CSE 351 Midterm - Winter 2015

February 09, 2015

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 4 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 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. 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 (no summary sheets, no calculators, no mobile phones, no laptops). 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.

Good Luck!

Name: ______________________

Student ID: ______________________

Section: ______________________

Problem	Max Score	Score
1	20	
2	20	
3	25	
4	35	
EC	15	
TOTAL	100	

1. Number Representation (20 points)

Integers

(a) Assuming unsigned integers, what is the result when you compute UMAX+1?

(b) Assuming two's complement signed representation, what is the result when you compute TMAX+1?

Floating Point

(c) Give M and E in the floating point representation of 3.75. Express each in both decimal and binary. (Remember, E is the actual value of the exponent, not the encoding with bias)

	Binary	Decimal
E		
M		

(d) Is the '==' operator a good test of equality for floating point values? Why or why not?

Casting and Pointers

(e) Given the following code:

```
float f = 5.0;
int i = (int) f;
int j = *((int *)&f);
```

Does `i==j` return true or false? Explain.

2. Assembly and C (20 points)

Consider the following x86-64 assembly and C code:

```
<do_something>:
    test   %rsi,%rsi
    _____  <end>
    xor    %rax,%rax
    sub    $0x1,%rsi

<loop>:
    lea    (%rdi,%rsi,_____),%rdx
    add    (%rdx),%ax
    sub    $0x1,%rsi
    jns    <loop>

<end>:
    retq


int do_something(short* a, int len) {
    int result = 0, i;
    for (i =______; i >= 0 ;______) {
        ______________;
    }
    return result;
}
```

(a) Both code segments are implementations of the unknown function `do_something`. Fill in the missing blanks in both versions. (Hint: `%rax` and `%rdi` are used for `result` and `a` respectively. `%rsi` is used for both `len` and `i`)

(b) Briefly describe the value that `do_something` returns and how it is computed. Use only variable names from the C version in your answer.

3. Pointers and Values (25 points)

Consider the following variable declarations:

```
int x;
int y[11]  = {0,1,2,3,4,5,6,7,8,9,10};
int z[][5] = {{210, 211, 212, 213, 214}, {310, 311, 312, 313,314}};
int aa[3]  = {410, 411, 412};
int bb[3]  = {510, 511, 512};
int cc[3]  = {610, 611, 612};
int *w     = {aa, bb, cc};
```

Variable	Address
aa	0x000
bb	0x100
cc	0x200
w	0x300
x	0x400
y	0x500
z	0x600

(a) Fill in the table below with the address, value, and type of the given C expressions. Answer N/A if it is not possible to determine the address or value of the expression. The first row has been filled in for you.

C Expression	Address	Value	Type (int/int*/int**)
`x`	0x400	N/A	int
`*&x`			
`y`			
`*y`			
`y[0]`			
`*(y+1)`			
`&(y[10])`			
`z[0]+1`			
`*(z[0]+1)`			
`z[0][6]`			
`w[1]`			
`w[2][0]`			

4. Recursion (35 points)

The fictional Fibonatri sequence is defined recursively for `n=0,1,...` by the following C code:

```
int fibonatri(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else if (n == 2) {
        return 2;
    } else {
        return fibonatri(n-3) - fibonatri(n-2) + fibonatri(n-1);
    }
}
```

Here is a disassembly of `fibonatri()`:

```
000000000040057b <fibonatri>:
  40057b:       53                      push   %rbx
  40057c:       48 83 ec 10             sub    $0x10,%rsp
  400580:       89 7c 24 0c             mov    %edi,0xc(%rsp)
  400584:       83 7c 24 0c 00          cmpl   $0x0,0xc(%rsp)
  400589:       75 07                   jne    400592 <fibonatri+0x17>
  40058b:       b8 00 00 00 00          mov    $0x0,%eax
  400590:       eb 4c                   jmp    4005de <fibonatri+0x63>
  400592:       83 7c 24 0c 01          cmpl   $0x1,0xc(%rsp)
  400597:       75 07                   jne    4005a0 <fibonatri+0x25>
  400599:       b8 01 00 00 00          mov    $0x1,%eax
  40059e:       eb 3e                   jmp    4005de <fibonatri+0x63>
  4005a0:       83 7c 24 0c 02          cmpl   $0x2,0xc(%rsp)
  4005a5:       75 07                   jne    4005ae <fibonatri+0x33>
  4005a7:       b8 02 00 00 00          mov    $0x2,%eax
  4005ac:       eb 30                   jmp    4005de <fibonatri+0x63>
  4005ae:       ?? ?? ?? ??             ___________________________
  4005b2:       83 e8 03                sub    $0x3,%eax
  4005b5:       89 c7                   mov    %eax,%edi
  4005b7:       e8 bf ff ff ff          callq  40057b <fibonatri>
  4005bc:       89 c3                   mov    %eax,%ebx
  4005be:       8b 44 24 0c             mov    0xc(%rsp),%eax
  4005c2:       83 e8 02                sub    $0x2,%eax
  4005c5:       89 c7                   mov    %eax,%edi
  4005c7:       ?? ?? ?? ??             ___________________________
  4005cc:       29 c3                   sub    %eax,%ebx
  4005ce:       8b 44 24 0c             mov    0xc(%rsp),%eax
  4005d2:       ?? ?? ?? ??             ___________________________
  4005d5:       89 c7                   mov    %eax,%edi
  4005d7:       e8 9f ff ff ff          callq  40057b <fibonatri>
  4005dc:       ?? ?? ?? ??             ___________________________
  4005de:       48 83 c4 10             add    $0x10,%rsp
  4005e2:       5b                      pop    %rbx
  4005e3:       c3                      retq
```

(a) Fill in the four blanks in the disassembly. You should be able to gather hints from the surrounding code.

(b) What register is used to pass the single argument to `fibonatri()`?

(c) Why is the register `%rbx` pushed onto the stack at the beginning of the function?

(d) Why are iterative solutions generally preferred over recursive solutions from a memory usage perspective? How much of the stack is used during each iteration of `fibonatri()`?

(e) What pattern do numbers in the Fibonatri sequence follow?

Extra Credit (15 points)

Write a non-recursive function in C with the same output as `fibonatri()` using only a switch statement (Hint: use the modulus `%` operator)

References

Powers of 2:

$2^0 = 1$	
$2^1 = 2$	$2^{-1} = 0.5$
$2^2 = 4$	$2^{-2} = 0.25$
$2^3 = 8$	$2^{-3} = 0.125$
$2^4 = 16$	$2^{-4} = 0.0625$
$2^5 = 32$	$2^{-5} = 0.03125$
$2^6 = 64$	$2^{-6} = 0.015625$
$2^7 = 128$	$2^{-7} = 0.0078125$
$2^8 = 256$	$2^{-8} = 0.00390625$
$2^9 = 512$	$2^{-9} = 0.001953125$
$2^{10} = 1024$	$2^{-10} = 0.0009765625$

Hex help:

$0x00 = 0$
$0x0A = 10$
$0x0F = 15$
$0x20 = 32$
$0x28 = 40$
$0x2A = 42$

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 (1^{st} operand) to dst (2^{nd}) with result stored in dst (2^{nd})
`sub`	subtract src (1^{st} operand) from dst (2^{nd}) with result stored in dst (2^{nd})
`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 1^{st} operand
`jmp`	jump to address
`jne`	conditional jump to address if zero flag is not set
`jns`	conditional jump to address if sign flag is not set
`cmp`	subtract src (1^{st} operand) from dst (2^{nd}) 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