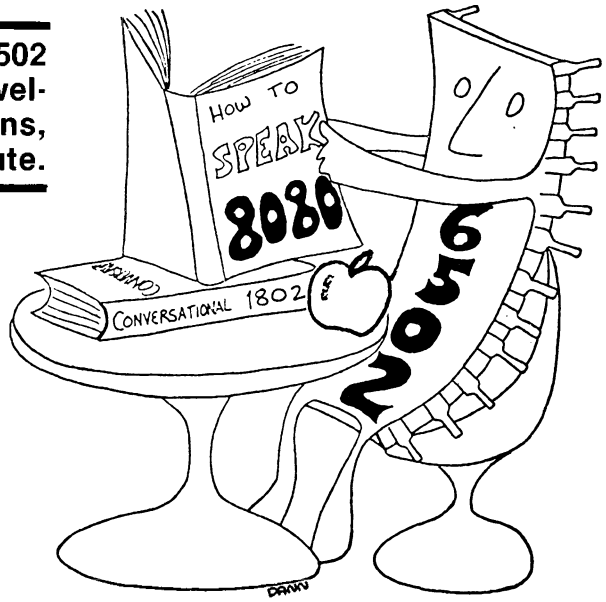
8080 Simulation with a 6502

The design for an 8080 simulator running on the 6502 illustrates how your micro can assist program development for other machines, master new applications, and double the quantity of software it will execute.



Dann McCreary
Box 16435-M
San Diego, CA 92116

Why Bother to Simulate?

While many advantages of simulating one microprocessor with another might be cited, there are several which I believe stand out above the rest.

Educators and students can use simulation software as an enhancement to introductory courses in microprocessing. Such courses often make use of single board microcomputers like the Commodore KIM-1. These computers provide invaluable hands-on experience. The addition of simulation software can multiply their effectiveness by enabling the study of alternate architectures and instruction sets without the expense of purchasing more hardware.

The entrepreneur and the hobbyist are typically owners of systems based on a single type of processor. Should a situation arise in which they would like to develop software for some other processor, they are faced with another significant capital investment. The availability of simulation packages which can run on their present hardware can make it economical to design and debug code for other processors.

Applications software fulfilling particular functions is sometimes hard to come by. Some might claim that the availability of a given application varies inversely with the need for a version written for the microprocessor available to run it on. Enter simulation software and your choice of applications can be easily doubled. One good example might be the use of an inexpensive 8080 assembler for a one-time task rather than going to the time or expense of producing a cross-assembler.

The experimenter, never quite satisfied with the status quo, can use simulation techniques to try out his theories about an optimized instruction set. He can, in software, model the processor of his design and by doing so he can gather actual data about the validity of his ideas.

The major and most obvious drawback to simulating one microprocessor with another is the large speed penalty. In the Cosmac 1802 Simulator which I have implemented on the 6502, about fifty 6502 instructions are executed in the course of executing one 1802 instruction. In my 8080 Simulator, twice as many or more are required for each 8080 instruction executed. High speed real-time code or applications requiring precise timing relationships derived from instruction cycle timing are clearly outside the scope of this technique.

A somewhat lesser problem involved is the space occupied by the simulator program, which must be co-resident in memory with the application program. Careful design here can make the simulator quite compact but it does take up a finite amount of space.

For a majority of applications, I feel that the advantages of using a simulator overshadow the drawbacks, making this type of modeling very worthwhile.

Optimizing the Approach

A simulation of sorts could be accomplished by compiling or translating the code of an 8080 into 6502 code. This approach would in fact be advantageous from an execution speed standpoint and would be a good choice if running application software were the only consideration. It would, however, generate large amounts of code and would not meet some of the other objectives I had for an 8080 simulator.

The interpretive approach seemed to best fulfill my self-imposed requirements. It would provide an accurate model of the 8080 processor, complete with all internal registers and duplicating all 8080 instructions. It would allow for single stepping or tracing through an 8080 program invaluable for debugging and for educational purposes. An interpreter could be very code-efficient, not only using little memory itself but also allowing 8080 object code to run unmodified in a 6502 environment.

I could have taken a "brute force" approach to interpretation, using perhaps

a table lookup scheme and transfering to a separate routine for each 8080 op-code. This offered some advantages in simplicity and execution speed but it required far more memory than I cared to use.

A careful analysis of the 8080 instruction set suggested that the 256 table entries and routines required by a "brute force" technique could be reduced by 25 by grouping the 8080 op-codes into categories sharing common functions.

In addition, certain judicious tradeoffs could be made between simplicity and ideal features, taking best advantage of the addressing modes and features of the 6502. For instance, the 6502 stack resides in page one and many of the 6502's instructions and addressing modes make use of page zero. To avoid memory use conflicts it would have been nice to simulate 8080 memory starting at 0200 HEX, making that address equivalent internally to 0000 HEX. This would have required a great deal of overhead in the form of a special monitor to show addresses minus the 200 HEX offset.

The addresses being used by the 8080 program while running would have to be converted dynamically, and in order to use indirect addressing a special set of simulated registers would have to be maintained in page zero. Besides requiring much more code, this would slow execution speed down considerably. I decided instead to simply require the user to patch around the small areas in page zero and page one being used by the simulator.

Final Design Overview

Laying out the 8080 instruction set graphically on a hexadecimal grid, as illustrated, reveals some interesting features. Four major divisions are apparent, neatly dividing the instruction set into quadrants. The second quadrant is composed alomst entirely of MOV instruction op-codes. This MOV group most clearly illustrates the way that 8080 op-codes break down into source and destination fields, and suggests the best way to organize simulated 8080 registers in memory.

With simulated registers arranged properly in memory, source and destination field data can be extracted from the op-code and used as indexes to the registers involved. In every case where instructions act upon individual registers their order, as dtermined by this source/destination indexing scheme is B.C,D,E,H,L,M,A—where M is not an actual register but rather the content of the mrmory location pointed to by the HL register pair.

This order suggests a general method for accessing individual registers with some slight exceptional logic for the M "pseudo-register". By inverting the source and destination indexes and reversing the order of the 8080 registers in memory it becomes possible to use the HL register pair directly as an indirect pointer to memory. Adding the Stack Pointer and Program Counter to the register array in the same reversed order completes the simulated register set.

Looking again at the instruction set grid it can be seen that a symmetry exists based on the source field of the op-code. For instance, all INR instructions have source fields containing 04 HEX while all DCR instructions have 05 HEX as their source field. The fourth quadrant exhibits similar symmetry. The third quadrant is more logically defined by the destination field, but still divides into 8 groups of similar instructions as do the first and fourth quadrants. These, along with the entire MOV quadrant, total 25 groups of similar instructions. A major task, then, of the simulator mainline is to determine from the op-code which of the 25 groups it belongs in so tht control can be transfered to the proper routine to interpret it.

To keep the simulator as compact as possible it is advantageous to perform as many common operations as possible in the mainline. Fetching the op-code, extracting source and destination indexes from it and incrementing the Program Counter are fundamental. The mainline also fetches the content of memory pointed to by the HL register pair, clears a flag used by many simulator routines, saves data from the register pointed to by the destination index for later operations, tests for and handles interrupts and handles other "housekeeping" type functions.

At the end of the mainline the address of the selected interpreter routine is pushed onto the stack along with a preset status. A 6502 RTI instruction is executed, transferring control to the proper module entry.

It is the responsibility of each module to correctly interpret all op-codes which result in a call to that module. Each module is constructed as a subroutine, returning control to the mainline via an RTS. This also enables certain modules to be used as subroutines by other modules. A brief look at the modules and their support subroutines will help to illustrate their functions.

MOV. While encompassing the largest number of op-codes of any module, MOV has perhaps one of the simplest tasks. It merely takes the content of the register indicated by the source index and stores it in the register pointed to by the destination index. No condition flags in the PSW (Processor Status Word) are affected. The only slight complication whether the destination is memory, in which case the HL register pair is used as an indirect pointer to store the result in memory.

INX/DCX. This module must increment or decrement a selected register pair. The least significant bit of the destination index is tested to determine whether the instruction is an increment or a decrement instruction. The bit is then dropped and what remains is an index to the proper register pair—except for the cases of 33 HEX and 3B HEX when the Stack Pointer is the register pair of interest. In these cases, the proper index for the Stack Pointer is substituted.

With the proper index set, a call is made to INCDEC. INCDEC is a 16 bit adder designed to add two zero page 16 bit operands. With the 6502's X and Y indexes properly set at the entry to this support routine, the content of a double precision one (0001 HEX) or a double precision minus one (FFFF HEX) is added to the chosen register pair, performing the increment or decrement.

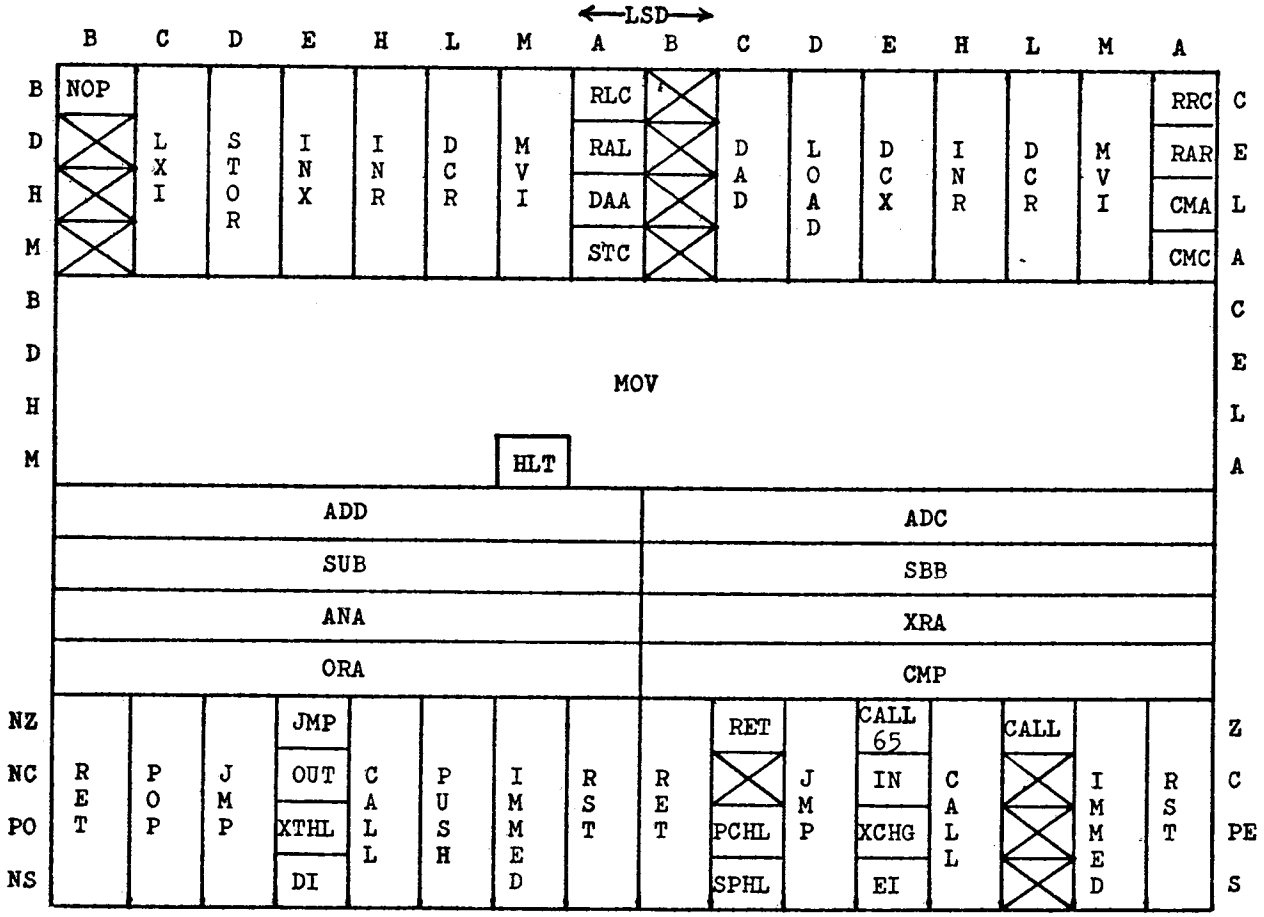
The proper register pair is selected in the same fashion for the DAD and LXI instructions also.

INR, DCR, MVI. These instructions are very consistent in their use of the destination index for determining which register (or memory as the case may be) they operate on. INR and DCR have the added complication of modifying the PSW condition bits, with the exception of the 8080 Carry.

Rotates. This is a mixture of quite different instructions lumped into one module. Proper execution depends on separating the Rotate instructions from the DAA, STC, CMC and CMA instructions, providing special logic for each and insuring the proper setting of PSW flags.

PUSH/POP. While handling register pairs somewhat like INX, DCX, DAD and LXI do, these instructions differ in substituting a register pair made up of the 8080 Accumulator and PSW for the Stack Pointer. The simulator handles this by looking for the special case and then decrementing the destination index to the proper position. The Stack Pointer is then incremented or decremented appropriately and the register pair data transferred to or from the stack as required.

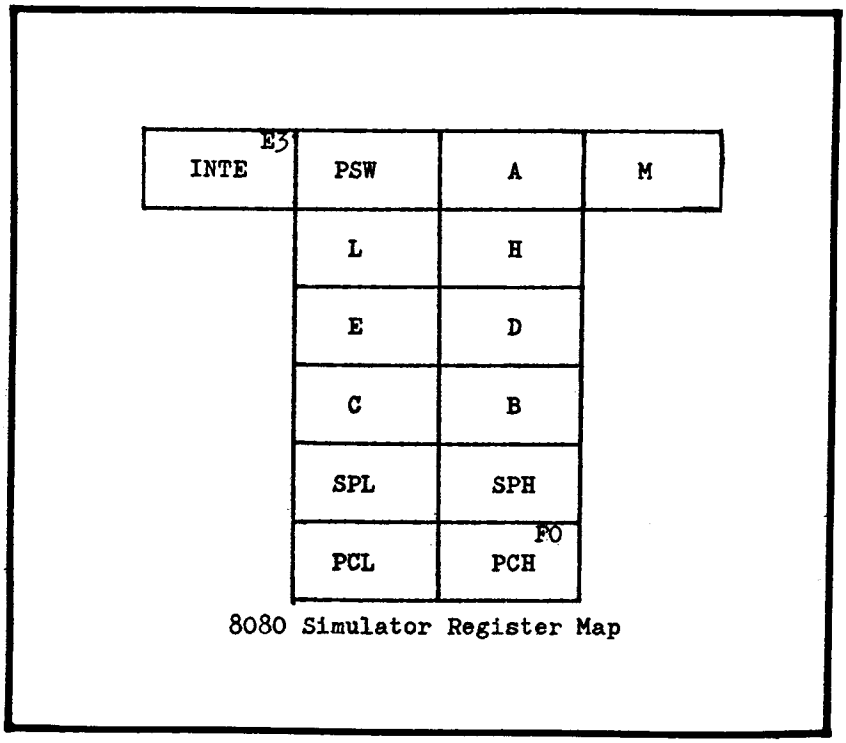
Several support routines come into play, including INCDEC and various routines for transferring the content of register pairs between each other and memory. An intermediate register pair (not illustrated) is utilized as a temporary storage location during the exchange of register pairs. I've labeled it simply "SCR", though I believe it bears an actual hardware analog in the 8080 in the form of a hidden register pair, temporary registers W and Z.



8080 Instruction Set Diagram
"X"s = Unimplemented

CALL and RETURN. These also manipulate the stack, using it as a storage location for the content of the Program Counter. The same set of support routines are used to get the transfer address from memory (for the CALL instruction) and to move data to and from the stack memory. RST is treated like a CALL instruction, except that the transfer address is computed from the destination field of the op-code rather than taken from memory. Conversely, JUMP gets its transfer address from memory, but does not save any return address on the stack.

Condition Codes. CALL, RET, and JMP all make use of a subroutine called CONDIT. CONDIT examines the destination index derived from the op-code and subdivides it into a condition index and a True/False indicator bit. The index is used to select a PSW bit mask from a table of masks. These masks align with the appropriate bit in the PSW. Based on the state of the selected PSW bit and the True/False indicator bit, CONDIT returns an indication of whether or not the JMP, CALL, or RET should take place.



8080 Simulator Register Map

Arithmetic and Logic. These instructions occupy the third quadrant of the instruction set. Rather than being grouped vertically by their source fields they are grouped horizontally by their destination fields. This is due to the fact that while they may have different sources of data, they all have one implied destination—the 8080 Accumulator. The CMP instruction is the only one of this group which does not place its results in the Accumulator. It merely discards the result, setting only the PSW flags accordingly. This is accomplished by forcing the destination index to point to a scratchpad location.

Probably one of the most difficult things to simulate successfully is the proper setting of the Processor Status Word. Different instruction groups affect different subsets of PSW flags but the Arithmetic and Logic group affect all the flags. Zero, Sign and Parity flags are always affected as a group. A routine called STATUS sets these three flags simultaneously when a result is passed to it. Carry and Auxilliary Carry are handled separately as they may be affected in isolation by some instructions and not affected at all by others.

Special Features. For the purpose of using the simulator as a debugging tool, I chose to trap unimplemented op-codes. When the 8080 Simulator determines that the current instruction is an illegal op-code it forces a jump to the system monitor. This can be used to advantage as a simple type of breakpoint. Alternately, a table of breakpoint addresses may be set up in memory. After each instruction, the 8080 Program Counter is compared to each address in the breakpoint table. If a match is found, a jump is forced to the system monitor. This makes it possible to step from breakpoint to breakpoint, seeing the result of groups of steps rather than only individual steps.

I/O Instructions. I/O is also handled via a table of addresses. Each entry in the table is the address of a port in the 6502 system. The entries in the table are associated with 8080 ports in sequential ascending order. Setting of the Data Direction Register, as in a 6530 PIO, is handled transparently to the user.

Call 65. I have "borrowed" one of the 8080's unimplemented op-codes for a special purpose function—calling 6502 subroutines from an 8080 program. This enables you to use existing system I/O routines and other utilities. All that is required is to add brief header and trailer routines to transfer the required parameters to and from simulator registers and 6502 registers used by the subroutine. The CALL 65 instruction may also be useful for handling time dependent code segments.

Summary

Modeling one microprocessor with another is a technique which provides many potential benefits. It has certain significant drawbacks, most notable of which is a large penalty in execution speed. These drawbacks, however, are not of paramount importance in a large number of applications in instructional, personal and experimental use.

Designing such a simulator involves tradeoffs between the complexity and quantity of the coding required for the task on one hand, and the features and execution speed of the final product on the other. I chose to minimize the quantity of code, emphasizing commonality of functions within the simulator.

Simulators for the 8080 and Cosmac 1802 microprocessors are available from the author in versions designed to run on the Commodore/MOS Technology KIM-1.

Thanks to Gary Davis for his generous support in the form of access to his 8080 system and his assistance in running comparison tests. μ