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 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 6 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!

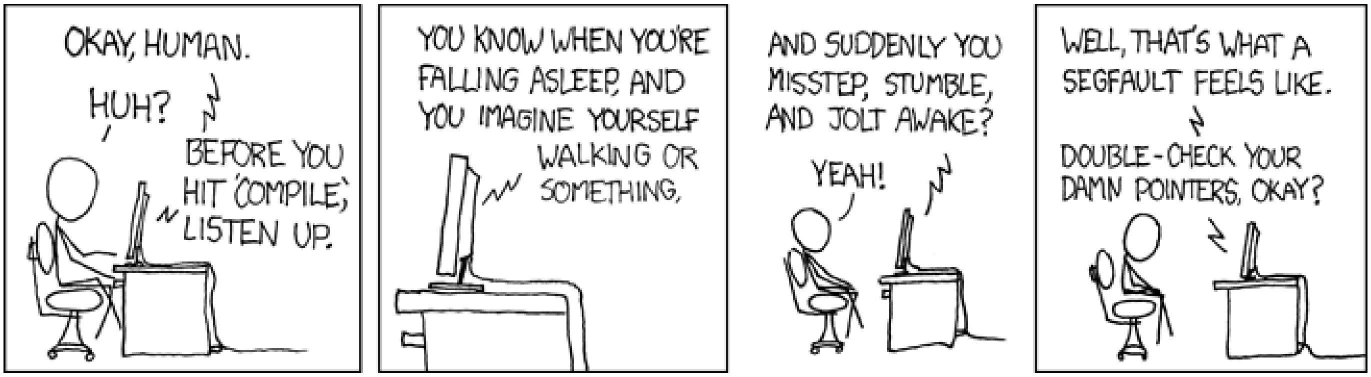
Name ________Solution________________________________

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. For this question you will create a "dense bool" type. A dense bool uses 1 bit to hold each boolean value, 0 is false and 1 is true. Our dense bool will be represented by a `uint8_t` type, so it will contain 8 boolean values:

```
uint8_t dense_bool = 00000001   // bool 0 is true, everything else is false
uint8_t other_bool = 00001100   // bool 2 and 3 are set to true
```

a) **[8 points]** Implement the **set** function for a dense bool variable as described below:

```
/* Pre:  N is the bit we want to change, 0 would change bit 0, and so on.
 *       Value is the boolean value, so it will be either 1 or 0
 * Post: bit N in dense_bool is set to Value; no other bits of dense_bool
 *       are changed
 */
void DB_set(uint8_t *dense_bool, uint8_t N, uint8_t Value)
{

      uint8_t clear_bit = 1 << N;
      clear_bit = ~clear_bit;

      *dense_bool = clear_bit & *dense_bool;

      uint8_t Mask = Value << N;
      *dense_bool = *dense_bool | Mask;
```

b) **[8 points]** Implement the **get** function for a dense bool variable as described below:

```
/* Pre: N is the bit we want to get, 0 would get the value of bit 0, and so on.
 * Returns: a uint8_t with only bit N set. The value at bit N should contain
 *          the same value as bit N in the dense_bool.
 */
uint8_t DB_get(uint8_t dense_bool, uint8_t N)
{

      uint8_t Mask = 1 << N;
      return dense_bool & Mask;
```

2. **[12 points]** Recall that for an earlier assignment you created a `BinaryInt` type. For this question you will write a similar C function, in this case you will be incrementing an **unsigned** integer.

You **may not** use any of the functions from your `BinaryInt` assignment, however you may use the `add_one_bit` function declared (but not implemented) below. This function takes the two "bits" you are adding (`a` and `b`), plus a carry in "bit" `cin`, and then returns the sum while placing the carry out "bit" in `cout`.

```
uint8_t add_one_bit(uint8_t a, uint8_t b, uint8_t cin, uint8_t * cout);

/*   Increments an **unsigned** integer value by 1.
 *
 *   Pre:   (*Sum)[] is of dimension currSize, currSize > 0
 *          (*Sum)[] stores an unsigned integer value
 *
 *   Post:  (*Sum)[] == (*Sum)[] + One[] (where One[] contains 000...01)
 *          If an overflow would occur when computing sum of (*Sum)[] + One[];
 *          (*Sum)[] should be resized dynamically (currSize = 2*currSize)
 *          to accommodate the bigger number.
 *
 *          So if we had four "bit" integers, adding 15 + 1 should produce:
 *             1111
 *           + 0001
 *         --------
 *         00010000
 *   Ret: the (potentially changed) dimension of (*Sum)[]
 */
uint8_t BI_Increment(uint8_t ** Sum, uint8_t currSize)
{
    uint8_t result, i = 0, carry = 0;
  (*Sum)[i] = add_one_bit((*Sum)[i], 1, 0, &carry);

   do
   {
         if(i ==  currSize - 1 && carry == 1)
         {
               uint8_t newSize = 2*currSize;
               uint8_t *temp = realloc(*Sum, currSize);

               if (temp != NULL)
                     *Sum = temp;

               (*Sum)[currSize] = 1;

               for(int s = currSize + 1;  s <  newSize; s++)
               {
                     (*Sum)[s] = 0;
               }
               currSize = newSize;
               break;

          }
          else
          {
               i++;
               (*Sum)[i] = add_one_bit((*Sum)[i], 0, carry, &carry);
          }

        }
        while(i < currSize);

        return currSize;
}
```

3. Consider the following short C program.

```
/* Process data inside of a buffer.
 * Pre:
 *  *pBuffer contains at least 2 bytes.
 *  The first byte in *pBuffer is the count of elements in the rest of pBuffer.
 *  The second byte is number of bytes per element (unit_size).
 *  The remaining count*unit_size bytes are data
 */
 void Q3(uint8_t *pBuffer)
 {
      uint8_t count = *pBuffer, x = 0;
      uint8_t unit_size = *pBuffer + 1;
      pBuffer +=2;

      if(unit_size == 4)  /* each element is a 4 byte value */
      {
              uint32_t *pNew  = malloc(unit_size * count);
              uint32_t *newTemp = pNew;

              while(x < count)
              {
                      *newTemp = *((uint32_t *) pBuffer);
                      newTemp +=4;
                      pBuffer +=4;
                      x++;
              }
              /* do something with pNew, not relevant */
      }
 }
```

a) [10 points] There are two pointer bugs in this function, one is a buffer overflow, and the other is something subtler. Analyze the code and determine the location of these errors. You may assume the input buffer is correctly formatted. **Be specific, vague answers will receive no credit.**

(1) *pBuffer + 1. The * will happen before the + 1, so unit_size ends up being count + 1, instead of the value at (pBuffer + 1).

(2) newTemp +=4. newTemp is a pointer to 32 bit (4 byte) integers, however we are incrementing it by 4 each time in the while loop. That would move the pointer 16 bytes or by four 32 bit integers, rather than to the next 32 bit integer.

Those were the two "big" bugs that alter the intended behavior of the program.

Not checking the return value of malloc is a bug. Mentioning this issue got you couple points if you didn't mention one (or both) of the bugs above.

Not free'ing the malloc'ed memory is also a potential bug, however in this case more code comes after copying from pBuffer as indicated by the comment at the end of the code.

b) [10 points] How could you fix each of the two bugs while keeping the same functionality?

```
*(pBuffer + 1);
newTemp +=1;
```

4. Consider the x86-32 translation of a short C function shown at right.

Assume that all parameters and local variables are of type `int` or `int*`.

a) **[6 points]** How many parameters does the function `Q4()` receive?

Three.

State the stack address at which each parameter is stored (e.g., `%ebp - 48`).

%ebp + 8 # 20

%ebp + 12 # 6

%ebp + 16 # 23

```
     . . .
Q4:
     pushl   %ebp             #  1
     movl    %esp, %ebp       #  2
     subl    $16, %esp        #  3
     movl    $0, -16(%ebp)    #  4
     movl    $0, -12(%ebp)    #  5
     movl    12(%ebp), %eax   #  6
     subl    $1, %eax         #  7
     movl    %eax, -8(%ebp)   #  8
     movl    $0, -4(%ebp)     #  9
     jmp    .L2               # 10
.L4:                          # 11
     movl    -16(%ebp), %eax  # 12
     sall    $2, %eax         # 13
     addl    8(%ebp), %eax    # 14
     movl    (%eax), %eax     # 15
     cmpl    16(%ebp), %eax   # 16
     jle    .L3               # 17
     movl    -16(%ebp), %eax  # 18
     sall    $2, %eax         # 19
     addl    8(%ebp), %eax    # 20
     movl    (%eax), %eax     # 21
     addl    -12(%ebp), %eax  # 22
     subl    16(%ebp), %eax   # 23
     movl    %eax, -12(%ebp)  # 24
     jmp    .L2               # 25
.L3:                          # 26
     addl    $1, -4(%ebp)     # 27
.L2:                          # 28
     movl    -16(%ebp), %eax  # 29
     cmpl    -8(%ebp), %eax   # 30
     jle    .L4               # 31
     movl    -12(%ebp), %eax  # 32
     leave                    # 33
     ret                      # 34
      . . .
```

b) **[6 points]** How many local variables does the function `Q4()` have?

Four.

State the stack address at which each local variable is stored (e.g., `%ebp - 48`).

%ebp - 4 # 9

%ebp - 8 # 8

%ebp - 12 # 5

%ebp - 16 # 4

c) [6 points] Examine the given x86-32 code. If there are any `if` or `if-else` control structures in the code, for each such structure, what range of statements (e.g., lines 17 through 23) belong to that structure (including any relevant labels and the boolean test)?

We recognize an if-statement by a conditional forward branch. An if-else-statement would have an unconditional forward branch, shortly before the target of the first forward branch.

The forward branches in #17 and #25 meet the criteria for an if-else-statement:

```
if-else   begins:   # 16 (comparison for if-test)
          ends:     # 28 (label for the jump over the else-clause)
```

(One could argue that the if-else begins a few statements earlier, where the necessary value for the comparison is placed into %eax.)

d) [6 points] Examine the given x86-32 code. If there are any loops in the code, for each loop, what range of statements (e.g., lines 17 through 23) belongs to the body of the loop (including any relevant labels and the loop test)?

We recognize a loop by a backward branch (usually conditional) to a label, from which execution falls back to the branch statement.

Such a branch occurs in #31. The target of than branch is preceded (#10) by an unconditional jump to the test for the loop, indicating this is a while-loop:

```
while     begins:   # 10 (jump forward to loop test)
          ends:     # 31 (jump backward to beginning of loop body)
```

e) [4 points] Examine the given x86-32 code. How many bytes are in the stack frame for `Q4`, including the backed up value for the frame pointer `ebp`? Justify your answer.

The first three instructions create the stack frame:

```
pushl   %ebp            #  1
movl    %esp, %ebp      #  2
subl    $16, %esp       #  3
```

The push instruction decrements %esp by 4, allocating space to store the old value of %ebp. The subl instruction allocates 16 more bytes.

So, the stack frame contains 20 bytes altogether.

5. A developer has an executable file that contains a C function and a `main()` function that calls it, but doesn't know much about the function except that it is named `mystery()`. So, she tries a gdb analysis. A partial transcript follows:

```
CentOS > gdb mdriver

(gdb) break mystery
Breakpoint 1 at 0x80483c2

(gdb) run
Breakpoint 1, 0x080483c2 in mystery ()
```

a)

```
(gdb) p *(int*)($ebp + 8)
$1 = 73

(gdb) p *(int*)($ebp + 12)
$2 = 14

(gdb) disassem
Dump of assembler code for function mystery:
   0x080483bc < +0>:    push   %ebp
   0x080483bd < +1>:    mov    %esp,%ebp
   0x080483bf < +3>:    sub    $0x10,%esp
=> 0x080483c2 < +6>:    mov    0x8(%ebp),%eax
   0x080483c5 < +9>:    mov    %eax,%edx
   0x080483c7 <+11>:    sar    $0x1f,%edx
   0x080483ca <+14>:    idivl  0xc(%ebp)
   0x080483cd <+17>:    mov    %eax,-0x4(%ebp)
   0x080483d0 <+20>:    mov    -0x4(%ebp),%eax
   0x080483d3 <+23>:    imul   0xc(%ebp),%eax
   0x080483d7 <+27>:    mov    %eax,-0x4(%ebp)
   0x080483da <+30>:    mov    -0x4(%ebp),%eax
   0x080483dd <+33>:    mov    0x8(%ebp),%edx
   0x080483e0 <+36>:    mov    %edx,%ecx
   0x080483e2 <+38>:    sub    %eax,%ecx
   0x080483e4 <+40>:    mov    %ecx,%eax
   0x080483e6 <+42>:    mov    %eax,-0x4(%ebp)
   0x080483e9 <+45>:    mov    -0x4(%ebp),%eax
   0x080483ec <+48>:    leave
   0x080483ed <+49>:    ret
End of assembler dump.

(gdb) ni
0x080483c5 in mystery ()
(gdb) ni
0x080483c7 in mystery ()
(gdb) ni
0x080483ca in mystery ()
(gdb) ni
0x080483cd in mystery ()

(gdb) disassem
   0x080483c7 <+11>:    sar    $0x1f,%edx
   0x080483ca <+14>:    idivl  0xc(%ebp)
=> 0x080483cd <+17>:    mov    %eax,-0x4(%ebp)
   0x080483d0 <+20>:    mov    -0x4(%ebp),%eax
```

b)

```
 (gdb) p $eax
$3 = 5

(gdb) p $edx
$4 = 3

(gdb) ni
0x080483d0 in mystery ()

(gdb) ni
0x080483d3 in mystery ()

(gdb) ni
0x080483d7 in mystery ()

(gdb) disassem
   0x080483d3 <+23>:    imul   0xc(%ebp),%eax
=> 0x080483d7 <+27>:    mov    %eax,-0x4(%ebp)
   0x080483da <+30>:    mov    -0x4(%ebp),%eax

(gdb) p $edx
$5 = 3

(gdb) p $eax
$6 = 70

(gdb) ni
0x080483da in mystery ()

(gdb) ni
0x080483dd in mystery ()

(gdb) ni
0x080483e0 in mystery ()

(gdb) ni
0x080483e2 in mystery ()

(gdb) ni
0x080483e4 in mystery ()

(gdb) disassem
   0x080483e2 <+38>:    sub    %eax,%ecx
=> 0x080483e4 <+40>:    mov    %ecx,%eax
   0x080483e6 <+42>:    mov    %eax,-0x4(%ebp)
```

c)

```
(gdb) p $ecx
$8 = 3
```

For the following questions, label the parameters to `mystery()` as `P1`, `P2`, etc.

Each question that follows refers to the part of the gdb session above that is labeled to match the question, but you may consider other parts of the gdb session in answering each question.

a) [4 points] Explain what the two print commands tell us about the function. You might want to consider the `disassem` output in your explanation.

These are the values of the two parameters passed to the function mystery().

We know that only two parameters are used since there are no other accesses to parameters in the disassembly of mystery() that follows.

b) [4 points] The `idivl` instruction performs integer division of the value in `%eax` by the operand to `idivl`. The result of an integer division is a quotient and a remainder (as in Discrete Math); `idivl` puts one of those values into `%eax` and one into `%edx`.

Which value goes into which register?

%eax holds the quotient and %edx holds the remainder.

We establish this by tracing the preceding code, which shows that prior to the idivl instruction, %eax holds the value of the first parameter (73), and knowing already that 0xc(%ebp) refers to the second parameter (value 14).

c) [10 points] Give an algebraic expression, in terms of the parameters to `mystery()`, for the value that is in `$ecx`. Justify your answer by **showing work next to the disassembly on page 8**.

Tracing the code reveals that the value returned equals: P1 – (P1 / P2) * P2

However, since these are all integer computations, this is just P1 % P2.

6. **[6 points]** The following x86 assembly code is part of a C function:

```
.L3:
     movl   $0, -8(%ebp)           # 1
     movl   -8(%ebp), %eax         # 2
     movl   (%eax), %eax           # 3
     movl   %eax, -4(%ebp)         # 4
     movl   $0, %eax               # 5
```

Explain what would happen if this code were executed, and why.

Statement #1 loads 0 into a local variable stored at %ebp – 8 on the stack.

Statement #2 loads the value of that local variable into %eax.

Statement #3 dereferences %eax... which is unfortunate, since %eax == 0.

So, a segmentation fault will occur (null pointer dereference).