  401129: 8d 44 00 01           lea    0x1(%rax,%rax,1),%eax
  40112d: 48 83 c4 08           add    $0x8,%rsp
  401131: c3                    retq

0000000000401132 <phase_4>:
  401132: 48 83 ec 18           sub    $0x18,%rsp
  401136: 48 8d 4c 24 0c        lea    0xc(%rsp),%rcx
  40113b: 48 8d 54 24 08        lea    0x8(%rsp),%rdx
  401140: be 31 2b 40 00        mov    $0x402b31,%esi
  401145: b8 00 00 00 00        mov    $0x0,%eax
  40114a: e8 31 fb ff ff        callq  400c80 <__isoc99_sscanf@plt>
  40114f: 83 f8 02              cmp    $0x2,%eax
  401152: 75 0d                 jne    401161 <phase_4+0x2f>
  401154: 8b 44 24 08           mov    0x8(%rsp),%eax
  401158: ** **                 test   %eax,%eax
  40115a: 78 05                 js     401161 <phase_4+0x2f>
  40115c: 83 f8 0e              cmp    $0xe,%eax
  40115f: 7e 05                 jle    401166 <phase_4+0x34>
  401161: e8 9e 05 00 00        callq  401704 <explode_bomb>
  401166: ba 0e 00 00 00        mov    $0xe,%edx
  40116b: be 00 00 00 00        mov    $0x0,%esi
  401170: 8b 7c 24 08           mov    0x8(%rsp),%edi
  401174: e8 7b ff ff ff        callq  4010f4 <func4>
  401179: ** ** **              cmp    $0x3,%eax
  40117c: 75 07                 jne    401185 <phase_4+0x53>
  40117e: 83 7c 24 0c 03        cmpl   $0x3,0xc(%rsp)
  401183: 74 05                 je     40118a <phase_4+0x58>
  401185: e8 7a 05 00 00        callq  401704 <explode_bomb>
  40118a: 48 83 c4 18           add    $0x18,%rsp
  40118e: c3                    retq
```

Problem 6: (9 points)

(a) Please make a comparison between fixed and variable length instructions. Discuss each's advantages and disadvantages.(4pts)

Answer: The advantage of using variable-length instructions, is that each instruction can use exactly the amount of space it requires, so that variable length instructions reduce the amount of memory space required for a program.

Fixed length instructions occupy the same amount of space, every instruction must be long enough to specify a memory operand, even if the instruction does not use one. Hence, memory space is wasted by this form of instruction. The advantage of fixed length instructions, it is argued, is that they make the job of fetching and decoding instructions easier and more efficient, which means that they can be executed in less time than the corresponding variable length instructions.

(b) Suppose we were to modify the Y86 PIPE implementation to include a small, hidden hardware stack for return address prediction. We'd push this stack on call instructions (in addition to doing what we normally due to the programmer-visible stack). We'd pop it in the fetch stage of ret instructions, and use it to predict the address of the next instruction. If the average subroutine is 50 cycles long, and if our prediction is right 90% of the time, what percentage improvement in performance can we expect?(5pts)

Answer: In the PIPE design, ret forces three bubbles into the pipeline, expanding our 50 cycles to 53. If we get rid of this penalty 90% of the time, we can expect to cut the average subroutine latency from 53 cycles to 50.3, an improvement of 2.7/53, or just over 5%.

By the way: you might expect the hardware predictor to be right more than 90% of the time, but only in the absence of recursion. The hidden hardware stack will be limited in size, and once recursion goes deeper than that, prediction will cease to be effective.

Problem 7: (5 + 5 + 5 = 15points)

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

Part 1

```
/*
 * absVal - absolute value of x
 *     Example: absVal(-1) = 1.
 *     You may assume -TMax <= x <= TMax
 *     Legal ops:   ! ~ & ^ | + << >>
 *     Max ops: 10
 */
int absVal(int x) {
      /* Please fill your code here*/
```

```
        int mask = x>>31;
        return (x ^ mask) + ~mask + 1L;




}
```

Part 2

```
/*
  * bitParity - returns 1 if x contains an odd number of 0's
  *     Examples: bitParity(5) = 0, bitParity(7) = 1
  *     Legal ops:   ! ~ & ^ | + << >>
  *     Max ops: 20
  */
int bitParity(int x) {
        /* Please fill your code here*/
        int wd16 = x ^ (x>>16); /* Combine into 16 bits */
        int wd8    = wd16 ^ (wd16>>8); /* Combine into 8 bits */
        int wd4    = wd8 ^ (wd8>>4);
        int wd2    = wd4 ^ (wd4>>2);
        int bit    = (wd2 ^ (wd2>>1)) & 0x1;
        return bit;





}
```

Part 3

```
/*
  * float_f2i - Return bit-level equivalent of expression (int) f
  *     for floating point argument f.
  *     Argument is passed as unsigned int, but
  *     it is to be interpreted as the bit-level representation of a
  *     single-precision floating point value.
  *     Anything out of range (including NaN and infinity) should return
  *     0x80000000u.
  *     Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
  *     Max ops: 30
  */
int float_f2i(unsigned uf) {
        /* Please fill code in blanks. */
        unsigned sign = uf >> 31;
        unsigned exp = ___(uf >> 23) & 0xFF___;
        unsigned frac = uf & 0x7FFFFF;
        /* Create normalized value with leading one inserted,
           and rest of significand in bits 8--30. */
        unsigned val = 0x80000000u + (frac << 8);
        if (      exp < 127       ) {   /* Absolute value is < 1 */
           return 0;
        }
        if (exp > 158) {   /* Overflow */
           return 0x80000000u;
        }
        /* Shift val right */
        val = val >>      (158 - exp)     ;
        if (sign) { /* Negative */
              /* Check if out of range */
              return      val > 0x80000000u ? 0x80000000u : -val;
        } else { /* Positive */
              /* Check if out of range */
              return      val > 0x7FFFFFFF ? 0x80000000u : val;
        }
}
```

Problem 8: (18pts)

Suppose we want to add a new instruction **irsubl** with the following format:

Byte		0		1		2	3	4	5
irsubl	**V, rA, rB**	C	1	rA	rB	V			

This instruction substracts rA from the constant value V and save the result to register rB, i.e rB ← V-rA. Describe the computations performed to implement this instruction. Please fill the blank of certain stage with "*Nothing to do*" if the stage has no work to do to accomplish this instruction.

Stage	**irsubl V, rB**

Fetch	icode:ifun ← M_1[PC]
	rA:rB ← M_1[PC+1]
	valC ← M_4 [PC+2]
	valP ← PC+6

Decode	valA ← R[rA]
Execute	valE ← valC - valA

Memory	Nothing to do
Write Back	R[rB]←valE
PC Update	PC←valP