CENG 325Program #2 – Due 12/07/2022

1. Introduction

You have been hired by the CEO of Old Fashion Computing to help develop the company's first offering, to be named the T34. The T34 is an emulator of a simple accumulator-style architecture. Your supervisor has decided that your first task will be to work on the assembler for the emulator

---

It is possible to program the computer manually by entering numbers one at a time into successive memory locations. A program of this sort is called a machine language program because the processor can directly run the coded program steps. Humans, however, find this type of data difficult to read and are more likely to make mistakes while working with it.

A more convenient method of programming is to assign some kind of code word to each value. The computer will translate this word into the correct number to store in memory. This translation is done by an assembler, and programs entered or displayed in this manner are called assembly-language programs.

While we are waiting for the engineers to finish their work with the T34 hardware we will write the assembler for our machine. The assembler will take a source code file: FILE.s and convert it to an object file: FILE.o. The engineers (hardware and software) have already provided us with specs for the machine, so we should have enough information to complete this assignment.

---

2. CPU Architecture

The T34 is an 8-bit byte-addressable accumulator architecture. It supports up to 56 instructions, up to 13 addressing modes, and 2^{16} (65536) bytes of memory, as specified below.

2.1. 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.

2.2. The Instruction Set

Included later in this documentation

3. Assembler Requirements

The first part is to create an assembler that takes a source file, translates the source and stores the created machine code as an object file (a T34 program). The T34 has the advantage that you can store a program at any address in memory (not really true, address $`0000` - $`00FF` are called zero pages and have a special use), and each line of object code holds the memory address where to be stored and the opcodes.

Here is a sample object file:

```
300: 20 58 FC
303: A2 FF
305: 8E 00 07
308: CA
309: D0 FA
30B: 60
30C: D0
```

The first number on each line is the memory address (in hex), the second bytes of T34 opcodes in hex. Ex, line 1, tells us that the first address for that line is `300`, there are three bytes, and that `20` should be stored in `300`, `58` stored in `301` and `FC` in `302`. The next three lines are interpreted the same as line 1, so `A2` is stored in `303`, and `FF` stored in `304`.

3.1 Task

Create a program in either C++ or Python that reads in a source-file, assembles the source and stores all the bytes in an object-file.

3.2. Source and Object Files

The source code file name should be taken as a command-line argument by your program, and the object file saved with the same name but with the extension `.o`.

T34 ASSEMBLER

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 E</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-Faf] .... hex
%[01]       .... binary
O[0-7]      .... octal
[0-9]       .... decimal
<           .... LO-byte portion
>           .... HI-byte portion
```

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:

1. A label
2. A decimal number
3. A hexadecimal number (preceded by a "$").
4. A binary number (preceded by "%").
5. Any ASCII character either, preceded or enclosed by double or single quotes.
6. 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.

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).
           LDA  #<VAL1  Load LO-byte of VAL1.
```

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.

Instructions by Name:

```
    ADC     ....  add with carry
    AND     ....  and (with accumulator)
    ASL     ....  arithmetic shift left
    BCC     ....  branch on carry clear
    BCS     ....  branch on carry set
    BEQ     ....  branch on equal (zero set)
    BIT     ....  bit test
    BMI     ....  branch on minus (negative set)
    BNE     ....  branch on not equal (zero clear)
    BPL     ....  branch on plus (negative clear)
    BRK     ....  interrupt
    BVC     ....  branch on overflow clear
    BVS     ....  branch on overflow set
    CLC     ....  clear carry
```

```
CLD     ....  clear decimal
CLI     ....  clear interrupt disable
CLV     ....  clear overflow
CMP     ....  compare (with accumulator)
CPX     ....  compare with X
CPY     ....  compare with Y
DEC     ....  decrement
DEX     ....  decrement X
DEY     ....  decrement Y
EOR     ....  exclusive or (with accumulator)
INC     ....  increment
INX     ....  increment X
INY     ....  increment Y
JMP     ....  jump
JSR     ....  jump subroutine
LDA     ....  load accumulator
LDY     ....  load X
LDY     ....  load Y
LSR     ....  logical shift right
NOP     ....  no operation
ORA     ....  or with accumulator
PHA     ....  push accumulator
PHP     ....  push processor status (SR)
PLA     ....  pull accumulator
PLP     ....  pull processor status (SR)
ROL     ....  rotate left
ROR     ....  rotate right
RTI     ....  return from interrupt
RTS     ....  return from subroutine
SBC     ....  subtract with carry
SEC     ....  set carry
SED     ....  set decimal
SEI     ....  set interrupt disable
STA     ....  store accumulator
STX     ....  store X
STY     ....  store Y
TAX     ....  transfer accumulator to X
TAY     ....  transfer accumulator to Y
TSX     ....  transfer stack pointer to X
TXA     ....  transfer X to accumulator
TXS     ....  transfer X to stack pointer
TYA     ....  transfer Y to accumulator
```

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.

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

Many assemblers have an equivalent pseudo-op called BLT (Branch Less Than), since BCC is often used immediately following a comparison to see whether the Accumulator is less than the specified value.

BCS: Branch Carry Set

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

Some assemblers support the pseudo-op BGT (Branch Greater Than), since this command is used to test whether the Accumulator is equal to or greater than the specified value.

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

Occurs when the instruction is not valid (perhaps misspelled) or the instruction is in the label column.

BAD ADDRESS MODE

The addressing mode is not a valid T34 instruction; for example, JSR (LABEL) or LDX (LABEL),Y.

BAD BRANCH

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

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

On the first pass, the assembler finds two identical labels.

MEMORY FULL

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.

The T34 Instruction Set

LO-NIBBLE

HI	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK impl	ORA X,ind	??? ---	??? ---	??? ---	ORA zpg	ASL zpg	??? ---	PHP impl	ORA #	ASL A	??? ---	??? ---	ORA abs	ASL abs	??? ---
1x	BPL rel	ORA ind,Y	??? ---	??? ---	??? ---	ORA zpg,X	ASL zpg,X	??? ---	CLC impl	ORA abs,Y	??? ---	??? ---	??? ---	ORA abs,X	ASL abs,X	??? ---
2x	JSR abs	AND X,ind	??? ---	??? ---	BIT zpg	AND zpg	ROL zpg	??? ---	PLP impl	AND #	ROL A	??? ---	BIT abs	AND abs	ROL abs	??? ---
3x	BMI rel	AND ind,Y	??? ---	??? ---	??? ---	AND zpg,X	ROL zpg,X	??? ---	SEC impl	AND abs,Y	??? ---	??? ---	??? ---	AND abs,X	ROL abs,X	??? ---
4x	RTI impl	EOR X,ind	??? ---	??? ---	??? ---	EOR zpg	LSR zpg	??? ---	PHA impl	EOR #	LSR A	??? ---	JMP abs	EOR abs	LSR abs	??? ---
5x	BVC rel	EOR ind,Y	??? ---	??? ---	??? ---	EOR zpg,X	LSR zpg,X	??? ---	CLI impl	EOR abs,Y	??? ---	??? ---	??? ---	EOR abs,X	LSR abs,X	??? ---
6x	RTS impl	ADC X,ind	??? ---	??? ---	??? ---	ADC zpg	ROR zpg	??? ---	PLA impl	ADC #	ROR A	??? ---	JMP ind	ADC abs	ROR abs	??? ---
7x	BVS rel	ADC ind,Y	??? ---	??? ---	??? ---	ADC zpg,X	ROR zpg,X	??? ---	SEI impl	ADC abs,Y	??? ---	??? ---	??? ---	ADC abs,X	ROR abs,X	??? ---
8x	??? ---	STA X,ind	??? ---	??? ---	STY zpg	STA zpg	STX zpg	??? ---	DEY impl	??? ---	TXA impl	??? ---	STY abs	STA abs	STX abs	??? ---
9x	BCC rel	STA ind,Y	??? ---	??? ---	STY zpg,X	STA zpg,X	STX zpg,Y	??? ---	TYA impl	STA abs,Y	TXS impl	??? ---	??? ---	STA abs,X	??? ---	??? ---
Ax	LDY #	LDA X,ind	LDX #	??? ---	LDY zpg	LDA zpg	LDX zpg	??? ---	TAY impl	LDA #	TAX impl	??? ---	LDY abs	LDA abs	LDX abs	??? ---
Bx	BCS rel	LDA ind,Y	??? ---	??? ---	LDY zpg,X	LDA zpg,X	LDX zpg,Y	??? ---	CLV impl	LDA abs,Y	TSX impl	??? ---	LDY abs,X	LDA abs,X	LDX abs,Y	??? ---
Cx	CPY #	CMP X,ind	??? ---	??? ---	CPY zpg	CMP zpg	DEC zpg	??? ---	INY impl	CMP #	DEX impl	??? ---	CPY abs	CMP abs	DEC abs	??? ---
Dx	BNE rel	CMP ind,Y	??? ---	??? ---	??? ---	CMP zpg,X	DEC zpg,X	??? ---	CLD impl	CMP abs,Y	??? ---	??? ---	??? ---	CMP abs,X	DEC abs,X	??? ---
Ex	CPX #	SBC X,ind	??? ---	??? ---	CPX zpg	SBC zpg	INC zpg	??? ---	INX impl	SBC #	NOP impl	??? ---	CPX abs	SBC abs	INC abs	??? ---
Fx	BEQ rel	SBC ind,Y	??? ---	??? ---	??? ---	SBC zpg,X	INC zpg,X	??? ---	SED impl	SBC abs,Y	??? ---	??? ---	??? ---	SBC abs,X	INC abs,X	??? ---

Key

A .... Accumulator
abs .... absolute
abs,X .... absolute, X-indexed
abs,Y .... absolute, Y-indexed
.... immediate
impl .... implied
ind .... indirect
X,ind .... X-indexed, indirect
ind,Y .... indirect, Y-indexed
rel .... relative
zpg .... zeropage
zpg,X .... zeropage, X-indexed
zpg,Y .... zeropage, Y-indexed

Recommended order of work:

1. Create the code to read in the source file.
2. Create the code to ignore entire line comments, and to only extract the label, instruction and operands from a line of source code. We will always ignore the comment field.
3. By now you have probably figured out that an assembler goes through the source code one line at the time, and generates machine code. For most instruction this is no problem, as most instructions will reference a register. We have no problem with this as the assembler knows the location of the register. Defining a label is no problem either. We should have the current address when we encounter the label. If we use the EQU pseudo instruction to define constants and locations outside the current program (i.e. addresses that is not part of the assembled code), we should be safe. The trouble is with the branching. Consider the following two pieces of code:

```
JMP   LATER            LOOP1      ...
...                               ...
...                               ...
LATER                             JMP   LOOP1
```

Branching or jumping back in the code as in the right hand example (i.e. some kind of loop instruction) is not a problem either, as LOOP1 is already defined. Our problem is in the left hand piece of code. When we try to skip a section (i.e. some kind of if-statement in a higher language code). We at that moment of assembling the code simply do not know what the address of the future location is.

This is known as a forward reference. If the assembler is processing the file one line at a time, then it does not know where LATER is when it first encounters the jump instruction. So, it doesn't know if the jump is a short jump, a near jump or a far jump. There is no difference amongst JMP and branch instructions by themselves (a JMP or JSR is always three bytes long, and all Branch instructions are two bytes. Hmmm is this a coincident?). The trouble is caused by the instructions in between, they can be 1, 2 or three bytes, and we don’t even know how many there are!

Anyway, at this point in the code the assembler would have to guess how far away the instruction is in order to generate the correct instruction. If the assembler guesses wrong, then the addresses for all other labels later in the program would be wrong, and the code would have to be regenerated.

So, what can we do to allow the assembler to generate the correct instruction? Simple: scan the code twice. The first time, just count how long each machine code instructions will be, just to figure out the addresses of all the labels. As we do this, we will create a table that has a list of all the labels and where they will be in the program. This table is known as the symbol table. On the second pass, we generate the actual machine code, and use the symbol table to determine where the address for the jump is. Now you know what the table at the bottom of the output is! Well that was a rather long explanation to tell what has to happen next. Implement the code to create the symbol table (I recommend you use the python dictionary maps to solve this), you might want to start with coding the implementation of the ORG and EQU pseudo instructions. Remember at this moment we are 'only' trying to get the address correct, we will not focus on mapping an instruction to a machine code instruction, just get the pc count correct and map each label to an address.

4. After this we are left with four major tasks: We need to implement the remaining pseudo instructions, we need to solve the problem with creating the machine code instruction (map an instruction to an OPCODE and the operand to and ADDRESS), we need to implement all the error messages, and we need to get the output (both to screen and to file) correct.
5. There are 'only' four pseudo codes, two of them were tightly coupled with the handling of the labels (ORG and EQU), they should been solved by now. The remaining two are tightly coupled with the output (END and CHK). So let's skip them for now and begin our work with the op-code problem.
6. Start working on the Implied addressing only instructions, i.e. the 25 T-34 instructions that have an empty operand field.
   Each time you have a valid instruction you should store it together with the current Program Counter (PC), and then increment the program counter with one byte.
   A common way to handle implied instructions is to ignore whatever is in the OPERAND field. Most assembler does this with implied addressing. This has the advantage that we do not need to create an error code for incorrect addressing mode.
7. Next work on the branching instructions (T-34 has eight of them), as all of them have relative addressing mode.
   Here we will need to check that the addressing mode is correct, but for now we can comment this in our code, and come back when we fix all the rest of the addressing errors.

Because the branching is relative to the current location we might have to calculate the offset if the operand is a label.

```
   LOOP     STX  $700
            DEX
            BNE  LOOP
```

In this case we are jumping backwards so the offset (or relative address) has to be negative. That means we have to calculate the address difference from the current location and the address of the label and express it as a signed byte using two’s complement.

So what a branching instruction will create is: a one byte OPCODE a one byte relative address, and increment the PC by two.

8. We are now ready to add the two jump instructions. We can either add them to branching instruction, or we can create a category by itself. These instructions will create: a one byte OPCODE a two byte absolute or indirect address, and increment the PC by three.
9. By now we have only 21 instructions, in 5 categories left to cover. We have the Arithmetic and Logic instructions, Shift and Rotate, Store and Load instructions, Compare instructions and some miscellaneous instructions (that does not really fit under any of the remaining categories).
10. Next try to classify your current instruction based on how the operand field is looking. The goal in the end is to relate the current instruction with the current addressing mode. We will use this both to check if we have a valid instruction, but also to figure out which of the different OPCODES the current instruction will have.
11. The easiest way is probably to start with the Compare instructions. If we look at the instruction set table, we notice some patterns. Take advantage of these patterns as you are going forward.
12. Now work on the remaining instructions, using the instruction set table to create groups that behave in a similar manner.
13. Next finish all the pseudo instructions. There is really nothing to do with the END instruction, it should not produce any opcode, nor add anything to the object-file. The CHK instruction should calculate the Exclusive OR of all the bytes produced in the object-file, and add this to the output.
14. Time to get all the output correct, both the output to the screen and to the object-file.

15. Ok, now when all this is working, it is time to work on the error messages:
    - BAD OPCODE: Caught during the first pass. Present the error message "Bad opcode in line: <line>", and terminate the assembler (we don't know how to interpret this line).
    - BAD ADDRESS MODE: Caught during the second pass (the actual assembly). Temporarily halt the assembler, present the error message "Bad address mode in line: <line>", increment the 'error' count, and don't create an assembly line output. Have the user press a key to resume the execution of the assembler.
    - DUPLICATE SYMBOL: Caught during the first pass. Temporarily halt the assembler, present the error message "Duplicate symbol in line: <line>", increment the 'error' count, and ignore the attempt to assign a new address. Have the user press a key to resume the execution of the assembler.
    - BAD BRANCH: Caught during the second pass (the actual assembly). Temporarily halt the assembler, present the error message "Bad branch in line: <line>", increment the 'error' count, and don't create an assembly line output. Have the user press a key to resume the execution of the assembler.
    - MEMORY FULL: Caught during the first pass. Present the error message "Memory Full", and terminate the execution of the assembler. As we are not writing the editor, we only need to bother about two types for this error: As soon as the object code is larger than the memory (i.e. we try to get beyond memory location $FFFF), this could been caused by either a bad choice of ORG, or that our code is simply too large! The other type is that the symbol table is too large. If the code were to be assembled on the T-34, the symbol table (together with the assembler) as to be stored in the T-34's main memory, so there is a limit on how large the symbol table (labels and addresses) can be, normally it is limited to 4096 bytes. Let us limit our symbol table to no more than 255 labels, so we throw an error when we try to create the 256^{th} label.
    - BAD OPERAND: By now you should be able to figure out when to catch the error and what to do with it. Just read the error message description.

4. Turning In Your Solution

4.1. General Information

The program must be submitted by the end of the day on the specified due date (i.e., no later than 11:59:59pm that day). The files (your source code, documentation etc) need to be tarred and gzipped prior to submission. Please empty the directory prior to tar/zip. [You will tar up the directory containing the files: `tar -czf prog1.tgz  prog1` .]

Provide documentation in PDF that tells the user (me) how to build and run the program. Further, it has to describe all functions you have written, how you tested all the functions and the program.

4.2. Submitting Your Solution

Use the D2L dropbox to submit your archive-file.

5. Sample code and output

```
********************************
*        SAMPLE PROGRAM 1      *
********************************
*
         ORG  $F000
*
START    SEI
         CLD
         LDX  #$FF
         TXS
         LDA  #$00
*
ZERO     STA  $00,X
         DEX
         BNE  ZERO
         END

Assembling
                1 ********************************
                2 *        SAMPLE PROGRAM 1      *
                3 ********************************
                4 *
                5          ORG  $F000
                6 *
F000: 78        7 START    SEI
F001: D8        8          CLD
F002: A2 FF     9          LDX  #$FF
F004: 9A       10          TXS
F005: A9 00    11          LDA  #$00
               12 *
F007: 95 00    13 ZERO     STA  $00,X
F009: CA       14          DEX
F00A: D0 FB    15          BNE  ZERO
               16          END

--End assembly, 12 bytes, Errors: 0

Symbol table - alphabetical order:
    START    =$F000    ZERO     =$F007

Symbol table - numerical order:
    START    =$F000    ZERO     =$F007
```

```
Assembling
                 ******************************
                 *        SAMPLE PROGRAM 2      *
                 ******************************
                 *
                 N1       EQU  $06
                 RSLT     EQU  $0A
                 *
8000: A5 06      START    LDA  N1
8002: 18                  CLC
8003: 69 80               ADC  #$80
8005: 85 0A               STA  RSLT
8007: 60         END      RTS
8010: BD                  CHK

--End assembly, 9 bytes, Errors: 0

Symbol table - alphabetical order:
    END      =$8007    N1       =$06    RSLT     =$0A    START    =$8000


Symbol table - numerical order:
    N1       =$06    RSLT     =$0A    START    =$8000    END      =$8007

Assembling
                 ******************************
                 *        SAMPLE PROGRAM 3      *
                 ******************************
                 *
                          ORG  $300
                 BELL     EQU  $FF3A
                 *
300: 18          ENTRY    CLC
301: 90 01                BCC  EXPT
                 *
303: EA          FILL     NOP
                 *
304: 20 3A FF    EXPT     JSR  BELL
                 *
307: 60          DONE     RTS
308: E6                   CHK

--End assembly, 9 bytes, Errors: 0

Symbol table - alphabetical order:
    BELL     =$FF3A    DONE     =$307    ENTRY    =$300    EXPT     =$304
    FILL     =$303

Symbol table - numerical order:
    ENTRY    =$300    FILL     =$303    EXPT     =$304    DONE     =$307
    BELL     =$FF3A
```

```
Duplicate symbol in line: 15
Assembling
                1 ********************************
                2 *        SAMPLE PROGRAM 4      *
                3 ********************************
                4 *
                5          ORG  $300
                6 BELL     EQU  $FF3A
                7 *
300: B8         8 ENTRY    CLV
Bad branch in line: 9
50 FC           9          BVC  BELL
               10 *
303: EA        11 FILL1    NOP
               12 *
304: 50 01     13 STEP     BVC  EXPT
               14 *
306: EA        15 FILL1    NOP
               16 *
Bad address mode in line: 17
               17 EXPT     JSR  (BELL)
               18 *
307: 60        19 DONE     RTS
308: 75        20          CHK

--End assembly, 8 bytes, Errors: 3

Symbol table - alphabetical order:
    BELL     =$FF3A    DONE     =$30A    ENTRY    =$300    EXPT     =$307
    FILL1    =$303    STEP     =$304

Symbol table - numerical order:
    ENTRY    =$300    FILL1    =$303    STEP     =$304    EXPT     =$307
    DONE     =$30A    BELL     =$FF3A
```

```
Assembling
                1 ******************************
                2 *        SAMPLE PROGRAM 6     *
                3 ******************************
                4 *
                6          ORG  $300
                7 CTR      EQU  $06
                8 HOME     EQU  $FC58
                9 COUT     EQU  $FDED
               10 *
300: 20 58 FC  11 START    JSR  HOME
303: A9 FF     12          LDA  #$FF
305: 85 06     13          STA  CTR
307: A5 06     14 LOOP     LDA  CTR
309: 20 ED FD  15          JSR  COUT
30C: C6 06     16          DEC  CTR
30E: F0 03     17          BEQ  END
310: 4C 07 03  18          JMP  LOOP
313: 60        19 END      RTS

--End assembly, 20 bytes, Errors: 0

Symbol table - alphabetical order:
    COUT     =$FDED    CTR      =$06     END      =$313    HOME     =$FC58
    LOOP     =$307     START    =$300

Symbol table - numerical order:
    CTR      =$06      START    =$300    LOOP     =$307    END      =$313
    HOME     =$FC58    COUT     =$FDED

Assembling
                1 ******************************
                2 *        SAMPLE PROGRAM 7     *
                3 ******************************
                4 *
                5          ORG  $300
                6 *
                7 COUT     EQU  $FDED
                8 *
300: A2 00      9 START    LDX  #$00
302: BD 13 03  10 LOOP     LDA  $0313,X
305: 20 ED FD  11          JSR  COUT
308: E8        12          INX
309: E0 05     13          CPX  #$05
30B: 90 F5     14          BCC  LOOP
30D: A9 8D     15          LDA  #$8D
30F: 20 ED FD  16          JSR  COUT
312: 60        17 EXIT     RTS

--End assembly, 19 bytes, Errors: 0

Symbol table - alphabetical order:
    COUT     =$FDED    EXIT     =$312    LOOP     =$302    START    =$300


Symbol table - numerical order:
    START    =$300    LOOP     =$302    EXIT     =$312    COUT     =$FDED
```

```
Assembling
                1 ******************************
                2 *        SAMPLE PROGRAM 8     *
                3 ******************************
                4 *
                5          ORG  $300
                6 *
                7 PTR      EQU  $06
                8 *
300: A9 04      9 ENTRY    LDA  #$04
302: 85 07     10          STA  PTR+1
304: A0 00     11          LDY  #$00
306: 84 06     12          STY  PTR
               13 * SETS PTR (6,7) TO $400
308: A9 A0     14 START    LDA  #$A0
30A: 91 06     15 LOOP     STA  (PTR),Y
30C: C8        16          INY
30D: D0 FB     17          BNE  LOOP
30F: E6 07     18 NXT      INC  PTR+1
311: A5 07     19          LDA  PTR+1
313: C9 08     20          CMP  #$08
315: 90 F1     21          BCC  START
317: 60        22 EXIT     RTS

--End assembly, 24 bytes, Errors: 0

Symbol table - alphabetical order:
    ENTRY    =$300   EXIT     =$317    LOOP     =$30A    NXT      =$30F
    PTR      =$06    START    =$308

Symbol table - numerical order:
    PTR      =$06    ENTRY    =$300    START    =$308    LOOP     =$30A
    NXT      =$30F   EXIT     =$317
```

```
Assembling
                1 ********************************
                2 *        SAMPLE PROGRAM 9       *
                3 ********************************
                4 *
                5          ORG  $300
                6 *
                7 N1       EQU  $06
                8 N2       EQU  $08
                9 RSLT     EQU  $0A
               10 *
300: 18        11 START    CLC
301: A5 06     12          LDA  N1
303: 65 08     13          ADC  N2
305: 85 0A     14          STA  RSLT
307: A5 07     15          LDA  N1+1
309: 65 09     16          ADC  N2+1
30B: 85 0B     17          STA  RSLT+1
30D: 60        18 END      RTS

--End assembly, 14 bytes, Errors: 0

Symbol table - alphabetical order:
    END      =$30D    N1       =$06    N2       =$08    RSLT     =$0A
    START    =$300

Symbol table - numerical order:
    N1       =$06    N2       =$08    RSLT     =$0A    START    =$300
    END      =$30D
```

```
Assembling
                 ******************************
                 *        SAMPLE PROGRAM 10    *
                 ******************************
                 *
                          ORG  $300
                 *
                 NUM      EQU  $06
                 MEM      EQU  $07
                 RSLT     EQU  $08
                 STAT     EQU  $09
                 *
                 YSAV1    EQU  $35
                 COUT1    EQU  $FDF0
                 CVID     EQU  $FDF9
                 COUT     EQU  $FDED
                 PRBYTE   EQU  $FDDA
                 *
                 *
300: A9 00       OPERATOR LDA  #$00
302: 48                   PHA
303: 28                   PLP
304: A5 06                LDA  NUM
306: 25 07                AND  MEM
308: 85 08                STA  RSLT
30A: 08                   PHP
30B: 68                   PLA
30C: 85 09                STA  STAT
30E: 60                   RTS
                 *
30F: A9 A4       PRHEX    LDA  #$A4
311: 20 ED FD             JSR  COUT
314: A5 06                LDA  NUM
316: 4C DA FD             JMP  PRBYTE
                 *
319: A5 06       PRBIT    LDA  NUM
31B: A2 08                LDX  #$08
31D: 0A          TEST     ASL
31E: 90 0D                BCC  PZ
320: 48          P0       PHA
321: A9 B1                LDA  #$B1
323: 20 ED FD             JSR  COUT
326: A9 A0                LDA  #$A0
328: 20 ED FD             JSR  COUT
32B: B0 0B                BCS  NXT
                 *
32D: 48          PZ       PHA
32E: A9 B0                LDA  #$B0
330: 20 ED FD             JSR  COUT
333: A9 A0                LDA  #$A0
335: 20 ED FD             JSR  COUT
                 *
338: 68          NXT      PLA
339: CA                   DEX
33A: D0 E1                BNE  TEST
                 *
33C: 60          EXIT     RTS
```

```
                *
33D: EA                   NOP
33E: EA                   NOP
33F: EA                   NOP
                *
340: C9 80         CSHOW  CMP  #$80
342: 90 10                BCC  CONT
344: C9 8D                CMP  #$8D
346: F0 0C                BEQ  CONT
348: C9 A0                CMP  #$A0
34A: B0 08                BCS  CONT
                *
34C: 48                   PHA
34D: 84 35                STY  YSAV1
34F: 29 7F                AND  #$7F
351: 4C F9 FD             JMP  CVID
                *
354: 4C F0 FD      CONT   JMP  COUT1
                *
357: 00            EOF    BRK
                *
358: 87                   CHK

--End assembly, 89 bytes, Errors: 0

Symbol table - alphabetical order:
    CONT     =$354    COUT     =$FDED   COUT1    =$FDF0    CSHOW    =$340
    CVID     =$FDF9   EOF      =$357    EXIT     =$33C
    MEM      =$07     NUM      =$06     NXT      =$338
    OPERATOR =$300    P0       =$320    PRBIT    =$319
    PRBYTE   =$FDDA   PRHEX    =$30F    PZ       =$32D
    RSLT     =$08     STAT     =$09     TEST     =$31D
    YSAV1    =$35

Symbol table - numerical order:
    NUM      =$06     MEM      =$07     RSLT     =$08     STAT     =$09
    OPERATOR =$300    PRHEX    =$30F    PRBIT    =$319
    TEST     =$31D    P0       =$320    PZ       =$32D
    NXT      =$338    EXIT     =$33C    CSHOW    =$340
    YSAV1    =$35     CONT     =$354    EOF      =$357
    PRBYTE   =$FDDA   COUT     =$FDED   COUT1    =$FDF0
    CVID     =$FDF9
```