CS237 Midterm Exam

Practice

This is a closed-book exam (no calculators, no phones, no outside materials). You have 1 hour and 30 minutes to complete the exam. All intended answers should fit in the space provided. You may use the backs of exam pages as scratch paper if you wish, but mark your final answers clearly on the exam. When you are done, submit your exam along with the signed honor code statement at the bottom of this page.

Be sure to give yourself enough time to answer each question as best you can. The point values should help you to plan your time. Partial credit will be given when appropriate, although the most credit will be given to the most correct answers.

Question	Value	Score
Bitwise and logical operations	8	
Twos complement	10	
Floating point	14	
Matching Assembly	8	
Evaluating Assembly	10	
Converting C	14	
Arrays	8	
Total	**72**	

I have neither given nor received unauthorized aid on this examination.

Signature: _____

Name: SAMPLE SOLUTIONS

Problem 1. (8 points):

Suppose `char x` and `char y` have byte values `0x63` and `0xA9`. Fill in the following table with the byte values of the following C expressions. You may express byte values in either binary or hexadecimal format.

Expression	Value
`x & y`	0010 0001 or 0x21
`x && y`	0000 0001 or 0x01
`x | y`	1110 1011 or 0xEB
`x || y`	0000 0001 or 0x01
`~x | ~y`	1101 1110 or 0xDE
`!x || !y`	0000 0000 or 0x00
`x & !y`	0000 0000 or 0x00
`x && ~y`	0000 0001 or 0x01
`x ^ x`	0000 0000 or 0x00
`x ^ ~x`	1111 1111 or 0xFF

Scratch space.

Problem 2. (10 points):

Consider a **8-bit** two's complement representation. Fill in the empty boxes in the following table:
(Note: all answers should be expressed as values that fall within the valid 8-bit two's complement range)

Number	Binary Representation	Decimal Representation
Zero	0000 0000	0
-1	1111 1111	-1
$1100\ 0100_2$	1100 0100	-60
TMin	1000 0000	-128
TMax	0111 1111	127
TMax+1	1000 0000	-128
64 + 96	1010 0000	-96

Scratch space.

Problem 3. (14 points):

The following procedure takes a double-precision floating point number in IEEE format, converts it to an integral value that has the same bit representation, and then prints out information about what category of number it is. Fill in the missing code so that it performs this classification correctly. (Recall that double-precision floating point numbers have a 52-bit `frac` field, an 11-bit `exp` field, and a bias of $2^{10} - 1 = 1023$. You may find it useful to think about the value of a floating point number v as $v = (-1)^s \times M \times 2^E$.)

```
void classify_float(double d) {
  /* Initialize unsigned value u to have the same bit pattern as d */
  unsigned long u = *(unsigned long *) &d;

  /* Split u into the different parts */
  int sign = (u >> 63) & 0x1;      // The sign bit

  int exp  = (u >> 52) & 0x7FF;    // The exp field (0x7FF for 11 ones)

  int frac = u & 0xFFFFFFFFFFFFF;  // The frac field (14 F's for 52 ones)

  /* The remaining expressions can be written in terms of the
  values of sign, exp, and frac */

  if (exp == 0 && frac == 0) {
    printf("Plus or minus zero");

  } else if (exp == 0 && frac != 0) { // frac != 0 optional
    printf("Nonzero, denormalized");

  } else if (exp == 0x7FF && frac == 0) {
    printf("Plus or minus infinity");

  } else if (exp == 0x7FF && frac != 0) { // frac != 0 optional
    printf("NaN");

  } else if (exp >= 1023) { // exp - 1023 >= 0
    printf("Normalized, |v| >= 1.0");
  } else {
    printf("Normalized, |v| < 1.0");
  }
}
```

Scratch space.

Problem 4. (8 points):

Match each of the assembler routines on the left with the equivalent C function on the right.

```
foo1:
    movq %rdi, %rax
    andq $15, %rax
    ret
```

```
foo2:
    movq %rdi, %rax
    salq $4, %rax
    subq %rdi, %rax
    ret
```

```
foo3:
    leaq 15(%rdi), %rax
    testq %rdi, %rdi
    cmovns %rdi, %rax  # see footnote
    sarq $4, %rax
    ret
```

```
foo4:
    movq %rdi, %rax
    salq $4, %rax
    ret
```

```
int choice1(int x) {
    return (x < 0);
}
```

```
int choice2(int x) {
    return x / 15;
}
```

```
int choice3(int x) {
    return x * 15;
}
```

```
int choice4(int x) {
    return x % 16;
}
```

```
int choice5(int x) {
    return x / 16;
}
```

```
int choice6(int x) {
    return x * 16;
}
```

Fill in your answers here:

foo1 corresponds to choice 4.
foo2 corresponds to choice 3.
foo3 corresponds to choice 5 (if x negative, add bias to round toward zero).
foo4 corresponds to choice 6.

*`cmovns`: conditional move if non-negative

Problem 5. (10 points):

Assume the following values are stored at the indicated registers and memory addresses:

Address	Value
0x100	0xEE
0x108	0xA8
0x110	0x33
0x118	0x11

Register	Value
%rax	0x100
%rcx	0x1
%rdx	0x3
%r12	0x0

Fill out the following table shosing the effects of the following instructions. The destination should either be a register (e.g, `%rax`) or a memory address (e.g., `0x100`). The first entry has been completed for you.

Instruction	Destination	Value
`movq $0x8, %r12`	`%r12`	`0x8`
`addq %rcx, (%rax)`	`0x100`	`0xEE + 0x11 = 0xEF`
`subq %rdx, 8(%rax)`	`0x108`	`0xA8 - 0x03 = 0xA5`
`movq (%rax,%rdx,8), %rcx`	`%rcx`	`0x11`
`leaq (%rax,%rdx,8), %rcx`	`%rcx`	`0x118`

Problem 6. (14 points):

A function `func` has the following general structure:

```
long func(unsigned long x) {
    long val = 0;
    while ( ... ) {
        .
        .
        .
        .
    }
    return ... ;
}
```

Suppose the assembly code below is generated by the compiler:

```
func:
        movl    $0, %eax
        jmp     .L5
.L6:
        xorq    %rdi, %rax
        shrq    $1, %rdi
.L5:
        testq   %rdi, %rdi
        jne     .L6
        andq    $1, %rax
        ret
```

(*Note:* `x` is stored in register `%rdi`)

Now, use your knowledge of C and assembly to do the following:

- Use the assembly code to fill in the missing C code. Rewrite the completed function on the next page.
- Below your code, provide an English-language description of the function's purpose. In other words, provide a sentence that would be suitable for use as a top-level function comment; do not give a line-by-line description of C syntax.

```
long func(unsigned long x) {
    long val = 0;


    while (x != 0) {
        val = val ^ x;
        x = x >> 1;
    }

    return val & 1;

}
```

```
// Intermediate w/ goto
  val = 0
  goto .L5;

.L6:
  val = val ^ x;
  x >>= 1

.L5:
  if (x != 0)
    goto .L6
  return val & 1;
```

Description of `func` (in English):

Returns 'true' if the number of 1's in the bitwise representation of x is odd, 'false' if even (aka bitwise parity).

Problem 7. (8 points):

Consider the source code below, where `M` and `N` are constants declared with `#define`.

```
long mat1[M][N];
long mat2[N][M];

long sum_element(long i, long j)
{
    return mat1[i][j] + mat2[j][i];
}
```

This generates the following assembly code:

```
sum_element:                    # i = rdi, j = rsi
        leaq 0(,%rdi,8), %rdx   # rdx = 8*rdi
        subq %rdi,%rdx          # rdx = 7*rdi
        addq %rsi, %rdx         # rdx = 7*rdi + rsi
        leaq (%rsi,%rsi,4), %rax # rax = 5*rsi
        addq %rax, %rdi         # rdi = rdi + 5*rsi
        movq mat2(,%rdi,8), %rax # rax = mat2 + 8*(5*rsi + rdi)
        addq mat1(,%rdx,8), %rax # rax += mat1 + 8*(7*rdi + rsi)
        ret
```

(Recall that registers `%rdi` and `%rsi` are used to pass the first two integral arguments.)

What are the values of `M` and `N`? Show your work and justify your answers. If you are stuck, think about drawing a picture of the arrays in memory.

`M = 5`

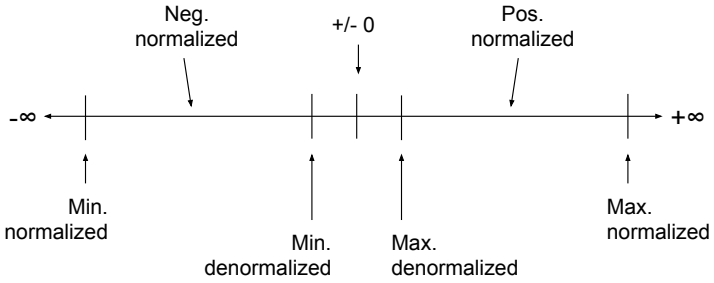
`N = 7`

Powers of 2:

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Powers of 16:

16^{3}	16^{2}	16^{1}	16^{0}
4096	256	16	1



When distinguishing between the floating point representations of the special values for infinity and NaN, NaN has a non-zero `frac` field.

C Declaration	Intel data type	Assembly code suffix	Size (bytes)
char	byte	b	1
short	word	w	2
int	double word	l	4
long	quad word	q	8
char *	quad word	q	8
float	single precision	s	4
double	double precision	l	8

There are sixteen 64-bit registers in `x86-64`: `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rdi`, `%rsi`, `%rbp`, `%rsp`, and `%r8-r15`.

Registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` are used to pass the first six integer or pointer parameters to procedures. Register `%rax` is used to pass the return value.

Registers `%rax`, `%rcx`, `%rdx`, `%rdi`, `%rsi`, `%rsp`, and `%r8-r11` are caller-save registers, `%rbx`, `%rbp`, and `%r12-r15` callee-save registers, and `%rsp` is the stack pointer.

Instruction	Description
`cmpX` S, D	computes $(D - S)$ and sets the condition codes.
`testX` S, D	computes $(S \& D)$ and sets the condition codes.
`subX` S, D	computes $(D - S)$ and stores the result in D.
`addX` S, D	computes $(S + D)$ and stores the result in D.
`sal` k, D	left shifts D by k and stores the result in D.
`sar` k, D	right shifts D by k (arithmetic) and stores the result in D.
`shr` k, D	right shifts D by k (logical) and stores the result in D.
`lea` S, D	computes $\&S$ and stores the result in D.
`cmovX` S, D	moves the contents of the register S to register D if and only if some condition is met.

Instruction	Jump Condition
`je`	`ZF`
`jne`	`~ZF`
`js`	`SF`
`jns`	`~SF`
`jg`	`~(SF ^ OF) & ~ZF`
`jge`	`~(SF ^ OF)`
`jl`	`SF ^ OF`
`jle`	`(SF ^ OF) | ZF`
`ja`	`~CF & ~ZF`
`jae`	`~CF`
`jb`	`CF`
`jbe`	`CF | ZF`
`jmp`	`unconditional jump`

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111