Instructions:

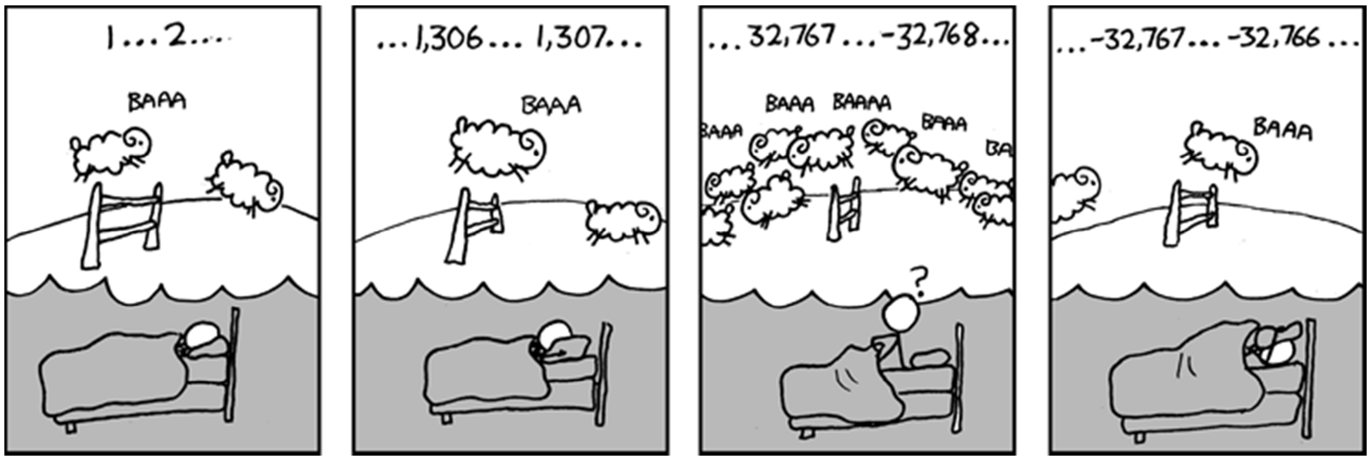
- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet. No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test 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 8 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


1... 2...
BAAA
...1,306... 1,307...
BAAA
...32,767...-32,768...
BAAA BAAAA BAAA BAAA
?
...-32,767...-32,766...
BAAA

xkcd.com

1. [10 points] Two registers in an x86 processor contain the representations of two integer values shown below. Which of the values is/are positive? Justify your conclusion clearly.

```
ebx:   1011 0011 0000 1010 0110 1001 0001 1000

ecx:   0010 1001 1111 0010 0100 0101 0110 0111
```

Note: you are not told whether these are representations of signed or unsigned integers. A full answer must take that into account, and consider both interpretations.

The bits in ecx represent a positive integer, whether the interpretation is signed or unsigned. We know this because unsigned values are never negative (and it's obvious the value isn't 0), and because signed values are represented in 2's complement form, and the high bit indicates the sign of the integer (0 for non-negative and 1 for negative).

The bits in ebx represent a positive value if the interpretation is unsigned, and a negative value if the interpretation is signed.

---

2. [12 points] Consider the following x86 assembly code fragment:

```
. . .
movl   -8(%ebp), %edx
movl   %edx, (%eax)
movl   %eax, 4(%esp)
. . .
```

Which of the parameters to the given x86 instructions refer to values that are stored on the stack. Explain how you know each of those parameters is stored on the stack.

Note: "parameters" are the operands supplied to commands. Each of the given commands has two parameters. Some parameters specify addresses, and so refer to values stored in memory; other parameters specify registers in the processor. And, some addresses correspond to values on the stack, and some addresses correspond to values in other regions of memory.

-8(%ebp)	**refers to a location 8 bytes below the target of the frame pointer, which must be within the frame for the currently-running function, since the frame pointer points to the first element within the current frame**
4(%esp)	**refers to a location 4 bytes above the target of the stack pointer, which must also be within the frame for the currently-running function, since the stack pointer points to the last element within the current frame**
($eax)	**refers to an address in memory, but there's no way to know if that's a location on the stack or not**

The remaining parameters all specify registers, not addresses.

3. [12 points] Circle the strings below that would match the following regular expression: `0+[3-5]*[abc]`

[✓ `0443b`] `06b` `3abc` [✓ `035c`] [✓ `0353535abc`] `0005`

The second answer uses '6', which is not in the given regular expression.
The third answer starts with '3', but the regular expression requires 1 or more 0's at the start.
The sixth answer doesn't end with an 'a' or 'b' or 'c'.
The fifth answer does not actually match the regular expression, because of the "bc" at the end, but it does contain a matching string…

---

4. [12 points] Formulate a regular expression that would match all, and only, those strings that satisfy the following conditions:

- The string starts with at least three occurrences of base-8 digits.
- Those are optionally followed by exactly two occurrences of the string SSDD.
- The next part of the string consists of one or more occurrences of A, or one or more occurrences of R, or one or more occurrences of Z.
- That is followed by a space character. (What follows that is irrelevant.)

Here is one solution:

```
[0-7]{3,}((SSDD){2})?(A+|R+|Z+)[ ]
```

Analysis:

Octal digits are 0 through 7; there are several ways to specify the necessary repetition, besides the solution shown above, including:

```
[0-7] [0-7] [0-7]+
[0-7] [0-7] [0-7] [0-7]*
```

For the second element, we need either two copies of SSDD or nothing at all; aside from the solution shown above, the following would do:

```
(SSDDSSDD)?
```

For the third element, we need either a sequence of A's or of R's or of Z's, but note that we can only use one of those letters. The following will NOT work:

```
(A|R|Z)+      that would include ARZ for instance
[ARZ]+        that suffers the same defect
```

For the final space, I used a set containing a space character, but there are several alternatives; see the regular expressions notes or reference for them.

5. [18 points] Write an implementation for the C function that is specified below. Pay attention to the pre- and post-conditions, and the specified return value. (You should consider whether a "helper" function would be helpful.)

```
// Pre:
//         A[] contains ASZ values
//         B[] contains BSZ values
//         The dimension of C[] is at least CSZ
// Post:
//         Barring an error, C[] contains all the elements of A[]
//         that are not also in B[]; the order in which the elements
//         occur in C[] is not specified
// Returns:
//         The number of elements copied into C[]; -1 if an error
//         is detected.
//
int Difference(int A[], int ASZ, int B[], int BSZ, int C[], int CSZ);
```

For example, given the arrays below, the function should respond as indicated:

```
A = {1, 7, 5, 4, 2} and B = {2, 3, 5, 4, 6} --> C = {1, 7} and returns 2
A = {1, 2, 3, 4, 5} and B = {2, 3, 5, 4, 1} --> C = {} and returns 0
```

```
int Difference(int A[], int ASZ, int B[], int BSZ, int C[], int CSZ) {

   int nCopied = 0;               // # of values copied into C[] from A[]

   for (int pos = 0; pos < ASZ; pos++) {

      if ( !isIn(A[pos], B, BSZ) ) {

         if ( nCopied >= CSZ )
            return -1;

         C[nCopied] = A[pos];
         ++nCopied;
      }
   }
   return nCopied;
}

bool isIn(int x, int List[], int LSZ) {

   for (int pos = 0; pos < LSZ; pos++) {
      if ( x == List[pos] )
         return true;
   }
   return false;
}
```

Notes:

- You are to copy values into C[] from A[] if they do not occur in B[], and only those values.
- The preconditions do not guarantee that C[] is big enough, but even if CSZ < ASZ, it may be that C[] will be large enough; BSZ is irrelevant (for this issue) since nothing that's in B[] will ever be copied to C[]; the only correct test is shown above.
- When searching B[] for a match, you should stop immediately if you find a match; do not enter Zombie Panic Mode and run off to the end of B[].

6. Consider the following short C function:

```
// Merges two ordered arrays A[] and B[] into a third ordered array C[].
//
// Example:
//                                          A[]:  3  4  7  8  9
//    C[]: 1  3  3  4  6  7  8  9  9 11 <--+
//                                          B[]:  1  3  6  9 11
//
// Pre:      A[] contains ASZ elements in sorted order
//           B[] contains BSZ elements in (the same) sorted order
//           C[] has room for the elements that will be copied into it
// Post:
//           Barring any errors, C[] contains the merged contents of A[]
//           and B[], in sorted order.
//
// Returns:  Barring any errors, the number of elements copied into C[];
//           -1 if an error occurred.
//
uint32_t G(int32_t A[], uint32_t ASZ, int32_t B[], uint32_t BSZ,
                                      int32_t C[], uint32_t CSZ) {

   if ( CSZ < ASZ + BSZ ) return -1;       // reject if C[] is too small

   uint32_t Apos = 0, Bpos = 0, Cpos = 0;

   while ( Apos < ASZ && Bpos < BSZ ) {    // walk and merge...
      if ( A[Apos] <= B[Bpos] ) {
         C[Cpos] = A[Apos];
         Apos++;
      }
      else {
         C[Cpos] = B[Bpos];
         Bpos++;
      }
      Cpos++;
   }
   return Cpos;    // return number of elements copied
}
```

a) [4 points] The implementer of the C function shown above has assumed something that is not guaranteed by the preconditions. What is that?

There are actually several serious things that are wrong here:

- **The implementation expects an ascending-order sort, and the preconditions do not require that.**
- **The loop will terminate as soon as all the elements in one of the arrays (either A[] or B[]) have been processed, and the rest of the other array will not be processed. The observation that the function fails when A[] is empty or B[] is empty, but not both, is just a special case of this.**
- **The preconditions do not actually say that CSZ is the dimension of C[], only that C[] is large enough to hold whatever is necessary.**

b) [4 points] And, assuming all of the given preconditions are satisfied, under what circumstances will this assumption affect the results, and why?

If the elements of both arrays are in descending order, then the resulting contents of C[] will be incorrect; for example, if A = {5, 1} and B ={4, 3}, then C will become {4,3}.

In any case, C[] will not include the final element(s) of one array; if A = {1, 5, 9} and B = {3, 7}, then C will become {1, 3, 5, 7}.

And, if CSZ is smaller than the dimension of C[], it's possible the function will terminate at the initial if statement and nothing will be copied.

---

7. [8 points] Consider the C language rules for scope, storage duration and linkage. How can you ensure that a variable X is not available to any function in any other file? Be complete.

The question does not specify whether we want X to be available to every function within the file, or just to one.

In the former case, we must declare X at file scope level (outside all blocks), and make it static (which restricts it to internal linkage so only functions in this file can access X).

In the latter case, we must declare X within the body of the function that needs to access X.

Notes:

- The question does say "Be complete".
- There is no "intern" or "internal" reserved word in C.
- "extern" does not make a variable available to other files, it enables linking a declaration of a variable in one file to a declaration in another file. So, "extern" is irrelevant.

8. The C code shown below compiles to the x86 assembly code shown below. (Some irrelevant assembly code has been omitted.)

```
// q7.c
int main() {

   int X = 32;     //   i
   int Y = 64;     //  ii
   int Z;          // iii

   Z = 8*X + Y;    //  iv
   Y = Z + 4;      //   v

   return Y;       //  vi
}
```

```
       .file "A.c"
       . . .
main:
       . . .
       movl  $32, -4(%ebp)       #  1
       movl  $64, -12(%ebp)      #  2
       movl  -4(%ebp), %eax      #  3
       sall  $3, %eax            #  4
       addl  -12(%ebp), %eax     #  5
       movl  %eax, -8(%ebp)      #  6
       movl  -8(%ebp), %eax      #  7
       addl  $4, %eax            #  8
       movl  %eax, -12(%ebp)     #  9
       movl  -12(%ebp), %eax     # 10
       . . .
```

a) [12 points] At what addresses (if anywhere) are the C variables X, Y, and Z stored on the stack? Explain how you determined your answers; references to the assembly code above would be useful. (You cannot give exact, numeric addresses but you can give exact symbolic ones.)

Comparing the initializations of X and Y in the C code to the assembly code we see that:

Line #1 must be initializing X, so X is at the address -4(%epb) or ebp - 4.
Line #2 must be initializing Y, so Y is at the address -12(%ebp) or ebp - 12.

Z is more challenging, since it's not initialized at its declaration. However, it is the next target of an assignment statement, in line iv of the C code, so that must correspond to the next time a value is copied to the stack, which occurs in line #6, so Z is at the address -8(%ebp) or ebp - 8.

b) [8 points] Which line(s) of assembly code correspond to statement `iv` in the C code? Explain your logic.

Lines #3 through #6 correspond to the execution of line iv of the C code:

```
movl   -4(%ebp), %eax      This fetches the value of X into eax, necessary for…
sall    $3, %eax           This shifts the value of X, multiplying it by 8.
addl   -12(%ebp), %eax     This adds the value of Y to the value of X.
movl   eax, -8(%ebp)       This copies the computed value into Z.
```