1F

UCLA Computer Science 33 (Fall 2015)
Midterm 1
99 points, 99 minutes, open book, open notes.
Questions are equally weighted (11 min. each).
Use a separate sheet of paper for each answer.
Put a big problem number at each sheet's top.
Turn in your sheets in increasing numeric order.

Name: Yeun Hoon Lee Student ID: 104469?6

1	2	3	4	5	6	7	8	9	sum
11	11	11	11	11	11	2	9	8	85

1 (11 minutes). You want to create a repeated bit pattern in a 64-bit unsigned word. The pattern repeats every 8 bits. For example, repeating the bit-pattern 10011011 would yield the word 0x9b9b9b9b9b9b9b9b. Write a C function rbp(p) that returns such a word, given an 8-bit pattern p. Have your function execute as few instructions as possible.

2 (11 minutes). The PDP-11 architecture is "mixed-endian": within a 16-bit short word, the least significant byte comes first, whereas within a 32-bit long word, the *most* significant short word comes first. Diagram how the signed 32-bit number -25306982 (-0x1822766) is represented as a series of unsigned 8-bit bytes (a) on a PDP-11, (b) on an x86-64 machine, and (c) on a bigendian machine like the SPARC. Your diagram should list the offset of each byte.

3 (11 minutes). Consider these two functions:

```
#include <stdbool.h>
bool pushme (unsigned long v) {
  return 255 <= (v >> 3);
}
bool pullyou (long v) {
  return ! (0 <= (v >> 3) && (v >> 3) < 255);
}
```

and this assembly-language implementation:

```
pushme: cmpq    $2039, %rdi
        seta    %al
        ret
pullyou:cmpq    $2039, %rdi
        seta    %al
        ret
```

a. Explain why those "2039"s are correct, even though the source code does not mention 2039.

b. How can pushme and pullyou have identical machine code, even though the functions have different types and implementations? Explain.

4 (11 minutes). Would the following be a valid implementation of (3)'s pushme and pullyou functions? If not, explain why not. If so, give another implementation of pushme and pullyou that would be even shorter (i.e., would take fewer bytes of machine code).

```
pushme: cmpq    $2039, %rdi
        seta    %al
        ret
pullyou:jmp     pushme
```

5 (11 minutes). The following is a buggy implementation of (3)'s pushme function. Three of its instructions are incorrect. Fix the bugs with as few changes as you can and briefly explain why your fixes are needed.

```
pushme: pushq   %rbx
        movq    %rsp, %rbp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rax
        shrq    $3, %rax
        cmpq    $255, %rax
        seta    %al
        popq    %rbx
        ret
```

6 (11 minutes). Explain what the following assembly-language function does, at a high level. Give C source code that corresponds to its behavior as closely as possible.

```
mystery: movzbl %dil,%eax
         movabs $0x101010101010101,%rdx
         imul   %rdx,%rax
         retq
```

7 (11 minutes). What does the following assembly-language code do? Briefly explain how to use it from C source code, how it executes, and what its behavior is from the C point of view.

```
callme: leaq    (%rdi,%rsi), %rax
        callq   .L1
.L1:    ret
```

8 (11 minutes). Consider the following C code:

```
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  long n = 3;
 5  extern int (*p) (void);
 6
 7  void
 8  output (int n)
 9  {
10    printf ("0x%x\n", n);
11  }
12
13  int
14  badfun (void)
15  {
16    int i;
17    memcpy (&i + 3, &p, sizeof p);
18    output (*(&i + n));
19    return i;
20  }
21
22  int
23  main (void)
24  {
25    return ! badfun ();
26  }
27
28  int (*p) (void) = main;
```

and the following machine code generated for two of its functions, in GDB disassembly format:

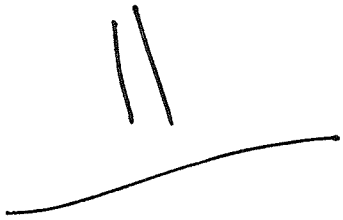
```
Dump of assembler code for function badfun:
   0x400550 <+0>:  sub    $0x18,%rsp
   0x400554 <+4>:  mov    0x200ae5(%rip),%rax
    # 0x601040 <p>
   0x40055b <+11>: mov    %rax,0x18(%rsp)
   0x400560 <+16>: mov    0x200ae1(%rip),%rax
    # 0x601048 <n>
   0x400567 <+23>: mov    0xc(%rsp,%rax,4),%edi
   0x40056b <+27>: callq  0x400540 <output>
   0x400570 <+32>: mov    0xc(%rsp),%eax
   0x400574 <+36>: add    $0x18,%rsp
   0x400578 <+40>: retq
Dump of assembler code for function main:
   0x400430 <+0>:  sub    $0x8,%rsp
   0x400434 <+4>:  callq  0x400550 <badfun>
   0x400439 <+9>:  test   %eax,%eax
   0x40043b <+11>: sete   %al
   0x40043e <+14>: add    $0x8,%rsp
   0x400442 <+18>: movzbl %al,%eax
   0x400445 <+21>: retq
```

For each instruction in the machine code, identify the corresponding source-code line number. If an instruction corresponds to two or more source-code line numbers, write them all down and explain.

9 (11 minutes). When (8)'s program is run it outputs about a million lines of text and then dumps core with a segmentation fault. Explain why this happens, in as much detail as you can. Your explanation should include what those text lines look like, and why.

1.

```
unsigned long rbp(char p)
{
    return p* 0x0101010101010101
}
```



= 0xFE7DD89A

a. 7D FE 9A D8 +2

+2
+3

b. [illegible] +2

c. [illegible] +2

3. a. right shifting an unsigned integer is equivalent to dividing by the corresponding power of 2. So 255 <= (v>>3) is equal to 255 <= v/8, which is then 2040 <= v, which is equal to 2039 < v as v is an integer.

+5

b. They have identical machine code as they do the same things mathematically. Although in python, v is signed but the code will always return false if v is negative.

Although v is more limited, the machine uses the same code regardless as it is a x86-64 bit machine/instruction and does not really care if it could be done in 64 bits or 63 bits. The rest of the function is simply the inverse of the inverse of pushing, and therefore is the same.

+6

4. This machine code is valid.

```
pushme:
pushyou: cmpq
         sete
         ret
```

5.

pushq %rbx	should be	pushq %rbp
seta %al	should be	setae %al
popq %rbx	should be	popq %rbp

The change to pushq %rbx and popq %rbp are necessary as otherwise the %rbp register's previous value is lost, which will prevent the function from returning to the part of the stack used previously before the function was called.

The change to seta %al is necessary as the function is evaluating if V>73 is greater than or equal to 255, not simply greater than. Without the change, the answer for V=2047 will be incorrect.

6. The function mystery first extends the 8-bit input to a 32-bit by filling the front with 24 zeroes. This is then multiplied by 0x0101010101010101, in effect repeating the 8-bit input 8 times in the 64-bit return value, as in problem #1.

```
unsigned long rbp(char p)
{
    return p * 0x0101010101010101;
}
```

11

7. The function callme returns the value &(x+y), where x and y are the first and second arguments. +2 +1

This could be used if x was a pointer to an array and y was the offset to the desired element, i.e. n · sizeof(array element) for the nth element of the array.

+1

8.

sub	$0x18, %rsp	x 13, 16, 18 : sees that it needs space for an int and a long
mov	0x200ae5(%rip), %rax	✓ 17, accesses p in line 17
mov	%rax, 0x18(%rsp)	✓ 17
mov	0x200ae1(%rip), %rax	✓ 18, 4 ; accesses n in line 18
mov	0xc(%rsp, %rax, 4), %edi	x 17
callq	0x400540 <output>	✓ 18, 8, accesses function output
mov	0xc(%rsp), %eax	x 17
add	$0x18, %rsp	x 13, 16, 18 : same as with sub, but restoring rsp to prev. value
retq		✓ 19

sub	$0x8, %rsp	✓ 22
callq	0x400550 <badfun>	✓ 25, accesses function badfun
test	%eax, %eax	✓ 25
sete	%dl	✓ 25
add	$0x8, %rsp	✓ 25
movzbl	%al, %eax	✓ 25
retq		✓ 25

11 − 4/2 = 11 − 2 = 9

9. badfun is allocated 24 bytes, but ends up using 32 bytes and overwrites the ret with a pointer to function main causing a stack ~~underflow~~ overflow and causing it to repeatedly function until program crash. The lines will contain ~~0x601040~~ 0x400430 as it keeps getting the location of P, prints a newline, and repeats this process. The function crashes as each time it runs, it loses some space beneath the function until it simply reaches the end of the stack at ~~ffffffffffffff~~. Then, when it reaches for a point below the stack, the computer crashes.