Sum Complement and Parity checksum

COMP 273 Assignment 3 - Fall 2023, Dr. Elsaadawy

Available: 24 October 2023 - Due date: 09 November 2023

Submission instructions

All work must be your own, and must be submitted by MyCourses. **Include your name and student number in a comment at the top of your source file**. Also use comments at the top of your code to provide any other information you would otherwise include in a README file. Submit only one file, `Checksum.asm`, i.e., you will need to rename the provided file! Do not use a zip archive. **Check your submission** by downloading your submission from the server and checking that it was correctly submitted. You will not receive marks for work that is incorrectly submitted.

NO submissions through email will be accepted.

1 Overview

In this assignment, you will use the Run I/O console to calculate and validate text using checksum algorithms. A checksum is a small-sized block of data derived from another block of data for the purpose of detecting errors that may have been introduced during its transmission or storage. In this assignment, you will generate and validate text with two types of checksum algorithms: (1) a sum complement and (2) odd parity checksum.

1.1 Sum Complement Checksum

In this checksum algorithm, we add the ASCII codes for all characters in the input text as unsigned binary numbers, discarding any overflow bits, and append the two's complement of the total as the "checksum byte". To validate that calculated checksum, the receiver adds all the ASCII codes for all characters in the received text in the same manner, including the "checksum byte"; and the result should equate to zero. Otherwise, an error must have occurred.

For example, to calculate the sum complement checksum of an input text of "CAT", we should first add the ASCII code of "C", "A" and "T". Looking at the ASCII table shown on the right[1], we found that ASCII code of "C", "A", and "T" letters are 0x43, 0x41 and 0x54, respectively. The result of adding up those three ASCII codes is 0xD8. Then, two's complement of this value should be computed, which is 0x28, and this will be the generated sum complement checksum for "CAT" text. To validate, the validation function should again add the ASCII codes of letters in "CAT" with the calculated checksum, that is 0xD8 (calculated sum of the ASCII codes of "CAT" as shown above) + 0x28 (calculated sum complement checksum) which equals to 0x00. Then, the checksum validation is succeeded. Otherwise, checksum validation will be failed.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

1.2 Parity word checksum

We learned about the odd parity earlier in this course and it basically ensure that the total number of 1s in a transmitted text is an odd number. In particular, the ASCII byte code of each letter in the text is listed and then a parity byte should be calculated by deriving the parity bit for each group of bits in the same position. For example, the parity byte of text "CAT" should be calculated as following:

Note that this operation is executing on a bit level, i.e., to decide on Bit0 (LSB) value of the parity byte,

[1] complete ASCII table can be found here: https://www.ascii-code.com/

Letter	ASCII Code
C	0100 0011
A	0100 0001
T	0101 0100
Odd Parity Byte	1010 1001

we should check the number of bits that has value 1 in this LSB position for all the inputs, if it's odd number then we set Bit0 in the parity byte to 0, otherwise, it's set to 1.

For the validation process, it does the same operation between the ASCII codes of all characters in text in addition to the received Parity byte. In particular, the number of 1's is counted at each bit position of for each bit position of text's characters and the parity byte, and If the number of ones are odd, the corresponding bit in the validation byte is set to 0, meaning no error. Otherwise, a value of 1 is set to the corresponding bit in the validation byte to flag an error. Thus, the output validation byte should be 0x00 to indicate no error. The following table shows the validation output byte for the "CAT" example with "0xA9" being the received odd parity byte.

Letter	ASCII Code
C	0100 0011
A	0100 0001
T	0101 0100
Odd Parity Byte	1010 1001
Validation output Byte	0000 0000

Notes

- Remember that what is stored in memory for a character is the ASCII code for that character. ASCII code for each character can be represented in 8-bits (a byte).
- The ASCII codes for blank, newline (also called line feed), and dot/period are considered in the checksum calculation.

Provided code and files

You are provided some code to help you get started, along with a small collection of test files for testing purposes. Once you've completed the objectives of the assignment, you should be able to calculate the checksum of those files using any of the two aforementioned checksum algorithms.

The provided code implements a simple command menu to let you read a text file into a buffer, calculate the checksum of the buffer, validate the checksum of the buffer, write the buffer to a file, and quit. The code handles reading from and writing to files and a few other basic procedures to let you focus on the job of writing procedures for generating and validating checksum of text using the two aforementioned algorithms. Here follows an example session.

```
Commands (read, generate, validate, write, quit):r
Enter file name:file0.txt
CAT
Commands (read, generate, validate, write, quit):g
Choose checksum algorithm (1- Sum complement, 2- Odd Parity Byte): 1
(
Commands (read, generate, validate, write, quit):w
Enter file name:checksum0.txt
Commands (read, generate, validate, write, quit):v
Choose checksum algorithm (1- Sum complement, 2- Odd Parity Byte): 1
```

```
"Checksum validated"
Commands (read, generate, validate, write, quit):r
Enter file name:file0.txt
CAT
Commands (read, generate, validate, write, quit):g
Choose checksum algorithm (1- Sum complement, 2- Odd Parity Byte): 2
©
Commands (read, generate, validate, write, quit):w
Enter file name:checksum1.txt
Commands (read, generate, validate, write, quit):v
Choose checksum algorithm (1- Sum complement, 2- Odd Parity Byte):2
"Checksum validated"
Commands (read, generate, validate, write, quit):w
Enter file name:checksum0Error.txt
Commands (read, generate, validate, write, quit):v
Choose checksum algorithm (1- Sum complement, 2- Odd Parity Byte):1
"Checksum not validated"
Commands (read, generate, validate, write, quit):q
```

Note that MARS will open files by default in the directory in which it is started. You can place your files in that location for easy reading and writing, or specify the full path, or likewise specify the directory in which to start the MARS jar.

Additionally, the calculated checksum has to be added to the text buffer to be written to the file using the write command in the menu for the validation process.

2 Checksum generation (5 marks)

Implement GenerateCheckSum procedure that generates the checksum of input text using one of the aforementioned algorithms according to the user choice. This procedure takes one argument, that is a null terminated ASCII string (i.e., the text buffer) and return one output that is the generated checksum as a character (1 byte). Please note that the ASCII code for the null character is 0x00. Besides printing out the calculated checksum to the user, the procedure should add it to the text buffer as mentioned above.

3 Validate CheckSum (5 marks)

Implement ValidateCheckSum procedure that validates the checksum of input text using one of the aforementioned algorithms according to the user choice. This procedure takes one argument that is a null terminated ASCII string to validate (i.e., the text buffer), and displays one message: Either "Checksum validated" or "Checksum not validated".

You are free to organize your solution as you please, **but must use register calling conventions and the stack as necessary**. Feel free to write additional procedures as necessary or even add commands to the menu that might help (e.g., print the text in the buffer).

HOW IT WILL BE GRADED

- Your program must execute to be graded.
- The grader should test your code against some test text files and compare the output with the expected one (part marks will be given).
- Not following the submission instruction will result in losing 1.5 marks.
- Not following MIPs register convention will result in losing 2.5 marks.