T34 User Manual

Contents

Running the Program

In a terminal, navigate to the folder containing t34.py. Then run

```
python3 t34.py {objfile}
```

objfile is an optional parameter. It is the name of a .obj file written in Intel HEX file format. If an obj file is supplied, the program will load the values specified by it into memory. Otherwise, memory will initialize to all 0s.

Usage

Once the program is started, you will see a terminal prompt. From there, you can input the desired command.

Display the Contents of a Specific Address

Simply enter the address you want to view and press enter. For example:

```
> 200
 200   A9
```

Display the Contents of a Range of Addresses

Enter the start address, a period, and the ending address, all with no spaces. For example:

```
> 200.20F
 200   A9 00 85 00 A5 00 8D 00
 208   80 E6 00 4C 04 02 00 00
```

Edit Memory Contents

Enter the start address you want to write to, followed by a colon, followed by a space-separated list of values. Values should be two hexadecimal characters each. For example:

```
> 300: A9 04 85 07 A0 00 84 06 A9 A0 91 06 C8 D0 FB E6 07
> 300.310
 300   A9 04 85 07 A0 00 84 06
 308   A9 A0 91 06 C8 D0 FB E6
 310   07
```

Run a Program

Enter the starting address of the program, followed by an R. For example:

```
> 200R
 PC  OPC  INS   AMOD OPRND  AC XR YR SP NV-BDIZC
 200  00  BRK   impl -- --  00 00 00 FC 00010100
```

For details on specific commands, see the the appendix.

Exiting

To exit the program, either:

- Ctrl-C
- Ctrl-D
- Type “exit” at the monitor prompt

Program Functions

core.py holds the “base” code for things like the main program loop, getting input, and displaying to the screen.

const.py has a dictionary mapping each opcode to a command and addressing mode, as well as SR bits.

t34.py holds the actual code functions, e.g. ASL. For a description of the t34 functions, see the appendix.

main

The main program loop. Gets the input from the user and, based on the format of the input, either displays addresses in memory, writes to memory, runs a program, or exits.

loadFile

Called on initialization if an obj file was supplied. This will read the file and load the contents into the appropriate place in memory.

getInput

Helps the main loop parse input from the user. It prompts the user for input and understands the instruction type based on the format of the input. It returns the instruction type and any relevant information (e.g. the addresses to load from if a display instruction was given).

display

Takes a start and end address, and prints the values in memory between those addresses (inclusive)

write

Takes a start address and a list of values and places the values in memory starting at the address given.

run

Takes an address and runs the program stored at the specified address. It will continue executing commands until the break flag is set.

intToHex

A helper function that takes an integer and returns a string with the integer’s hexadecimal representation.

CheckNZ

Sets the N and Z bits of the status register based on the value supplied to it

Branch

Adds the value of arg1 (interpreted as a signed int) to the PC

GetValue/SetValue

Based on the addressing mode, these will fetch a value from memory or set a value in memory.

GetAddr

Gets the address specified by the current addressing mode. This just exists to reduce code duplication in GetValue & SetValue.

Testing Process

I downloaded the code3_X.obj files from the COA website and ran each of them to check that they matched the results in the design document. The only difference I found was in code3_4.obj. This is what my code produces, pay close attention to the last two lines:

```
> 300.31F
 300   A9 01 85 00 18 2A A5 00
 308   8D 00 80 A2 03 A0 03 88
 310   D0 FD CA D0 F8 4C 05 03
 318   00 00 00 00 00 00 00 00
> 300R
 PC  OPC  INS   AMOD OPRND  AC XR YR SP NV-BDIZC
 300  A9  LDA      # 01 --  01 00 00 FF 00000000
 302  85  STA    zpg 00 --  01 00 00 FF 00000000
 304  18  CLC   impl -- --  01 00 00 FF 00000000
 305  2A  ROL      A -- --  02 00 00 FF 00000000
 306  A5  LDA    zpg 00 --  01 00 00 FF 00000000
 308  8D  STA    abs 00 80  01 00 00 FF 00000000
 30B  A2  LDX      # 03 --  01 03 00 FF 00000000
 30D  A0  LDY      # 03 --  01 03 03 FF 00000000
 30F  88  DEY   impl -- --  01 03 02 FF 00000000
 310  D0  BNE    rel FD --  01 03 02 FF 00000000
 30F  88  DEY   impl -- --  01 03 01 FF 00000000
 310  D0  BNE    rel FD --  01 03 01 FF 00000000
 30F  88  DEY   impl -- --  01 03 00 FF 00000010
 310  D0  BNE    rel FD --  01 03 00 FF 00000010
 312  CA  DEX   impl -- --  01 02 00 FF 00000000
 313  D0  BNE    rel F8 --  01 02 00 FF 00000000
 30D  A0  LDY      # 03 --  01 02 03 FF 00000000
 30F  88  DEY   impl -- --  01 02 02 FF 00000000
 [...]
```

In the instruction document, there is another line between the last two instructions shown:

```
[...]
30D  A0  LDY      # 03 --   01 02 03 FF 00100000
310  D0  BNE    rel FD --   01 02 02 FF 00100000
30F  88  DEY   impl -- --   01 02 01 FF 00100000
[...]
```

I believe this was either a typo or a bug in the instruction document, since LDY immediate should only advance the PC by 2 (from 30D to 30F), not 3 (from 30D to 310).

Other than code3_4.obj, my program ran identically to the sample code provided in the instruction document.

Known Bugs

The program will crash if it tries to execute an opcode it's not familiar with or the user supplies an invalid instruction at the prompt.

Appendix: T34 Instructions & Addressing Modes

See const.py for opcodes

Instructions

Command	Description
ADC	Add memory to accumulator with carry
AND	Logical AND memory with accumulator
ASL	Arithmetic shift left (accumulator or memory)
BCC	Branch if C = 0
BCS	Branch if C = 1
BEQ	Branch if Z = 1
BIT	Bits 7 & 6 of operand are transferred to N and V bits respectively; Z is set to operand AND accumulator
BMI	Branch if N = 1
BNE	Branch if Z = 0
BPL	Branch if N = 0
BRK	Break
BVC	Branch if V = 0
BVS	Branch if V = 1
CLC	Clear the carry bit
CLD	Clear the decimal bit
CLI	Clear the interrupt bit
CLV	Clear the overflow bit
CMP	Subtract memory from accumulator, set NZC bits based on result
CPX	Subtract memory from X register, set NZC bits based on result
CPY	Subtract memory from Y register, set NZC bits based on result
DEC	Decrement memory by 1
DEX	Decrement the X register
DEY	Decrement the Y register
EOR	Logical XOR memory with accumulator
INC	Increment memory by 1
INX	Increment the X register
INY	Increment the Y register
JMP	Jump to new location
JSR	Jump to new location, storing PC on stack
LDA	Load memory into accumulator
LDX	Load memory into X register
LDY	Load memory into Y register
LSR	Logical shift right (accumulator or memory)
NOP	No operation
ORA	Logical OR memory with accumulator
PHA	Push the accumulator onto the stack
PHP	Push the status register onto the stack
PLA	Pull the accumulator from the stack
PLP	Pull the status register from the stack

ROL	Roll left (C into lowest bit, highest bit into C)
ROR	Roll right (C into highest bit, lowest bit into C)
RTI	Return from interrupt (Pull SR, pull PC)
RTS	Return from subroutine (Pull PC)
SBC	Subtract memory from accumulator with borrow
SEC	Set the carry bit
SED	Set the decimal bit
SEI	Set the interrupt bit
STA	Store accumulator into memory
STX	Store X register into memory
STY	Store Y register into memory
TAX	Transfer accumulator to X register
TAY	Transfer accumulator to Y register
TSX	Transfer stack pointer to X register
TXA	Transfer X register to accumulator
TXS	Transfer X register to stack pointer
TYA	Transfer Y register to accumulator

Addressing Modes

Addressing Mode	Abbr	Value Used
Implied	impl	No value
Accumulator	A	accumulator
Immediate	#	arg1
Absolute	abs	memory[arg1 + (arg2 << 8)]
Absolute, X-indexed	abs,x	memory[arg1 + (arg2 << 8) + X]
Absolute, Y-indexed	abs,y	memory[arg1 + (arg2 << 8) + Y]
Zeropage	zpg	memory[arg1]
Zeropage, X-indexed	zpg,x	memory[(arg1 + X) & 0xFF]
Zeropage, Y-indexed	zpg,y	memory[(arg1 + Y) & 0xFF]
X-indexed, Indirect	x,ind	memory[memory[(arg1 + X) & 0xFF] + (memory[(arg1 + 1 + X) & 0xFF] << 8)]
Indirect, Y-indexed	ind,y	memory[memory[arg1] + (memory[arg1 + 1] << 8) + Y]
Indirect	ind	memory[arg1] + (memory[arg1 + 1] << 8)
Relative	rel	arg1 (signed)