MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.S077(6.1900)(6.0004): Introduction to Low-level Programming in C and Assembly

Fall 2022, Quarter 1

Name:	**Kerberos:**
	MIT ID #:

#1 (16)	#2 (18)	#3 (13)	#4 (9)	#5 (16)	#6 (14)	#7 (14)	Total

Enter your answers in the spaces designated in each problem. Show your work for potential partial credit. You can use the extra white space and the backs of the pages for scratch work.

Problem 1. Binary Arithmetic (16 points)

A. (4 points): What is `0x68 ^ (0x9C | 0x5A)`? Provide your result in both unsigned 8-bit binary and unsigned 8-bit hexadecimal.

Result in unsigned 8-bit binary (0b):

Result in unsigned 8-bit hexadecimal (0x):

B. (2 points): What is the result of `((0b001 > 0b101) && 0b100) == 0b001)`? Assume all numbers are unsigned. Provide your result in decimal.

Result in Decimal:

C. (4 points): What are `14` and `31` in 8-bit 2's complement notation? What is `-31` in 8-bit 2's complement notation? **Show** how to compute `14-31` using 2's complement addition. What is the result in 8-bit 2's complement notation?

14 in 8-bit 2's complement notation (0b):

31 in 8-bit 2's complement notation (0b):

–31 in 8-bit 2's complement notation (0b):

14–31 in 8-bit 2's complement notation (0b):

D. (2 points): How many bits are required to encode decimal values ranging from `-128` to `127` in two's complement representation? How many bits are required to encode decimal values ranging from 0 to 127 in unsigned binary representation? Provide your answer in decimal.

Bits required for two's complement (in decimal):

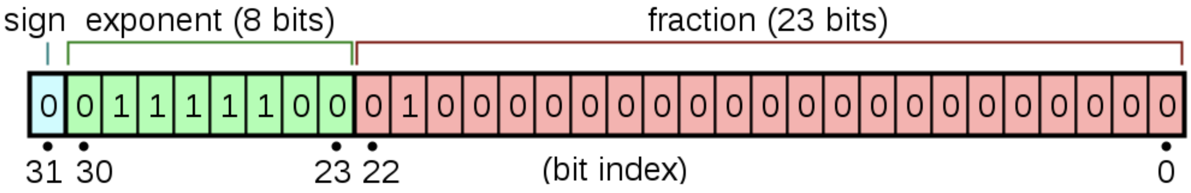
Bits required for unsigned binary (in decimal):

E. (2 points) What is the result of the **logical** right shift `0b11011010 >> 2` in **2's complement** notation? What is the result of the **arithmetic** right shift `0b11011010 >> 2` in **2's complement** notation? Provide your answer in binary.

Logical right shift (in binary):

Arithmetic right shift (in binary):

G. (2 Points) What is the decimal equivalent of the 32-bit floating point number `0x41080000`? The format of 32-bit floating point encoding is shown below.
Show your work.



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

Decimal Equivalent:

Problem 2. What If (18 points)

Suppose that we have a function `int is_uppercase(char c)` that returns `1` if `c` is an uppercase alphabetical character, and `0` otherwise. Similarly, suppose also that we have a function `int is_lowercase(char c)` that returns `1` if `c` is a lowercase alphabetical character, and `0` otherwise.

Find below another function `flip_case`, which flips the case of each character in the given string.

```
// Given a string, flip the case of each alphabetical character.
void flip_case(char *x) {
   while (*x != 0) {
       if (is_uppercase(*x)) {
           *x += 'a' - 'A';  // e.g., 'G' becomes 'g'
       } else if (is_lowercase(*x)) {
           *x += 'A' - 'a';  // e.g., 'g' becomes 'G'
       }
       x++; // <- For part A and B
   }
}
```

A. (6 points) Assume that this code is to be run on a 32-bit RISC-V processor. Determine if each of the following candidates is functionally equivalent to `x++;` in the above program.

Candidate	CIRCLE ONE:
`&x = &x + 1;`	YES NO
`*(&x) = x + 1;`	YES NO
`x = (char *)((uint32_t)x + 4);`	YES NO

B. (4 points) Suppose we are instead to replace `x++;` in the implementation of `flip_case` with `f(&x);` and implement `f` as follows.

```
void f(__1__ x) {
   __2__++;
}
```

Fill in the two blanks so that `flip_case` would retain its correctness. Pay attention to the existing code that precedes and/or follows the blanks; you are not allowed to modify or get rid of it. **Take note of the order of operations.**

Blank 1:	Blank 2:

C. (8 points) Consider the following code, which makes a call to `flip_case`. The implementation of `flip_case` is repeated below for your convenience.

```
#include <stdio.h>
int is_uppercase(char x); // implemented elsewhere
int is_lowercase(char x); // implemented elsewhere
// Given a string, flip the case of each alphabetical character.
void flip_case(char *x) {
   while (*x != 0) {
      if (is_uppercase(*x)) {
         *x += 'a' - 'A';  // e.g., 'G' becomes 'g'
      } else if (is_lowercase(*x)) {
         *x += 'A' - 'a';  // e.g., 'g' becomes 'G'
      }
      x++;
   }
}
int main() {
   char str[] = "ababABAB";
   char *p = str;
   char **q = &p;
   flip_case(???); // <- REPLACE HERE
   printf("%s\n", str);
   return 0;
}
```

For each expression, determine the string that the program above would print if we were to replace **???** with that expression. If the code would not compile, write "WON'T COMPILE". If the exact output cannot be determined, write "CAN'T TELL".

Expression	**Answer**
`p`	
`str + 2`	
`(*q) + 8`	
`&(*(&p))[4] - 1`	

Problem 3. C structs (13 points)

When communicating with our RISC-V microcontroller, we usually did so through the serial monitor embedded in our IDE. A struct called `SerialBuffer` is defined below to represent our controller's serial communication buffer, which is responsible for handling incoming Serial messages.

```
struct SerialBuffer{
  char termChar;        // char to stop reading at
  char charBuffer[64];  // 64 element wide buffer to store characters in
  uint8_t size;         // number of chars stored in buffer
};
```

A. (3 points) First let's model how the buffer receives data. An example of an empty `SerialBuffer`, with the newline as the terminating character, is shown below:

```
struct SerialBuffer buf;
buf.termChar = '\n';
buf.size = 0;
```

When the buffer receives a character, the buffer adds the char to the `charBuffer` array at the smallest available index. It then also increments the active buffer size count by one.

Write a function **`receiveChar`** that takes in a `SerialBuffer` struct (by value) and a character to add to the buffer and returns an updated `SerialBuffer` instance.
Assume the buffer has enough space for an incoming character, `c`.

```
struct SerialBuffer receiveChar(struct SerialBuffer buf, char c){









}
```

B. (4 points): Sometimes we want to peek into our serial buffer without removing any characters from the buffer. Write a function called **`peekChars`** that receives **a pointer to a buffer instance** and returns how many characters are in the buffer **up to and including** the first termination character (reflected in `termChar`). Assume the buffer contains at least one terminating character.

```
int peekChars(struct SerialBuffer *buf){




}
```

C. (6 points): Consider the case where we want to read from an active `SerialBuffer` instance.

```
struct SerialBuffer buf;
```

This buffer has already been populated with multiple characters and at least one terminating character.

Create a function, **`readChars`**, that reads characters from the buffer **up to and including the buffer's termination character**. Store this string of characters as a properly terminated C-string in `char *message`, which you can assume will be large enough to store the resulting message.

Be sure to update the buffer by both updating the `charBuffer` array and the buffer size.
As an example:

```
// up to this point buf has been populated and contains:
// buf.charBuffer = {'h','e','l','l','o','\n','b','y','e','\n', ...}
```

```
printf("%c", buf.termChar); // prints: "\n"
printf("%d", buf.size);     // prints: "10"

char msg[65];           // large enough to store message from buffer
readChars(&buf, msg);   // move chars up to termChar from buffer to msg

printf("%s", msg);      // prints: "hello\n"
printf("%d", buf.size); // prints: "4"

// now at this point buf.charBuffer = {'b','y','e','\n', ...}
```

Note that `readChars` is passed a **pointer** to a `SerialBuffer`.
Be sure to leverage the **`peekChars`** function you just wrote.

```
void readChars(struct SerialBuffer *buf, char *message){



}
```

Problem 4. An Average Filter (9 points)

Suppose we have a function `void find_mean(const float *arr, int n, float *mean)` that goes through the array `arr` of `n` real numbers, finds the arithmetic mean (i.e. average), and puts the result in the variable pointed to by `mean`.

Implement the function `mean_filter`, which takes an array `input` of `num_elems` real numbers, finds the arithmetic mean for each contiguous window of exactly `window_size` elements, and puts the results in the array `output`.

```
void find_mean(const float *arr, int n, float *mean); // Defined elsewhere.

void mean_filter(const float *input, int num_elems, int window_size, float *output) {
   for (int i = 0; i+window_size-1 < num_elems; i++) {
      float buffer;
      float *ptr = &buffer;
      find_mean(___(A)___, ___(B)___, ___(C)___);
      *(output+i) = ___(D)___;
   }
}

// For example, if num_elems=4, input={3, 2, 7, 6}, window_size=3, then
// there are two contiguous windows, each with the following arithmetic means:
// output[0] = (3+2+7)/3.0 = 4.0
// output[1] = (2+7+6)/3.0 = 5.0
```

Fill in the blank (A):

Fill in the blank (B):

Fill in the blank (C):

Circle ALL correct answers (D):

`buffer`	`*buffer`	`&buffer`
`ptr`	`*ptr`	`&ptr`
`ptr[0]`	`*ptr[0]`	`&ptr[0]`

Problem 5. Assembly Language (16 points)

(A) (2 points) What is the **hexadecimal** encoding of the instruction `srai t3, a2, 6`? You can use the template below to help you with the encoding. Please show your work for partial credit.

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
0100000	shamt	rs1	funct3	rd	opcode

`srai t3, a2, 6` **instruction encoding (0x):**________________________

For the RISC-V instruction sequences below, provide the hexadecimal values of the specified registers after each sequence has been executed. Assume that each sequence execution ends when it reaches the `end` label. Also assume that all registers are initialized to 0 before execution of each sequence begins.

(B) (4 points)

The first instruction executed is located at address `0x20`.

```
. = 0x20
      li x11, 0x600
      lw x11, 0x0(x11)
      bge x11, x0, L1
      xori x12, x11, 0xA55
      j end

L1:  srli x12, x11, 8
end:

. = 0x600
X:   .word 0xC0C0A0A0
```

Value left in x11: 0x______________________

Value left in x12: 0x______________________

(C) (10 points)

The first instruction executed is located at address `0x100`.

```
. = 0x20
f:
      slli x13, x12, 8
      ret

. = 0x100
      lui x11, 0x3
      lw x12, 0x4(x11)
      jal x1, f
      ori x14, x1, 0xC2
end:

. = 0x3000
.word 0x11112222
.word 0x22224444
.word 0x33336666
```

Value left in x1: 0x______________________

Value left in x11: 0x_____________________

Value left in x12: 0x_____________________

Value left in x13: 0x_____________________

Value left in x14: 0x_____________________

Problem 6. Calling Convention (14 points)

You decided to write Snake in RISC-V assembly. You implement a `drawBoard` function to render the game board. `drawBoard` uses one helper function, `setPixel`. to set a given pixel to be 0 (off) or 1 (on). It's C function signature is shown below:

```
void setPixel(uint32_t *screen_buffer, uint8_t location, uint8_t val);
```

You can assume that setPixel works as expected and follows calling convention. You do not have access to the assembly implementation of `setPixel,` so you cannot make any further assumptions about its implementation.

Unfortunately, your program does not work, and you suspect that it is due to calling convention. Please add appropriate instructions (**either increment/decrement stack pointer, load word from stack, or save word to stack only**) into the blank spaces on the right to make `drawBoard` follow calling convention. You can assume that `drawBoard` will work as expected once it follows the calling convention.

If the procedure already follows calling convention, write NO INSTRUCTIONS NEEDED. For full credit, you should only save registers that must be saved onto the stack, restore registers that must be restored, and minimize the number of instructions used. You may not need to use all the blank lines.

```
# drawBoard Arguments:
# (1) screen_buffer
# (2) locations: array holding locations of snake segments on board
# (3) num_locations: length of locations array.
# (4) food: location of food

drawBoard:
      slli a2, a2, 2
      add a2, a2, a1
      mv s0, a0
loop:
      bge a1, a2, end
      mv a0, s0
      lw a1, 0(a1)
      li a2, 1
      call setPixel
      addi a1, a1, 4
      j loop
end:
      mv a0, s0
      mv a1, a3
      li a2, 1
      call setPixel
      ret
```

```
drawBoard:

      ____________________________

      ____________________________

      ____________________________

      ____________________________

      ____________________________

      ____________________________

      slli a2, a2, 2
      add a2, a2, a1
      mv s0, a0

      ____________________________

      ____________________________

      ____________________________
```

```
# code copied here
drawBoard:
      slli a2, a2, 2
      add a2, a2, a1
      mv s0, a0
loop:
      bge a1, a2, end
      mv a0, s0
      lw a1, 0(a1)
      li a2, 1
      call setPixel
      addi a1, a1, 4
      j loop
end:
      mv a0, s0
      mv a1, a3
      li a2, 1
      call setPixel
      ret
```

```
loop:
      bge a1, a2, end
      ___________________________
      ___________________________
      ___________________________
      ___________________________
      ___________________________
      ___________________________
      mv a0, s0
      lw a1, 0(a1)
      li a2, 1
      ___________________________
      ___________________________
      ___________________________
      ___________________________
      ___________________________
      ___________________________
      call setPixel
      ___________________________
      ___________________________
      ___________________________
      ___________________________
      ___________________________
      ___________________________
      addi a1, a1, 4
      j loop
```

```
# code copied here
drawBoard:
      slli a2, a2, 2
      add a2, a2, a1
      mv s0, a0
loop:
      bge a1, a2, end
      mv a0, s0
      lw a1, 0(a1)
      li a2, 1
      call setPixel
      addi a1, a1, 4
      j loop
end:
      mv a0, s0
      mv a1, a3
      li a2, 1
      call setPixel
      ret
```

```
end:
      ___________________________

      ___________________________

      ___________________________

      ___________________________

      ___________________________

      ___________________________

      mv a0, s0
      mv a1, a3
      li a2, 1

      ___________________________

      ___________________________

      ___________________________

      ___________________________

      ___________________________

      call setPixel

      ___________________________

      ___________________________

      ___________________________

      ___________________________

      ___________________________

      ___________________________

      ret
```

Problem 7. Stack Detective (14 points)

Consider the following C function which takes an array of unsigned 32 bit integers `a` of length `b` and computes their product. We don't have a multiply instruction in our RV32I system, so we use the `mult` procedure (which you used in class, provided in appendix for reference) in order to actually do the multiplication:

```
int arrayProd(uint32_t* a, uint32_t b){
   // uint32_t *a: pointer to array
   // uint32_t b: length of array
   if (b == 1) {
       return a[0];
   }else {
       // multiply both numbers:
       return mult(arrayProd(a+1, b-1), a[0]);
   }
}
```

The equivalent assembly procedure for this function is below:

```
arrayProd:
   lw t0, 0(a0)
   li a3, 1
   beq a1, a3, end
   addi sp, sp, -8
   sw ra, 0(sp)
   sw t0, 4(sp)
   # SAMPLE POINT - prints ra, sp, a0, a1, a part of the stack
   addi a0, a0, 4
   addi a1, a1, -1
   jal arrayProd
   lw a1, 4(sp)
   jal mult  # returns product of a0 and a1 (see appendix)
   ______________
   lw ra, 0(sp)
   addi sp, sp, 8
end:
   mv a0, t0
   ret
```

Note the sample point line above. When this line is encountered, the four registers `ra, sp, a0,` and `a1` are printed as well as a region of the stack.

A. (2 points): What line of assembly should be substituted into the blank line in the `arrayProd` procedure above?

Line of assembly:

B. (12 points): A user creates an array and passes it and its length into `arrayProd`. Immediately prior to and immediately after the procedure call, a sample of the `ra, sp, a0,` and `a1` is collected as well as a region of the stack.

```
# SAMPLE POINT – prints ra, sp, a0, a1, a part of the stack
jal arrayProd
# SAMPLE POINT – prints ra, sp, a0, a1, a part of the stack
```

When the code is run, six sample snapshots are generated (including the one from before the initial call to `arrayProd`). They are shown **in chronological order** below:

```
#1  sp  =0x00080280 ra  =0x00000000
    a0  =0x00004000 a1  =0x00000005
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000035c
0x80274: 0x00000011
0x80278: 0x00000808
0x8027c: 0x0000a321
0x80280: 0x00000781
```

```
#2  sp  =0x00080278 ra  =0x00000204
    a0  =0x00004000 a1  =0x00000005
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000035c
0x80274: 0x00000011
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#3  sp  =0x00080270 ra  =0x0000025C
    a0  =0x00004004 a1  =0x00000004
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x00000008
0x8026c: 0x00000001
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#4   sp  =0x00080268 ra  =0x0000025C
     a0  =0x00004008 a1  =0x00000003
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000035c
0x80264: 0x00000011
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#5  sp  =0x00080260 ra  =0x0000025C
    a0  =0x0000400C a1  =0x00000002
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

```
#6  sp  =0x00080280 ra  =0x00000204
    a0  =0x000000F0 a1  =0x00000000
Address: Data:
0x80258: 0x000ff3af
0x8025c: 0x00000018
0x80260: 0x0000025c
0x80264: 0x00000008
0x80268: 0x0000025c
0x8026c: 0x00000005
0x80270: 0x0000025c
0x80274: 0x00000003
0x80278: 0x00000204
0x8027c: 0x00000001
0x80280: 0x00000781
```

Answer the following questions:

(3 points) What is the hexadecimal address of the instruction that originally calls `arrayProd`?

Address: (0x)

(3 points) What is the hexadecimal address of the instruction that is responsible for the recursive calls to `arrayProd`?

Address: (0x)

(2 points) What is the hexadecimal address of the array `a` provided to the initial call of `arrayProd`?

Address: (0x)

(4 points) Specify a C array below that is identical to the one the user must have handed into `arrayProd`.

```
uint32_t a[    ] =
```

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.

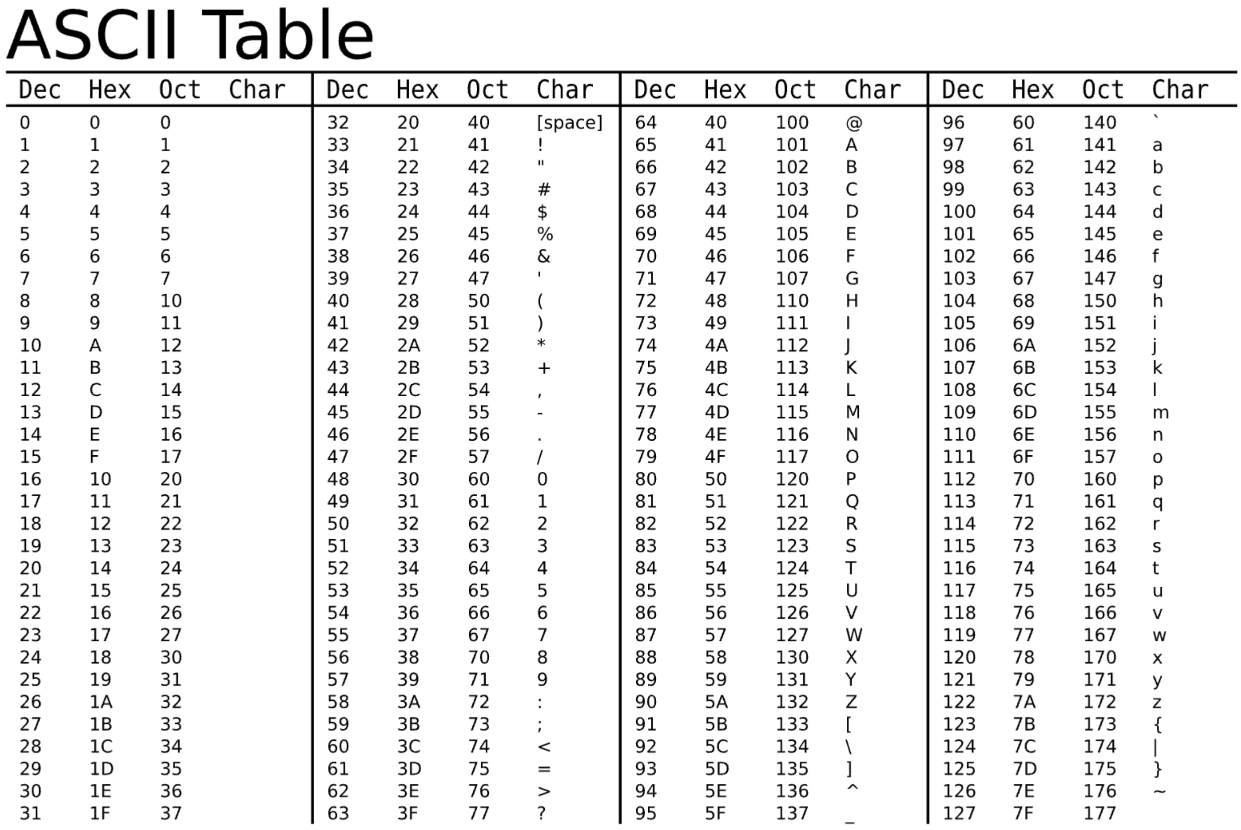
char *strchr(const char *str, int c) - searches for the first occurrence of the character c (an unsigned char) in the string pointed to by the argument str. This returns a pointer to the first occurrence of the character c in the string str, or NULL if the character is not found.

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: mult function

```
mult:
   bge a0, a1, init
   mv t0, a0
   mv a0, a1
   mv a1, t0
init:
   mv t0, zero
loop:
   beq  a1, zero, done
   addi a1, a1, -1
   add t0, t0, a0
   j loop
done:
   mv a0, t0
   ret
```