HYBRID TOOL FOR UNIVERSAL MICROPROCESSOR DEVELOPMENT

A single-board universal interface attached to a personal computer can act as an intelligent ROM emulator for software upgrading of an existing product, or it can function with CPU personality cards as a development system for a new product design

David McCracken **Thera Institute, Aptos, California**

Microprocessor development tools presently available suffer substantial drawbacks. One-processor, dedicated, single-board products allow inexpensive development of a simple product but offer limited program diagnostic capabilities because of their restricted input/output facility. Also, they provide no means of benchmarking different processors in a given application. More flexible and powerful emulators can be used to benchmark different processors but are correspondingly more expensive. Both approaches restrict engineers to use of only those microprocessors for which development systems are manufactured.

A hybrid approach interfaces the microprocessor under development to a low cost, interactive, personal computer to afford a complete development environment with all the features of an emulator at the approximate cost of a dedicated single-board system. Since all of the hardware and most of the software are microprocessor independent, the hybrid may be adapted for use with virtually any microprocessor. It is flexible enough to be used with both 8- and 16-bit microprocessors.

Integration of a computer and a microprocessor requires a means of controlling the microprocessor without interfering with its application operating system. In addition, the implementation must use hardware and software that are easily adapted to any microprocessor, including, hopefully, future products. The described integrated approach developed hardware and software simultaneously for the first half of the design cycle, enabling tradoffs that were crucial in making the interface transparent to the microprocessor. Once the transparency problem was solved, the remaining goal of hardware and software adaptability was achievable.

Communication Channel

Under control of the host computer, a Superboard II, the interface serves as a sophisticated, writable program store for the microprocessor. The interface also provides a bidirectional communications channel between the microprocessor, which is completely unaware of its presence, and the host computer. In many cases, the interface can be attached to the microprocessor by means of a simple ribbon cable connection to a read only memory (ROM) socket. Alternately, a central processing unit (CPU) personality card, or satellite, including data memory if needed, plugs into the interface to provide a development environment for new applications (Fig 1).

The host computer writes application programs into shared memory, called program memory, for subsequent execution by the microprocessor. It also sets the microprocessor's particular state by initializing registers and flags to selected values. Finally, the microprocessor reports its state at various times, such as at breakpoints, and supplies information for video displays generated by the host computer.

Two communication channels are offered by the interface: a program channel to access shared program memory and handshake channel to pass control information to the microprocessor and return status information to the host computer via shared handshake memory. Under host computer control, the interface multiplexes 16 data lines, 10 address lines, and 8 control lines, allowing either the host computer or the microprocessor to access the shared program and handshake memory banks (Fig 2).

An operating system is necessary for the microprocessor to communicate with the interface; therefore, it must be programmed to interact in a development environment. This is a common requirement in microprocessor development tools, many of which dedicate certain addresses to a development communication channel and almost invariably require use of restart routines during development, making it difficult to test new application software fully until the hardware is built and the memory is programmed. Of course, the application restart routine cannot coexist with the development restart routine because both must reside at the same address. The entire address mapping is similarly restricted until the final hardware has been constructed. By then, it is difficult to modify microprocessor system software because the development system cannot coexist with the application system.

These problems are solved by overlapping development (handshake) memory and application (program) memory within the microprocessor address space. Now, the microprocessor can have two restart routines: an application routine in the program bank and a development routine at the same address in the handshake memory. Program and handshake memory distinctions remain transparent to the microprocessor, which need

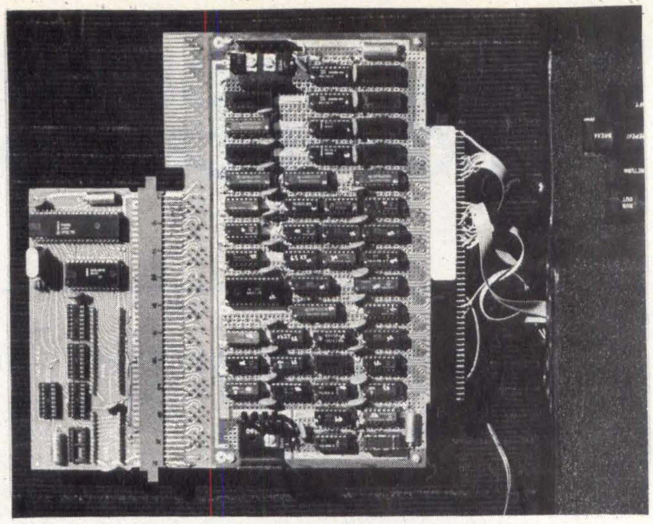


Fig 1 Universal interface. Simple 8035 satellite plugs into edge connector at lower left. Ribbon cable at right attaches to host computer. Edge connector at upper left allows expansion of multiplexed program RAM. Satellite may be replaced by ribbon cable connection to application system ROM socket

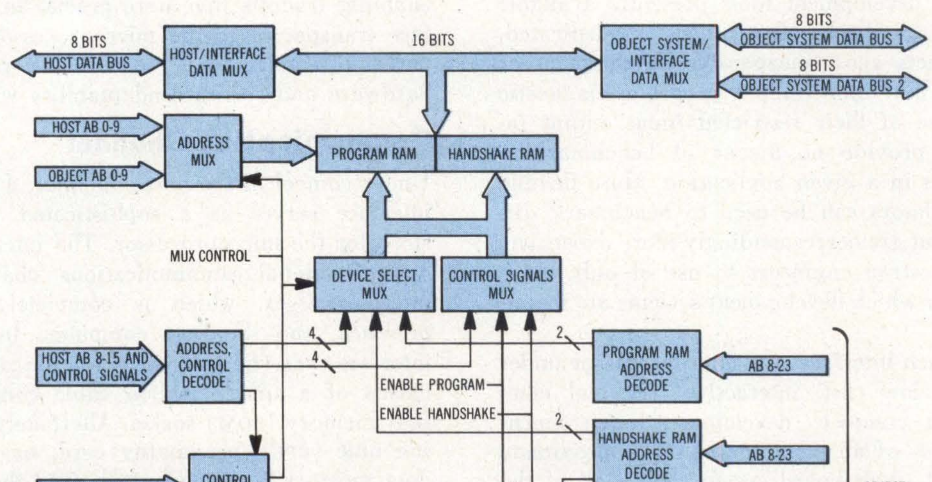


Fig 2 Universal interface block diagram. Multiplexers give either object microprocessor or host computer access to shared memory. Control register determines which of these gains memory access. Memory bank switch latch is key to microprocessor transparent operating system

not be programmed to select an appropriate restart routine. In practice, the two restart routines must be invoked in strict alternating sequence, beginning with the development restart routine that prepares the microprocessor to run a program—either the application restart routine or the continuation of some other routine previously interrupted, perhaps by a breakpoint.

The host computer initially sets a memory bank switch (Fig 2), that directs all microprocessor instruction fetches to handshake memory. When the microprocessor reset pin is released by the host computer, it is the development routine generated by the host computer in handshake memory that executes on the microprocessor. The last instruction of the development restart routine resides at a specific address (F7), which is decoded by hardware when fetched by the microprocessor and is used to trigger the memory bank switch. Subsequent instruction fetches now reference program memory, and the memory bank switch can be restored only by the host computer. As shown in Fig 3, memory bank switching occurs only on the trailing edge of the read signal. This avoids a race condition between the switch and the microprocessor memory access.

A mechanism for transferring breakpoint information from the microprocessor to the host computer completes the transparent communication requirements. Breakpoint—the interruption of program execution at an arbitrary, preselected location—ranks among the most valuable program development aids. Necessary data include extensive microprocessor status information used by the host to generate a video display, along with information required to resume microprocessor program execution. Breakpoints can be implemented in hardware or software. A hardware solution compares the current instruction address with the breakpoint address and switches microprocessor instruction fetches to the handshake memory, when these match, to execute a breakpoint routine. This approach allows breakpoints to be placed anywhere in the application program. A software solution, in which the breakpoint entry routine replaces a block of instructions beginning at the breakpoint address, prevents use of breakpoints within about 15 locations of the end of actual memory space. Despite this restriction, a software implementation was adopted to dispense with about ten extra integrated circuits (ICs) required for a hardware solution.

The host computer loads the breakpoint routine at the specified address in program memory. When the microprocessor executes the breakpoint routine, it records microprocessor status information in handshake memory. Interface hardware decodes control signals and directs all microprocessor write accesses to handshake memory, so that neither breakpoint nor unplanned program writes can modify program memory.

Interface Implementation

The interface can be adapted to virtually any microprocessor having external program memory. Individual microprocessor timing and control requirements must be considered during implementation. This is particularly true for 16-bit microprocessors such as the Z8000 with 24 address lines, using A0 to distinguish between high and low order bytes, or the 8086 with 20 address lines, using A0 to select low order bytes and an additional signal, BHE, to select high order bytes.[1,2]

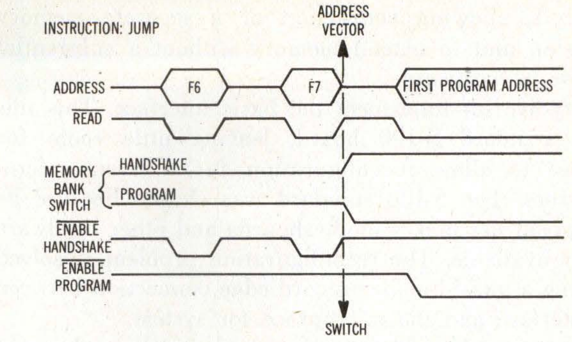


Fig 3 Memory bank switching. Microprocessor access to last restart instruction (F7, program starting address vector) initiates switching from handshake memory to program memory. Switching occurs on trailing edge of READ to avoid race condition

Still, microprocessor similarities outweigh their differences, and a nearly universal communication protocol requires the interface to make only minor changes to the microprocessor control signals. For example, the protocol assumes that read and write are low true signals, invariably generated when the address and data are valid. If a microprocessor lacks the read signal, as with the 6502, read is generated automatically. If the microprocessor system uses high true read and write signals as on the S-100 bus, these are inverted by the interface.

OR and NAND gates allow further manipulation of control signals such as MEMRQ in the Z80, and the 8035 PSEN program store enable used to read program memory.[3,4] Microprocessor system address lines in Fig 2 are decoded by two different hardware subsystems during device selection to allow mapping of program memory anywhere in the microprocessor address space.

The host computer loads the high order address into program memory address decode circuitry registers under operator direction. These are compared with microprocessor address lines to select the appropriate memory bank. Meanwhile, handshake memory decode circuitry selects handshake memory whenever the high order address is 000XXX or FFFXXX, chosen to match the microprocessor restart location. Again, these addresses may overlap because microprocessor control signals combine with memory bank switch status to arbitrate the actual memory selection. Both memory banks are 16 bits wide; the program bank is 1k bytes by 16 bits in size, and the handshake bank is 256 bytes by 16 bits. The program bank can be treated as 2k by 8-bit memory for 8-bit microprocessors.

The host, assumed to be an 8-bit computer, addresses all memory and control registers as sequential memory locations within a 4k-byte block. Pertinent control signals and all multiplexed lines to the memory array are brought to the edge connector in the upper left corner of Fig 1, allowing connection of a separate memory expansion unit to extend memory without a substantial increase in hardware.

Forty-five ICs implement the basic interface. This fills out a standard S-100 board, leaving little room for switches to allow reconfiguration for different microprocessors. The S-100 standard was chosen because its widespread use makes motherboards and other hardware readily available. The reconfiguration problem is solved by using a quasi-intelligent card edge connection between the interface and the microprocessor system.

Interface address, data, and control signals are brought to an edge connector at the lower left corner in Fig 1. Here, a microprocessor satellite attaches directly to that edge connector. Alternately, a ribbon cable would be installed between the edge connector and a microprocessor application system ROM socket, allowing the interface to function as a ROM emulator. The plug-in board (or ribbon cable) is microprocessor specific and completes all connections necessary for reconfiguration. Because the simple 8035 satellite attaches directly to the edge connector in Fig 1, this typical configuration requires only one additional cable connection to the host computer.

Hardware and Software Flexibility

The 6502 CPU was chosen as host because of its widespread use in inexpensive personal computers and its associated ease of programming. The Superboard II was selected for its low price and easy expansion to accommodate the interface. This single-board computer includes a built-in, programmable, full-size keyboard and a 25 by 25 element television display interface for operator interaction, as well as a cassette tape interface for storage of both the development operating system and the application software. The 8035 object microprocessor implementation demonstrates the flexibility of the design because, since the 8035 is a single-chip microcomputer with its own memory, input/output (I/O), and unusual protocol for external circuit communication, the task of fitting it to a universal standard appears difficult. Other existing implementations support the 6502 and 8086 microprocessors.

Flexibility was the primary software development goal, with efficiency secondary. One of the most powerful aspects of the device is provision of a development environment for virtually any microprocessor. To achieve this, adaptation to different microprocessors must be as easy in software as in hardware. Flexibility has been achieved in that 65% of the software is totally object microprocessor transparent. The remaining 35% was kept microprocessor specific, because of gross inefficiencies required to generalize the routines; however, once the software functions were well defined by the first implementation, the problem of generating equivalent software for different object microprocessors became less difficult.

Rigid adherence to rules of structuring allows software flexibility. There are no program jumps, even when a subroutine call requires additional instructions. Subroutines are used in preference to straight-line programming wherever there is no associated memory penalty. All parameters are passed between subroutines through page zero, making the index registers, accumulator, and stack fully available for most flexible use within the subroutines.[5] These characteristics simplify coding long sequences of subroutine calls and allow future software to utilize all existing subroutines.

All tables, which are largely object microprocessor dependent, reside in a dedicated memory block isolated from the software routines. Table expansion and alteration therefore occur without program relocation. Table manipulation was identified as the area in which to concentrate programming effort because it provides both flexibility and efficiency.

Design Complications

Two unusual aspects of the interface complicate the software design more than a cursory inspection would indicate. One problem develops because shared memory is accessed by different addressing schemes in the host computer and the object microprocessor. This results from the operator's need to reference program memory from the object microprocessor viewpoint, while the host computer employs a different addressing convention.

Memory addressing is further complicated because the interface itself can be mapped into any 4k segment within host computer address space. Furthermore, the operator can change the location of program memory, as accessed by the microprocessor, to allow development of application software anywhere in microprocessor address space. While handshake memory addresses change when the interface is remapped, they do not change when program memory is remapped; in fact, handshake memory addressing must never change in the object microprocessor system, because handshake memory provides the development restart.

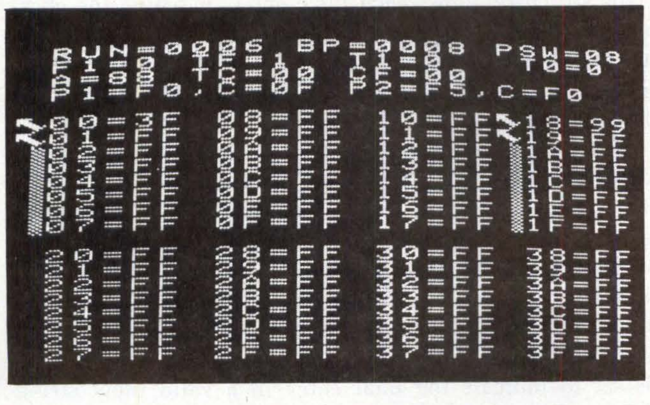
Dual memory addressing constraints define a scheme of virtual addresses and physical addresses, either of which can be translated to the other by an algorithm. The operator employs only the virtual addresses used by the object microprocessor. If the operator relocates program memory within microprocessor address space, the virtual mapping changes while handshake memory remains fixed at the physical restart location. The host computer uses physical addressing to access both the program and the handshake memory banks. It converts the operator specified virtual addresses to physical addresses by subtracting the lowest program memory bank virtual location, adding the interface lowest physical address, and checking to guarantee that the result falls within program memory physical space. If the result is not a program memory physical address, it must reference an object microprocessor memory location not shared by the host computer. Since the host cannot manipulate microprocessor memory that is not shared, it passes the virtual address to the microprocessor, along with a communication program, via handshake memory.

For example, suppose the interface is mapped to host physical addresses 9000 through 93FF. If program memory begins at virtual address 1000 and an address reference designates virtual address 1200, the virtual address is converted to a physical address by subtracting the lowest program memory virtual address and adding the interface lowest address: 1200 − 1000 + 9000 = 9200. In contrast, virtual address 1600 corresponds to host physical address 1600 − 1000 + 9000 = 9600, an unimplemented program memory bank address; therefore, an address reference of 1600 must designate the object microprocessor address of a memory location not shared by the host.

The second unusual problem that complicated software development is that handshake routines must be written simultaneously in both host computer assembly language and object microprocessor assembly language, because of tradeoffs required to achieve program efficiency. This problem was most evident when writing the 8035 restore status and continue routine, which directs the host computer to assemble a program in handshake memory for the object microprocessor to execute. Unless the operator designates status changes, the program in 8035 assembly language restores the 8035 to its exact condition at the location where a breakpoint was encountered and then jumps to the required application program address.

The major tradeoff consideration is whether the object microprocessor should pick up its status information through a table access loop or through a straight-line routine. A table access loop, the obvious choice in most situations, requires less memory. In this case, however, a high efficiency table access loop cannot access memory outside the 8035 internal random access memory (RAM) because the additional status words are directed to unique destinations such as the 8035 timer, program status word (PSW), and I/O port.[6]

Another consideration is the 8035's unusual addressing technique to retrieve data from its program memory, which it generally considers to be only a source of instructions. The MOVPA, @A instruction concatenates the accumulator content with the current program page to address program memory data; the data are retrieved to the accumulator where they overwrite the original address.[7] This architecture precludes forming efficient loops to pick up status information. A standard alternative would be straight-line programming, where each word is loaded directly into the desired location. For example, MOV R0,#FF in only two bytes restores register 0 to the designated value.



```
RUN=0006, BP=0008  PSW=08
F1=0  TF=1  T1=0  T0=0
A=08  TC=00  CF=00
P1=F0,C=0F  P2=F5,C=F0

00=3F  08=FF  10=FF  18=99
01=FF  09=FF  11=FF  19=FF
02=FF  0A=FF  12=FF  1A=FF
03=FF  0B=FF  13=FF  1B=FF
04=FF  0C=FF  14=FF  1C=FF
05=FF  0D=FF  15=FF  1D=FF
06=FF  0E=FF  16=FF  1E=FF
07=FF  0F=FF  17=FF  1F=FF

20=FF  28=FF  30=FF  38=FF
21=FF  29=FF  31=FF  39=FF
22=FF  2A=FF  32=FF  3A=FF
23=FF  2B=FF  33=FF  3B=FF
24=FF  2C=FF  34=FF  3C=FF
25=FF  2D=FF  35=FF  3D=FF
26=FF  2E=FF  36=FF  3E=FF
27=FF  2F=FF  37=FF  3F=FF
```

Fig 4 Breakpoint status display. Standard television set displays extensive 8035 status information including full 64 bytes of internal RAM

Thought must be given to how the host 6502 computer would set up such a program for the object 8035 to execute. Because the 6502 is unaware of 8035 instructions, the program must be moved as a data string into the proper location, using a table and loop for efficiency. If a straight-line approach were used on the 8035, the 6502 would have to determine where to insert status information into the 8035 program it handles as an arbitrary data string. This is possible, but very inefficient. Instead, the host computer loads a complete, straight-line, 8035 restore program, which specifies data sources through a table that is modified by the host computer using other, unrelated subroutines.

Table Structure

Development operating systems for different microprocessors require equivalent software routines such as the program/display routine, which allows the operator to examine and modify object programs, and additional routines used to insert breakpoints, run programs, continue execution, single step through a program, display breakpoint status, modify status at breakpoints, disassemble program code, and transfer application software to or from tape. There are 52 such subroutines in the entire program.

To reduce future programming effort, adaptability of the routines for use with any object microprocessor was an important goal. This was achieved by writing microprocessor independent routines that are driven by microprocessor dependent tables. Extension to additional microprocessors by changing predefined table entries is much easier than changing software routines, especially when each routine interacts with many others. This is reflected by the final result, in which 65% of the executable code and only 4% of the table code are microprocessor independent.

Status Display

A breakpoint display routine does not appear to be easily generalized. Every microprocessor has a unique internal architecture. The 8035 is a good example, because as a single-chip computer it maintains considerably more internal information than other microprocessors, as shown in Fig 4. The 8035 breakpoint display routine must show a standard, internal status form, filling in the blanks with information passed by the object microprocessor via handshake memory when the breakpoint is encountered. The difficulty is that while the blank form can originate in ROM, the fill-in information cannot.

One solution is to devise a table containing both the American Standard Code for Information Interchange (ASCII) characters to be displayed and the addresses of the sources of data to fill in the blanks. This table also contains control characters that use character codes not defined in the limited ASCII character set. Table extracts in Fig 5 show the four ASCII table entries representing PSW. The next characters displayed would be the breakpoint value of the program status word. The display routine knows that codes directing it to a data location will follow any "=" character. A packed control word as the next entry designates the number of display digits and indicates whether the source is the host computer zero page or the handshake memory. The following entry is the low byte address.

The display routine fetches the data, converts to ASCII format, and displays the required digits. Because the number of digits is specified, the same routine can handle 16-bit addresses, 8-bit data, and 1-bit flags. The multiple space character, followed by the number of spaces required, saves table space when long, empty fields are required in the display. Use of the end of transmission (EOT) character makes table driven software independent of table entry length.

An addressing limitation of the host 6502 computer is most noticeable in this situation because the remappable interface must be addressed indirectly. The 6502 offers two forms of indirect addressing: indexed indirect with the X register as a pre-index, and indirect indexed with the Y register as a post-index.[8] The post-indexed form addresses breakpoint information in handshake memory, and should also be used to show status information if a full screen is to be displayed.

Competition between these applications for the Y register would create substantial problems, but this was avoided by displaying only eight video lines comprising 256 entries in the video memory. This allows use of an absolute address indexed by X for the video, reserving indirect addressing indexed by Y for exclusive use in handshake memory access. Eight lines suffice to display the internal state of any general purpose microprocessor. While the 8035 status display uses a full screen, most of this information is an orderly internal RAM array loaded into video memory by a relatively simple program loop.

Command Parsing

The next table handling example demonstrates the use of table lookup to generalize a sequential keystroke parsing routine employed by the breakpoint processor change status utility. With this debugging tool the operator can change any object microprocessor internal status information before resuming program execution. The procedure is to examine each keyboard entry until a specified register or other microprocessor resource has been entered correctly and identified, then to use subsequent keys as new data to be stored in the register. The program will actually modify a table in handshake memory. Host computer page zero may also change, in which case another routine uses the page zero information to modify a table in handshake memory because the object microprocessor cannot access host computer page zero. Then, as previously described, the restore status and continue routine sets up the object microprocessor to retrieve table data for restoring status before continuing program execution.

The unique microprocessor architecture and nomenclature rule out keyboard parsing with prior knowledge of the actual keys to be entered or the number of entries required to identify a resource, particularly when abbreviations are permitted to reduce operator effort and program length. Handshake memory or page zero table destination corresponding to the resource is also unknown. All of this information must be stored in a microprocessor specific table.

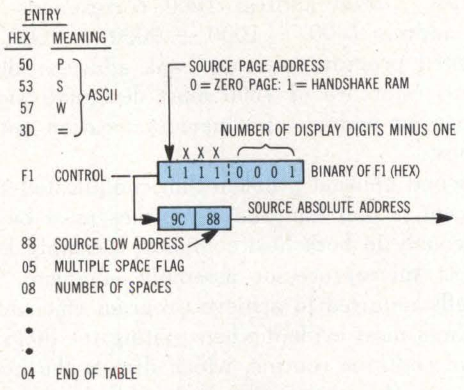


Fig 5 Breakpoint display tab!e entry format. General purpose routine driven by table entries displays status of virtually any microprocessor. ASCII "=" preceeds reference to data outside table. Subsequent entries serve to locate, adjust, and format data

For program efficiency, keyboard parsing is synchronous with operator input so that data need not be stored and then deciphered. Fig 6(a) shows the table divided into numbered key fields for the first key entered, the second key, and so on. These fields are further divided into subfields, each of which contains all possible characters that might follow a particular character in a valid input string.

Each table entry comprises three bytes. First is an ASCII character to be compared with the input keystroke. This is followed by two control bytes. Generally, the first control byte directs the parsing routine to the starting address of the next sequential key subfield, linking the entries for all keys that could follow to produce valid input. The second control byte supplies the subfield length used to terminate searching for a key match. If no match is found, the parsing routine waits for another key entry and searches the subfield again with the new character. This structure allows each table entry matching the character entered by the operator to direct the parsing routine to the next set of table entries, so that much of the parsing operation is controlled by the table rather than the parsing routine.

Because the routine cannot know the required number of entries in advance, the table must also terminate parsing. The second control byte, which is the third byte of each complete table entry, normally supplies the number of entries in the following subfield. Only a few bits of this word are needed as there are never many characters that could follow a particular character in a valid input string. By convention, bit 7 of the second control byte is set to indicate the final entry in a valid input string [Fig 6(b)]. In the same way, bit 6 is used as a special control flag, and bit 0 distinguishes between host page zero locations and handshake memory locations.

Disassembly

Disassembly is achieved by searching a table to find a match for each instruction operation code, in sequence, skipping over the operands. A match directs display of a character string corresponding to the instruction mnemonic. Both the instruction operation code and the character string mnemonic are supplied by a single table. Each entry consists of a hexadecimal operation code and its associated ASCII mnemonic.

Microprocessor instruction mnemonics may vary in length. Mnemonics for the 8035 range in fact from two to nine characters (Fig 7); therefore, a control word in each table entry supplies the length of each mnemonic. Fig 8 shows the table structure, beginning with the control word that addresses the operation code. Since this requires no more than four bits for mnemonics up to 16 characters long, the unused high order bits afford additional table control. Bit 7 is set to signal a decrease in operation code length, relative to the previous entry, while bit 6 is set to indicate an increase in mnemonic length. Organizing table entries in decreasing order by operation code length and in increasing order by mnemonic length permits the search routine to keep track of these parameters with very little overhead.

Many character strings are used in more than one mnemonic, such as MOV, which appears in 17 different 8035 instructions.[9] A multicharacter table entry is addressed by a special character, with bit 7 set, and the remaining bits supplying an offset to the address of the character string. Use of a multicharacter table conserves memory by handling frequently needed strings as an extension of the single character processing.

This disassembly table structure appears efficient but overlooks an important characteristic of all microprocessor instruction sets—the property of operation code regularity. The table structure treats every operation code as an isolated entity when, in reality, instruction sets tend to use a sequence of consecutive operation codes to perform a sequence of related operations. For example, the 8035 uses F8 through FF, A8 through AF, 28 through 2F, and nine similar operation code sequences for related operations that manipulate registers R0 through R7, with the low order three bits of the operation code designating the particular register.[10] To save memory, operation codes for this class of instructions, branch instructions, and I/O instructions are presorted by an 8035-specific routine before a general table search is attempted. While the operation code presort requires more than 20% of the microprocessor specific programming, the resulting economies are worth the investment.

Summary

Built-in flexibility makes the universal interface a valuable engineering tool at every stage of product development. Typically, the design of a microprocessor based product begins with a problem description. This should lead to a list of microprocessor requirements such as word length, controller or data manipulator orientation, single chip or bus orientation, and so on. These requirements, in turn, form the basis for a list of potential microprocessors. At this point, the interface can be used with the desired satellites, which are inexpensive because they are small printed circuits containing the central processor and data memory, to benchmark the microprocessor in operations characteristic of the application.

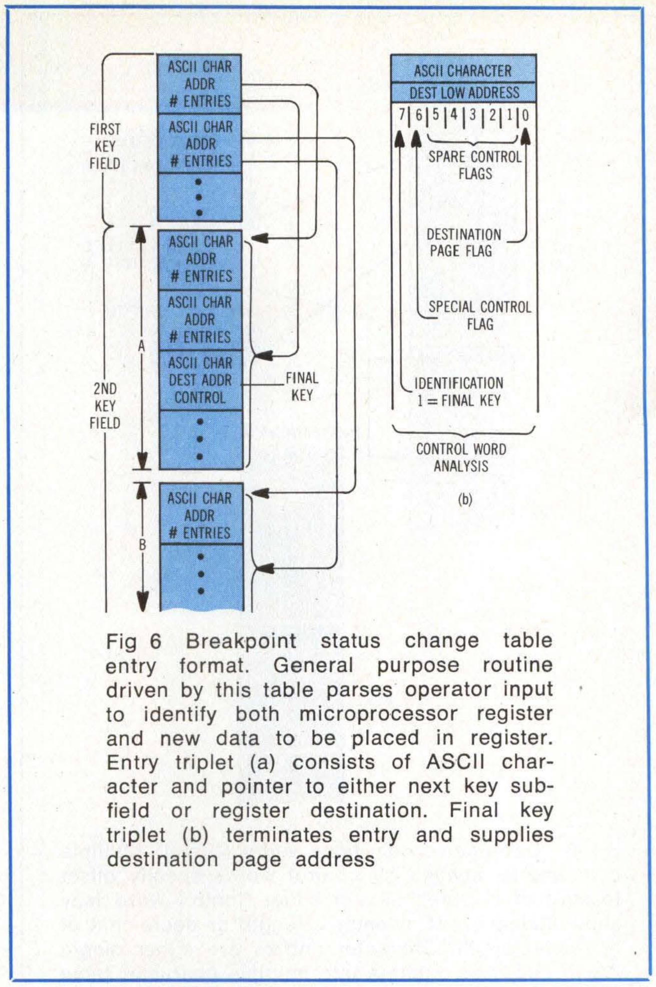
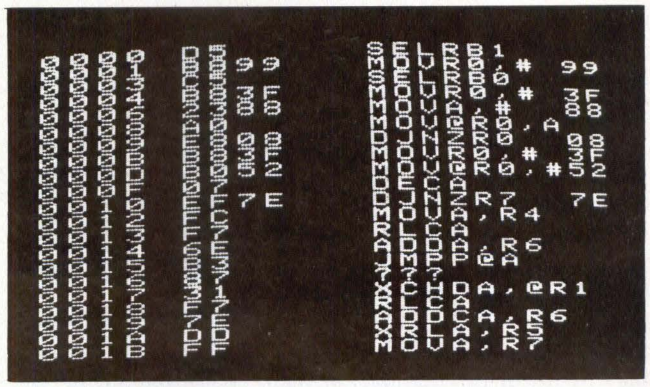


Fig 6 Breakpoint status change table entry format. General purpose routine driven by this table parses operator input to identify both microprocessor register and new data to be placed in register. Entry triplet (a) consists of ASCII character and pointer to either next key subfield or register destination. Final key triplet (b) terminates entry and supplies destination page address



```
0000 D5    SEL RB1
0001 B8 99 MOV R0,#99
0003 C5    SEL RB0
0004 B8 3F MOV R0,#3F
0006 23 88 MOV A,#88
0008 A0    MOV @R0,A
0009 B8 08 MOV R0,#08
000B B9 3F MOV R1,#3F
000D BA 52 MOV R2,#52
000F [illegible]
0010 EF 7E DJNZ R7,7E
0012 FC    MOV A,R4
0013 [illegible]
0014 6E    ADD A,R6
0015 B3    JMPP @A
0016 [illegible] ???
0017 21    XCH A,@R1
0018 [illegible]
0019 DE    XRL A,R6
001A 2D    XCH A,R5
001B FF    MOV A,R7
```

Fig 7 Program disassembly display segment. General purpose routine generates two-to-nine character mnemonic plus operands from machine instructions in program memory. Typical 8035 instructions are shown, but virtually any microprocessor instruction set is handled. Disassembly is a powerful aid to locating both software errors and program entry errors