CSci 2021 Section 010
Spring 2020
Midterm Exam 1 (solutions)
February 24th, 2020
Time Limit: 50 minutes, 3:35pm-4:25pm

- This exam contains 6 pages (including this cover page) and 4 questions. Once we tell you to start, please check that no pages are missing.
- Before starting the exam, you can fill out your name and other information of this page, but don't open the exam until you are directed to start. Don't put any of your answers on this page.
- You may use any textbooks, notes, or printouts you wish during the exam, but you may not use any electronic devices: no calculators, smart phones, laptops, etc.
- You may ask clarifying questions of the instructor or TAs, but no communication with other students is allowed during the exam.
- Please read all questions carefully before answering them. Remember that we can only grade what you write on the exam, so it's in your interest to show your work and explain your thinking.
- Students often find that the questions vary in difficulty. Your best strategy is usually to skim over all the questions, and then start working on the ones that look easiest. We also suggest that you leave time at the end to attempt every question, since we can't give you any partial credit if you leave a question blank.
- By signing below you certify that you agree to follow the rules of the exam, not to share exam material with other students before their exams, and that the answers on this exam are your own work only.

The exam will end promptly at 4:25pm. Good luck!

Your name (print): _____

Your UMN email/X.500: _____@umn.edu

Sign and date: _____

Question	Points	Score
1	20	
2	30	
3	30	
4	20	
Total:	100	

1. (20 points) Integer expressions in C.

   Below are 10 statements of C code that each assign an integer value between 1 and 10 to a variable `x`. In fact, each of the values 1 through 10 is used exactly once. Write the value that will be assigned to `x` in the blank next to each statement.

   You can assume that the integer `x`, the integer array `arr`, and the string `str` are declared as:

```
int x;
int arr[5] = {3, 1, 4, 1, 5};
char *str = "42";
```

   (a) __**6**__ `x = 1 + 2 * 2 + 1;`
   Multiplication has higher precedence than addition.

   (b) __**9**__ `x = 8 ^ 1;`
   $1000_2 \oplus 0001_2 = 1001_2$

   (c) __**8**__ `x = 1 << 3;`
   $1_2 << 3_{10} = 1000_2$

   (d) __**7**__ `x = arr[0] + arr[2];`
   `arr[0]` *is 3 and* `arr[2]` *is 4.*

   (e) __**1**__ `x = str[1 + 1] + 1;`
   `str[2]` *is the null terminator, a character whose numeric value is 0.*

   (f) __**2**__ `x = str[1] - '2' + 2;`
   `str[1]` *is the character* `'2'`*, and* `'2' - '2'` *is 0.*

   (g) __**10**__ `x = 0xa0 >> 4;`
   Shifting right by 4 bits is the same as shifting right by one hex digit. `0xa` *is 10.*

   (h) __**4**__ `x = (0xa >> 1) & (-1 << 1);`
   $000\ldots000101_2 \wedge 111\ldots11110_2 = 000\ldots000100_2$

   (i) __**5**__ `x = 4 | 1;`
   $100_2 \vee 001_2 = 101_2$

   (j) __**3**__ `x = ~(~0 << 2);`
   `(~0 << 2)` *is* $111\ldots111100_2$*. Inverting that gives* 11_2*.*

2. (30 points) A string operation using C pointers.

   One way to reverse the order of characters in a string is to interchange each character in the first half of the string with a corresponding character in the second half: swapping the first character with the last character, the second character with the second-to-last character, and so on. For instance if you start with the string `AbyZ`, you could first swap `A` with `Z` to get `ZbyA`, and then swap `b` with `y` to get `ZybA`. In this question, your job is to finish implementing a C function that implements this general approach, using pointers.

   Part of the function has already been written, but you need to fill in the body of a loop that swaps characters using pointers and updates the pointers. You shouldn't need to declare any more variables, and you shouldn't use array syntax (like `[]` brackets) to access the characters. Just use the pointers `p` and `q` that have been initialized, and the temporary variable `temp` if necessary. Write one statement per blank line; you may not need to fill in every blank.

```
void reverse_str(char *str) {
    /* P points at the first (0th) character of the string */
    char *p = str;
    /* Q points at the last (non-null) character of the string */
    char *q = p + strlen(str) - 1;
    char temp;

    /* Repeat the loop while P points before Q */
    while (p < q) {
        /* Swap the characters pointed to by P and Q */

        /* Swapping is an operation on the characters that the
           pointers point to, so you dereference the pointers with
           the * operator. */
        ____ temp = *p; _____________________________
        ____ *p = *q; _____________________________
        ____ *q = temp; _____________________________

        /* Move P and Q to the next characters to swap */

        /* P advances forward, while Q goes backwards. This is an
           operation on the pointer themselves, so there's no
           dereference operator. */
        ____ p++; _____________________________________
        ____ q--; _____________________________________

    }
}
```

3. (30 points) Integer representations.

   Suppose we have a very small microprocessor with a 4-bit word size. A pattern of bits in one of its registers might be interpreted either as a signed (two's complement) integer or as an unsigned integer. Each row of the following table shows how one bit pattern can be interpreted as a signed or unsigned number. Given the various kinds of descriptions in the left column, your job is to fill in all the remaining table entries (we've done "five" as an example, and filled in a few other obvious ones). Remember that UMin, UMax, TMin and TMax refer to the smallest (Min) and largest (Max) representable unsigned (U) and signed (T) values. Some values may be repeated.

Description	Binary	Decimal (signed)	Decimal (unsigned)
zero	0000	0	0
five	0101	5	5
negative one	1111	-1	15
decimal 6	0110	6	6
unsigned 12	1100	-4	12
binary 1010	1010	-6	10
binary 0011	0011	3	3
UMax	1111	-1	15
TMin	1000	-8	8
TMax	0111	7	7
TMin + TMin	0000	0	0
-TMax	1001	-7	9

   The best way to convert small values like this from binary to decimal is just to remember the place values and add the place values together for all of the 1 bits. For unsigned the place values are 8, 4, 2, and 1, while signed the only difference is that the sign bit is negated, so they are -8, 4, 2, and 1. So for instance 1010 signed is $-8+2=-6$*, and unsigned it is* $8+2=10$*. For the operations you can either do them mechanically (addition right to left with carries, negation by inverting and then adding 1), or by doing the operation in decimal and then accounting for overflow. You should also notice the patterns that the signed and unsigned values match between rows whenever the binary values match, and the signed and unsigned values are always either the same if they are between 0 and 7, or the unsigned value is larger by 16 otherwise.*

4. (20 points) Integer operations in assembly.

   Each of the x86-64 assembly-language code snippets shown on the left below eventually puts a fixed number into the `%rdi` register (the previous contents of the register don't matter). Match each sequence of instructions with the number it generates, chosen among the numbers shown in the right-hand column, by writing the letter from the right in the blank. Numbers prefixed with `0x` are hexadecimal, while other numbers are decimal. Each number is used exactly once. Suggestion: don't try to do the arithmetic here by converting to decimal: the operations on hexadecimal values are all simpler if you keep them in binary/hexadecimal.

(a) __**A**__
```
xorq %rdi, %rdi // x^x == 0
```

(b) __**D**__
```
movq $0, %rdi
leaq 4(%rdi), %rdi // rdi = rdi + 4
```

(c) __**G**__
```
movq $0x120, %rdi
xorq $0x100, %rdi // 0x120^0x100 == 0x20
```

(d) __**C**__
```
movq $0x20, %rdi
shrq $4, %rdi // 0x20 >> 4 == 0x2
```

(e) __**J**__
```
movq $0x20, %rdi
leaq 0x60(%rdi,%rdi,4), %rdi
// 0x20 + 4*0x20 + 0x60 = 16*0x10
```

(f) __**H**__
```
subq %rdi, %rdi // x-x == 0
notq %rdi // ~0 = 111..111
andq $0x40, %rdi // 111..111 & 0x40 == 0x40
```

(g) __**I**__
```
movq $0x10, %rdi
leaq (,%rdi,8), %rdi // rdi = 8*rdi
```

(h) __**F**__
```
movq $0x50, %rdi
subq $0x40, %rdi // 0x50-0x40 = 0x10
```

(i) __**E**__
```
movq $1, %rdi
shlq $3, %rdi // 1 << 3 == 8
```

(j) __**B**__
```
movq $11, %rdi
xorq $10, %rdi // 1011 ^ 1010 = 0001
```

A. `0x0 (0)`

B. `0x1 (1)`

C. `0x2 (2)`

D. `0x4 (4)`

E. `0x8 (8)`

F. `0x10 (16)`

G. `0x20 (32)`

H. `0x40 (64)`

I. `0x80 (128)`

J. `0x100 (256)`

This extra page has a summary about x86-64 assembly language (AT&T format) for your reference:

Instructions	
`add` X, Y	add $X + Y$ and store in Y
`and` X, Y	compute bitwise AND of X and Y and store in Y
`lea` X, Y	compute address of X and store in Y
`mov` X, Y	copy value X to location Y
`not` X	compute bitwise complement of X and store in X
`shl` X, Y	shift Y left by X positions
`shr` X, Y	logical shift Y right by X positions
`sub` X, Y	compute $Y - X$ and store in Y
`xor` X, Y	compute bitwise XOR of X and Y and store in Y
Addressing modes	
(R)	mem[reg[R]]
D(R)	mem[D + reg[R]]
D(,R,S)	mem[D + reg[R] * S]
D(B,R,S)	mem[D + reg[B] + reg[R] * S]
Size suffixes	
b	8-bit byte
w	16-bit value
l	32-bit value
q	64-bit value