MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.S077: Introduction to Low-level Programming in C and Assembly

Spring 2023, Quarter 1

Name: Brian Kernighan	**Kerberos:** briank
	MIT ID #:

#1 (15)	15
#2 (6)	6
#3 (10)	10
#4 (24)	24
#5 (10)	10
#6 (12)	12
#7 (15)	15
#8 (8)	8
Total (100)	100

Exam content is on **both sides** of the exam sheets.

Enter your answers in the boxes designated for each problem. Show your work for potential partial credit.

This page intentionally left blank

Problem 1. Give a Little Bit (15 points)

A. (2 points) Consider this code:

```
uint8_t a = 0x56;
uint8_t b = 0b10101010;
uint8_t c = 3;
```

Given the variable initializations above, evaluate `(a && b) | c`. Provide your answer in both unsigned 8-bit binary and decimal encodings.

Unsigned 8-bit binary (0b): 0000_0011
Decimal: 3

B. (2 points) Convert 16 to 8-bit two's complement binary and hexadecimal encoding:

8 bit two's complement binary (0b): 0001_0000
8 bit two's complement hexadecimal (0x): 10

C. (2 points) Convert -16 to 8-bit two's complement binary and hexadecimal encoding:

8 bit two's complement binary (0b): 1111_0000
8 bit two's complement hexadecimal (0x): F0

D. (2 points): An 8-bit C variable contains the value `0xE0`. What would the decimal value be if the variable was a `uint8_t`? An `int8_t`?

`uint8_t` decimal value: 224
`int8_t` decimal value: -32

E. (2 points): Consider this code:

```
int8_t y = 0x7;
int8_t z = 0b10000001;
```

Evaluate the two operations and provide the resulting value in **decimal** form:

`y << 2` (in decimal): 28
`z >> 1` (in decimal): -64

F. (2 Points) Consider this code:

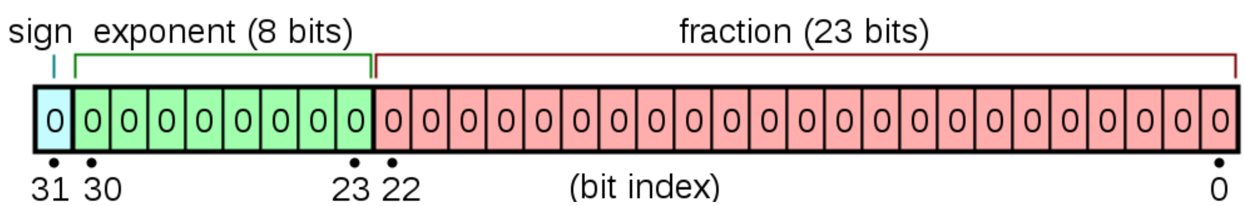
```
int8_t x = 0xEF;
int8_t y = 0x1A;
int8_t z = x+y;

uint8_t a = 250;
uint8_t b = 7;
uint8_t c = a+b;
```

After this code executes, what are the decimal values of `z` and `c`?

Value of `z` **(in decimal)**: **9**
Value of `c` **(in decimal)**: **1**

G. (3 Points) What is the 32-bit floating point representation of the number -128.0? The format of 32-bit floating point encoding is shown below. Show your work for full credit. *Note that the number shown in the figure is not -128.0.*



$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

32 bit floating point representation of -128.0. Provide your answer in hexadecimal: **0xC3000000**

Problem 2. Who Loves the Sum (6 points)

You are writing a function that computes the sum of the given array. Here is what you have so far.

```
// Computes the sum in the given array
// x: address of the first int in the array
// n: array length
int compute_sum(int* x, const unsigned int n) {
    int* y = x;
    int sum = 0;
    while (__BLANK 1__) {
        sum += __BLANK 2__;
        y += __BLANK 3__;
    }
    return sum;
}
```

Fill in the blanks (using the table below) to complete the implementation.

Please note that you may not alter `n` because the variable is declared with the `const` (constant) keyword.

BLANK 1:	BLANK 2:	BLANK 3:
`y - x < n`	`*y`	`1`

Problem 3: Hex's & Oh's (10 points)

The nth hexagonal number can be calculated via the following formula:

$$h_n = \frac{2n \times (2n-1)}{2}$$

Please write an assembly procedure, `hexagonal`, that calculates the nth hexagonal number using the formula above. Its C declaration is: `int hexagonal(int n);`. It should obey the RISC-V calling convention and return to its caller once it's done. *Solutions that make unnecessary memory accesses will not be given full credit.*

You have access to an additional **instruction**, `mul`, that performs integer multiplication. However, it is very slow so **you may only use it once.** The RISC-V ISA describes `mul` as:

Inst.	Syntax	Description	Execution
MUL	`mul rd, rs1, rs2`	Integer multiplication	`reg[rd] ← (reg[rs1] * reg[rs2])[31:0]`

In other words, it performs 32 bit * 32 bit multiplication and places the lower 32 bits of the product in `rd`. *For this problem, we do not need to handle the case where the product is more than 32 bits.* The encoding of `mul` is as follows:

31–25	24–20	19–15	14–12	11–7	6–0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

A. (8 points) Please write your implementation of `hexagonal` in the box below.

```
Hexagonal:

    slli a0, a0, 1
    addi a1, a0, -1
    mul a0, a0, a1
    srai a0, a0, 1
    ret

    OR

    slli a1, a0, 1
    addi a1, a1, -1
    mul a0, a0, a1
    ret
```

B. (2 points) How much space in memory (in **bytes**) does your implementation of `hexagonal` take up?

Number of **bytes** occupied by your `hexagonal` instructions: **4 * number of instructions**

Problem 4. Money, Money, Money (24 points)

A procedure is written in RISC-V assembly that calculates the composition of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent) needed to represent a certain amount of money specified in cents. The C definition is:

```
void makeChange(int amount, int* change_array);
```

- `int amount:` The amount of money (in US cents) to analyze
- `int* change_array`: An array used as the function's output. It lists the number of quarters, dimes, nickels, and pennies, at indices 0, 1, 2, and 3, respectively.

```
makeChange:
   addi sp, sp, -4
   sw ra, 0(sp)
quarter:
   addi t0, zero, 25
   bgt t0, a0, dime
   lw t1, 0(a1)
   addi t1, t1, 1
   sw t1, 0(a1)
   addi a0, a0, -25
   call makeChange
   j done
dime:
   addi t0, zero, 10
   bgt t0, a0, nickel
   lw t1, 4(a1)
   addi t1, t1, 1
   sw t1, 4(a1)
   addi a0, a0, -10
   call makeChange
   j done
nickel:
   addi t0, zero, 5
   bgt t0, a0, penny
   lw t1, 8(a1)
   addi t1, t1, 1
   sw t1, 8(a1)
   addi a0, a0, -5
   call makeChange
   j done
penny:
   addi t0, zero, 1
   bgt t0, a0, done
   lw t1, 12(a1)
   addi t1, t1, 1
   sw t1, 12(a1)
   addi a0, a0, -1
   call makeChange
   j done
done:
   lw ra, 0(sp)
   addi sp, sp, 4
   ret
```

The procedure is run. You are not given the value for `amount`, and `change_array` is an array that starts with zeroed-out elements. **Eight coins are dispensed.**

You obtain a stack trace from *immediately after the procedure is run*:

Address:	Value:
0x3fc93ef4:	0x420000e8
0x3fc93ef8:	0x3c020184
0x3fc93efc:	0x3fc93ef8
0x3fc93f00:	0x3fc93ef8
0x3fc93f04:	0x00000001
0x3fc93f08:	0x3fc91000
0x3fc93f0c:	0x00000000
0x3fc93f10:	0x42004e2c
0x3fc93f14:	0x00000002
0x3fc93f18:	0x000000a3
0x3fc93f1c:	0x420000a7
0x3fc93f20:	0x4200002c
0x3fc93f24:	0x4200002c
0x3fc93f28:	0x420000ec
0x3fc93f2c:	0x420000a8
0x3fc93f30:	0x420000a8
0x3fc93f34:	0x420000a8
0x3fc93f38:	0x42000068
0x3fc93f3c:	0x42000068
0x3fc93f40:	0x42000048
0x3fc93f44:	0x42000048
0x3fc93f48:	0x42000048
0x3fc93f4c:	0x4200012c
0x3fc93f50:	0x00000003
0x3fc93f54:	0x00000002
0x3fc93f58:	0x00000000
0x3fc93f5c:	0x00000003

Answer the following questions:

A. (2 points) What is the value of the stack pointer (`sp`) at the time of the snapshot above?

0x3fc93f50

B. (2 points) What is the address of the instruction that makes the initial call to `makeChange`?

0x42000128

C. (4 points) What are the final values in `change_array` after the call to `makeChange`?

{3, 2, 0, 3}

D. (1 point) What is the value of input variable `amount` in the call to `makeChange`?

98

E. (5 points) What is the 32 bit value in memory address `0x42000080`? Specify in binary or in hexadecimal.

addi a0, a0, -5
0b111111111011_01010_000_01010_0010011
0b1111_1111_1011_0101_0000_0101_0001_0011
0xFFB50513

F. (10 points) Next, the following code is run (`sp` starts at `0x3fc93f40`):

```
//int arrays coins_1 and coins_2 previously declared
for (int i=0; i<4; i++){
  coins_1[i] = 0;
  coins_2[i] = 0;
}
//time point 1
makeChange(52,coins_1); //corresponding call executed when pc=0x42004e18
//time point 2
makeChange(16,coins_2); //corresponding call executed when pc=0x42004e24
//time point 3
```

The values in a certain portion of memory are shown at `time point 1.` On the next page, fill in the values at `time point 2` and `time point 3`.

Leave the cell blank if the values are unchanged from the values at `time point 1.`

Address	time point 1	time point 2	time point 3
0x3fc93f04	0x00000001		
0x3fc93f08	0x3fc91000		
0x3fc93f0c	0x00000000		
0x3fc93f10	0x42004e2c		
0x3fc93f14	0x00000000		
0x3fc93f18	0x00000000		
0x3fc93f1c	0x00000000		
0x3fc93f20	0x00000000		
0x3fc93f24	0x0000002a		
0x3fc93f28	0x00000111		
0x3fc93f2c	0xa0a0a0a0	**0x420000a8**	**0x420000a8**
0x3fc93f30	0x0000008a	**0x420000a8**	**0x420000a8**
0x3fc93f34	0x0000008a	**0x42000048**	**0x42000088**
0x3fc93f38	0x42004e6a	**0x42000048**	**0x42000068**
0x3fc93f3c	0x42004e6a	**0x42004e1c**	**0x42004e28**
0x3fc93f40	0x00000000		
0x3fc93f44	0x00000001		
0x3fc93f48	0x00000004		
0x3fc93f4c	0x00000003		
0x3fc93f50	0x420165ac		
0x3fc93f54	0x420165b0		
0x3fc93f58	0x12004e2c		
0x3fc93f5c	0x00000000		

Problem 5: COPYCAT (12 points)
Belly Eyelash is writing an assembly program that she can use to retrieve information from different sources.

Part of this program is a procedure, `arr_copy`, that copies the values of an input array into an output array. It uses one other procedure, `copy`. Belly does not have access to the C or assembly implementations of `copy`, but she can assume that it works as expected and follows the RISC-V calling convention.

`arr_copy`	`copy`
Arguments: 1. `int *src` – pointer to input array 2. `int *dest` – pointer to destination array 3. `int length` – length of source array *Copies all* `length` *elements in the input array (*`src`*) into another array (*`dest`*).* Returns nothing	Arguments: 1. `int *src` – pointer to input array 2. `int *dest` – pointer to destination array 3. `int idx` – index of element to copy *Copies* `src[idx]` *into* `dest[idx]` Returns nothing

She uses a working C implementation of `arr_copy` for reference.

Working C Implementation

```
void arr_copy(int *src, int *dest,
              int length) {
   int i = 0;
   while (i < length) {
       copy(src, dest, i);
       i++;
   }
}
```

Belly's Assembly Implementation

```
arr_copy:
   li s0, 0
   mv s1, a2
   j compare
loop:
   mv a2, s0
   call copy
   addi s0, s0, 1
compare:
   blt s0, s1, loop
   ret
```

The code compiles, however, it runs into some issues at run-time. Please answer the questions on the following page.

Please provide a short explanation for the following run-time behaviors. The entire assembly program works as expected when she uses the C implementation of `arr_copy` rather than her assembly version, so **she has narrowed down the root cause to her assembly implementation of `arr_copy`.**

A. (4 points) Belly's processor crashes due to an instruction accessing a memory address that it is not allowed to. This occurs at the instruction `lw s0, 0(a0)` within the `copy` procedure. She already used a debugger to verify that the correct arguments were passed into `arr_copy`.

Explanation: **`arr_copy` does not save `a0` before calling `copy` (or restore it after returning from `copy`). So, `copy` could have overwritten the value of `a0` to be some value that, when used as a memory address, would result in an illegal memory access.**

B. (4 points) Belly observes that her program gets trapped in an infinite loop within `arr_copy`. She already used a debugger to verify that the correct arguments were passed into `arr_copy`.

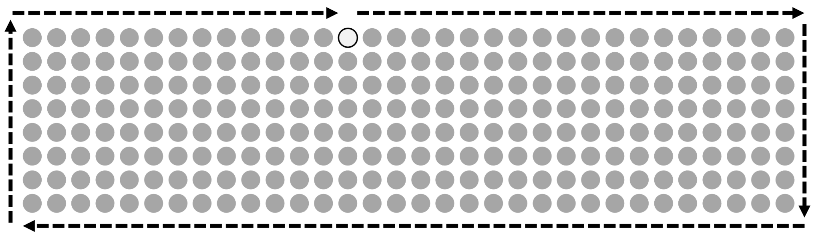
Explanation: **`arr_copy` does not save `ra` before calling `copy` (or restore it after returning from `copy`). However, `ra` gets overwritten by the `call copy` instruction. Since the old value of `ra` (which would refer to an instruction outside of `arr_copy`; specifically the instruction after the one that called `arr_copy`) was not saved to the stack, there is no way to recover it and return to the proper instruction in the program. So, when `arr_copy` encounters the `ret` instruction, it will jump to the `addi s0, s0, 1` instruction within `arr_copy`.**

C. (4 points) Belly's processor crashes (again) due to an instruction accessing a memory address that it is not allowed to. This time, it occurs *after* `arr_copy` returns back to its caller procedure, at the instruction `lw t1, 0(s1)`. She already used a debugger to verify that `arr_copy` wrote to the destination array as expected.

Explanation: **`arr_copy` uses `s0`, which is a callee-saved register, without saving its initial value to the stack and restoring it before returning. However, the procedure that called `arr_copy` was expecting `s0` to be a particular value. Because `arr_copy` overwrites the value of `s0` without restoring it to its initial value, the value in `s0` may no longer be a valid memory address.**

Problem 6. Lights on Broadway (10 points)

As you remember from the labs and postlabs, our lab kit's display is an 8×32 array of LEDs that we control through a length-8 `uint32_t` array. You can assume the zeroth bit of the zeroth array element corresponds with the upper right corner of the display. We'd like to make a border-scrolling LED pattern where a single illuminated LED traces the entire border in a clockwise fashion, like shown below:



Your friend started implementing a function, `chasingBorder`, that takes in a pointer to the screen array, `sb`, and based on the array's state, updates it to the next appropriate value in the border animation. Complete the function so that each call to the function `chasingBorder` moves the illuminated LED one spot in the clockwise direction. You can assume the LEDs are already following this border-scrolling pattern when `chasingBorder` is called. *You should not use any helper functions from lab.*

Unfortunately, you spilled boba on your keyboard, disabling the `'['` and `']'` keys, so you can't use them in your code. Fill in the ten blanks in the code below using the table on the next page.

```
void chasingBorder(uint32_t* sb){
  if(___________BLANK 1__________){
    //move right
    _________BLANK 2_________;
    return;
  } else {
    //move down
    for(int i = 0; i < 7; i++){
      if (__________BLANK 3_________){
        ____________BLANK 4________;
        ____________BLANK 5________;
        return;
      }
    }
  }
  if (__________BLANK 6_________){
    //move left
    ____________BLANK 7_________;
    return;
  } else {
    //move up
    for(int i = 0; i < 7; i++){
      if (_________BLANK 8__________){
        ___________BLANK 9_________;
        ___________BLANK 10________;
        return;
      }
    }
  }
}
```

Blank #:	**Line of Code:**
BLANK 1	`*sb && !(*sb == 1) OR *sb>1`
BLANK 2	`*sb >>= 1  or *sb = *sb>>1`
BLANK 3	`*(sb + i) == 1`
BLANK 4	`*(sb + i) = 0  or 3 and 4 can be interchanged`
BLANK 5	`*(sb + i + 1) = 1`
BLANK 6	`*(sb + 7) && !(*(sb + 7) == (1 << 31))`
BLANK 7	`*(sb + 7) <<= 1`
BLANK 8	`*(sb + 7 - i ) == (1 << 31)`
BLANK 9	`*(sb + 7 - i) = 0 or 8 and 9 can be interchanged`
BLANK 10	`*(sb + 6 - i) = (1 << 31)`

Could also do. (going down on left side instead of up)

BLANK 8	`*(sb + i + 1 ) != 0`
BLANK 9	`*(sb + i) = *(sb + i + 1)`
BLANK 10	`*(sb + i + 1) = 0`

Problem 7. MM..FOOD (15 Points)

We're in charge of managing a BurgerTime franchise which serves meals that look like this:

```
struct Meal{
  uint16_t burger;
  uint8_t fries;
};
```

We are going to focus on the burgers. Each burger contains only four possible ingredients: patties, cheese, tomatoes, and pickles. A burger is represented by a `uint16_t` that uses four bits to represent the count of each ingredient, so each burger can contain up to 15 units of each ingredient.

Ingredient	**Pickles**				**Tomatoes**				**Cheese Slices**				**Patties**			
Burger bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

A. (7 points) Due to popular demand, corporate has requested a function that can quickly remove pickles from a meal's burger. Write a function **`removePickles`** that takes in a pointer to a `Meal` struct, removes the pickles in the burger, and returns how many pickles were removed .

```
uint8_t removePickles(struct Meal *m){
    uint16_t burger = m->burger;
    uint16_t mask = 0xF << 12;
    m->burger &= ~mask;
    return (uint8_t)(burger >> 12);
```

```
}
```

B. (8 points) Someone called in sick, and now it's on you to manage one of the stations.

Write a function called **`stationOne`** that adds `patty_num` patties, and `tomato_num` tomato slices to a meal's burger. *You can assume the meal has no patties or tomato slices before the function is called.*

```
void stationOne(struct Meal *m, uint8_t patty_num, uint8_t tomato_num){
   uint16_t mask = (patty_num & 0xF) | ((tomato_num & 0xF) << 8);
   m->burger |= mask;
}
```

Problem 8. The End (8 points)

Read through the following functions so that you understand what they do. An ASCII table is provided to you for reference in the exam Appendix. Assume that a `char` acts like an unsigned 8 bit integer.

```
#include <stdio.h>

void mystery1(char input) {
    for(int i=7; i>=0; i--) {
        printf("%d", (input >> i) & 1);
    }
    printf("\n");
}

char mystery2(char input) {
    input = ((0b11110000 & input) >> 4) | ((0b00001111 & input) << 4);
    input = ((0b11001100 & input) >> 2) | ((0b00110011 & input) << 2);
    input = ((0b10101010 & input) >> 1) | ((0b01010101 & input) << 1);
    return input & 0b11111111;
}
```

A. (4 points) Consider the test code below:

```
char input1 = 0b11001100;

mystery1(input1); // PRINT A
```

What will be printed by the line indicated by `PRINT A`?

11001100

B. (4 points) Consider the test code below:

```
char input2 = 'G';

mystery1(mystery2(input2));  // PRINT B
```

What will be printed by the line indicated by `PRINT B`?

11100010

This page intentionally left blank

This page intentionally left blank

Appendix 1: String functions

`char *strcat(char *dest, const char *src)` - appends the string pointed to by `src` to the end of the string pointed to by `dest`. This function returns a pointer to the resulting string `dest`.

`char *strncat(char *dest, const char *src, size_t n)` - appends the string pointed to by `src` to the end of the string pointed to by dest up to `n` characters long. This function returns a pointer to the resulting string `dest`.

`char *strcpy(char *dest, const char *src)` - copies the string pointed to, by `src` to `dest`. This returns a pointer to the destination string `dest`.

`char *strncpy(char *dest, const char *src, size_t n)` - copies up to `n` characters from the string pointed to, by `src` to `dest`. In a case where the length of `src` is less than that of `n`, the remainder of `dest` will be padded with null bytes. This function returns the pointer to the copied string.

`int strcmp(const char *str1, const char *str2)` - compares the string pointed to, by `str1` to the string pointed to by `str2`. This function return values that are as follows −

- if Return value < 0 then it indicates `str1` is less than `str2`.
- if Return value > 0 then it indicates `str2` is less than `str1`.
- if Return value = 0 then it indicates `str1` is equal to `str2`.

`int strncmp(const char *str1, const char *str2, size_t n)` - compares at most the first `n` bytes of `str1` and `str2`. This function return values that are as follows −

- if Return value < 0 then it indicates `str1` is less than `str2`.
- if Return value > 0 then it indicates `str2` is less than `str1`.
- if Return value = 0 then it indicates `str1` is equal to `str2`.

`char *strchr(const char *str, int c)` - searches for the first occurrence of the character `c` (an `unsigned char`) in the string pointed to by the argument `str`. This returns a pointer to the first occurrence of the character `c` in the string `str`, or `NULL` if the character is not found.

`char *strrchr(const char *str, int c)` - searches for the last occurrence of the character `c` (an `unsigned char`) in the string pointed to, by the argument `str`. This function returns a pointer to the last occurrence of character in `str`. If the value is not found, the function returns a `null` pointer.

`char *strstr(const char *haystack, const char *needle)` - function finds the first occurrence of the substring `needle` in the string `haystack`. The terminating `'\0'` characters are not compared. This function returns a pointer to the first occurrence in `haystack` of any of the entire sequence of characters specified in `needle`, or a null pointer if the sequence is not present in `haystack`.

`char *strtok(char *str, const char *delim)` - breaks string `str` into a series of tokens using the delimiter `delim`. This function returns a pointer to the first token found in the string. A null pointer is returned if there are no tokens left to retrieve.

Appendix 2: ASCII Table

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Appendix 3: C Operator Precedence

<table>
<tr><th>Precedence</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td rowspan="5">1</td><td>++ --</td><td>Suffix/postfix increment and decrement</td><td rowspan="5">Left-to-right</td></tr>
<tr><td>()</td><td>Function call</td></tr>
<tr><td>[]</td><td>Array subscripting</td></tr>
<tr><td>.</td><td>Structure and union member access</td></tr>
<tr><td>-></td><td>Structure and union member access through pointer</td></tr>
<tr><td rowspan="6">2</td><td>++ --</td><td>Prefix increment and decrement</td><td rowspan="6">Right-to-left</td></tr>
<tr><td>+ -</td><td>Unary plus and minus</td></tr>
<tr><td>! ~</td><td>Logical NOT and bitwise NOT</td></tr>
<tr><td>(type)</td><td>Cast</td></tr>
<tr><td>*</td><td>Indirection (dereference)</td></tr>
<tr><td>&</td><td>Address-of</td></tr>
<tr><td>3</td><td>* / %</td><td>Multiplication, division, and remainder</td><td rowspan="11">Left-to-right</td></tr>
<tr><td>4</td><td>+ -</td><td>Addition and subtraction</td></tr>
<tr><td>5</td><td><< >></td><td>Bitwise left shift and right shift</td></tr>
<tr><td rowspan="2">6</td><td>< <=</td><td>For relational operators < and ≤ respectively</td></tr>
<tr><td>> >=</td><td>For relational operators > and ≥ respectively</td></tr>
<tr><td>7</td><td>== !=</td><td>For relational = and ≠ respectively</td></tr>
<tr><td>8</td><td>&</td><td>Bitwise AND</td></tr>
<tr><td>9</td><td>^</td><td>Bitwise XOR (exclusive or)</td></tr>
<tr><td>10</td><td>|</td><td>Bitwise OR (inclusive or)</td></tr>
<tr><td>11</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>12</td><td>||</td><td>Logical OR</td></tr>
<tr><td>13</td><td>?:</td><td>Ternary conditional</td><td rowspan="6">Right-to-left</td></tr>
<tr><td rowspan="5">14</td><td>=</td><td>Simple assignment</td></tr>
<tr><td>+= -=</td><td>Assignment by sum and difference</td></tr>
<tr><td>*= /= %=</td><td>Assignment by product, quotient, and remainder</td></tr>
<tr><td><<= >>=</td><td>Assignment by bitwise left shift and right shift</td></tr>
<tr><td>&= ^= |=</td><td>Assignment by bitwise AND, XOR, and OR</td></tr>
</table>