MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.1900: Introduction to Low-level Programming in C and Assembly

Spring 2023, Quarter 4

Name: Answer Key	**Kerberos:**
	MIT ID #:

#1 (15)	
#2 (9)	
#3 (15)	
#4 (15)	
#5 (22)	
#6 (12)	
#7 (12)	
Total (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. Commit to the Bit (15 points)

A. (2 points) For each expression below, write an equivalent expression using **bitwise operators.** Assume `x` and `y` are type `uint8_t`. **(You may only use the following operators: `|` , `&` , `^` , `~` , `<<` , `>>`. Do not use `==` or `!=` in your answer.)**

`while ((x/8) != y);`	`while(` `(x >> 3) ^ y` `);`
`if (x || (y*2))`	`if (` `x | (y << 1)` `)`

B. (2 points) Convert the decimal number 12 to 8-bit two's complement binary and hexadecimal encoding. *You must indicate all 8 bits.*

8-bit two's complement binary (0b): **0000_1100**
8-bit two's complement hexadecimal (0x): **0x0C**

C. (2 points) Convert the decimal number -12 to 8-bit two's complement binary and hexadecimal encoding. *You must indicate all 8 bits.*

8-bit two's complement binary (0b): **1111_0100**
8-bit two's complement hexadecimal (0x): **0xF4**

D. (2 points) Consider this code:

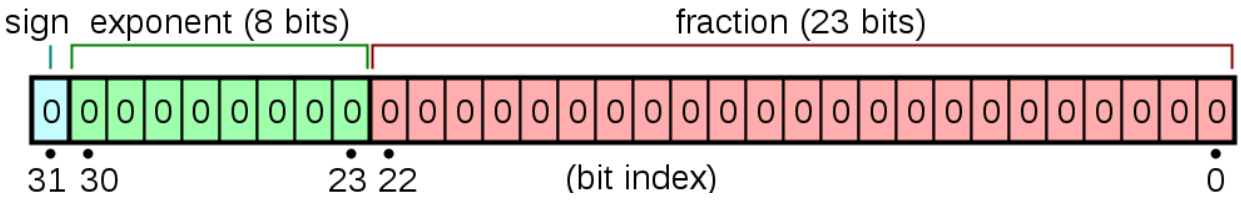
```
uint8_t a = 0x5;
uint8_t b = 0xA;
uint8_t c = a-b;

int8_t d = 0x80;
int8_t e = d>>1;
```

After this code executes, what are the values of `c` and `e`? *Your answer should be in decimal encoding.*

Value of `c` **(in decimal)**: **251**
Value of `e` **(in decimal)**: **-64**

E. (3 points) What is the 32-bit floating point representation of the number -64.5? 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 may not be -64.5.*



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

32 bit floating point representation of -64.5. Provide your answer in hexadecimal: **0xC2810000** **(sign: 1, exponent: 1000_0101, fraction: 0000_0010_0000…0)**

F. (4 points) The following is the 32-bit binary **floating point representation** of -38.25**:**

```
0b1_10000100_00110010000000000000000
```

Write the floating point representation of the value **`38.25`** in hexadecimal:

0x42190000
(sign: 0, everything else unchanged)

Write the floating point representation of the value **`-153.0`** in hexadecimal:

0xC3190000
(exp: 10000110, everything else unchanged)

Problem 2. Set It and Forget It (9 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. Your friend has written a function, `setPixel`, that takes in a game board, an x-coordinate, a y-coordinate and a value to set the pixel to (1 for on, 0 for off).

They started the function but left it incomplete. **Without using the `'['` and `']'` keys, complete the code:**

```
1 void setPixel(uint32_t* gb, int8_t x, int8_t y, int8_t val){
2  if (__BLANK 1__){
3    __BLANK 2__ = __BLANK 3__;
4  }else{
5    __BLANK 4__ = __BLANK 5__;
6  }
7 }
```

Blank #:	**Line of Code:**
BLANK 1	`val`
BLANK 2	`*(gb + y)`
BLANK 3	`*(gb + y) | (1 << x)`
BLANK 4	`*(gb + y)`
BLANK 5	`*(gb + y) & ~(1 << x)`

Problem 3. strTiK strToK (15 points)
Study the function and determine what it does. An ASCII table is provided in the Appendix.

```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 void mystery(char* in, int index, char* out, int out_size) {
 5     char* tok = strtok(in, "-");
 6     while(tok != NULL && index > 0) {
 7         index--;
 8         tok = strtok(NULL, "-");
 9     }
10
11     if (index == 0 && tok != NULL) {
12         for(int i=3; i>=0; i--) {
13             if (out_size > 1) {
14                 *out = 48 + (((*tok) >> (i * 2)) & 3);
15                 out ++;
16                 out_size --;
17             }
18         }
19         *out = 0;
20         out_size --;
21
22         tok ++;
23         strncat(out, tok, out_size);
24     }
25 }
```

A. (6 points) Consider the test code below:

```
char in[] = "strs-chars-ints-floats";
int index = 0;
char out[100] = "";
int out_size = 100;

mystery(in, index, out, out_size);
printf("%s", out);            // PRINT A
```

Determine what will get printed by the line `PRINT A`

What gets printed from `PRINT A`

1303trs

Program reproduced from the previous page for reference.

```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 void mystery(char* in, int index, char* out, int out_size) {
 5     char* tok = strtok(in, "-");
 6     while(tok != NULL && index > 0) {
 7         index--;
 8         tok = strtok(NULL, "-");
 9     }
10
11     if (index == 0 && tok != NULL) {
12         for(int i=3; i>=0; i--) {
13             if (out_size > 1) {
14                 *out = 48 + (((*tok) >> (i * 2)) & 3);
15                 out ++;
16                 out_size --;
17             }
18         }
19         *out = 0;
20         out_size --;
21
22         tok ++;
23         strncat(out, tok, out_size);
24     }
25 }
```

B. (3 points) Consider the test code below:

```
char in[] = "strs-chars-ints-floats";
int index = 0;
char out[100] = "";
int out_size = 6;   // << Changed here

mystery(in, index, out, out_size);
printf("%s", out);     // PRINT B
```

Determine what will get printed by the line `PRINT B`

What gets printed from `PRINT B`

1303t

C. (6 points) Consider the test code below:

```
char in[] = "strs-chars-ints-floats";
int index = 2;        // << Changed here
char out[100] = "";
int out_size = 100;  // << Changed here

mystery(in, index, out, out_size);
printf("%s", out);   // PRINT C
```

Determine what will get printed by the line `PRINT C`

What gets printed from `PRINT C`

1221nts

Problem 4. The Things We Take at MIT (15 points)

An MIT class can be separated into two parts:

- **Department number**: The number associated with the department offering the class. For example, any class offered by the MIT EECS department will have a department number of 6. You may assume all departments are represented numerically (don't worry about STS or 21A).
- **Subject number**: A number used to identify each class within a department. For example, the class 6.190 has a subject number of 190. The class 20.190 also has a subject number of 190. You may assume all subjects are represented numerically (no 6.UAT here).

The registrar encodes each MIT class as a 32-bit value, where:

- The upper 8 bits represent the **department** number.
- The lower 16 bits represent the **subject** number.
- The remaining 8 bits in the middle are **unused**, so we do not know or care about their values.

This encoding is shown in the table below. *Note: `X` means that we don't care about the value of the bit.*

`number[31:0]`	`number[31:24]`	`number[23:16]`	`number[15:0]`
MIT Class	**Department Number**	**Unused Bits**	**Subject Number**
6.101	`0b00000110`	`0bXXXXXXXX`	`0b0000000001100101`
18.06	`0b00010010`	`0bXXXXXXXX`	`0b0000000000000110`
NULL	`0b00000000`	`0bXXXXXXXX`	`0b0000000000000000`

A. (5 points) Write a function **`getDept`** that takes in one argument:

- `uint32_t mit_class`: an MIT class number

`getDept` should return the number of the department that offers the class represented by `mit_class` . For example, if `mit_class` is representing 18.06, `getDept` should return 18.

```
int8_t getDept(uint32_t mit_class){

    return (mit_class >> 24) & 0xFF;

}
```

The registrar uses a struct, `deptCatalog`, to store all of the classes currently offered by a given department. It is shown below:

```
#include <stdint.h>
#define MAX_CLASSES 65536
struct deptCatalog{
  uint8_t dept;          // department number
  uint32_t num_classes;  // number of classes offered by the department
  uint32_t mit_classes[MAX_CLASSES]; // array of MIT classes
};
```

The first `num_classes` elements of `mit_classes` contains every active class, encoded as described in the previous part, offered by the department, `dept`. You may not assume these classes are stored in any particular order.

For the remainder of the problem, assume a `deptCatalog` struct named `course6` is defined, where:

- `dept == 6`
- `0 <= num_classes < MAX_CLASSES`

B. (5 points) Write a function, `classExists`. It takes 2 arguments:

- `deptCatalog *dc`: A pointer to a department catalog struct.
- `uint16_t subject_number`: The subject number, as previously defined, of interest. *Note that this is not the same as an MIT class.*

Ex: To check that 6.101 is in the Course 6 catalog, you would call `classExists(&course6, 101)`.

`classExists` should return a `1` if a class with `subject_number` is present in a given department's catalog, `dc`. Otherwise, it should return `0`.

```
int classExists(struct deptCatalog *dc, uint16_t subject_number){

    int exists = 0;
    for (int i = 0; i < dept->num_classes; i++){
        uint32_t class = dc->dept_classes[i];
        if ((class & 0xFFFF) == subject_number){
            exists = 1;
        }
    }
    return exists;









}
```

C. (5 points) Write a function, `addClass`, that has two arguments:

- `deptCatalog *dc`: A pointer to a department catalog struct, as previously defined.
- `uint16_t new_subject`: The subject number, as previously defined, to add to the department.

`addClass` should first check if the department already offers a class with the subject number `new_subject`. If it does not, `addClass` should add a class with that subject number to the department struct. Specifically, it should:

- Place the new class number (containing both the department number and subject number) at the lowest unoccupied address in the `mit_classes` array.
- Update the number of classes offered by the department.

Ex: To add 6.190 to the Course 6 department catalog, you'd call `addClass(&course6, 190)`.

You may assume that a correct implementation of `classExists` from Part B is available to use.

```
void addClass(struct deptCatalog *dc, uint16_t new_subject){

    int exists = classExists(dc, new_subject);
    if (!exists){
        uint32_t new_class = new_subject | (dc->dept << 24);
        dc->dept_classes[num_classes] = new_class;
        dc->num_classes++;
    }









}
```

Problem 5. Logs (22 points)

Computing $\log_2(x)$ is readily achievable on a digital computer due to the inherent base 2 nature of the underlying binary representation. Computing logarithms of an arbitrary base of the form $\log_a(x)$ is not so easy, however. There is a workaround, though! We can use the change of base logarithm formula:

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

with $b = 2$. Therefore, to compute a logarithm of an arbitrary base for an integer, you just need to be able to:

1. Calculate the log-base-2 of a number
2. Divide.

These operations are implemented by two RISC-V assembly procedures, `ilog2` and `idiv`. In addition, a higher-level procedure, named `log_a_x` is created and utilizes `ilog2` and `idiv` as discussed above. The source code for these three procedures is shown on the next page.

The following line of code is run. At various points throughout the program (denoted `TIME POINT X`), the values in certain memory locations and registers are saved. Some of these values are shown in the table on the next page. Using the code below and those values, fill in the missing cells in the table.

```
jal ra, log_a_x    # <--------- TIME POINT 0 (after this line is executed)
```

```
 1 ilog2: # produce ilog2 of a0
 2   addi sp, sp, -4
 3   sw ra, 0(sp)
 4   addi t1, zero, 1
 5   blt t1, a0, ilog_else
 6   addi a0, zero, 0
 7   beq zero, zero, ilog_ret
 8 ilog_else:
 9   srli a0, a0, 1
10   jal ra, ilog2
11   addi a0, a0, 1
12 ilog_ret:
13   lw ra, 0(sp)
14   addi sp, sp, 4
15   jalr zero, 0(ra)
16
17 idiv: #produce idiv of a0/a1
18   addi sp, sp, -4
19   sw ra, 0(sp)
20   addi t1, zero, 0
21   bge a0, a1, idiv_else
22   addi a0, zero, 0
23   beq zero, zero, idiv_ret
24 idiv_else:
25   sub a0, a0, a1
26   jal ra, idiv
27   addi a0, a0, 1
28 idiv_ret:
29   lw ra, 0(sp)
30   addi sp, sp, 4
31   jalr zero, 0(ra)
32
33 log_a_x: #compute the log of a0 in base a1
34   addi sp, sp, -12
35   sw ra, 0(sp)
36   sw s0, 4(sp)
37   sw s1, 8(sp)
38   add s0, a0, zero   #   <--------- TIME POINT 1 (after line 38 executed)
39   addi a0, a1, 0
40   jal ra, ilog2
41   addi s1, a0, 0
42   addi a0, s0, 0     #   <--------- TIME POINT 2 (after line 42 executed)
43   jal ra, ilog2
44   addi a1, s1, 0
45   jal ra, idiv
46   lw s1, 8(sp)       #   <--------- TIME POINT 3 (after line 46 executed)
47   lw s0, 4(sp)
48   lw ra, 0(sp)
49   addi sp, sp, 12
50   jalr zero, 0(ra)   #   <--------- TIME POINT 4 (after line 50 executed)
```

Complete this table for Problem 5, using the code on the previous page.

Address	TIME POINT 0	TIME POINT 1	TIME POINT 2	TIME POINT 3	TIME POINT 4
0x3fc93f00	0xffffffff	0xffffffff	0xffffffff	0xffffffff	0xffffffff
0x3fc93f04	0xa5a5a5a5	0xa5a5a5a5	0xa5a5a5a5	0x42001620	0x42001620
0x3fc93f08	0xa5a5a5a5	0xa5a5a5a5	0xa5a5a5a5	0x42001620	0x42001620
0x3fc93f0c	0x00000004	0x00000004	0x00000004	0x42001620	0x42001620
0x3fc93f10	0x000007c2	0x000007c2	0x000007c2	0x42001620	0x42001620
0x3fc93f14	0x00000123	0x00000123	0x00000123	0x42001650	0x42001650
0x3fc93f18	0xffffffff	0xffffffff	0x42001620	0x42001650	0x42001650
0x3fc93f1c	0x00000123	0x00000123	0x42001620	0x42001650	0x42001650
0x3fc93f20	0x420165b0	0x420165b0	0x4200167c	0x42001690	0x42001690
0x3fc93f24	0x3fc91000	0x4201540a	0x4201540a	0x4201540a	0x4201540a
0x3fc93f28	0x3fc91000	0x0000000f	0x0000000f	0x0000000f	0x0000000f
0x3fc93f2c	0x00000011	0x00000001	0x00000001	0x00000001	0x00000001
0x3fc93f30	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x3fc93f34	0x00000111	0x00000111	0x00000111	0x00000111	0x00000111
Register	**TIME POINT 0**	**TIME POINT 1**	**TIME POINT 2**	**TIME POINT 3**	**TIME POINT 4**
a0	0x0000008a	0x0000008a	**0x0000008a**	**0x00000003**	**0x00000003**
a1	0x00000005	0x00000005	0x00000005	**0x00000002**	**0x00000002**
s0	0x0000000f	**0x0000008a**	**0x0000008a**	**0x0000008a**	**0x0000000f**
s1	0x00000001	0x00000001	**0x00000002**	**0x00000001**	**0x00000001**
ra	0x4201540a	0x4201540a	**0x4200167c**	**0x42001690**	**0x4201540a**
t1	0x0000000a	0x0000000a	**0x00000001**	**0x00000000**	**0x00000000**
sp	0x3fc93f30	**0x3fc93f24**	**0x3fc93f24**	**0x3fc93f24**	**0x3fc93f30**

Problem 6: RISC-Y Business (12 points)
Please convert the following C expressions into equivalent RISC-V assembly language instructions. Full credit will be given for correct solutions that minimize the number of RISC-V instructions used. You may assume all question parts are independent.

These are all of the previously defined C variables and their types:

- `x (int *)`
- `y (int)`
- `z (int)`

Assume:

- `x = a0`
- `y = a1`
- `z = a2`
- `q = a3`

You may use any RISC-V registers in your answer, provided that the functionality is equivalent to that given in the C code. **You may not use pseudoinstructions in your answers.**

A. (2 points) `int q = *x;`

```
lw a3, 0(a0)
```

B. (2 points) `int q = *(x + y);`

```
slli t0, a1, 2
add t0, a0, t0
 lw a3, 0(t0)
```

C. (2 points) `int q = *(x + 1);`

```
lw a3, 4(a0)
```

D. (2 points) `int q = 5 * y;`

```
slli a3, a1, 2
add a3, a3, a1
```

E. (2 points) `int q = y > z;`

```
slt a3, a2, a1
```

F. (2 points) `*x = 0x90007101;`

```
lui t0, 0x90007
addi t0, t0, 0x101
sw t0, 0(a0)
```

Problem 7. Let's Call it a Night (12 points)

Translate the C function, `hypotenuse`, into a RISC-V assembly procedure. You may assume that `exp` and `sqrt` are already defined and that these procedures follow calling convention and work as expected, but you cannot make any other assumptions about their implementations. **You must use them just as they are used in the C program.** Your assembly procedure must adhere to RISC-V calling conventions. **Additionally, there are only 12 bytes available to use on the stack, so your implementation cannot use more than 12 bytes of the stack.**

C Implementation:

```
int hypotenuse(int a, int b){
    return sqrt(exp(a, 2) + exp(b, 2));
}
```

RISC-V Implementation

```
hypotenuse:

    addi sp, sp, -12
    sw ra, 0(sp)
    sw s0, 4(sp)
    sw a1, 8(sp)
    addi a1, zero, 2
    jal ra, exp
    add s0, zero, a0
    lw a0, 8(sp)
    addi a1, zero, 2
    jal ra, exp
    add a0, a0, s0
    jal ra, sqrt
    lw s0, 4(sp)
    lw ra, 0(sp)
    addi sp, sp, 12
    jalr x0, 0(ra)
```

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

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