

## 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, uint32\_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`.

**int sprintf(char \*str, const char \*format, ...)** - writes formatted output to the string pointed to by `str`, using the format string and additional arguments in the same way as `printf`. Returns the number of characters written (excluding the null terminator), or a negative value on error.

**uint32\_t strlen(const char \*str)** - computes the length of the string pointed to by `str`, not including the terminating `'\0'` character. Returns the number of characters in the string.

**int strncmp(const char \*str1, const char \*str2, uint32\_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, char 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, char 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: strtok Operation

## CHAR ARRAY CREATION:

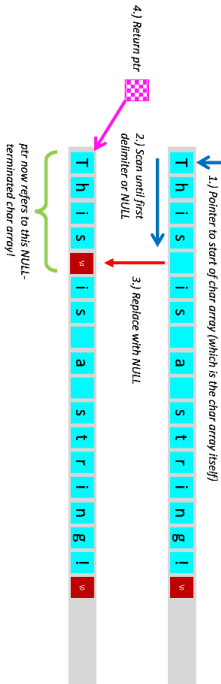
char message[22] = "This is a message!";

*Array/Array Address values (ASCII)*



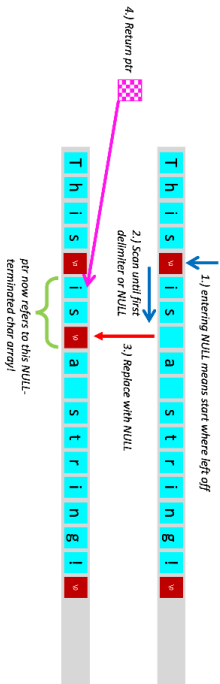
### FIRST CALL:

char\* ptr = strtok(message, " ");



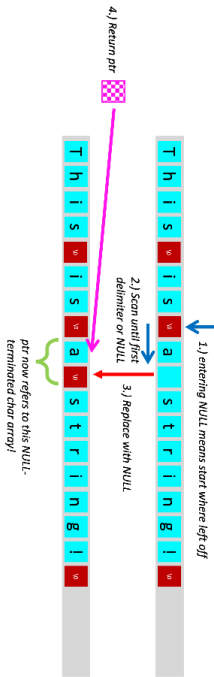
### SECOND CALL:

char\* ptr = strtok(NULL, " ");



### THIRD CALL:

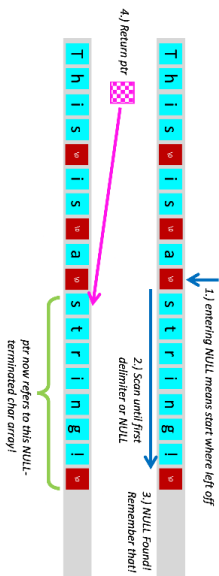
char\* ptr = strtok(NULL, " ");



- = assignment/overwrite
- = some non-NULL char
- = path of scan/checking
- ↖ = returned pointer

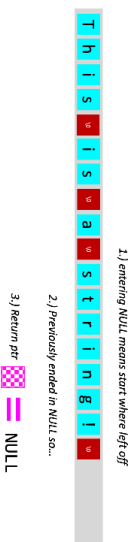
### FOURTH CALL:

char\* ptr = strtok(NULL, " ");



### FIFTH, ETC... CALL:

char\* ptr = strtok(NULL, " ");



## Appendix 3: 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 4: 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	
	( <i>type</i> )	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size 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	
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	

## MIT 6.190 ISA Reference Card: Instructions

Instruction	Syntax	Description	Execution
LUI	<b>lui</b> rd, luiConstant	Load Upper Immediate	reg[rd] <= luiConstant « 12
JAL	<b>jal</b> rd, label	Jump and Link	reg[rd] <= pc + 4 pc <= label
JALR	<b>jalr</b> rd, offset(rs1)	Jump and Link Register	reg[rd] <= pc + 4 pc <= {(reg[rs1] + offset)[31:1], 1'b0}
BEQ	<b>beq</b> rs1, rs2, label	Branch if =	pc <= (reg[rs1] == reg[rs2]) ? label : pc + 4
BNE	<b>bne</b> rs1, rs2, label	Branch if ≠	pc <= (reg[rs1] != reg[rs2]) ? label : pc + 4
BLT	<b>blt</b> rs1, rs2, label	Branch if < (Signed)	pc <= (reg[rs1] < <sub>s</sub> reg[rs2]) ? label : pc + 4
BGE	<b>bge</b> rs1, rs2, label	Branch if ≥ (Signed)	pc <= (reg[rs1] >= <sub>s</sub> reg[rs2]) ? label : pc + 4
BLTU	<b>bltu</b> rs1, rs2, label	Branch if < (Unsigned)	pc <= (reg[rs1] < <sub>u</sub> reg[rs2]) ? label : pc + 4
BGEU	<b>bgeu</b> rs1, rs2, label	Branch if ≥ (Unsigned)	pc <= (reg[rs1] >= <sub>u</sub> reg[rs2]) ? label : pc + 4
LW	<b>lw</b> rd, offset(rs1)	Load Word	reg[rd] <= mem[reg[rs1] + offset]
SW	<b>sw</b> rs2, offset(rs1)	Store Word	mem[reg[rs1] + offset] <= reg[rs2]
ADDI	<b>addi</b> rd, rs1, constant	Add Immediate	reg[rd] <= reg[rs1] + constant
SLTI	<b>slti</b> rd, rs1, constant	Compare < Immediate (Signed)	reg[rd] <= (reg[rs1] < <sub>s</sub> constant) ? 1 : 0
SLTIU	<b>sltiu</b> rd, rs1, constant	Compare < Immediate (Unsigned)	reg[rd] <= (reg[rs1] < <sub>u</sub> constant) ? 1 : 0
XORI	<b>xori</b> rd, rs1, constant	Xor Immediate	reg[rd] <= reg[rs1] ^ constant
ORI	<b>ori</b> rd, rs1, constant	Or Immediate	reg[rd] <= reg[rs1]   constant
ANDI	<b>andi</b> rd, rs1, constant	And Immediate	reg[rd] <= reg[rs1] & constant
SLLI	<b>slli</b> rd, rs1, shamt	Shift Left Logical Immediate	reg[rd] <= reg[rs1] « shamt
SRLI	<b>srl</b> rd, rs1, shamt	Shift Right Logical Immediate	reg[rd] <= reg[rs1] » <sub>u</sub> shamt
SRAI	<b>srai</b> rd, rs1, shamt	Shift Right Arithmetic Immediate	reg[rd] <= reg[rs1] » <sub>s</sub> shamt
ADD	<b>add</b> rd, rs1, rs2	Add	reg[rd] <= reg[rs1] + reg[rs2]
SUB	<b>sub</b> rd, rs1, rs2	Subtract	reg[rd] <= reg[rs1] - reg[rs2]
SLL	<b>sll</b> rd, rs1, rs2	Shift Left Logical	reg[rd] <= reg[rs1] « reg[rs2][4:0]
SLT	<b>slt</b> rd, rs1, rs2	Compare < (Signed)	reg[rd] <= (reg[rs1] < <sub>s</sub> reg[rs2]) ? 1 : 0
SLTU	<b>sltu</b> rd, rs1, rs2	Compare < (Unsigned)	reg[rd] <= (reg[rs1] < <sub>u</sub> reg[rs2]) ? 1 : 0
XOR	<b>xor</b> rd, rs1, rs2	Xor	reg[rd] <= reg[rs1] ^ reg[rs2]
SRL	<b>srl</b> rd, rs1, rs2	Shift Right Logical	reg[rd] <= reg[rs1] » <sub>u</sub> reg[rs2][4:0]
SRA	<b>sra</b> rd, rs1, rs2	Shift Right Arithmetic	reg[rd] <= reg[rs1] » <sub>s</sub> reg[rs2][4:0]
OR	<b>or</b> rd, rs1, rs2	Or	reg[rd] <= reg[rs1]   reg[rs2]
AND	<b>and</b> rd, rs1, rs2	And	reg[rd] <= reg[rs1] & reg[rs2]

Note: *luiConstant* is a 20-bit value. *offset* and *constant* are signed 12-bit values that are sign-extended to 32-bit values. *label* is a 32-bit memory address or its alias name. *shamt* is a 5-bit unsigned shift amount.

## MIT 6.190 ISA Reference Card: Pseudoinstructions

Pseudoinstruction	Description	Execution
<b>li</b> rd, liConstant	Load Immediate	reg[rd] <= liConstant
<b>mv</b> rd, rs1	Move	reg[rd] <= reg[rs1] + 0
<b>not</b> rd, rs1	Logical Not	reg[rd] <= reg[rs1] ^ ~1
<b>neg</b> rd, rs1	Arithmetic Negation	reg[rd] <= 0 - reg[rs1]
<b>j</b> label	Jump	pc <= label
<b>jal</b> label	Jump and Link (with ra)	reg[ra] <= pc + 4 pc <= label
<b>call</b> label		
<b>jr</b> rs1	Jump Register	pc <= reg[rs1] & ~1
<b>jalr</b> rs1	Jump and Link Register (with ra)	reg[ra] <= pc + 4 pc <= reg[rs1] & ~1
<b>ret</b>	Return from Subroutine	pc <= reg[ra]
<b>bgt</b> rs1, rs2, label	Branch > (Signed)	pc <= (reg[rs1] > <sub>s</sub> reg[rs2]) ? label : pc + 4
<b>ble</b> rs1, rs2, label	Branch ≤ (Signed)	pc <= (reg[rs1] <= <sub>s</sub> reg[rs2]) ? label : pc + 4
<b>bgtu</b> rs1, rs2, label	Branch > (Unsigned)	pc <= (reg[rs1] > <sub>u</sub> reg[rs2]) ? label : pc + 4
<b>bleu</b> rs1, rs2, label	Branch ≤ (Unsigned)	pc <= (reg[rs1] <= <sub>u</sub> reg[rs2]) ? label : pc + 4
<b>beqz</b> rs1, label	Branch = 0	pc <= (reg[rs1] == 0) ? label : pc + 4
<b>bnez</b> rs1, label	Branch ≠ 0	pc <= (reg[rs1] != 0) ? label : pc + 4
<b>bltz</b> rs1, label	Branch < 0 (Signed)	pc <= (reg[rs1] < <sub>s</sub> 0) ? label : pc + 4
<b>bgez</b> rs1, label	Branch ≥ 0 (Signed)	pc <= (reg[rs1] >= <sub>s</sub> 0) ? label : pc + 4
<b>bgtz</b> rs1, label	Branch > 0 (Signed)	pc <= (reg[rs1] > <sub>s</sub> 0) ? label : pc + 4
<b>blez</b> rs1, label	Branch ≤ 0 (Signed)	pc <= (reg[rs1] <= <sub>s</sub> 0) ? label : pc + 4

Note: *liConstant* is a 32-bit value.

## MIT 6.190 ISA Reference Card: Calling Convention

Registers	Symbolic names	Description	Saver
x0	zero	Hardwired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporary registers	Caller
x8-x9	s0-s1	Saved registers	Callee
x10-x11	a0-a1	Function arguments and return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

## MIT 6.190 ISA Reference Card: Instruction Encodings

31	25	24	20	19	15	14	12	11	7	6	0	
funct7			rs2		rs1	funct3		rd	opcode			R-type
imm[11:0]					rs1	funct3		rd	opcode			I-type
imm[11:5]			rs2	rs1	funct3		imm[4:0]		opcode			S-type
imm[12:10:5]			rs2	rs1	funct3		imm[4:1:11]		opcode			B-type
imm[31:12]								rd	opcode			U-type
imm[20:10:1:11:19:12]								rd	opcode			J-type

## RV32I Base Instruction Set (MIT 6.190 subset)

imm[31:12]								rd	0110111			LUI
imm[20:10:1:11:19:12]								rd	1101111			JAL
imm[11:0]				rs1	000		rd	1100111			JALR	
imm[12:10:5]			rs2	rs1	000		imm[4:1:11]		1100011		BEQ	
imm[12:10:5]			rs2	rs1	001		imm[4:1:11]		1100011		BNE	
imm[12:10:5]			rs2	rs1	100		imm[4:1:11]		1100011		BLT	
imm[12:10:5]			rs2	rs1	101		imm[4:1:11]		1100011		BGE	
imm[12:10:5]			rs2	rs1	110		imm[4:1:11]		1100011		BLTU	
imm[12:10:5]			rs2	rs1	111		imm[4:1:11]		1100011		BGEU	
imm[11:0]				rs1	010		rd	0000011			LW	
imm[11:5]			rs2	rs1	010		imm[4:0]		0100011		SW	
imm[11:0]				rs1	000		rd	0010011			ADDI	
imm[11:0]				rs1	010		rd	0010011			SLTI	
imm[11:0]				rs1	011		rd	0010011			SLTIU	
imm[11:0]				rs1	100		rd	0010011			XORI	
imm[11:0]				rs1	110		rd	0010011			ORI	
imm[11:0]				rs1	111		rd	0010011			ANDI	
0000000			shamt	rs1	001		rd	0010011			SLLI	
0000000			shamt	rs1	101		rd	0010011			SRLI	
0100000			shamt	rs1	101		rd	0010011			SRAI	
0000000			rs2	rs1	000		rd	0110011			ADD	
0100000			rs2	rs1	000		rd	0110011			SUB	
0000000			rs2	rs1	001		rd	0110011			SLL	
0000000			rs2	rs1	010		rd	0110011			SLT	
0000000			rs2	rs1	011		rd	0110011			SLTU	
0000000			rs2	rs1	100		rd	0110011			XOR	
0000000			rs2	rs1	101		rd	0110011			SRL	
0100000			rs2	rs1	101		rd	0110011			SRA	
0000000			rs2	rs1	110		rd	0110011			OR	
0000000			rs2	rs1	111		rd	0110011			AND	

- For JAL and branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU), the immediate encodes the target address as an offset from the current pc (i.e.,  $pc + imm = label$ ).
- Not all immediate bits are encoded. Missing lower bits are filled with zeros and missing upper bits are sign-extended.