CS-2011 Machine Organization and Assembly Language
Professor Hugh C. Lauer
Quiz date: Tuesday, March 25, 2014

WPI, D-term 2014
Quiz #1 (20 points)

Answers

1. (5 points) For each of the following values of `k`, write an expression in *C* to multiply an arbitrary 32-bit signed integer `x` by `k`, using only the operations `+`, `-`, and `<<` and no more than three operations total.

   `k = 96:`

   ```
   (x << 6) + (x << 5)
   ```

   `k = -15:`

   ```
   x - (x << 4)
   ```

   `k = 160:`

   ```
   (x << 7) + (x << 5)
   ```

   `k = -192:`

   ```
   (x << 6) - (x << 8)
   ```

   `k = 31:`

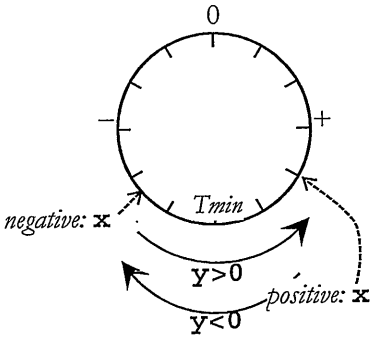
   ```
   (x << 5) - x
   ```

2. (5 points) Write a function in *C* that returns one if the machine is big endian and zero if it is little endian (regardless of word size). The header of the function should be:–

   ```
   int isBigEndian(void) {
      int x = 1;
      char *p = (char *) &x;
      return !(*p);
   }
   ```

3. (5 points) Two's-complement signed integers resemble the face of a clock, but with zero at the top, `Tmin` at the bottom, positive integers on the right, and negative integers on the left. `Tmax` is the last integer on the right before `Tmin`.

   Write a fragment of code in C to tell if the difference `(x - y)` of two signed integers overflows or not. Alternatively, you may explain it words. In either case, indicate on the "clock" on the right the various overflow cases.


   ```
   int tsub_ok(int x, int y){
      int diff = x - y;
      int negOverflow = x <  0 && y >= 0 && diff >= 0;
      int posOverflow = x >= 0 && y <  0 && diff <  0;
      return !negOverflow && !posOverflow;
   }
   ```

4. (5 points) Convert the numbers 13 and 17 to binary and show the (binary) multiplication of these two numbers.

$13_{10} = 1101_2$; $17_{10} = 10001_2$

$$\begin{array}{r} 1101 \\ \times\ 10001 \\ \hline 1101 \\ 1101 \\ \hline 11011101 \end{array}$$

$11011101 = DD_{16} = 221_{10}$

5. (5 points) Write a function in *C* that returns the number of bits in a `long long int`.

```
int countBits(void) {
    long long L = 1;
    int i;
  // shift one bit at a time till L is zero
    for (i = 0; L; i++, L <<= 1);
    return i;
} // int countBits()
```

CS-2011 Machine Organization and Assembly Language
Professor Hugh C. Lauer
Quiz date: Tuesday, April 1, 2014

WPI, D-term 2014
Quiz #2 (20 points)

Answers

1. (5 points) Consider an IEEE floating-point number with a *k*-bit exponent and an *n*-bit fraction. Write the exponent *E* and significand *M* and describe the bit representation for the following values:–

   (a) The number 5.0

   The number 5.0 will have *E* = 2 and *M* = *1.01*. The bit representation will be 100...01 and the fraction representation will be 0100...00.

   (b) The largest even integer that can be represented exactly

   The largest even integer is the largest normalized number that can be represented, namely $(2.0 - \varepsilon) \times 2^{bias-1}$. It consists of all ones in the fraction and a single zero in the low-order bit of the exponent.

   (c) The reciprocal of the smallest positive normalized number

   The smallest positive normalize number has an exponent field of *1* and all zeros in the fraction. This is $1.0 \times 2^{-(bias-1)}$. Since *1* is its own reciprocal, the reciprocal of this smallest number will be $1.0 \times 2^{bias-1}$. This is an exponent with all ones except the low order bit and a fraction with all zeros.

2. (5 points) Assume that **`unsigned f2u(float)`** is a function that returns an **`unsigned`** integer with the same bit pattern as its **`float`** argument. Using only **`unsigned`** operations, write a function with the prototype

   ```
   int float_ge(float fx, float fy);
   ```

   that returns *1* if **`fx >= fy`** and returns zero otherwise. You may assume that neither argument is **`NaN`**. Note that **`-0`** and **`+0`** are considered equal.

```
int float_ge(float fx, float fy){
   unsigned ux = f2u(fx);
   unsigned uy = f2u(fy);
   unsigned sx = ux >> 31;          // sign of fx
   unsigned sy = uy >> 31;          // sign of fy

   return (ux << 1 == 0 && uy << 1 == 0) ||  //both zero
      (!sx && sy) ||                // fx >= 0, fy < 0
      (!sx && !sy && ux >= uy) ||   // both >= 0
      (sx && sy && ux <= uy);       // both < 0

}
```

3. (5 points) Write expressions in *C* that evaluate to *1* when the following conditions are true and that evaluate to zero when they are false. Assume that **`x`** is of type **`int`** (32 bits). You must follow the integer and bit-level integer coding rules of Datalab, with the additional restriction that you may not use equality (**`==`**) or inequality (**`!=`**).

(a) Any bit of **x** equals **1**

```
!!x
```

(b) Any bit of **x** equals **0**

```
!!~x
```

(c) Any bit in the least significant byte of **x** equals **1**

!!(x & 0xff)

(d) Any bit in the most significant byte of **x** equals **0**

!!(~x & (0xff << 24))

4. (5 points) Let the following registers have the following values:–

%edx = 0x808d400
%eax = 0xc8 (= 200_{10})
%ecx = 0x2c (= 44_{10})
%esi = 0x20 (= 32_{10})

Evaluate the following address expressions. (Be sure to indicate whether your answer is in decimal or hexadecimal).

a) **%edx**

0x808d400

b) **0x24(%edx)**

0x808d424

c) **(%edx, %ecx)**

0x808d42c

d) **(%esi, %esi, 4)**

0xa0 (= 160_{10})

e) **0x28(%edx, %esi, 8)**

0x808d500

5. (5 points) Consider the function **mystery** with the header below and the assembly language below left. Write the *C* code for this function. Assume **x** is at **4(%esp)**, **y** is at **8(%esp)**, and **z** is at **12(%esp)** and that the result is returned in **%eax**. The **leal** instruction evaluates the "address" of its first operand and stores it into the register represented by its second operand.

```
int mystery(int x, int y, int *z);
```

```
 1.   movl    4(%esp),%edx
 2.   movl    8(%esp),%eax
 3.   leal    (%edx,%edx,2),%edx
 4.   leal    (%eax,%eax,4),%eax
 5.   negl    %eax
 6.   leal    (%eax,%edx,4),%edx
 7.   movl    12(%esp),%eax
 8.   movl    %edx,(%eax)
 9.   movl    $0x0,%eax
10.   ret
```

```
int mystery(int x, int y, int *z) {
   *z = 12 * x - 5 * y;
   return 0;
}
```

Answers

1. (5 points) The following function prototype

```
int decode4(int x, int y, int z);
```

generates the following IA32 assembly code:–

```
/* x at %esp+4, y at %esp+8, z at %esp+12, result in %eax
*/
   movl    8(%esp), %eax      // t1 = y
   andl    4(%esp), %eax      // t1 = x & y
   sarl    $3, %eax           // t2 = t1 >> 3
   subl    12(%esp), %eax     // t3 = t2 - z
   notl    %eax               // t4 = ~t3
   ret
```

Write C code that has the same effect as the assembly code.

```
int decode4(int x, int y, int z) {
    int t1 = x&y;
    int t2 = t1 >> 3;
    int t3 = t2-z;
    int t4 = ~t3;
    return t4;
}
```

2. (5 points) Which of the functions, shown below on the left, compiled into the assembly code on the right? Explain your answer

```
int fun1(int a, int b) {
   if (a < b)
      return a;
   else
      return b;
} // fun1(...)


int fun2(int a, int b) {
   if (b < a)
      return b;
   else
      return a;
} // fun2 (...)
```

```
int fun3(int a, int b){
   unsigned ua =
         (unsigned) a;
   if (ua < b)
      return b;
   else
      return ua;
} // fun3 ( ...)
```

```
 movl    4(%esp), %edx
 movl    8(%esp), %eax
 cmpl    %eax, %edx
 cmovae  %edx, %eax
 ret
```

fun3 — it is the only function of the three that does an unsigned compare. (Interestingly, fun1 and fun2 generate exactly the same machine code.)

3. (5 points) In the assembly code of Question 2, the traditional `push` and `pop` of `%ebp` have been optimized out. Explain why this is okay — i.e., why is saving of `%ebp` on the stack not necessary in this case?

   None of these functions calls another function or uses %ebp. Therefore, its value will be the same at the end of the function as when the function is called. Moreover, the compiler can find the arguments relative to %esp. That is, %ebp serves no purpose at all in this function, and its value does not change. Therefore, there is no point putting it on the stack, just to pop it off again.

4. (5 points) The following is the object code generated from a switch statement. Reverse engineer it to produce the C code for the function `int switch_prob(int x, int n)`.

```
   movl 4(%esp), %eax
   movl 8(%esp), %edx
   subl $50, %edx
   cmpl $5, %edx
   ja .L2
   jmp  *.L7(,%edx,4)
.L7:
   .long.L3
   .long.L2
   .long.L4
   .long.L3
   .long.L5
   .long.L6
.L4:
   sarl $3, %eax
   ret
.L3:
   sall $3, %eax
   .p2align 4,,6
   ret
.L5:
   leal (%eax,%eax,2), %eax
.L6:
   imull%eax, %eax
.L2:
   addl $12, %eax
   ret
```

```
int switch_prob(int x, int n){
    int result = x;
    switch(n) {
        case 52:
            result >>= 3;
            break;
        case 50:
        case 53:
            result <<= 3;
            break;
        case 54:
            result *= 3;
            /* Fall through */
        case 55:
            result *= result;
            /* Fall through */
        default:
            result += 12;
    }  //switch
    return result;
}   // int switch_prob
```

5. (5 points) Consider the following listing of assembly code for a C **for**-loop:–

```
loop:
   pushl %edi
   pushl %esi
   pushl %ebx
   movl 16(%esp), %esi
   movl 20(%esp), %edi
   cmpl %edi, %esi
   jle  .L4
   movl $1, %edx
.L3:
   movl %esi, %ebx
   subl %edx, %ebx
   leal (%edx,%edi), %ecx
   incl %edx
   cmpl %ecx, %ebx
   jg .L3
   movl %edx, %eax
.L2:
   popl %ebx
   popl %esi
   popl %edi
   ret
.L4:
   movl $1, %eax
   jmp  .L2
```

Based on this assembly code, fill in the blanks below with its corresponding C source code. (**Note**: you may only use the symbolic variables **x**, **y**, and **result** in your expressions below — *You may not use register names.*)

```
int loop(int x, int y) {
   int result;
   for (result = 0; x > y; result++ ) {
      x--;
      y++;
   }
   result++;
   return result;
} // loop(int x, int y)
```

CS-2011 Machine Organization and Assembly Language
Professor Hugh C. Lauer
Quiz date: Tuesday, April 15, 2014

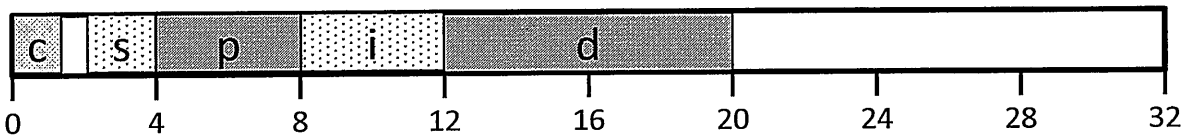
WPI, D-term 2014
Quiz #4 (20 points)

Answers

1. (5 points) (5 points) Consider the following **struct** defined in *C*.

```
typedef struct {
   char c;
   short s;
   double *p;
   int i;
   double d; } struct1;
```

Lay out on the following diagram the IA32 memory allocation of the members of this **struct** *as required by 32-bit Linux*. Preserve the natural alignments required of each member of the **struct**, but assume no re-ordering of the members. Indicate the wasted space. [Note that the **struct** itself must be aligned to the same byte multiple as its most restrictive member.]



Does this layout change for *32-bit Windows*? If so, explain how.

Yes. 32-bit Windows requires that doubles be aligned to 8-byte boundaries. Therefore, field d would start at byte 16 and continue to byte 24. The other fields would remain the same.

2. (5 points) Write a function called **copy_line()** that reads one line from **stdin** and writes it to **stdout**. Your function needs to work correctly for arbitrarily long input lines, and it needs to check for error and exceptional conditions. Use the library function **fgets()** to read the input from the stream **stdin**. The function **fgets()** is defined as follows:–

```
char *fgets(char *s, int size, FILE *stream);
```

The function **fgets** reads at most **size-1** characters from **stream** and stores them into the buffer pointed to by **s**. Reading stops after an **EOF** or a newline (**'\n'**). If a newline is encountered, it is stored into the buffer. A terminating null byte (**'\0'**) is stored after the last character in the buffer. **fgets()** returns **s** if the read was successful and **NULL** on error or when end of file occurs before any characters have been read.

[Warning: no matter how big your buffer is, the graders' input lines are longer!]

```
#define BUFSIZE 64
 void copy_line(){
   char buf[BUFSIZE];
   int i;
   while (1) {
      if (!fgets(buf, BUFSIZE, stdin))
```

```
        return; /* End of file or error */
    /* Print characters in buffer */
    for (i = 0; buf[i] && buf[i] != '\n'; i++)
        if (putchar(buf[i]) == EOF)
       return; /* Error */
    if (buf[i] == '\n') {
    /* Reached terminating newline */
       putchar('\n');
       return;
       }
    }
  }
```

There are many other correct answers.

3. (5 points) Consider the x86-64 architecture, in which `long` and `long int` both denote 64-bit (signed) integers and `int` denotes 32-bit (signed) integers. Will the following functions return the same result for all `x` and `y`? Explain your answer.

```
long int F1(int x, int y) {
   return (long) x + y;
} // long
```

```
long int F2(int x, int y) {
   return (long) (x + y);
} // long
```

They are not the same. F1 does long arithmetic, because the (long) cast is applied to x, hence the arithmetic is promoted to long. F2 does int arithmetic to add x and y, then casts the result to long. The int arithmetic is subject to rounding adjustment before casting.

If x = y = $Tmin_{32}$, F1 returns a result of 2* $Tmin_{32}$, whereas F2 returns a result of zero.

4. (5 points) Explain the concept of *canary* and how it can be used to thwart buffer overflow attacks. What does the term suggest to you?

The term *canary* suggests the "canary in the coal mine." When the canary stopped signing, it was an indication of the presence of explosive gases.

In the case of the buffer overflow, a program (or the compiler) can insert a non-predictable value immediately after a character array. In the case of a buffer overrun, the value is corrupted, and the corruption can be detected by comparison with the original value (which is kept is a separate segment). The non-predictability is important, so that hackers cannot thwart the canary by inserting its own value while overrunning the buffer.

5. (10 points) Consider the following source code, where **M** and **N** are constants specified with **#define**:–

```
int matrix1[M][N];
int matrix2[N][M];

int copy_element(int i, int j) {
   matrix1[i][j] = matrix2[j][i];
} //copy_element(...)
```

This generates the following x86-64 assembly code:–

```
copy_element:
   movslq  %esi, %rsi          // %rsi = j
   movslq  %edi, %rdi          // %rdi = i
   movq %rdi, %rax
   salq $4, %rax               // 16 * i
   addq %rdi, %rax             // 17 * i
   addq %rsi, %rax             // j + 17 * i
   leaq (%rsi,%rsi,2), %rdx    // 3 * j
   leaq (%rsi,%rdx,4), %rdx    // j + 4 * 3 * j = 13 * j
   addq %rdx, %rdi             // jth row
   movl matrix2(,%rdi,4), %edx // ith column
   movl %edx, matrix1(,%rax,4) // jth column, ith row
   ret
```

What are the values of **M** and **N**? Explain your answer. You may annotate the assembly code above to support your explanation.

M = 13
N = 17

CS-2011 Machine Organization and Assembly Language
Professor Hugh C. Lauer
Quiz date: Tuesday, April 29, 2014

WPI, D-term 2014
Quiz #5 (20 points)

Answers

1. (5 points) Mark up the following *x86_64* assembly code to indicate which registers are *caller save*, which registers are *callee save*, how many arguments there are and which registers hold them, and which register contains the return value. Also indicate what this function does.

```
mystery:
   movq %rbx, -16(%rsp)
   movq %rbp, -8(%rsp)
   subq $24, %rsp
   movq %rdi, %rbx
   movl $0, %eax
   testq%rdi, %rdi
   je .L2   //test for zero
   movb $1, %al //%rax = 1
   cmpq $1, %rdi
   je .L2   //test for one
   leaq -1(%rdi), %rdi
   call mystery
   movq %rax, %rbp
   leaq -2(%rbx), %rdi
   call mystery
   addq %rbp, %rax //%rax =
      mystery(%rdi-1) +
      mystery(%rdi-2)
.L2:
   movq 8(%rsp), %rbx
   movq 16(%rsp), %rbp
   addq $24, %rsp
   ret
```

%rbx. %rbp are callee save and are used as temporaries in the function mystery.

%rdi is the only argument.

%rax is the return value.

This is a recursive version of
int fibonacci(int n),
which returns the nth Fibonacci number.

2. (5 points) Explain the difference between a *direct mapped* cache and a *two-* or *four-way set associative cache.* Are there any circumstances in which a cache with less associativity would have superior performance to a cache with more associativity, given the same number of bits/bytes of cache memory? If not, why would a system designer even consider building direct mapped caches?

A direct mapped cache is one in which there is only one cache line per set; a two-way associative cache has two cache lines per set, and a four-way associative cache has four lines per set. In a direct mapped cache, if two memory locations collide in the same set, one causes the other to be evicted. This could be a costly performance hit in the unfortunate case that two very active memory locations map to the same cache location.

In a multi-way associative cache, each set can hold multiple cache lines at the same time (the number being the "way" of the associativity). Therefore, when two or more locations map to the same set, the cache can hold them both. This minimizes the number of evictions due to pure coincidence.

However, a multi-way cache requires more hardware circuitry, because the tag of the memory location has to be compared with all of the tags of the

set at the same time, in order to identify which (if any) of the cache lines in the set matches the given memory location. Therefore, direct mapped caches save time and chip space, which may or may not be important.

It is difficult to imagine a direct-mapped cache outperforming a set associative cache, because every pattern of cache hits, misses, and evictions of the direct mapped cache can be mimicked by a set associative cache, and possibly improved upon.

3. (5 points) In the memory hierarchy of a particular processor, data items can be retrieved from the Level 1 (L1) cache in 4 cycles. However, on a L1 cache miss, it takes 24 cycles to transfer a data item from the L2 cache to the L1 cache. If the L2 cache misses, it takes an average of 120 cycles to retrieve the item from main memory (D-RAM). Assume that an average of 4% accesses to the L1 cache result in misses, and assume that an average of 1% of the accesses to the L2 cache result in misses, what is the average time it takes to retrieve a data item? (You may ignore page faults.)

Average Access Time$_{L1}$ = hit time$_{L1}$ + miss rate$_{L1}$×miss penalty$_{L1}$.

Miss penalty$_{L1}$ = Average Access Time $_{L2}$ = hit time$_{L2}$ + miss rate$_{L2}$ × miss penalty$_{L2}$

Miss penalty$_{L2}$ = Average Access Time$_{D\text{-}RAM}$ = 120 cycles.

Average Access Time $_{L2}$ = 24 + 0.01 × 120 = 25.2 cycles

Average Access Time$_{L1}$ = 4 + 0.04 × 25.2 = 4 + 1.008 = 5.008 cycles

4. (5 points) Suppose that a 16-megabyte file consisting of 4096-byte logical blocks is stored on a disk drive with the following characteristics:

Parameter	Value
Rotational rate	7,500 RPM
$T_{avg\ seek}$	4 milliseconds
Average # sectors/track	2048
Surfaces	8
Sector size	4096 bytes

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first block is $T_{avg\ seek} + T_{avg\ rotation}$.

a) Estimate the fastest time in milliseconds required to read the entire file with the best possible mapping of logical blocks to disk sectors. (You may assume that the seek time from one track to the immediately adjacent track is negligible.)

The file consists of 4096 blocks of 4096 bytes each. The best mapping would be to lay out the logical blocks of the file consecutively within one track and on two separate tracks within the same *cylinder* so that they can be read without moving the head. 7500 RPM = 125 rotations per second ⇒ 8 ms per rotation. Therefore $T_{avg\ rotation}$ = 4 ms.

The total time to read all 4096 blocks at maximum speed is therefore $T_{avg\ seek} + T_{avg\ rotation}$ + 2 total rotations = 4 + 4 + 8*2 = 24 milliseconds

b) Estimate the time in milliseconds required to read the entire file if its logical blocks are scattered randomly across all of the sectors of the disk.

If the blocks are scattered randomly, the time to access each block is about $T_{avg\ seek} + T_{avg\ rotation}$ = 8 ms. Since there are 4096 blocks, this works out to slightly more than 32 seconds!

5. (5 points) Consider the following program, which is based on code from the lectures:–

```
/* main.c */
int buf[2] = {1, 2};

int main(){
   swap();
   return 0;
} // int main()
```

```
/* swap.c */
#include <stdio.h>
extern int buf[];
char main;
int *bufp0 = &buf[0];
static int *bufp1;

void swap(){
   int temp = *bufp0;
   bufp1 = &buf[1];
   *bufp0 = *bufp1;
   *bufp1 = temp;
   printf("0x%x\n", main);
} // void swap()()
```

When this program is compiled and executed on a 32-bit Linux system, it prints the string `"0x55\n"` and terminates normally, even though `swap` never initializes the variable `main`. Explain this.

The function "main" in main.c is a strong symbol. The global "char main" in swap.c is a weak symbol. Under Rule 2 of the Linker's Symbol Rules, the weak symbol resolves to the strong one. Therefore, the character main in swap.c is resolved to the executable function main() in main.c.

The first instruction of main() happens to be "push %ebp"; this instruction occupies a single byte and happens to have the value 0x55.