Upward Migration

Part 1: Translators

Using translation programs to move CP/M-86 programs to CP/M and MS-DOS

Roger Taylor and Phil Lemmons
C/o BYTE Publications Inc.
POB 372
Hancock, NH 03449

In software migrations as in any other, the first thing you must know is where you are and where you are going. If a Californian decides to move to Australia, he can call an airline, ask for a ticket from San Francisco to Sydney, and the airline clerk will be happy to reserve a seat for him. If the Californian asks for a ticket from Los Angeles-or-San Francisco to Sydney, however, the airline clerk will find the request confusing. If the Californian asks for a ticket from Los Angeles-or-San Francisco to Sydney under Catholic rule, or a ticket from Los Angeles-or-San Francisco to Sydney under Protestant rule, the airline clerk will probably say, "You've already spent too much time in the hot tub, buddy. Stay in California. You're right where you belong."

In this little parable of modern times, Los Angeles is the 8080, San Francisco is the Z80, Sydney is the 8086, Catholicism is CP/M-86, and Protestantism is MS-DOS. The operating systems are represented by religions because they generate similar passions, controversies, true believers, and skeptics.

Is there really any need to explain why Los Angeles stands for the 8080 and San Francisco for the Z80? Or that the airlines stand for the software houses that have written translation programs?

You've probably guessed who the guy is who's trying to buy the ticket. He's the experienced 8080 or Z80 programmer, and the hot tub symbolizes his strong preference for staying right where he is—on one familiar processor with one familiar operating system. What could be cozier?

The programmer may not have the urge for going, but he has to go to one unfamiliar processor and to both operating systems. And so do the rest of us.

We're lucky to have software from at least three companies to help us along. In this article, the first of two parts, we will review three CP/M-80-to-8086 translator programs. We'll...
Figure 1: Architecture of the 8080 microprocessor showing the internal registers.

show the output that each of the three translators produced from the same original code. In addition, we'll make some observations about the differences in adapting the translated code to CP/M-86 and MS-DOS. Next month, we'll take a closer (although still not comprehensive) look at CP/M-86 and MS-DOS.

Orientation to the 8086

The first thing we have to do is examine the differences between the familiar 8080 and Z80 microprocessors and the 8086. For reference, figure 1 shows the registers and architecture of the 8080; figure 2 shows the registers and architecture of the Z80. We'll make few comments about these registers because they are familiar to you if you have 8080 or Z80 source code that you want to translate.

Figure 3 shows the registers and architecture of the 8086. Since the 8086 is less familiar, we'll take a brief look at it for orientation. (For further enlightenment, see The 8086 Book, Russell Rector and George Alexy, Osborne/McGraw-Hill, 1980, and The 8086 Primer, Stephen P. Morse, Hayden, 1980.) The 8086 is, of course, a 16-bit microprocessor. The 8088 can access up to 1 megabyte of memory and as many as 65,000 input/output ports. The megabyte of memory is $2^{20}$ 8-bit bytes; any two consecutive bytes are a 16-bit word. Some 8086 instructions access bytes; others access words.
Sixteen bits are not enough to address a megabyte of memory. The 8086 manages to do so, however, by dividing the megabyte of memory into a number of segments of 64K bytes each. Each segment begins at an address that yields an even result when divided by 16.

All calculations of memory addresses in the 8086 involve four special registers called segment registers. The 8080 family has a 16-bit address bus that allows addressing of 65,536 bytes of memory. While the internal registers of the 8088/8086 family also have 16 bits, the external address bus has 20 bits. To get the 20-bit address, the 8086 extends a segment register with 4 low-order bits of 0, and adds the segment register to a 16-bit address from another register, as shown in figure 4.

Each segment register defines what is known as its own "current" segment. Each instruction specifies an offset into a segment. The segment registers, which cannot be used interchangeably, are as follows:

- **CS**—The Code Segment register defines the 64K-byte current code segment. When an instruction is fetched, the contents of the program counter are added to the CS register contents to calculate the address of the instruction to be fetched.

- **DS**—The Data Segment register defines the current data segment. With three exceptions, all data memory references are understood in relation to the DS register. (The exceptions are that the stack pointer is used to calculate stack addresses, any data memory addresses calculated using the BP register are taken in relation to the stack segment, and any string operations involving the destination are taken in relation to the extra segment. See SS and ES immediately below.)

- **SS**—The Stack Segment register identifies the current stack segment. References to data memory that use the BP or SP register in calculating the address are understood in relation to the SS register. For example, the PUSH, POP, CALL, INT, and RET instructions use the SS register.

- **ES**—The Extra Segment register plays...
The 8086 extends a segment register with 4 low-order bits of 0. It then adds the segment to a 16-bit address from another register to achieve a 20-bit address.

Figure 5: Register-usage mapping between the two processor families is shown. Note that the IX and IY registers of the Z80 do not exist for the 8080/8085. More registers are in the 8088/8086 than in either the 8080 or Z80. Observe that the 8088/8086 segment registers have no parallel in the 8080/Z80 family. The 8080 family has a 16-bit address bus that allows addressing of 65,536 bytes of memory. While the internal registers of the 8088/8086 family also have 16 bits, the external address bus has 20 bits, formed by extending a segment register with 4 low-order bits of 0, and adding it to a 16-bit address from another register.
because its extensive instruction set has to be mapped to the 8086 instructions, as well as the 8080 registers.

The 8086 does, however, have a rich instruction set that covers all the bit instructions and most of the block instructions. The problems encountered in mapping instructions and registers can be formalized and solved by using a variety of software tools.

To convert a source file running under CP/M-80 so as to assemble it with an 8086 assembler requires either the use of a code translator or a considerable effort with a program editor. At least three commercial products come under the translator category:

- XLT68 from Digital Research Inc.
- TRANS86/ACT86 from Sorcim
- The Z80-to-8086 translator from Seagate Computer Products

Intel also has a translator, CONVERT 86, but it is sold only as part of a large software package intended for its OEM customers and doesn't run under CP/M. In addition, some reports indicate that Microsoft may soon bring out a translator.

Two issues must be addressed when converting from a CP/M-80 source:

1. Where is the source coming from?

Z80 instruction mnemonics?
(a) Zilog
(b) TDL (Technical Design Labs)
(c) Intel (Digital Research with macros)
(d) Sorcim/ACT-80

or

Straight 8080/8085 code with no macros?

2. Where is the resultant code going?

CP/M-86 or MS-DOS?

The three translators under review do not always approach these issues in the same way. Furthermore, they all handle register mapping somewhat differently. We'll look at the translators now one at a time. Then we'll see how they translated the same CP/M-80 listing. With the listing in hand, we'll see how each of the translators handled register mapping. Finally, we'll turn to the subject of how the two different 16-bit operating systems affect program translation.

**XLT86 from Digital Research**

Digital Research's XLT86 takes standard 8080 source code in a format compatible with ASM, MAC, or RMAC assemblers and converts the 8080 source code to 8086 source code in a format compatible with ASM86 or RMAC86. Since XLT86 is written in PL/1-80, the translator can run either stand-alone under CP/M-80 or for cross development under VAX/VMS. It produces optimized 8086 code in a five-phase, multipass process, doing global data-flow analysis to determine optimal register usage.

Although macro definitions are not supported, conditional-assembly directives are. The XLT86 User's Guide suggests that if you want macro expansion, you can use a pass through MAC or RMAC to produce a PRN file that can be edited (removing the first few columns of generated hexadecimal code) to produce an expanded source file for input acceptable to XLT86. XLT86 does not recognize Z80 instructions. XLT86 passes repeat loops through to the 8086 source code.

XLT86 analyzes the source program in its entirety, determining the block structure and the register/flag usage. Working from this information, it translates the code to 8086 assembler code in an optimized way. The decision algorithm for each instruction type is given in a section of the manual to allow the user to see what happens in each situation.

Register mapping generally follows the correspondence shown in figure 4, with a loose relationship between the 8086 AX and the 8080 PSW; the exact relationship is determined from register usage at translate time.

Many run-time options are available to control the translation pro-
The options control the disks that the work and output files are on, whether the block-analysis information is output to disk, whether code and data segments are to be intermixed or kept separate, and whether the condition flags are active on exiting from subroutines.

XL T86 is a sophisticated program that does a reasonable job of optimizing the translation of 8080 source code to 8086 source code. BDOS calls from CP/M-80 are mapped into BDOS calls that are compatible with CP/M-86.

XL T86 has special features for handling translation of conditional JMP and CALL instructions in 8080 source code. In the 8080 instructions, JMP and CALL instructions are capable of reaching any address within the 64K-byte region. The 8086 conditional JMP instructions can reach only 128 bytes on either side of the IP (Instruction Pointer) register. XL T86 examines the target of the conditional JMP, if the target cannot be reached, XL T86 changes the sense of the conditional JMP and skips over a long JMP to the target address. Since there are no conditional CALL or RET instructions in the 8080, the sense of the condition is changed and a short conditional JMP is performed to skip over an unconditional CALL or RET.

As noted earlier, the segment registers allow for separation of code and data regions. To reference data, you have to tell the 8086 whether data is in the code segment (CS) or the data segment (DS). For the Digital Research ASM86 assembler, the Offset directive handles this chore.

XL T86 examines an expression and determines the proper segment for the particular instruction.

XL T86 does have limits on the size of the 8080 source files that it can translate because the flow-analysis information must be in memory. In a 64K-byte CP/M system, the maximum source file that can be translated is approximately 6K bytes, depending on the structure of the program. Nothing is said in the manual about being able to deal with modular code using RMAC and external references. This implies that the entire source program must be converted at once, limiting the size of the program that can be translated by using XL T86 to 6K bytes.

In summary, if you're starting from 8080/8085 assembly code written for ASM or MAC and you want to go to CP/M-86, and if the source program does not exceed 6K bytes, XL T86 is the most useful translator. Code written for Z80s using MAC requires careful examination after the translation process to make sure that no
flags were inadvertently changed. You have to expand macros before using XLT86 unless the number of invocations was small. In that event, expanding the macros by hand with an editor might be just as easy.

Sorcim’s TRANS86

TRANS86 is an 8080/ACT80-to-8086 translator. It takes input to ACT86 source code as input and creates a file compatible with the input to ACT86, an 8086 assembler. The output of TRANS86 is incompatible with any assembler other than ACT86. The ACT86 mnemonics are different enough so that, unless the programmer has a sophisticated text processor and the talent (or patience) to do a great deal of text manipulation, TRANS86 should be used only with ACT86.

Both TRANS86 and ACT86 run on either 8080 or Z80 processors under CP/M-80, MP/M, or CDOS with a minimum of 24K bytes of RAM (random-access read/write memory). TRANS86 consists of an executable file, an overlay file, and a translation table. The input assembly source code must be in a form acceptable to TRANS86. The ACT80 instruction set has some ASM-style instructions, some Z80-style instructions, and some instructions that are unique to the ACT80 assembler. If the source code is written in 8080 ASM mnemonics, TRANS86 will process it and output ACT86 assembler code. The 8080 instruction SPHL, however, was translated incorrectly in the current version of TRANS86.

Another section describes the differences between ACT80 and Z80 mnemonics. This information allows the programmer to manually convert assembly source code to a form acceptable to TRANS86. The ACT80 instruction set has some ASM-style instructions, some Z80-style instructions, and some instructions that are unique to the ACT80 assembler. If the source code is written in 8080 ASM mnemonics, TRANS86 will process it and output ACT86 assembler code. The 8080 instruction SPHL, however, was translated incorrectly in the current version of TRANS86.

Another section in the manual contains suggested constructions that can be manually entered to deal with Z80 op codes that are flagged as errors. Block input/output instructions and input/output through register C are described in detail.

---

NEW FROM EXTENDED PROCESSING
THE BURNER I/O FOR THE S-100 BUS

The BURNER I/O has a complete EPROM programmer, two serial ports, one parallel I/O port with handshaking and memory management.

Programmer features:
- Most EPROM types from 2704 thru 2764 and 2508, 2516 and TMS2716.
- CPM compatible software supplied in EPROM that can be easily written on a diskette.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O features: (serial)</td>
<td>2 fully independent RS-232 serial ports.</td>
</tr>
<tr>
<td></td>
<td>RS-232 data ready supported.</td>
</tr>
<tr>
<td></td>
<td>Each serial port has independent baud rate</td>
</tr>
<tr>
<td></td>
<td>generators that are software programmable</td>
</tr>
<tr>
<td></td>
<td>from 50 to 19,200 baud.</td>
</tr>
<tr>
<td></td>
<td>Serial ports may be polled and/or interrupt</td>
</tr>
<tr>
<td></td>
<td>driven.</td>
</tr>
<tr>
<td>I/O features: (parallel)</td>
<td>Independent 8 bit output, input and status</td>
</tr>
<tr>
<td></td>
<td>flags.</td>
</tr>
<tr>
<td>I/O including flags are latched.</td>
<td></td>
</tr>
<tr>
<td>Memory management features:</td>
<td>Controls the S-100 address lines from A16-A23.</td>
</tr>
<tr>
<td></td>
<td>Uses output instruction to load the address.</td>
</tr>
<tr>
<td></td>
<td>Normally 40H.</td>
</tr>
</tbody>
</table>

We are offering this board with all options, or just the portions that are needed may be purchased. Regardless of which version that is purchased, documentation for the entire board is supplied. All combinations are assembled and tested.

Option A: Complete board with programmer, I/O and memory management, $324.95
Option B: Programmer only, $199.95
Option C: I/O only, (2S + P) $199.95
Option D: Option B and C, $299.95
Option E: Memory management only $150.00
Memory management may be added to options B or C for $25.00.

AVAILABLE FROM
Microbyte Computer Systems
2798 So. Bascom, San Jose, CA 95124
(408) 377-4685
DEALER INQUIRIES INVITED

Circle 308 on inquiry card.
**Seattle Computer System**

Seattle Computer Products' (SCP) translation system consists of two programs: a Z80 translator (called TRANS86 on the disk, though not the same as Sorcim's) and a compatible 8086 cross assembler (ASM86). Both programs run on Z80 processors under CP/M-80 with a minimum of 24K bytes of RAM. The programs do not run on 8080/8085 processors.

The translator accepts source files in Zilog/Mostek mnemonics and produces an 8086 assembler source file in a form acceptable to the SCP 8086 cross assembler. Since the 8086 source format required by the SCP 8086 cross assembler is different from any other 8086 assembler that we know of, you must use these two programs together. The translator places its output on the same disk as the Z80 source code and gives it an .A86 file extension.

The translation is on an instruction-by-instruction basis with no optimization. There appears to be no limit on the file size that may be translated. Not all Z80 instructions are translated, however. Those in the following list will produce an op code error:

- Cpd
- Cpi
- Ind
- Indr
- Ini
- Inr
- Ldd
- Ldi
- Ottd
- Otir
- Outd
- Outi
- Rld
- Rd

These op codes are mostly in the block-manipulation set of instructions. Although programmers do use these instructions, they must be manually coded when converting to the 8086. The SCP translator does not support macros and permits use of the following pseudo-ops only:

- Db
- Dm
- Ds
- Dw
- Equ
- If/Endif
- Org

If the Z80 index registers IX and IY are used, they are mapped into memory locations with the labels IX: and IY:. The programmer has to define these locations; otherwise, they will show up on the assembly listing as undefined labels. The Z80 alternate register set (BC,DE,HL') is treated the same way, as memory locations that the programmer must define.

Either the DI or SI register can be used as a temporary IX register by loading one of them from the location IX when required to do indexed instructions. The programmer has to take care of this substitution; the translator does not.

When using the DI register, you must always keep in mind that the only 8086 segment base that can be used with the DI register is the ES segment; the SI register, on the other hand, can reference all the segment bases, defaulting to the DS segment. If this 8086 source code is going to be run under CP/M-86, you have to be careful about using the ES segment register. The CP/M-86 documentation specifically states that ES is not saved through a BDOS call.

For the SCP Z80-to-8086 translator's register usage, see figure 6b.

**Translating the Test File**

To determine how the three programs actually translate source text, we prepared a file acceptable as input to an assembler and containing all the op codes of the 8080 and Z80. Since the SCP translator could accept only Zilog/Mostek mnemonics, the test text was run through an 8080-to-Z80 filter program before the translation.

Listing 1 presents, side by side and line by line, the original 8080 code, the Sorcim TRANS86 translation, the Seattle translation, and the XLT86 translation. Here are reminders of some things to consider when you examine the translations:

- Because of the differences in the architectures of the 8086 and the 8080/Z80, some choices must be made when translating from one architecture to the other. Therefore, some difference in translation is to be expected.
NEW FROM NETRONICS
AUTO-PATCH HARD DISK
With plug-in multi-user ports
Automatically Installs Itself Into
Your Present CP/M® 2.2 Operating
system & Floppy Disk Hardware.
It's Exclusive!
6 megabytes... $2950.00
12 megabytes... $3495.00

What is the big concern of $100 owners when they consider adding Hard Disks? They worry that it will be difficult to install, that it won't be compatible with their present software and hardware, and that it may cause down-time on their $100 system.

Worry no more — Netronics new AUTOPATCH Hard Disk Systems are here. AUTOPATCH installs in just one-two-three: (1) plug in the hard disk S100 card; (2) run three short programs supplied on disk; (3) disable the boot on your floppy controller and enable the boot on your hard disk controller (this step not required if you wish to continue to boot to your floppy drives). And thats it: The AUTOPATCH feature automatically finds the end of your existing BIOS and then self relocates and patches itself into the existing BIOS. A copy of CCP and BIOS are loaded into memory, a customized $BOOT is added to the front of CCP and the whole memory image is written to the reserved tracks on your hard disk. You can add up to 4 hard disks to the controller supplied. The new BIOS will automatically rename any old devices as B and C and define the hard disk as drive A. All with the lift of one finger!!! If your BIOS is large you may have to re-size your system down 1 or 2 k. If this is necessary the AUTOPATCH program will prompt you to do so.

AUTOPATCH Hard Disk Systems are available in 6 and 12 megabyte models. Included in the system: 6 or 12 megabyte Hard Disk Drive - Controller for up to 4 Hard Disk drives - S100 Hard Disk card with provisions for adding 8 additional I/O ports to be used when adding a multi-user operating system - Power Supply - Deluxe Steel Cabinet - All necessary cables - AUTOPATCH Programs supplied on either 8” or 5 1/4” IBM formatted single density diskettes (specify style required) - Complete installation instructions - Fully wired and tested, ready to go.

SPECIFICATIONS
Unformatted Recording Capacity: 6.4 or 11.6 M
Number of Tracks: 612 or 1056
Bytes/sector format: 512
Communication Port: 
DO (other ports available on special order). Proograms supplied on 5 1/4” or 8” single density IBM formatted diskettes (North Star CP/M® version available on special order)

10 DAY MONEY BACK OFFER
Continental U.S.A. Credit Card Buyers Outside Conn.
CALL TOLL FREE 800-243-7428
To Order From Connecticut Or For Tech. Assist. Call (203) 354-9375

NETRONICS R&D LTD. Dept.
333 Litchfield Road, New Milford, CT 06776
Print Name ________________________________
Signed ______________________________________
Address ______________________________________
City __________________ State ________ Zip ___________
Listing 1: The original 8080 code, the Sorcim TRANS86 translation, the Seattle translation, and the XLT86 translation.

<table>
<thead>
<tr>
<th>Original 8080 code</th>
<th>TRANS86</th>
<th>Seattle</th>
<th>XLT86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stax B</td>
<td>Xchg BX,CX</td>
<td>Mov DI,CX</td>
<td>Mov SI,CX</td>
</tr>
<tr>
<td></td>
<td>Sto AL,[BX]</td>
<td>Stob</td>
<td>Mov [SI],AL</td>
</tr>
<tr>
<td></td>
<td>Xchg BX,CX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inx B</td>
<td>Inc CX</td>
<td>Lahf</td>
<td>Lahf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inc CX</td>
<td>Inc CX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sahf</td>
<td>Sahf</td>
</tr>
<tr>
<td>Dad B</td>
<td>Add BX,CX</td>
<td>Lahf</td>
<td>Lahf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add BX,CX</td>
<td>Add BX,CX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rcr SI</td>
<td>Sahf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rcl SI</td>
</tr>
<tr>
<td>Ldax B</td>
<td>Xchg BX,CX</td>
<td>Mov SI,CX</td>
<td>Mov SI,CX</td>
</tr>
<tr>
<td></td>
<td>Ld AL,[BX]</td>
<td>Lodb</td>
<td>Mov AL,[SI]</td>
</tr>
<tr>
<td></td>
<td>Xchg BX,CX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stax D</td>
<td>Xchg BX,DX</td>
<td>Mov DI,DX</td>
<td>Mov SI,DX</td>
</tr>
<tr>
<td></td>
<td>Sto AL,[BX]</td>
<td>Stob</td>
<td>Mov [SI],AL</td>
</tr>
<tr>
<td></td>
<td>Xchg BX,DX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dad H</td>
<td>Add BX,BX</td>
<td>Lahf</td>
<td>SBL BX,1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add BX,BX</td>
<td>Add BX,BX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rcr SI</td>
<td>Sahf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rcl SI</td>
</tr>
<tr>
<td>Dec H</td>
<td>Dec BX</td>
<td>Lahf</td>
<td>Dec BX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sahf</td>
<td></td>
</tr>
<tr>
<td>Inr M</td>
<td>Incb [BX]</td>
<td>Inc B,[BX]</td>
<td>Inc Byte Ptr 0[BX]</td>
</tr>
<tr>
<td>Mvi M,3</td>
<td>Stob #3,[BX]</td>
<td>Mov B,[BX],3</td>
<td>Mov Byte Ptr 0[BX],3</td>
</tr>
<tr>
<td>Dad SP</td>
<td>Add BX,SP</td>
<td>Lahf</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Add BX,SP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rcr SI</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sahf</td>
<td></td>
</tr>
<tr>
<td>Dcx SP</td>
<td>Dec SP</td>
<td>Lahf</td>
<td>Dec SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sahf</td>
<td></td>
</tr>
<tr>
<td>Mov M,A</td>
<td>Sto AL,[BX]</td>
<td>Mov [BX],AL</td>
<td>Mov Byte Ptr 0[BX],AL</td>
</tr>
<tr>
<td>Rnz</td>
<td>Jz :G1</td>
<td>Jnz Ret</td>
<td>Jz L 1</td>
</tr>
<tr>
<td></td>
<td>Ret</td>
<td></td>
<td>Ret</td>
</tr>
<tr>
<td></td>
<td>:G1</td>
<td></td>
<td>L 1:</td>
</tr>
<tr>
<td>Jnz Help</td>
<td>Jz :G2</td>
<td>Jnz Help</td>
<td>Jz L 2</td>
</tr>
<tr>
<td></td>
<td>Jmp Help</td>
<td></td>
<td>Jmp Help</td>
</tr>
<tr>
<td></td>
<td>:G2</td>
<td></td>
<td>L 2:</td>
</tr>
</tbody>
</table>

*Please call for quote.*
translated text, also. We therefore suspect that the program has a minor bug with regard to this particular instruction.

Since this test program was nonsense as far as logical program flow is concerned, XLT86 was at a loss to determine the active registers, and sometimes chose simpler instructions than it would have in a real program. The results for XLT86 could be different in different situations.

The conditional jumps for TRANS86 and XLT86 can vary, depending on the distance of the target label from where the jump is.

The conditional return in Seattle's translator references a label called RET. This refers to any RET within 128 bytes on either side of the statement. This is one reason why Seattle's translator should be used with ASM86; no other assembler will take advantage of this feature.

Note that TRANS86 and the Seattle translator treat the DJNZ instruction differently. TRANS86 uses a 16-bit register, CX, and the Seattle translator uses CH, an 8-bit register. A warning message comes out of the Seattle translator reminding the programmer to take care of the Z80's IX, IY, BC, DE, and HL registers.

Summing Up the Translators
A general view is that Sorcim's TRANS86 is a useful product if the original source is in 8080 or ACT80 form and the user has ACT86 as a target 8086 assembler. The register and flag usage appear to be a little looser than for the other two programs. This requires more knowledge and more involvement from the programmer to make sure that the sense of the translated code is maintained. No limitations exist as to the size of the source file and macros are supported if the input is in ACT80 format.

Sorcim's TRANS86 is sold separately from ACT86, but they should be used together.

The Seattle Computer Products' Z80-to-8086 translator is a straightforward code translator that uses Zilog mnemonics and runs only on Z80-based processors. There appear to be no limitations as to the size of the source program that may be translated since the program translates one instruction at a time. Register and flag usage are very conservative, protecting the source architecture as much as possible and providing warnings when potential problems could arise. The converted program has more of a chance of working the first time than a less conservative translation would have.
Once the program is working correctly, the programmer can reedit the 8086 source to trim out the unnecessary register saves and flag manipulations to speed up the code and make it smaller. If the source code is in Zilog mnemonics, or if a filter program is available to convert to Zilog mnemonics, we recommend the pair of programs from Seattle Computer Products. The Seattle translator and its ASM86 assembler should be used together for optimum benefit.

XLT86 from Digital Research is a program that takes 8080 source code and converts it into 8086 source code in an intelligent manner using data-flow-analysis techniques. It will produce better code than either of the other two translators if two conditions are met: (1) no 8080 instructions are used, and (2) the source code is not bigger than 6K bytes (assuming the CP/M system is a 64K-byte system). XLT86 should be used with ASM86 under either CP/M-80 or CP/M-86.

**MS-DOS versus CP/M-86**

In the trade press and various advertisements, we see claims that conversion of CP/M-80 programs to MS-DOS (IBM PC-DOs, Lifeboat Associates' SB-86, Seattle Computer Products' 86-DOS) is as easy, or even easier, than from CP/M-80 to CP/M-86. In fact, some differences are seen in interfacing to the two 16-bit operating systems. With the assumption that the program had to operate in a 64K-byte region under CP/M-80, we will bypass memory-allocation questions. The remaining issues that have to be addressed are:

- How does a program gain access to operating system resources?
- How are file control blocks used to manipulate files?

Gaining access to the operating system under CP/M-80 requires placing the function code in the C register, placing the information address in the DE registers, and calling location 05 hexadecimal, the CP/M-80 entry point. If the system call returns a value, it returns the value to the A register. A so-called warm boot under CP/M-80 is accessed by jumping to location 00 hexadecimal, which reads in the operating system and resets the disk system.

Now let's look at similar functions under CP/M-86 and MS-DOS.

**Gaining Access to CP/M-86**

Gaining access to CP/M-86 requires placing the function code in the CL register, placing the byte parameter in the DL register or placing the word parameter in the DX register, placing the data segment in the DS register (the data segment is usually not changed for a converted program), and executing a software interrupt, INT #224. The result is returned in the AL register if it is a byte value; if the result is a word value, it is returned in both the AX and BX registers. Double-word values are returned with the offset in the BX registers and the segment in the ES register. Conversion of programs from CP/M-80 to CP/M-86, then, requires replacing the call to location 5 with the software interrupt INT #224.

Another necessary change involves the warm boot. Under CP/M-80, the warm boot may be accessed by a system call with a function code of 0 for a jump to location 0. CP/M-86, however, does not support the jump to location 0. As a result, you must change this program exit in the translated program if the program is to run correctly.

Provided that the call to location 5 is replaced with INT #224, that the warm boot change is made, and that the registers are mapped correctly, there should be little problem in getting the translated program to access the CP/M-86 system functions.

**Gaining Access to MS-DOS**

Although MS-DOS has a "preferred" mechanism through a software interrupt, INT #33, for accessing the system, an additional mechanism is provided for "preexisting" programs that is compatible with CP/M-80 calling conventions, at least for functions in the range of
0-36. As far as system calls within the allowed function range are concerned, the programmer doesn't have to do anything to translated programs to get them to run under MS-DOS other than to correctly map the registers.

MS-DOS also supports the warm boot function of CP/M-80. A jump to location 0 under MS-DOS executes a software interrupt, INT #32, which is functionally a program end and the normal way to exit from a program.

Manipulating File Control Blocks

The file control block used in CP/M-80 consists of a 36-byte block, which describes the disk drive on which to find or create the file, the file name, and information relating to which record of the file is desired.

At least so far as normal file-access requests are concerned, both MS-DOS and CP/M-86 treat this block of information the same.

System-level information is quite different in the two cases, and programs that look at system bytes within the file control block need to be changed for MS-DOS to function correctly. The MS-DOS file control block has many more features, including the date the file was created or last updated, the logical record size, and the file size. These system-information bytes are in areas within the file control block that application programs normally do not access. Nevertheless, converting programs to make use of MS-DOS file control blocks should take little effort.

Conclusion

There is, in fact, little if any difference in the difficulty of translating sound CP/M-80 programs to CP/M-86 or MS-DOS. With CP/M-86, the programmer will have to make minor changes to gain access to the operating system. With MS-DOS, the programmer will have to make minor changes to handle the extra features of the MS-DOS file control blocks.

Next month, we will make further comparisons between MS-DOS and CP/M-86. We will include some benchmarks made with the Compu­pro 8085/8088 dual-processor S-100 system. We will report not only the results of running programs under both CP/M-86 and MS-DOS on the same 8088 in the same machine, but also the results of running the same programs under MS-DOS running Emulator-86 on the same 8088 in the same machine. Although that may sound more like a cat chasing its own tail than a test of operating systems, we will try to keep it all straight.