24

## Question 1. The bits are a lie. (24, 4 pts each)

You see the following bytes in increasing addresses on your x86-64 based laptop.

Table 1: My caption

| Address | 0xA0 | 0xA1 | 0xA2 | 0xA3 | 0xA4 | 0xA5 | 0xA6 | 0xA7 | 0xA8 |
|---|---|---|---|---|---|---|---|---|---|
| Data | 0x49 | 0x3C | 0x33 | 0x63 | 0x73 | 0x33 | 0x33 | 0x00 | 0xFE |

Interpret this data as different data types. You can write numbers using hexadecimal, and floating point numbers using hexadecimal + equations. Please don't convert to decimal.

Always assume the data type starts at 0xA0. (does not need to use all bytes)

✓ 1. `int` 0x63333C49

✓ 2. Say we have the definition:
`struct S1 {char a; short b; int c; char d;} instance;`.

Say instance starts at 0xA0, interpret as `instance.d` as a char. 0xFE

✓ 3. `unsigned short` 0x3C49

✓ 4. `float` $(-1)^0(0x333C49)(2^{71})$

63 3 33C49

✓ 5. C string (ie, dereference a char*) I<3cs33

01001001

✓ 6. The first byte as a set of numbers. {0,3,6}

## Question 2. ISA Madness (9 pts)

6

✗ 1. Is the procedure calling convention specified by the instruction set architecture? (3 points)

-3

Yes

✓ 2. Is the size of the address space determined by the instruction set architecture? (3 points)

No

✓ 3. Which part of x86-64 annoys you the most? (3 points)

Little endian

16

## Question 3. Hey, I've seen these before! (16, 4 pts each)

Match the following datalab implementations to their descriptions.

```
int func1(int x, int y) {
  return !!(x ^ y);
}
int func2(int x) {
  return !(x >> 31);
}
int func3(int x, int y) {
  return (~x) & (~y);          ⟷ ~(x | y)
}

int func4(int x, int n) {
  return 0xff & (x >> (n << 3));
}

int func5(int x) {
 int wd16 = x ^ x>>16;
 int wd8 = wd16 ^ wd16>>8;
 int wd4 = wd8 ^ wd8>>4;
 int wd2 = wd4 ^ wd4>>2;
 int bit = (wd2 ^ wd2>>1) & 0x1;
 return bit;
}

int func6(int x) {
  int test = x >> 31;
  return (~test & x) | (test & (~x + 1));
}

int func7(int x) {
  int m8 = 0xAA;                       10101010
  int m16 = m8 | m8 << 8;
  int m32 = m16 | m16 <<16;
  int fillx = x | m32;
  return !~fillx;
}
```

1. Return true if two numbers are not equal. func1
2. Return true if all even bits of a number are set. func7
3. Return true of a number is not negative. func2
4. Get a particular byte out of an integer. func4

## Question 4. Addressing an Array (12 points)

11 | 12

Consider the following array declaration.

```
int *my_array[4][3];
```

1. If we do nothing else, will the program segfault if we access element [3][2]? (4 points)

   No ✓

2. Write a mathematical expression for accessing element [x][y] for this array. (4 poitns)

   12*x + 4*y     24x + 8y     −1

3. Can the above access be performed using one x86-64 mov instruction? (4 points)

   No ✓

## Question 5. Stack of Lists (24)

The following is a program which constructs a linked list, and multiplies a value stored in each node together.

```
typedef struct S{ //Defining the linked list node
  unsigned val;
  struct S* next;
} X;

int mult(X* x, int y) { // Do a multiply
  return x->val * y;
}

int follow(X* x, int y) { // iterate over the linked list
   while(x) {
     y=mult(x,y);
     x = x->next;
   }
   return y;
}

X s1, s2, s3; //items in the linked list, of type X.

void setup_list() {
  s1.val = 1; //set up the linked list
  s2.val = 2;
  s3.val = 3;
  s1.next=&s2;
  s2.next=&s3;
  //Oops, should have set s3.next to 0
}

int main(int argc, char** argv) {
  setup_list();
  printf("%d\n",follow(&s1, 1));
}
```

1. What is the size of the struct X in bytes? (3 points)

   16 bytes

2. What is type of the expression: x->val * y (3 points)

   unsigned

3. This code was supposed to print the value 6, but instead it segfaults (it crashes) because the last pointer of the list was not set to NULL. Looking at the assembly on the next page, what is the address of the instruction that the program segfaults on? (4 points)

   400548

4. In the box, draw the stack at the moment the program segfaults. The top of the page should have higher addresses; please indicate which values belong to which function. If you do not know a register's value that is pushed to the stack, write "old" register (eg. old rax). (10 points)

```
0000000000400546 <mult>:
  400546: 89 f0                mov    %esi,%eax
  400548: 0f af 07             imul   (%rdi),%eax
  40054b: c3                   retq

000000000040054c <follow>:
  40054c: 53                   push   %rbx
  40054d: 48 89 fb             mov    %rdi,%rbx
  400550: 89 f0                mov    %esi,%eax
  400552: eb 0e                jmp    400562 <follow+0x16>
  400554: 89 c6                mov    %eax,%esi
  400556: 48 89 df             mov    %rbx,%rdi
  400559: e8 e8 ff ff ff       callq  400546 <mult>
  40055e: 48 8b 5b 08          mov    0x8(%rbx),%rbx
  400562: 48 85 db             test   %rbx,%rbx
  400565: 75 ed                jne    400554 <follow+0x8>
  400567: 5b                   pop    %rbx
  400568: c3                   retq

000000000040059e <main>:
  40059e: 48 83 ec 08          sub    $0x8,%rsp
  4005a2: b8 00 00 00 00       mov    $0x0,%eax
  4005a7: e8 bd ff ff ff       callq  400569 <setup_list>
  4005ac: be 01 00 00 00       mov    $0x1,%esi
  4005b1: bf 60 10 60 00       mov    $0x601060,%edi
  4005b6: e8 91 ff ff ff       callq  40054c <follow>
  ...
```

...
Return address } main

(trash val.) rdi/rbx — 0x601088
old rax/esi — 6
40055e } follow

eax — 6 } mult

400566

-1

List the address of any conditional branch. (4 points)

400565

23/24

## Question 6. Double recursion all the way ... across the sky! (20 pts)

Dump of assembler code for function func(unsigned int m, unsigned int n)

```
0x0000000000400546 <+0>:    test   %edi,%edi
0x0000000000400548 <+2>:    jne    0x40054e <func+8>
0x000000000040054a <+4>:    lea    0x1(%rsi),%eax
0x000000000040054d <+7>:    retq
0x000000000040054e <+8>:    push   %rbx
0x000000000040054f <+9>:    mov    %edi,%ebx
0x0000000000400551 <+11>:   test   %esi,%esi
0x0000000000400553 <+13>:   jne    0x400564 <func+30>
0x0000000000400555 <+15>:   lea    -0x1(%rdi),%edi
0x0000000000400558 <+18>:   mov    $0x1,%esi
0x000000000040055d <+23>:   callq  0x400546 <func>
0x0000000000400562 <+28>:   jmp    0x400576 <func+48>
0x0000000000400564 <+30>:   sub    $0x1,%esi
0x0000000000400567 <+33>:   callq  0x400546 <func>
0x000000000040056c <+38>:   lea    -0x1(%rbx),%edi
0x000000000040056f <+41>:   mov    %eax,%esi
0x0000000000400571 <+43>:   callq  0x400546 <func>
0x0000000000400576 <+48>:   pop    %rbx
0x0000000000400577 <+49>:   retq
```

if m=0, return n+1

ZF=0

m to ebx

n & n

if n≠0, jump

if n=0

m=m-1

n=1

return func(?,?)

n=n-1

func(m,?)

m=m-1

1. Fill in the blanks in the function based on the assembly code. (18 Points)

```
                              edi                 esi
unsigned int func(unsigned int m, unsigned int n)
{
    if( m == 0 ){
        return n+1;
    }
    if( n == 0 ) {
        return func(m-1, 1);
    }
    return func( m-1 , func( m , n-1 ));
}
```

2. How many instructions read a data value from memory? (2 points)

3

## Question 7. The Digit Lock Bomb (15 pts)

The final question of the midterm, "I'm almost home free!" you think to yourself. After finishing the Bomblab at 11:59 last night, and then studying until 4:00am, you thought your troubles were just about to be over. However, as you skim over the last question, you realize you couldn't have been more wrong. Luckily, the evil professor left one helpful comment in the assembly...

Solve the final Bomblab phase.

The following is `void phase_8(char*)`:

```
            0x00000000004007ac <+0>:    mov     $0x0,%edx
            0x00000000004007b1 <+5>:    mov     $0x0,%ecx
            0x00000000004007b6 <+10>:   jmp     0x4007f3 <phase_8+71>
            0x00000000004007b8 <+12>:   movslq  %edx,%rax
            0x00000000004007bb <+15>:   movzbl  (%rdi,%rax,1),%eax  //get ith char of input
            0x00000000004007bf <+19>:   sub     $0x30,%eax          //convert ascii to int
            0x00000000004007c2 <+22>:   cmp     $0x6,%al                ecx - 6 > 0     ecx > 54
            0x00000000004007c4 <+24>:   ja      0x4007f0 <phase_8+68>                   7,8,9
            0x00000000004007c6 <+26>:   movzbl  %al,%eax
            0x00000000004007c9 <+29>:   jmpq    *0x4008f8(,%rax,8)
input = 0   0x00000000004007d0 <+36>:   or      $0x1,%ecx               ecx | 1   [illegible]
            0x00000000004007d3 <+39>:   jmp     0x4007f0 <phase_8+68>
input = 1   0x00000000004007d5 <+41>:   or      $0x2,%ecx               ecx | 2   10
            0x00000000004007d8 <+44>:   jmp     0x4007f0 <phase_8+68>
       =2   0x00000000004007da <+46>:   or      $0x4,%ecx               ecx | 4   ecx 100
            0x00000000004007dd <+49>:   jmp     0x4007f0 <phase_8+68>
       =3   0x00000000004007df <+51>:   or      $0x8,%ecx               ecx | 8   ecx 1000
            0x00000000004007e2 <+54>:   jmp     0x4007f0 <phase_8+68>
       =4   0x00000000004007e4 <+56>:   or      $0x10,%ecx              ecx | 16  ecx 10000
            0x00000000004007e7 <+59>:   jmp     0x4007f0 <phase_8+68>
       =5   0x00000000004007e9 <+61>:   or      $0x20,%ecx              ecx | 32  ecx 100000
            0x00000000004007ec <+64>:   jmp     0x4007f0 <phase_8+68>
       =6   0x00000000004007ee <+66>:   add     %ecx,%ecx           ecx + ecx = ecx
            0x00000000004007f0 <+68>:   add     $0x1,%edx
            0x00000000004007f3 <+71>:   cmp     $0x2,%edx               edx ≤ 20
            0x00000000004007f6 <+74>:   jle     0x4007b8 <phase_8+12>                       507
            0x00000000004007f8 <+76>:   sub     $0x8,%rsp
            0x00000000004007fc <+80>:   cmp     $0x21,%cl                   rcx - 33
            0x00000000004007ff <+83>:   jne     0x40080b <phase_8+95>       ZF=0
            0x0000000000400801 <+85>:   mov     $0x0,%eax
            0x0000000000400806 <+90>:   callq   0x400746 <phase_defused>
            0x000000000040080b <+95>:   test    %cl,%cl                 rcx & rcx      cl - lowest byte
            0x000000000040080d <+97>:   jns     0x400819 <phase_8+109> SF=0              (8 bits)
            0x000000000040080f <+99>:   mov     $0x0,%eax                              10000000
            0x0000000000400814 <+104>:  callq   0x400768 <secret_phase> ← rcx ≠ 33
            0x0000000000400819 <+109>:  mov     $0x0,%eax                              566
            0x000000000040081e <+114>:  callq   0x40078a <explode_bomb>
```

Also, you might need this: (gx just means print out "giant" 8 byte words)

```
(gdb) x/7gx 0x4008f8                  0                     1
0x4008f8:    0x00000000004007d0         0x00000000004007d5
0x400908:    0x00000000004007da 2       0x00000000004007df 3
0x400918:    0x00000000004007e4 4       0x00000000004007e9 5
0x400928:    0x00000000004007ee 6
```

1. What does the jmpq* instruction do, and what program control feature does it sometimes correspond to? (5 Points)

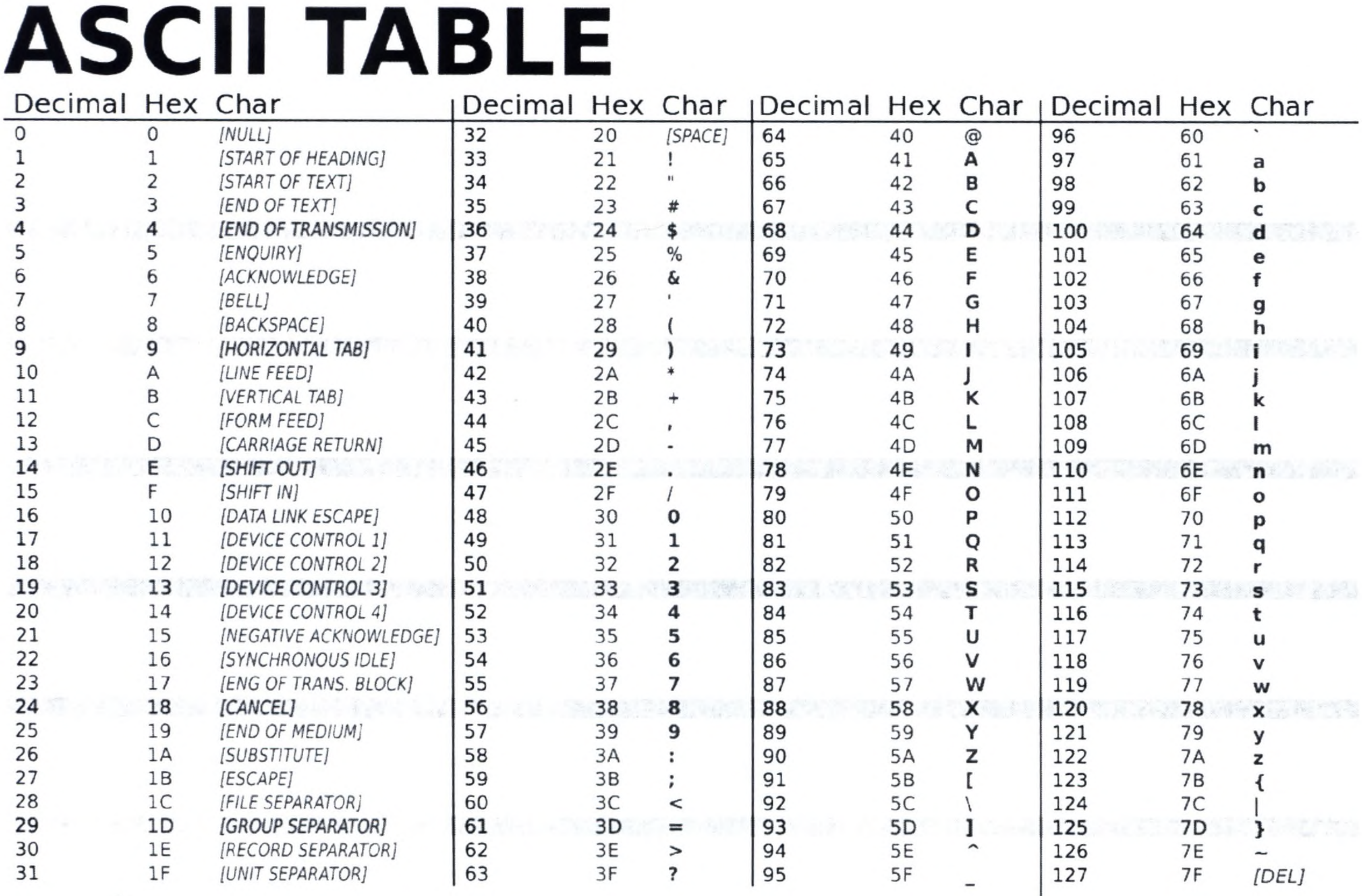
   It is an unconditional jump usually associated with a switch statement ✓

2. What input string would defuse the bomb? (5 Points)

   "507" ✓

3. What input string would activate the secret phase? (5 Points)

   "566" ✓



# ASCII TABLE

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

1.