T34 Assembler Documentation

Build and Run

WARNING: The assembler was only tested with Python 3.10.8 on Windows and likely does not work on any Python version below 3.10.0

INSTRUCTIONS

1. If haven't already, unzip "T34Assembler.tgz" then unzip the resulting "T34Assembler.tar"
2. Navigate to the directory/module named "Assembler" containing "main.py" (location may differ depending on how you completed the previous step).
3. Run "main.py" with the path to the desired source file as a command-line argument. Could look something like this:

   ```
   py main.py "..\sample_1.s"
   ```

   The format of the source file must be consistent with the format shown in the below section labeled **T34 Assembler**.

RESULTS

The console will output the generated assembly collated with their corresponding lines from the source code. This is then followed by the calculated symbol table in alphabetical then numerical order.

Also generated will be the object file with a name similar to the source file. In the above example the object file would be "..\sample_1.o". If a file with said name already exists, it will be overwritten. The object file will only contain the assembly (i.e. memory addresses and byte code).

Note: If any exceptions occur during assembly, the object file will not be generated.

Architecture

The T34 is an 8-bit byte-addressable accumulator architecture. It supports up to 56 instructions, up to 13 addressing modes, and 216 (65536) bytes of memory, as specified below.

Main Memory

Main memory consists of 65,536 words, each of which is one byte wide. Addresses are in lobyte-hibyte representation (Little-Endian), 16 bit address range, operands follow instruction codes. Within the byte, bits are numbered from right to left, with bit 0 the least significant bit and bit 7 the most significant bit. The memory interface transfers one byte of data at a time. Communication with memory is described in detail in later documentation.

DESIGN

The T34 source code has four fields separated by spaces:
LABEL, INSTRUCTION, OPERAND, and COMMENT.

The T34 editor produces fixed field sizes:
LABEL INSTRUCTION OPERAND COMMENT
[1-9] [10-14] [15-25] [26-79]

This means that the T34 editor will always produce a line of source code of one of the following formats:

- LABEL</t>INSTR</t>OPERAND</t>; COMMENT
- LABEL</t>INSTR</t>OPERAND
- LABEL</t>INSTR</t>; COMMENT
- LABEL</t>INSTR
- </t>INSTR</t>OPERAND</t>; COMMENT
- </t>INSTR</t>OPERAND
- </t>INSTR </t>; COMMENT
- </t>INSTR
- </t>; COMMENT
- COMMENT

Where </t> represent the number of spaces needed to fill out to the specified column.

All identifiers, numbers, opcodes, and pragmas are case insensitive and should be translated to upper case by the assembler.

A line containing only a comment must begin with either a "*" or ";". Comments starting with a ";" will be tabbed to the comment field, while comment lines beginning with a "*" will begin in column 1. The maximum allowable combined OPERAND+COMMENT length is 64 characters. The assembler will give an error message if this limit is exceeded. Also, a comment line by itself is limited to 64 characters. Same error message applies.

NUMBER FORMATS

- $[0-9A-Fa-f] .... hex
- %[01] .... binary
- O[0-7] .... octal
- [0-9] .... decimal

EXPRESSIONS

To make clear the syntax accepted and/or required by the assembler, we must define what is meant by an "expression". Expressions are built up

from "primitive expressions" by use of arithmetic and logical operations. The primitive expressions are:

- A label
- A decimal number
- A hexadecimal number (preceded by a "$").
- A binary number (preceded by "%").
- Any ASCII character either, preceded or enclosed by double or single quotes.
- The character "*" which stands for the present address.

All number formats accept 16-bit data and leading zeros are never required.
In case 5, the value of the primitive expression is the value of the ASCII character. The high bit will be on if the double quote (") is used, and off if the single quote (') is used. The assembler supports the four arithmetic operations: +, -, /, and *. It also supports the three logical operations: ! = Exclusive OR, . (period) = OR, and & = AND.

Some examples of legal operations are:

- LABEL1-LABEL2
- 2*LABEL+$231
- 1234+%10111
- K
- 0
- LABEL&$7F
- *-2
- LABEL.%10000000

Parentheses have another meaning and are not allowed in expressions. All arithmetic and logical operations are done from left to right (2+3*5 would assemble as 25 and not 17). Parentheses are normally used to change the order of evaluation in an expression. If the need arises to perform such an operation, partial "sums" can be collected in dummy labels and finally combined to obtain the desired effect. Using the above example where the answer was 25, and assuming the desired answer was 17:

- LABEL1 EQU 3*5
- LABEL2 EQU 2+LABEL1

LABELS and IDENTIFIERS
Identifiers must begin with a letter [A-Z] and contain letters, digits, and the underscore [A-Z0-9_]. No label can be longer than 8

characters. Labels and Identifiers must not be the same as valid opcodes or valid hex strings.
EXAMPLES:

- LABEL1 LDA #4 Define LABEL1 with the address of instruction LDA.
- JMP LABEL2 Jump to address of label LABEL2.
- STORE EQU $0800 Define STORE with value 0800.
- HERE EQU * Define HERE with current address (PC).
- HERE2 Define HERE2 with current address (PC).

COMMENTS
There are two ways to create comments: Make an entire line a comment, or use the comment field.
EXAMPLES:

- ; comment Any sequence of characters starting with a semicolon to the end of the line are ignored.
- * Any line starting with a * is ignored.

INSTRUCTIONS (OPCODE)
There are 56 instructions in the T34. Many instructions make use of more than one addressing mode and each instruction/addressing mode combination has a particular hexadecimal OPCODE that specifies it exactly. The instructions are always 3 letter mnemonics followed by an (optional) operand/address. Any pseudo instructions for the assembler has to be 3 letter mnemonics, and must not be the same as valid opcodes.
ADDRESSING MODES:
The T34 has 13 addressing modes:

- OPC .... implied
- OPC A .... Accumulator
- OPC #BB .... immediate
- OPC HHLL .... absolute
- OPC HHLL,X .... absolute, X-indexed
- OPC HHLL,Y .... absolute, Y-indexed
- OPC *LL .... zeropage
- OPC *LL,X .... zeropage, X-indexed
- OPC *LL,Y .... zeropage, Y-indexed
- OPC (BB,X) .... X-indexed, indirect
- OPC (LL),Y .... indirect, Y-indexed
- OPC (HHLL) .... indirect
- OPC BB .... relative

Where HHLL is a 16 bit word and LL or BB a 8 bit byte, and A is literal "A". There must not be any white space in any part of an instruction's address. As all addressing modes are not valid with all

opcodes, we will here give examples of each valid INSTR ADRESSING paring.

INSTRUCTIONS:

ADC: ADd with Carry

Addressing Modes	Common Syntax	Hex Coding
Immediate	ADC #$12	69 12
Zero Page	ADC $12	65 12
Zero Page,X	ADC $12,X	75 12
Absolute	ADC $1234	6D 34 12
Absolute,X	ADC $1234,X	7D 34 12
Absolute,Y	ADC $1234,Y	79 34 12
(Indirect,X)	ADC ($12,X)	61 12
(Indirect),Y	ADC ($12),Y	71 12

AND: Logical AND

Addressing Modes	Common Syntax	Hex Coding
Immediate	AND #$12	29 12
Zero Page	AND $12	25 12
Zero Page,X	AND $12,X	35 12
Absolute	AND $1234	2D 34 12
Absolute,X	AND $1234,X	3D 34 12
Absolute,Y	AND $1234,Y	39 34 12
(Indirect,X)	AND ($12,X)	21 12
(Indirect),Y	AND ($12),Y	31 12

ASL: Arithmetic Shift Left

Addressing Modes	Common Syntax	Hex Coding
Accumulator	ASL	0A
Zero Page	ASL $12	06 12
Zero Page,X	ASL $12,X	16 12
Absolute	ASL $1234	0E 34 12
Absolute,X	ASL $1234,X	1E 34 12

BCC: Branch Carry Clear

Addressing Modes	Common Syntax	Hex Coding
Relative	BCC $7F	90 7F

BCS: Branch Carry Set

Addressing Modes	Common Syntax	Hex Coding
Relative	BCS $F9	B0 F9

BEQ: Branch if EQual

Addressing Modes	Common Syntax	Hex Coding
Relative	BEQ $FF	F0 FF

BIT: compare Accumulator BITs with memory

Addressing Modes	Common Syntax	Hex Coding
Zero Page	BIT $12	24 12
Absolute	BIT $1234	2C 34 12

BMI: Branch on MInus

Addressing Modes	Common Syntax	Hex Coding
Relative	BMI $FF	30 FF

BNE: Branch Not Equal

Addressing Modes	Common Syntax	Hex Coding
Relative	BNE $FF	D0 FF

BPL: Branch on PLus

Addressing Modes	Common Syntax	Hex Coding
Relative	BPL $FF	10 FF

BRK: BReaK (software interrupt)

Addressing Modes	Common Syntax	Hex Coding
Implied	BRK	00

BVC: Branch on oVerflow Clear

Addressing Modes	Common Syntax	Hex Coding
Relative	BVC $FF	50 FF

BVS: Branch on oVerflow Set

Addressing Modes	Common Syntax	Hex Coding
Relative	BVS $FF	70 FF

CLC: CLear Carry

Addressing Modes	Common Syntax	Hex Coding
Implied	CLC	18

CLD: CLear Decimal mode

Addressing Modes	Common Syntax	Hex Coding
Implied	CLD	D8

CLI: CLear Interrupt mask

Addressing Modes	Common Syntax	Hex Coding
Implied	CLI	58

CLV: CLear oVerflow flag

Addressing Modes	Common Syntax	Hex Coding
Implied	CLV	B8

CMP: CoMPare to Accumulator

Addressing Modes	Common Syntax	Hex Coding
Immediate	CMP #$12	C9 12
Zero Page	CMP $12	C5 12
Zero Page,X	CMP $12,X	D5 12
Absolute	CMP $1234	CD 34 12
Absolute,X	CMP $1234,X	DD 34 12
Absolute,Y	CMP $1234,Y	D9 34 12
(Indirect,X)	CMP ($12,X)	C1 12
(Indirect),Y	CMP ($12),Y	D1 12

CPX: ComPare data to the X-Register

Addressing Modes	Common Syntax	Hex Coding
Immediate	CPX #$12	E0 12
Zero Page	CPX $12	E4 12
Absolute	CPX $1234	EC 34 12

CPY: ComPare data to the Y-Register

Addressing Modes	Common Syntax	Hex Coding
Immediate	CPY #$12	C0 12
Zero Page	CPY $12	C4 12
Absolute	CPY $1234	CC 34 12

DEC: DECrement a memory location

Addressing Modes	Common Syntax	Hex Coding
Zero Page	DEC $12	C6 12
Zero Page,X	DEC $12,X	D6 12
Absolute	DEC $1234	CE 34 12
Absolute,X	DEC $1234,X	DE 34 12

DEX: DEcrement the X-Register

Addressing Modes	Common Syntax	Hex Coding
Implied	DEX	CA

DEY: DEcrement the Y-Register

Addressing Modes	Common Syntax	Hex Coding
Implied	DEY	88

EOR: Exclusive OR with Accumulator

Addressing Modes	Common Syntax	Hex Coding
Immediate	EOR #$12	49 12
Zero Page	EOR $12	45 12
Zero Page,X	EOR $12,X	55 12
Absolute	EOR $1234	4D 34 12
Absolute,X	EOR $1234,X	5D 34 12
Absolute,Y	EOR $1234,Y	59 34 12
(Indirect,X)	EOR ($12,X)	41 12
(Indirect),Y	EOR ($12),Y	51 12

INC: INCrement memory

Addressing Modes	Common Syntax	Hex Coding
Zero Page	INC $12	E6 12
Zero Page,X	INC $12,X	F6 12
Absolute	INC $1234	EE 34 12
Absolute,X	INC $1234,X	FE 34 12

INX: INCrement X-Register

Addressing Modes	Common Syntax	Hex Coding
Implied	INX	E8

INY: INCrement Y-Register

Addressing Modes	Common Syntax	Hex Coding
Implied	INY	C8

JMP: JuMP to address

Addressing Modes	Common Syntax	Hex Coding
Absolute	JMP $1234	4C 34 12
Indirect	JMP ($1234)	6C 34 12

JSR: Jump to SubRoutine

Addressing Modes	Common Syntax	Hex Coding
Absolute	JSR $1234	20 34 12

LDA: LoaD Accumulator

Addressing Modes	Common Syntax	Hex Coding
Immediate	LDA #$12	A9 12
Zero Page	LDA $12	A5 12
Zero Page,X	LDA $12,X	B5 12
Absolute	LDA $1234	AD 34 12
Absolute,X	LDA $1234,X	BD 34 12
Absolute,Y	LDA $1234,Y	B9 34 12
(Indirect,X)	LDA ($12,X)	A1 12
(Indirect),Y	LDA ($12),Y	B1 12

LDX: LoaD the X-Register

Addressing Modes	Common Syntax	Hex Coding
Immediate	LDX #$12	A2 12
Zero Page	LDX $12	A6 12
Zero Page,Y	LDX $12,Y	B6 12
Absolute	LDX $1234	AE 34 12
Absolute,Y	LDX $1234,Y	BE 34 12

LDY: LoaD the Y-Register

Addressing Modes	Common Syntax	Hex Coding
Immediate	LDY #$12	A0 12
Zero Page	LDY $12	A4 12
Zero Page,X	LDY $12,X	B4 12
Absolute	LDY $1234	AC 34 12
Absolute,X	LDY $1234,X	BC 34 12

LSR: Logical Shift Right

Addressing Modes	Common Syntax	Hex Coding
Accumulator	LSR	4A
Zero Page	LSR $12	46 12
Zero Page,X	LSR $12,X	56 12
Absolute	LSR $1234	4E 34 12
Absolute,X	LSR $1234,X	5E 34 12

NOP: NO Operation

Addressing Modes	Common Syntax	Hex Coding
Implied	NOP	EA

ORA: inclusive OR with the Accumulator

Addressing Modes	Common Syntax	Hex Coding
Immediate	ORA #$12	09 12
Zero Page	ORA $12	05 12
Zero Page,X	ORA $12,X	15 12
Absolute	ORA $1234	0D 34 12
Absolute,X	ORA $1234,X	1D 34 12
Absolute,Y	ORA $1234,Y	19 34 12
(Indirect,X)	ORA ($12,X)	01 12
(Indirect),Y	ORA ($12),Y	11 12

PHA: PusH Accumulator

Addressing Modes	Common Syntax	Hex Coding
Implied	PHA	48

PHP: PusH Processor status

Addressing Modes	Common Syntax	Hex Coding
Implied	PHP	08

PLA: PulL Accumulator

Addressing Modes	Common Syntax	Hex Coding
Implied	PLA	68

PLP: PulL Processor status

Addressing Modes	Common Syntax	Hex Coding
Implied	PLP	28

ROL: ROtate Left

Addressing Modes	Common Syntax	Hex Coding
Accumulator	ROL	2A
Zero Page	ROL $12	26 12
Zero Page,X	ROL $12,X	36 12
Absolute	ROL $1234	2E 34 12
Absolute,X	ROL $1234,X	3E 34 12

ROR: ROtate Right

Addressing Modes	Common Syntax	Hex Coding
Accumulator	ROR	6A
Zero Page	ROR $12	66 12
Zero Page,X	ROR $12,X	76 12
Absolute	ROR $1234	6E 34 12
Absolute,X	ROR $1234,X	7E 34 12

RTI: ReTurn from Interrupt

Addressing Modes	Common Syntax	Hex Coding
Implied	RTI	40

RTS: ReTurn from Subroutine

Addressing Modes	Common Syntax	Hex Coding
Implied	RTS	60

SBC: SuBtract with Carry

Addressing Modes	Common Syntax	Hex Coding
Immediate	SBC #$12	E9 12
Zero Page	SBC $12	E5 12
Zero Page,X	SBC $12,X	F5 12
Absolute	SBC $1234	ED 34 12
Absolute,X	SBC $1234,X	FD 34 12
Absolute,Y	SBC $1234,Y	F9 34 12
(Indirect,X)	SBC ($12,X)	E1 12
(Indirect),Y	SBC ($12),Y	F1 12

SEC: SEt Carry

Addressing Modes	Common Syntax	Hex Coding
Implied	SEC	38

SED: SEt Decimal mode

Addressing Modes	Common Syntax	Hex Coding
Implied	SED	F8

SEI: SEt Interupt disable

Addressing Modes	Common Syntax	Hex Coding
Implied	SEI	78

STA: STore Accumulator

Addressing Modes	Common Syntax	Hex Coding
Zero Page	STA $12	85 12
Zero Page,X	STA $12,X	95 12
Absolute	STA $1234	8D 34 12
Absolute,X	STA $1234,X	9D 34 12
Absolute,Y	STA $1234,Y	99 34 12
(Indirect,X)	STA ($12,X)	81 12
(Indirect),Y	STA ($12),Y	91 12

STX: STore the X-Register

Addressing Modes	Common Syntax	Hex Coding
Zero Page	STX $12	86 12
Zero Page,Y	STX $12,Y	96 12
Absolute	STX $1234	8E 34 12

STY: STore the Y-Register

Addressing Modes	Common Syntax	Hex Coding
Zero Page	STY $12	84 12
Zero Page,X	STY $12,X	94 12
Absolute	STY $1234	8C 34 12

TAX: Transfer Accumulator to the X-Register

Addressing Modes	Common Syntax	Hex Coding
Implied	TAX	AA

TAY: Transfer Accumulator to the Y-Register

Addressing Modes	Common Syntax	Hex Coding
Implied	TAY	A8

TSX: Transfer Stack to the X-Register

Addressing Modes	Common Syntax	Hex Coding
Implied	TSX	BA

TXA: Transfer the X-Register to Accumulator

Addressing Modes	Common Syntax	Hex Coding
Implied	TXA	8A

TXS: Transfer the X-Register to Stack

Addressing Modes	Common Syntax	Hex Coding
Implied	TXS	9A

TYA: Transfer the Y-Register to Accumulator

Addressing Modes	Common Syntax	Hex Coding
Implied	TYA	98

PSEUDO OPCODES – DIRECTIVES

CHK:

Syntax

```
CHK
```

CHK places a checksum byte into the object code at the location of the CHK opcode (usually at the end of the program). The checksum byte is calculated by EXclusive OR all the previous bytes in the object code.

END:

Syntax

```
END
```

This opcode is not needed by T34. It is provided so T34 can assemble source code originally written for assemblers that do require an END statement. In any event, good programming dictates that it should be specified (Don't you feel better when you see both the ORG and END opcodes surrounding your precious source?).

EQU:

Syntax

```
LABEL EQU expression ; comment
```

The above example, is used to define the value of a LABEL, usually an exterior address or a constant for which a meaningful name is desired (good programming practices dictate that all constants be given a meaningful name and comment. The meaning of "magic" numbers tends to fade when the program source is read at a later time). In any case, it is recommended that the EQU’s all be located at the beginning of the program. The assembler will not permit an EQU to a zero page number after the label equated has been used, since bad code could result from such a situation.

ORG:
Syntax
ORG expression
Establishes the address at which the program is designs to run. It defaults to the present value of T34 HIMEM ($8000 by default). Usually, there will be one ORG and it will be at the start of the program. If more than one ORG is used, the first establishes the LOAD address. This can be used to create an object file that would load at one address even though it might be designed to run at another.

ERROR MESSAGES
BAD OPCODE - FATAL
Occurs when the instruction is not valid (perhaps misspelled) or the instruction is in the label column.

BAD ADDRESS MODE – NON-FATAL
The addressing mode is not a valid T34 instruction; for example, JSR (LABEL) or LDX (LABEL),Y.

BAD BRANCH – NON-FATAL
A branch (BEQ, BCC, &c) to an address that is out of range, i.e. further away than .... NOTE: Most errors will throw off the assembler's address calculations. Bad branch errors should be ignored until previous errors have been dealt with.

BAD OPERAND - FATAL
This occurs if the operand is illegally formed or if a label in the operand is not defined. This also occurs if you "EQU" a label to a zero page value after the label has been used. It may also mean that your operand is longer than 64 characters, or that a comment line exceeds 64 characters. This error will abort assembly.

DUPLICATE SYMBOL – NON-FATAL
On the first pass, the assembler finds two identical labels.

MEMORY FULL - FATAL
This is usually caused by one of four conditions: Incorrect ORG setting, source code too large, object code too large or symbol table too large.

Structure

The assembler is split into two pieces main and the CodeLine module. All main does is read in the source file and call the CodeLine module, so the majority of this section will be explaining the CodeLine module.

CodeLine

The CodeLine module is the bulk of the program. It does all of the input parsing and output formatting. The CodeLine module has 8 main files: Exceptions, Op_Param, Parser, Line, NOpcode, Memory, Opcode, and Checksum. Each of these files will be explained in turn.

Exceptions

All the classes in this file are custom, fatal exceptions that end assembly: BadOpcode, BadOperand, MemoryFull. Their descriptions can be found in the **Error Messages** section above. The odd-one-out is the CustomException class. All it does is act as an abstract base for the other exceptions. All of the methods in these classes are derived from the standard Exception class.

Op_Param

Op_Param has two functions, op_params and to_dec. These two functions are used throughout the module to convert mathematic expressions to hexadecimal.

The op_params function converts a string encoded math expression and converts it to a hexadecimal string. It supports the following operations:

- + addition
- - subtraction
- * multiplication
- / division
- & bit-wise AND
- . bit-wise OR
- ! bit-wise XOR

It will also raise a standard ValueError if there are spaces in the string.

The to_dec function converts a string encoded number without any operands into an integer. It supports these formats:

- HEX - $45
- DEC – 65
- OCT - o101 or O101
- BIN - %01000101
- ASC - 'A' or "A"

Will raise a standard ValueError if there are spaces or the string doesn't match one of these formats.

Parser

Parser is a container class for Lines. It generates and organizes Lines as well as the symbol table. It also writes assembled code to the console and output file.

The Parser constructor converts the source code into Lines. It also sorts said Lines into lists, so they can be operated on later without further sorting. Then, after the symbol table has been calculated, it goes back into the Lines and replaces any symbol with the corresponding value.

The add_memory method is a helper function that operates on Memory objects (Memories) and serves several purposes.

- Sorts Memories into their appropriate lists
- Extracts symbols from Memories and adds them to the symbol table
- Checks for duplicate symbols (**Design > Error Messages**)

The store_equ method is a helper function that parses a string containing the EQU instruction and stores its symbol in the symbol table. Also checks for duplicate symbols (**Design > Error Messages**).

The print method prints the assembled code and any non-fatal errors to the console.

The write_to_file method Writes the generated assembly to the given filename. The assembly consists of memory address and byte codes like so:

```
MEM1: B1 B2 B3
MEM2: B1 B2 B3
MEM3: B1 B2 B3
...
```

If the file already exists, its contents are overwritten.

Line
The Line class is an abstract, base class that acts as a base class for other, more complex Line classes.

The Line constructor stores a raw string and line number. It raises a BadOperand exception (**Structure > Exceptions**) if the line length is greater than 64.

The __len__ method returns the number of bytes in Line's byte code. Without an overwrite, it always returns 0.

The error method determines if there are any errors in the Line. Without an overwrite, it always returns an empty string.

The raw method returns the Line's raw string.

The num method returns the Line's line number.

The __str__ method returns the data's info in the following format:
" LN raw_string"
Where LN is the line number and raw_string is the Line's raw_string.

NOpcode
Nopcode is a type of Line that is not included in the object file. In other words, a NOpcode has neither a memory address nor a byte code. Nopcode inherits all its methods from Line.

Memory
Memory is a type of line that is included in the object file. In other words, it has a memory address and a byte code. Memory is an abstract, base class for Opcode and Checksum.

The Memory constructor extends the Line constructor and also stores an optional label and a memory address.

The __len__ method is an undefined, abstract method. It returns the number of bytes in the Memory's byte code.

The chk method is an undefined, abstract method. Its functionality differs based on the child that inherits it. See **Opcode** and **Checksum** below for more information.

The addr method returns the Memory's memory address.

The symbol method returns the Memory's optional label.

The assembly method is abstract, undefined method. It returns the Memory's generated assembly in a ready-to-write format.

Opcode

Opcode is the most common child of Line and Memory. It is used to implement almost every T34 instruction. This is done via the opcode table, which defines every instruction, their addressing modes, and their corresponding byte code. The exceptions are ORG, EQU, and END which are NOpcodes and CHK which has a dedicated class, Checksum.

The Opcode constructor extends the Memory constructor and also stores the Opcode's instruction and parameters. The constructor also tries to replace any symbol it finds with its corresponding value from the latest symbol table. Afterwards, it marks any remaining symbols so they can be replaced later.

The params_to_op method uses the op_params function, from **Structure > Op_Param** above, the convert the Opcode's parameters to hexadecimal.

The instr method returns the Opcode's instruction.

The params method returns the Opcode's parameters.

The __get_modes method is a helper function that returns all the addressing modes associated with the Opcode's instruction. If the instruction is not defined in the opcode table, this method raises a BadOpcode exception (**Structure > Exceptions**).

The byte_params method converts the Opcode's params to byte code. If there is a Bad branch error (**Design > Error Messages**), then the method returns dummy opcode since it is non-fatal. If the Opcode's instruction isn't defined in the opcode table or the Opcode's parameters cannot be converted to hex, then a BadOpcode or a BadOperand exception is thrown respectively (**Structure > Exceptions)**.

The __abs_to_rel method is a helper method that converts an absolute address to a relative address based on the Opcode's memory address.

The replace_symbols method attempts to replace symbols, using the final version of the symbol table, that weren't replaced during the Opcode's constructor.

The mode method determines the mode of the Opcode's instruction using the modes available in the opcode table, the Opcode's parameter format, and the Opcode's byte code.

The __len__ method returns the length of the Opcode's byte code. If there is an error in the opcode (**Design > Error Messages**), it defaults to 0.

The assembly method returns the Opcode's memory address and byte code in the following format:

"mADR: OP lo hi "

Where mADR is the memory address, OP is the opcode, lo is the lo-byte and hi is the hi-byte.

The __str__ method formats the Opcode's assembly and raw_string as follows:

"assembly LN raw_string"

Where assembly comes from the assembly method, LN is the line number, and raw string is the Opcode's raw string.
If there is a non-fatal error, the method includes the error description in the string. If there is a fatal error, the method and assembly stop (**Design > Error Messages**).

The chk method calculates the Opcode's checksum by finding the XOR of all the Opcode's bytes. If there is a non-fatal error, the method returns 0 (**Design > Error Messages**).

The opcode method determines the Opcode's (object) opcode (hex string) via its instruction and mode. If the given instruction-mode combo is not defined, the method returns an empty string.

The error method determines if there is a Bad branch, bad address mode, or Bad operand error in the Opcode, in that order (**Design > Error Messages**). If so, the method returns an abbreviation for Bad branch and bad address mode or else raises an exception for BadOperand (**Structure > Exceptions**).

Checksum

Checksum is a special Memory class that only implements the CHK instruction.

The Checksum constructor extends the Memory constructor and also stores a list of the previously generated Memory objects.

The chk method calculates and returns the checksum of the previous Memory objects by finding the XOR of all their bytes.

The __len__ method returns the length of the Checksum's byte code, which is always 1.

The assembly method formats and returns the Checksum’s memory address and byte code like so:

“mADR: CS ”

Where mADR is the memory address and CS is the checksum.

The __str__ method formats and returns the memory address, checksum, line number, and raw string of the Checksum like so:

“assembly LN raw_string”

Where assembly comes from the assembly method, LN is the line number and raw_string is the Checksum’s raw_string.

Testing

The assembler was tested by comparing the console output to the given sample files. I also created my own sample file to test some functionalities that weren't checked in the original 10.

```
Assembling
                1 ********************************
                2 *        SAMPLE PROGRAM C      *
                3 ********************************
                4 *
                5           ORG  $F000
                6 BIN       EQU  %10
                7 OCT       EQU  o10
                8 DEC       EQU  10
                9 HEX       EQU  $10
               10 ASCIA     EQU  'a
               11 ASCIB     EQU  "b"
               12 PLUS      EQU  $1003+2
               13 SUB       EQU  $1003-2
               14 MULT      EQU  $1003*2
               15 DIV       EQU  $1003/2
               16 AND       EQU  $1003&$2
               17 OR        EQU  $1003.o2
               18 XOR       EQU  $1003!%10
               19 BO        EQU  $FFFF
               20 COMBO     EQU  $7800*%10+05
               21 *
F000: 78       22 START     SEI
F001: D8       23           CLD
F002: A2 FF    24           LDX  #$FF
F004: 9A       25           TXS
F005: A9 00    26           LDA  #$00
               27 *
F007: 95 00    28 ZERO      STA  $00,X
F009: CA       29           DEX
F00A: D0 F9    30           BNE  COMBO
               31           END

--End assembly, 12 bytes, errors: 0

Symbol table - alphabetical order:
    AND      =$02     ASCIA    =$61     ASCIB    =$62     BIN      =$02
    BO       =$FFFF   COMBO    =$F005   DEC      =$0A     DIV      =$801
    HEX      =$10     MULT     =$2006   OCT      =$08     OR       =$1003
    PLUS     =$1005   START    =$F000   SUB      =$1001   XOR      =$1001
    ZERO     =$F007
Symbol table - numerical order:
    BIN      =$02     AND      =$02     OCT      =$08     DEC      =$0A
    HEX      =$10     ASCIA    =$61     ASCIB    =$62     DIV      =$801
    SUB      =$1001   XOR      =$1001   OR       =$1003   PLUS     =$1005
    MULT     =$2006   START    =$F000   COMBO    =$F005   ZERO     =$F007
    BO       =$FFFF
```

A copy of the custom sample file came with the program.