Assembler for the PET

BY MARK ZIMMERMANN

In the Commodore PET, the operating system and the BASIC interpreter fill the supplied ROMs to capacity. There was no room left for an assembler, which you'll need to develop and run machine-language programs. An assembler will not only make programming easier and more efficient, but also increase the speed of program execution.

This PET assembler, a BASIC program that takes assembly-language instructions like LDA 17 and STA 32768, Y and translates them to opcodes the 6502 microprocessor understands, fills that need not only for the PET but for other 6502 systems as well. The assembler occupies about 4K bytes of memory (and, by streamlining, should fit into a 4K PET's space). The assembler takes about half a second to interpret each line of your assembly language input (the program you wish to translate). The resulting program can then be expected for rapid output.

Assembly language programming is no more difficult than programming most pocket calculators. This BASIC program lets you write in assembly language on 6502 systems.

This tiny assembler neither makes symbol tables of variables or labels nor allows users to define "macros" (to add such features would greatly increase complexity, length and runtime). The assembler, given the absolute address of the target, calculates, branches and catches most (though not all) of the errors that operators can make with the 6502's various addressing modes. This assembler also contains all 6502 instructions, including the often omitted ROR.

Programming in assembly language isn't difficult; it's like programming most pocket calculators. Assembly language programs, which usually run 10 to 100 times faster than the same programs written in BASIC, are a must for applications where speed is essential. If you've never tried assembly language programming before, see *Programming a Microcomputer:* 6502, by Caxton C. Foster; *How to Program Microcomputers,* by William Barden, Jr.; and the MCS6500 *Microcomputer Family Programming Manual,* from MOS Technology, Inc.

Assembler structure

The assembler (see Program Listing) is written in BASIC; it would be fun to use it to write a machine-language version of itself, but that might be a long project! The BASIC version begins with DATA statements used to initialize the arrays M$, OP, and CA. M$ has 56 elements (numbered 0 to 55) which are the possible mnemonics for the 6502: ADC, AND, . . . , to TYA. For each element of M$, the corresponding element of OP contains the "base opcode". Many instructions have only one possible mode of addressing, and their "base opcode" is the actual (decimal) opcode understood by the microprocessor. Other instructions can have a variety of addressing modes; for example, LDA #27; LDA 32770; LDA 255; LDA (2,X); LDA (34), Y; LDA 47,X; LDA 2001,X; and LDA 1066,Y. For such instructions. the "base opcode" will have some integer added to it, depending on the addressing mode. Adam Osbourne's tables of opcodes and instructions in Volume 2 of his series on microcomputers makes this explicit by showing which bits of the binary instruction change for each mode. (Note that he omits the ROR instruction, however.)

So, M$(0), M$(1), . . ., M$(55) contain the mnemonics, OP(0) through OP(55) contain the corresponding base opcodes, and finally, CA(0) to CA(55) contain the instruction "category". For instance, all "branches" are category 8, a JMP is category 6, etc. The categories identify which instructions go with which addressing modes.

After initializing the mnemonics, opcodes and categories, the assembler asks for the starting address, the location for the output of the assembler to begin at. The PET assembler writes to tape (and to the video display of the PET). It's easily modified to write directly into memory (in fact, that's how the first version I wrote operated), but that's a dangerous modification! If an area of memory not used by the BASIC program itself is written into, users risk that the BASIC interpreter will use that region for string storage or other purposes. Even if an absolutely "safe" area is used, such as the "2nd cassette buffer" region, when the machine-language program has a bug in it, you can't interrupt it without turning power off and losing the whole thing. It's much better procedure to write all but the shortest machine-language programs on tape, where they can be preserved in case of catastrophic failure!