CSE351 Autumn 2013 – Midterm Exam (30 Oct 2013)

Please read through the entire examination first! We designed this exam so that it can be completed in 50 minutes and, hopefully, this estimate will prove to be reasonable.

There are 6 problems for a total of 100 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided. If you need more space (you shouldn't), you can write on the back of the sheet where the question is posed, but please make sure that you indicate clearly the problem to which the comments apply. Do NOT use any other paper to hand in your answers. If you have difficulty with part of a problem, move on to the next one. They are independent of each other.

The exam is CLOSED book and CLOSED notes. Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Name: ____*Sample Solution*______

ID#: ________________________

Problem	Max Score	Score
1	10	***10***
2	10	***10***
3	25	***25***
4	20	***20***
5	25	***25***
6	10	***10***
TOTAL	**100**	***100***

1. Number Representation – Integers (10 points)

A. Explain why we have a Carry-Flag and an Overflow-Flag in x86 condition codes. What is the difference between the two? (Explain in at most two sentences.)

(4 points)
The carry flag is used for unsigned numbers and indicates a carry-out of 1 during addition from the most-significant-bit. The overflow flag applies to signed arithmetic and indicates that the addition yielded a number that was too large a positive or too small a negative value.

B. Add 11011001 and 01100011 as two's complement 8-bit integers & convert the result to decimal notation.

(3 points)

$$\begin{array}{rrcrr} & 11011001 & = & & -39 \\ + & 01100011 & = & + & +99 \\ \hline & 00111100 & = & & +60 \end{array}$$

C. Convert your answer from the previous problem to a 2-digit hex value.

(3 points)
60 = 0x3c

2. Number Representation – Floats (10 points)

For this question, assume we are working with a 64-bit architecture.

A. For each of the casts below, circle T if a loss of precision is possible and F, otherwise.

(4 points)

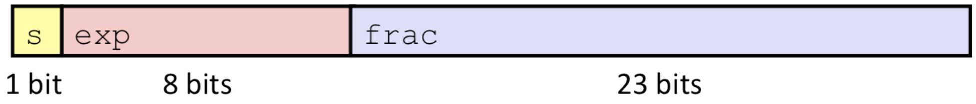
int → float [✓ T] F

float → int [✓ T] F

double → int [✓ T] F

int → double T [✓ F]

B. This is how single-precision floating point numbers are stored in memory.



Fill in the corresponding fields for the two numbers below. Please be sure to show the bits by writing “0”, “1”, “all 0s”, “all 1s”, or a pattern of 0s and 1s in the spaces provided.

(2 points)

0 (zero):

s = *0*

exp = *all 0s*

frac = *all 0s*

- ∞ (negative infinity):

s = *1*

exp = *all 1s*

frac = *all 0s*

C. Consider the following code snippet where the variables a and b are both floats.

```
if (a + (b – b)  == a) { printf(“Equals a\n”); }
if ((a + b) – b) == 0) { printf(“Equals 0\n”); }
```

Suppose the user runs this program and sees the following output:

```
Equals a
Equals 0
```

How is this possible given that addition and subtraction are associative, for example, (a + b) + c is equal to a + (b + c)? (Two sentences max.)

(2 points)
Because, the representation and range of floating point number representations are finite, associativity no longer can be relied upon. If the value b is very large and a very small, then the representation will not enough precision to represent a + b as different than just b and we will see the behavior above.

Give an example value for both a and b in decimal that would generate this output (do **not** consider the case where a or b are equal to 0)?

(2 points)

a = ____*1.0E-20*____ b = ____*1.0E20*____

3. C to Assembly Code (25 points)

Write x86-64 assembly instructions (see appendix for the list of instructions that you can use) that might be generated by the C code for the function `foo` (note: you are not being asked to write any code for the function `bar` which you can simply assume is at label `bar`). It may be a good idea to consult the register chart provided at the back of this exam.

```
int bar(int c) { … }

int foo(int a, int b) {
  int x;
  x = bar(a >> 4);
  if (x != 0)
    return x;
  else
    return b;
}
```

Place the assembly code for function `foo` here (you should only need between 5 and 10 instructions) and add a comment to each line:

```
  push   %rsi          ; save b (passed in %rsi) on the stack
                       ; must do this as bar may use %rsi
                       ; %rsi is a caller-saved register
  sar    $0x4,%rdi     ; a is in %rdi so shift it right by 4
                       ; and leave result there so it is ready
                       ; to be passed as sole argument to bar
  call   bar           ; bar(a >> 4), its return value will be
                       ; in %rax and this will be the variable x
  pop    %rsi          ; restore b, saved value of original %rsi
  test   %rax,%rax     ; test x to set condition codes
                       ; could be done with "cmp $0, %rax"
  jne    end           ; jump to return if x != 0
                       ; return value of x is already in %rax
  mov    %rsi, %rax    ; then, if x == 0
                       ; get b from %rsi and put in %rax
                       ; as return value in this branch

end:
  ret
```

4. Assembly Code to C (20 points)

Given the following assembly instruction for the function 'mystery', on a IA32 32-bit architecture derive the C code for the function (you can assume all values are signed integers).

```
mystery:
  pushl   %ebp
  movl    %esp,%ebp
  movl    8(%ebp),%eax
  addl    %eax,%eax
  addl    8(%ebp),%eax
  addl    $2,%eax
  subl    12(%ebp),%eax
  popl    %ebp
  ret
```

Please write the code for the function below (make sure to include return and parameter types; the body of the function should only need to be a few C statements at most) and add a comment to each statement you write.

```
int mystery(int a, int b) {

  int c = a     // temp variable c initially a
                // read first argument at 8(%ebp)
  c = c + c;    // c = 2*a
                // first add instruction
  c = c + a;    // c = 3*a
                // second add instruction
  c = c + 2;    // c = 3*a + 2
                // third add instruction
  c = c – b;    // c = 3*a + 2 – b
                // subtract 2nd argument at 12(%ebp)
  return c;

}
```

or, more simply,

```
int mystery(int a, int b) {

  return (3*a + 2 – b);

}
```

5. Stack Discipline (25 points)

The following table shows the contents of a part of stack memory just after calling a function in an x86-64 architecture.

Memory address	Value
`0x7fffffffffffad8`	`0xf00`
`0x7fffffffffffad0`	`0x7fffffffffffb00`
`0x7fffffffffffac8`	`0xcable`
`0x7fffffffffffac0`	`0xface`
`0x7fffffffffffab8`	`0xdeadbeef`
`0x7fffffffffffab0`	`0x12`
`0x7fffffffffffaa8`	`0x3`

```
%rbp = 0x7fffffffffffad0
%rsp = 0x7fffffffffffaa8
```

A. Assuming a 32-bit stack discipline, what is the size of the stack frame for the function shown in the diagram?

(8 points)
%rbp points to the first element of the stack frame,
%rsp points to the last,
therefore there are 6 entries in all or 48 bytes in the stack frame

B. What is the value of the program counter (%rip) after the function returns?

(4 points)
The return address is just above the stack frame, therefore the function returns to address 0xf00

C. Suppose that no parameters were passed into the function on the stack. What are the values of rsp and rbp when this function returns?

(7 points)
The base pointer is restored from the stack, it is the first value in the stack frame at the address of the current %rbp, therefore the new value of %rbp will be 0x7fffffffffffb00. The stack pointer will be pointing to the top of the stack just after popping off the return address, therefore %rsp will be 0x7fffffffffffae0.

D. The first 4 lines of assembly for this particular function are below. What is the value in register rbx after executing the fourth line of the assembly below?

```
push %rbp
mov %rsp, %rbp
push %rbx
mov $20, %rax
```

(6 points)

______*0xcab1e*________

6. Structs (10 points)

Suppose you are given the following struct definition for an x86-64 architecture which is used to implement a linked list of student records.

```
typdef struct node{
    char first [15];
    double gpa;
    int id;
    char last [15];
    node* next;
} student;
```

A. Given the following diagram for the bytes of this struct, specify the **byte offsets** of each of the **five** fields in the space provided below each and the **size** of the **two** shaded areas of internal fragmentation (wasted space) below them.

(4 points)

	first	■	gpa	id	last	■	next
Offset	*0*		*16*	*24*	*28*		*48*
Space		*1*				*5*	

B. What is the size of the struct?

(2 points)

56 bytes

C. How much internal fragmentation does this struct have?

(2 points)

6 bytes

D. How much external fragmentation does this struct have?

(2 points)

0 bytes

REFERENCES

Powers of 2:

2^{0} = 1	
2^{1} = 2	2^{-1} = .5
2^{2} = 4	2^{-2} = .25
2^{3} = 8	2^{-3} = .125
2^{4} = 16	2^{-4} = .0625
2^{5} = 32	2^{-5} = .03125
2^{6} = 64	2^{-6} = .015625
2^{7} = 128	2^{-7} = .0078125
2^{8} = 256	2^{-8} = .00390625
2^{9} = 512	2^{-9} = .001953125
2^{10} = 1024	2^{-10} = .0009765625

Assembly Code Instructions:

`push`	push a value onto the stack and decrement the stack pointer
`pop`	pop a value from the stack and increment the stack pointer
`call`	jump to a procedure after first pushing a return address onto the stack
`ret`	pop return address from stack and jump there
`mov`	move a value between registers and memory
`lea`	compute effective address and store in a register
`add`	add src (1st operand) to dst (2nd) with result stored in dst (2nd)
`sub`	subtract src (1st operand) from dst (2nd) with result stored in dst (2nd)
`and`	bit-wise AND of src and dst with result stored in dst
`or`	bit-wise OR of src and dst with result stored in dst
`sar`	shift data in the dst to the right (arithmetic shift) by the number of bits specified in 1st operand
`jmp`	jump to address
`jne`	conditional jump to address if zero flag is not set
`cmp`	subtract src (1st operand) from dst (2nd) and set flags
`test`	bit-wise AND src and dst and set flags

Register map for x86-64:

Note: all registers are caller-saved except those explicitly marked as callee-saved, namely, `rbx`, `rbp`, `r12`, `r13`, `r14`, and `r15`. `rsp` is a special register.

`%rax`	Return value	`%r8`	Argument #5
`%rbx`	Callee saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Caller saved
`%rdx`	Argument #3	`%r11`	Caller Saved
`%rsi`	Argument #2	`%r12`	Callee saved
`%rdi`	Argument #1	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved