Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page fact sheet. Your fact sheet may contain definitions and examples, but it may not contain questions and/or answers taken from old tests or homework. You may include examples from the course notes.
- No calculators or other electronic devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 5 questions, some with multiple parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- If you brought a fact sheet to the test, write your name on it and turn it in with the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

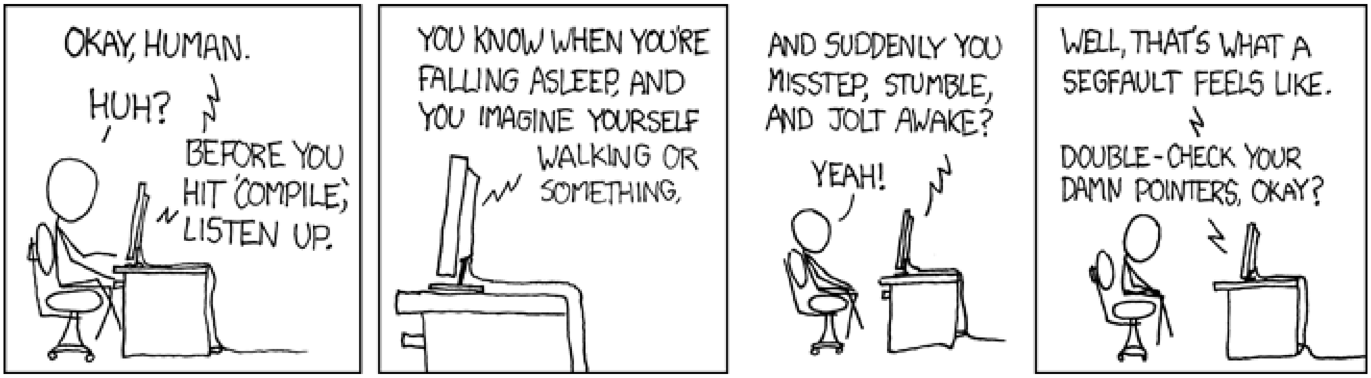
Name Solution

printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

____

signed


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

xkcd.com

1. **[14 points]** Consider the following two C data types:

```
struct _Score {
   uint8_t high;
   uint8_t low;
};
typedef struct _Score Score;
```

```
struct _Result {
   char  *winner;
   char  *loser;
   Score score;
};
typedef struct _Result Result;
```

Implement the C function shown below, conforming to the header comment. Violations of the restrictions will earn a score of zero. **Be careful of syntax in your answer!**

The function could be called in this manner:

```
const int nResults = 1000;
Result *pRes = malloc(nResults * sizeof(Result));
// initialize Result array

Score averageScore;
avgScore(&averageScore, pRes, nResults);
```

```
/**  Computes an "average" score for an array of Result objects.
 *
 *   Pre:  pScore points to a Score object
 *         pRes points to an array of nRes Result objects
 *   Post: *pScore has been updated to store the average high
 *            and low point values represented in the array pRes[]
 *   Restrictions:
 *     You may only use pointer syntax when accessing the elements in
 *     the array of Result objects.
 */
void avgScore(Score* const pScore, const Result* pRes, uint32_t nRes) {

   const Result* current = pRes;
   uint32_t sumHigh = 0, sumLow = 0;

   for (uint32_t idx = 0; idx < nRes; idx++) {

      sumHigh += current->score.high;
      sumLow  += current->score.low;

      current++;
   }

   pScore->high = sumHigh / nRes;
   pScore->low  = sumLow  / nRes;
}
```

Notes:

- A uint8_t will overflow if its value exceeds 255, so you really need to use a "wider" type to accumulate the sums (I did not deduct points for that).
- Nothing in the preconditions says that the fields in *pScore have been initialized, so you need to set them to zero if you use them to accumulate the sums.
- The "dot" operator must have the name of a struct variable on its left and the name of a field of that struct variable on its right
- `current` points to a `Result` object; `Result` objects have a field named `score` of type `Score`; so you access the `high` field of the `Score` object by writing `current->score.high`.

2. Suppose a hexdump of a memory region shows the following bytes:

```
          0   1   2   3   4   5   6   7       8   9   A   B   C   D   E   F
-----------------------------------------------------------------------------
     0 | 39  86  B5  BF  C5  52  A2  3E      8C  31  3B  1B  27  94  EE  5E
     1 | 69  2F  8E  FC  AF  F5  F5  56      C0  FE  20  A7  C4  45  9C  9D
     2 | CB  51  CC  91  A3  6E  CF  30      9F  0A  4B  C6  C6  39  24  07
     3 | 68  B3  04  18  A8  B5  6E  69      B3  8E  10  78  D3  AC  15  9F
```

a) **[8 points]** If the four bytes here are interpreted to represent the value of an `int32_t` variable, is that variable negative or nonnegative? Justify your conclusion.

Integers are stored in little-endian order (on our machines); the high byte is "BF", so the high nybble is 1011, so the high bit is 1, so the value is negative.

The question did not specify whether the data was stored in big-endian or little-endian order. Little-endian order is the rational assumption, given the nature of current systems. If you explicitly said you were assuming big-endian order, I intended to give full credit. If your answer implied you assumed that, but you were not explicit, I made a deduction.

b) **[8 points]** Suppose that the **`uint8_t*`** variable `P` points to the first byte in the memory region shown above. Complete the C statement below so that the variable `X` is assigned the value referred to in part a).

```
uint32_t X = *(uint32_t*)P;
```

To get a uint32_t value, you need to dereference a pointer whose target type is 4 bytes wide, hence the typecast of P to uint32_t*.

c) **[8 points]** Suppose that the **`uint32_t*`** variable `Q` points to the first byte in the memory region shown above. Complete the C statement below so that the pointer R will point to this byte in the memory region.

```
uint8_t* R = (uint8_t*)Q + 15;
```

The byte in question is 15 bytes after the target of Q; I typecast to get a pointer with a 1-byte target, then added 15. Other solutions are possible. There's no dereference because we want to make R point to the correct byte, not give R the value of that byte.

d) **[12 points]** Implement the following function, using only pointer notation:

```
/**  Returns the value of the first byte in the memory region that occurs
 *   in two or more consecutive locations.  For the region given above,
 *   the function would return the value 0xF5.
 *
 *   Pre:  P points to the first byte of a block of length nBytes.
 */
uint8_t firstDupe(const uint8_t* P, uint32_t nBytes) {

   for (uint32_t offset = 0; offset < nBytes - 1; offset++) {

      if ( *(P + offset) == *(P + offset + 1) ) {

         return *(P + offset);
      }
   }
   return 0;
}
```

There's a basic flaw in the question, since there's no value reserved (or specified) to return if no suitable pair of bytes is found… I was not picky about what you returned in that case. It would have been better to return a pointer to the relevant (first) byte; then NULL could have been used to indicate no pair was found.

Some errors (common or otherwise):

- Not allowing for the possibility that no match would be found… no matter what you return in that case, you cannot just assume there will be a match.
- Going one byte of bounds if the traversal reaches the end of the data region.
- Not stopping with the first match.
- Using array index notation instead of pointer notation (read the instructions).

3. The short C function shown below compiles to the assembly code shown below:

```
int q3(int x, int y) {

   int a = x + y;     // 1
   int b = x - y;     // 2

   return a * b;      // 3
}
```

```
q3:
   pushq  %rbp                       #  1
   movq   %rsp,      %rbp            #  2
   subq   $24,       %rsp            #  3
   movl   %edi,      -16(%rbp)       #  4
   movl   %esi,      -20(%rbp)       #  5
   movl   -20(%rbp), %eax            #  6
   movl   -16(%rbp), %edx            #  7
   addl   %edx,      %eax            #  8
   movl   %eax,      -12(%rbp)       #  9
   movl   -20(%rbp), %eax            # 10
   movl   -16(%rbp), %edx            # 11
   subl   %eax,      %edx            # 12
   movl   %edx,      %eax            # 13
   movl   %eax,      -4(%rbp)        # 14
   movl   -12(%rbp),  %eax           # 15
   imull   -4(%rbp), %eax            # 16
   leave                             # 17
   ret                               # 18
```

a) **[6 points]** Line 1 of the C code is translated into lines 6 through 9 of the assembly code. Why does gcc chose to use `movl` and `addl` instructions for this, but chooses `movq` and `subq` instructions for lines 2 and 3 of the assembly code?

Instructions 2 and 3 are manipulating 8-byte pointer values in registers rsp and rbp; instructions 6 through 9 are manipulating the parameters x and y and the local variable a, which are 4-byte int values.

The question was about WHY the compiler chose the specific instructions, not about the EFFECT of that choice. Many answers did not relate the choice to the data types that were being manipulated.

b) **[6 points]** In relation to the C code, what is the assembly instruction in line 16 doing?

Computing a * b and placing it in the eax register to be the return value from q3().

There were two parts to a complete answer, the product of the parameters is being computed, and the return value is being set. Most answers did not address the second issue.

c) **[6 points]** Many of the assembly instructions shown above access registers. Which of these assembly instructions access RAM (and possibly registers as well)? (Just list the numbers of the relevant instructions.)

1, 4, 5, 6, 7, 9, 10, 11, 14, 15, 16, 18

1 pushes a value to the stack; 18 pops a value from the stack; the stack is stored RAM.

Each of the other instructions listed above explicitly uses an address expression as a parameter.

d) **[8 points]** Using the variable names from the C code, show what (if anything) is stored at each occupied location in the stack frame for the function `q3`. If a frame location is not occupied, leave it blank.

rbp	old value of rbp
rbp - 4	b
rbp - 8	
rbp - 12	a
rbp - 16	x
rbp - 20	y
rbp - 24	
rbp - 28	

The assembly code shows the two parameters, x and y, being copied to the stack frame at addresses rbp - 16 and rbp - 20, respectively (lines 4 and 5).

The addl instruction (line 8) is computing x + y; the following movl instruction is storing that sum to rbp - 12 on the stack, which is must be where a is stored.

Similarly, the subl instruction (line 12) is computing x - y; the following movl instruction is storing that value to rbp - 4 on the stack, which must be where b is stored.

4. **[12 points]** Implement the following C function, obeying all restrictions. Violations of the restrictions will earn a score of zero.

```
/** Determines whether every byte in the word N represents an odd value.
 *
 *  Allowed operations:  ! ~ & | ^ << >>
 *  No more than 20 uses of operators.
 *  No use of loops, if, if-else, switches.
 *  No declarations of constants wider than one byte.
 *
 *  Returns 1 if every byte represents an odd value, 0 otherwise.
 *
 *  Examples:
 *    00000000 00000000 00000000 00000000 :  0
 *    00000000 00001110 01100010 10000010 :  0
 *    10000011 10000001 10000001 10000111 :  1
 *    01000011 10000001 00000001 00000001 :  1
 */
int bytesAreOdd(int N) {


   int lowBitMask = 0x01;

   int lowBit01 = N & lowBitMask;
   int lowBit02 = (N >>  8) & lowBitMask;
   int lowBit03 = (N >> 16) & lowBitMask;
   int lowBit04 = (N >> 24) & lowBitMask;

   return (lowBit01 & lowBit02 & lowBit03 & lowBit04);
}
```

First point: a value is odd iff its low bit is 1.

Therefore, we need to examine the low bit of each of the four bytes in N. We can isolate the low bit in a byte by ANDing the byte with the mask 0000 0001, which is just 0x01.

Second point: 0x01 assigned to an int variable will be 00000000 00000000 00000000 00000001 in binary.

So:

- N & 0x01 gives us the low bit of N (i.e., of N's low byte)
- if we shift N 8 bits to the right and apply the same mask, we get the low bit of the next byte of N
- … and so forth

(The C right shift gives us an arithmetic shift, which will fill the high bits with 1s if N < 0, but the mask wipes those to 0s, so we don't have to worry about that.)

Now, the function should return 1 iff all of those low bits were 1. If we AND the four variables found above, the result will be 1 iff all of the low bits were 1, and will be 0 iff one or more was 0.

So, we want to return the AND of those variables.

There are alternate approaches that work, but if you shift the mask up (to the left) instead of shifting N down (to the right), ORing might yield results that are not equal to either 0 or 1. So, if you took that approach, you needed to "post-process" so that the function always returns 0 or 1.

5. **[12 points]** The assembly code below was produced by gcc from a simple C function. The comments give some clues about what's going on in the function:

```
q5a:
     pushq  %rbp                       # stack frame setup
     movq   %rsp, %rbp
     subq   $20,  %rsp

     movl   %edi, -20(%rbp)            # int parameter:   X
     andl   $0x7FFFFFFF, -20(%rbp)     # X = X & 0x7FFFFFFF  (set high bit of X to 0)
     movl   $0,    -4(%rbp)            # int local auto:  Y = 0
.L2:
     movl   -20(%rbp), %eax            # eax = X
     andl   $15,  %eax                 # eax = X & 0xF  (low nybble of X)
     movl   %eax, -8(%rbp)             # int local auto:  Z = X & 0xF
     sarl   $4,    -20(%rbp)           # Z = X >> 4  (shift out low nybble of X)
     movl   -8(%rbp), %eax             # eax = Z
     addl   %eax, -4(%rbp)             # Y += Z
     cmpl   $5,    -8(%rbp)            # compare Z to 5
     jg     .L2                        # if Z > 5, goto L2  (repeat loop)
     movl   -4(%rbp), %eax             # set return value to Y

     leave
     ret
```

Examine the assembly code and write a C version of the function `q5a`. Your answer may not use a `goto` statement.

```
int q5a(int X) {

   X = X & 0x7FFFFFFF;

   int Y = 0;
   int Z;
   do {

      Z = X & 0xF;
      X = X >> 4;
      Y += Z;

   } while ( Z > 5 );

   return Y;
}
```

The comments I added in the assembly code above contain my analysis of the problem.

The backward jump to .L2 indicates there's a loop. The fact that there's no forward jump to the loop test from before .L2 shows it is a do-while loop, not a while loop (or a for loop).

The given comments identified the parameter (X) and the local automatic variables (Y and Z), so the rest was a matter of analyzing the individual assembly statements and then putting it all together as C code.