Adrian Azan
12/5/18
Revised

Purpose:

This program emulates the T34, it's stack, stack pointer, X register, Y register, accumulator, program counter, and status register. The emulator will read in an object file that contains a list of instructions in hexadecimals. These instructions are all a part of the T34 instruction set. This emulator can only interpret Instructions with implied or Accumulator addressing modes. These instructions include (ASL, BRK, CLC, CLD ,CLI ,CLV, DEX, DEY, INX, INY, LSR, NOP, PHA, PHP, PLA, PLP, ROL, ROR, SEC, SED, SEI, TAX, TAY, TSX , TXA, TXS, and TYA. This emulator can also interpret 8 branch instructions (BCC, BCS, BEQ, BMI, BNE, BPL, BVS, and BVC) and the two jump instructions(JMP and JSR).

A user can then view, edit, and run the instructions loaded in from the file. The memory supports up to $2^{16}(65536)$ bytes of memory. This memory is filled with string "0"'s when started up. Can also read from object files to the memory. Bytes are read in starting at memory address 0. The only library required for this program is copy. Copy was used to make pushing the status register onto the stack easier in both instructions PHP and BRK

Usage:

When prompted, enter a file name to load instructions from. The file extension is '.o' (for object). File extension can be included or excluded. If you have no file to read from, enter nothing and return. If no file is entered, memory will be filled with 0's. The instructions will be read in space delaminated order, starting at memory address 0. Instructions entered into the program or read from an object file must be in hexadecimal format and space delaminated per 2 bytes.

To see any individual byte of memory, type the position into the command line. 1F is the memory address and EA is the value. If the entered address is greater than 0 and less than 65536, then the address is printed out along with the value located at that memory (in hex)
Example:

```
 > 1F
1F          EA
```

To view a range of bytes in memory, use the period (.) operator.
<start_memory_address>.<end_memory_address> will display every byte from the starting memory address (inclusive) up until the end memory address (not inclusive). Starting address must be lower than end address.
Example:

```
>A.18
      A     C8    98    48    E8    8A    68    00
      12    88    E8    98    0A    48
```

First user input is searched for a "." And if the symbol is found then the input is split based on the period. The first part is stored into an int variable for the beginning address and the second part is stored in another in variable holding the end address. A loop from the beginning address to the end

Adrian Azan
12/5/18
Revised
address begins displaying the contents in memory 8 two-byte chunks with the first number being the memory address that row began on.

To change any bytes in memory, use the colon (:) operator. <start_memory_address>: <changes> will change the bytes located at the starting memory address and continue until no more changes are left. Changes must be in hexadecimal.
Example:

```
>14: C8 D8 F8

>14.17
        14      C8      D8      F8
```

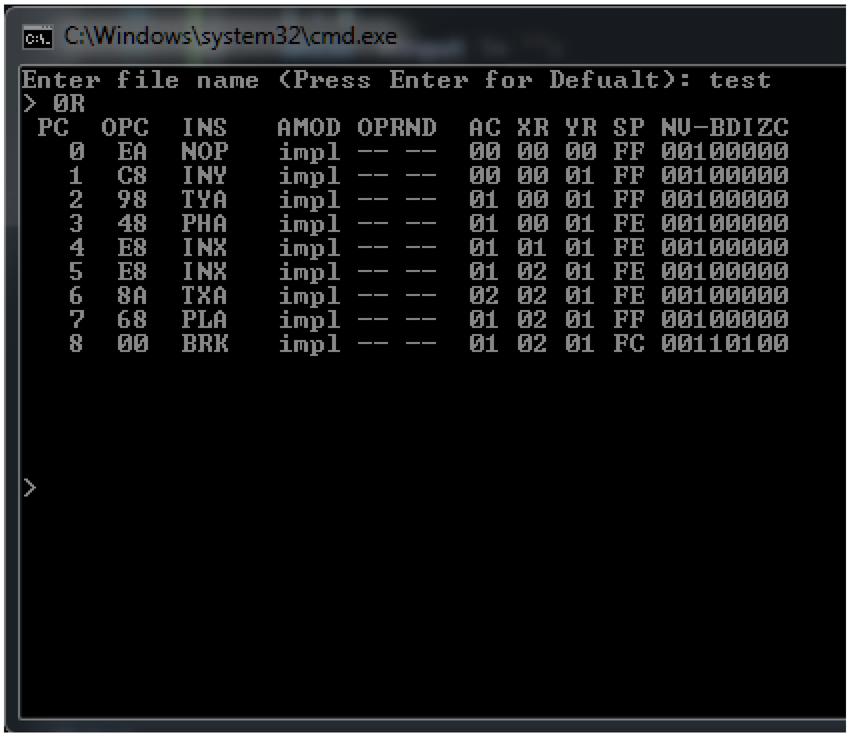
A ":" symbol is searched for in userInput, and if found is split into several parts. The first being where the edits begin, and the second part is split again by spaces. All of these are the edits being made and are stored in an array. A for loop goes from starting address found in the first split to however many edits were found in the second split. Memory at these addresses are over written by edits.

To run instructions from memory, use the 'R' or 'r' operator. <start_memory_address>R will emulate the instructions beginning at the start memory address and then continuing until a BRK is hit. The program counter, OPC, instruction, addressing mode, accumulator, X register, Y register, stack pointer, and status register are printed to the console as shown in Figure 1. All these values are also reset to their default values so that multiple program runs do not affect each other.

PC, OPC, AC, XR,YR, and SP are all written in hexadecimal. Currently, the emulator can only execute operations with implied or accumulator addressing modes and can only read instructions in hexadecimal. Each instruction is a part of a if/elif chain in which each instruction has its own if statement to edit the registers and other values. Once the right instruction has been found, it sets the necessary values and calls the functions described below to accomplish its task. After running an instruction, each register is checked to make sure that their values are still in bounds (>0 and <256). The instruction and registers are printed and the PC is increased.

The status register is an 8 integer array. Each status is updated after every instruction execution.
Status Register[0]: Negative status
Status Register[1]: Overflow status
Status Register[2]: Ignored
Status Register[3]: Break status
Status Register[4]: Decimal status
Status Register[5]: Interrupt status
Status Register[6]: Zero status
Status Register[7]: Carry status

Adrian Azan
12/5/18
Revised


```
C:\Windows\system32\cmd.exe
Enter file name (Press Enter for Defualt): test
> 0R
 PC  OPC  INS   AMOD OPRND  AC XR YR SP NV-BDIZC
   0  EA  NOP   impl -- --  00 00 00 FF 00100000
   1  C8  INY   impl -- --  00 00 01 FF 00100000
   2  98  TYA   impl -- --  01 00 01 FF 00100000
   3  48  PHA   impl -- --  01 00 01 FE 00100000
   4  E8  INX   impl -- --  01 01 01 FE 00100000
   5  E8  INX   impl -- --  01 02 01 FE 00100000
   6  8A  TXA   impl -- --  02 02 01 FE 00100000
   7  68  PLA   impl -- --  01 02 01 FF 00100000
   8  00  BRK   impl -- --  01 02 01 FC 00110100



>
```

Figure 1: Example Output

Branching and Jumping:

Branching and Jumping is identified similarly to other instructions. First it checks the OPC, sets the instruction name, and the Addressing mode. After that, the current PC is saved so that the branch instruction can be outputted with the PC is located at. The PC is incremented. Depending on the branch, different elements of the status register are checked. For example, BCC and BCS both check the carry register (SR[7]) to see if they are set. For branches, the operand the next byte in the memory and is loaded into the operand. The PC is then set to the operand in hexadecimal. This will effectively branch the code to run at the operand the branch was given.

The Jumping instructions work about the same. The main difference in JMP is that the operand is read in little endian and the PC is changed according to the operand. JSR also pushes the current PC (PC+2 since the beginning of the instruction) to the stack.

Functions:

printStep(PC, OPC, INS, AMOD, AC, XR, YR, SP, SR)

Prints PC count, operation code (OPC), assembly instruction (INS), addressing mode (AMOD) Accumulator (AC), X-register (XR), Y-register (YR), Stack pointer (SP), and status register (SR) to the screen (Figure 1). Instructions and addressing modes are formatted to be centered with three character width and right justified with 4 character width, in that order. All other variables are formatted to be 2 bytes wide in hexadecimal. All except PC which is formatted to be hexadecimal 4 bytes wide. A second string variable is used to print out status register appropriately at the end. Function the prints the output.

Adrian Azan
12/5/18
Revised
openFile(filename)

Checks if file extension exists and adds extension if it does not exist. The extension expected is .o for object and is checked by slicing the last two characters of the filename string passed in through the parameters. Opens file if it exists and returns file object.

checkRegisters(R)

Ensures that a registers value is between 0 and 255 at any time.

negative(R, SR)

Checks if the value in the register (R) is negative and sets status (SR) register accordingly. In order to keep consistency with the output formatting required, any number larger than 128 is considered negative and any number below 128 is positive.

zero(R, SR)

Checks if the value in the register (R) is zero. To account of decreasing and increasing to 0, both 0 and 256 are checked. Other wise the status register (SR) is set to 0. If the register is 256, it will soon be reset to 0 which is not negative so the negative status register is also set to 0.

carry(R, SR)

If the left most digit is equal to 1, then carry status register (SR) is set to 1. To check if carry exists, the register (R) is masked against 128 (10000000 in unsigned binary) and if the result is still 128 than the left most digit is 1.

Instructions by Name (From assignment 1 [Assembler] by Professor Christer Karlsson):
Instructions this emulator can process are **BOLDED**.

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

Adrian Azan
12/5/18
Revised
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

Test Case:

br.o contains 9 simple tests to check the branch and jump instructions. The only branch not tested in this file is BVS as none of the implied instructions affect the overflow status. Run the following addresses to see each branch and jump output.

Adrian Azan
12/5/18
Revised
0: Checks to make sure the **JMP** takes in 2 bytes for its operands and prints them accordingly and moves to the appropriate PC.
D: Checks that **JSR** pushes pc onto stack, reads and prints operand correctly. A bigger operand was chosen for this check.
1A: Checks **BCC** prints out operand properly and branches correctly.
27: Checks **BCS** set by setting carry just before branching.
34: Checks **BEQ** by incrementing, and then decrementing the x register to 0.
41: Checks **BMI** by decrementing the Y register several times to set the zero status.
4E: Checks **BNE** prints out operand properly and branches correctly.
5B: Checks **BLP** prints out operand properly and branches correctly by incrementing x register
68: Checks **BVC** by clearing carry register before branching (although that status begins at 0 anyway)