The 6502 Gets Microprogrammable Instructions

Dennette A Harrod
POB 9475
Rochester NY 14604

"Every programmer is part of a collective mind, and progress demands that he educate and be educated by others." So states H T Gordon in a letter published in the October 1977 issue of *Dr Dobbs Journal of Computer Calisthenics and Orthodontia.*

In this article I shall attempt to educate others by detailing a hardware approach to adding sixty-four user-defined instructions to the MOS Technology 6502 microprocessor. The 6502 device is used in the Apple II, PET, KIM-1, SYM-1, Rockwell, Ohio Scientific, and Atari microcomputers to name a few.

My own research concerning 6502 operation codes (ie: op codes) has closely paralleled the efforts of Dr Gordon (see his Technical Forum article "The XF and X7 Instructions of the MOS Technology 6502" December 1977 BYTE, page 72). Close investigation reveals that sixty-four of the unimplemented op codes can be detected by a simple circuit such as that shown in figure 1. Unimplemented op codes are any of the op codes with the two least significant bits set to 1.

About the Author

Dennette A Harrod is a systems programmer at Xerox Corp in the Computer Aided Drafting (CAD) Department.

A similar circuit, by C W Moser ("Add a Trap Vector for Unimplemented Op Codes" *Dr Dobbs Journal of Computer Calisthenics and Orthodontia* January 1979, volume 4, issue 1, pages 32 thru 34) can be used to detect *all* undefined op codes, but it requires a programmable read-only memory programmer and does not appear to be as cost-effective in its use of components. My circuit uses only three integrated circuits.

Simple Hardware Appendage

The circuit in figure 1 is deceptively simple. Its purpose is to cause the 6502 to receive an interrupt signal whenever it attempts to execute one of the sixty-four undefined op codes in which the right nybble (ie: 4-bit segment) has a hexadecimal value of 3, 7, B, or F. (The left nybble can have any value.) These values correspond to the situation where both of the two least significant bits are high.

The software interrupt-service routine then examines the instruction and jumps to a routine to perform the operation specified by that code. This facility enables the user to add instructions not available on the 6502 as supplied. With certain added instructions, 16-bit arithmetic and logical operations can be performed, or string comparison and move operations may be implemented. These added instructions are called *virtual operation codes*, or *v-codes*.

When the 6502 is in the op-code-fetch phase of the instruction cycle, the normally low SYNC line goes high. When the processor attempts to fetch an op code which has both of the two least significant bits set to 1, the three-input NAND gate (IC2a, 74LS10) output goes low. (Note: one of the NAND gates on the 74LS10 is wired as an inverter.) As a result, the data-bus transceiver (IC4, 74LS245) is disabled, and the 6502 never receives the op code. Instead, a buffer (IC3, 74LS244) is enabled, forcing all 0 values onto the data-bus lines by pulling them to ground potential.

The net effect is that the 6502 thinks it fetched a BRK (break) instruction (hexadecimal 00). The BRK instruction causes the microprocessor to go through an interrupt sequence under program control.

Three things happen when the 6502 executes a BRK instruction.

- The program counter (PC) is incremented by 2 and is pushed onto the stack (thus the processor treats BRK as a two-byte instruction).
- The BREAK bit (B) in the processor status word (PSW) is set to 1, and the PSW is pushed onto the stack.
- The 6502 transfers control to the

Number	Type	+5 V	GND
IC1	6502		
IC2	74LS10	14	7
IC3	74LS244	20	10
IC4	74LS245	20	10

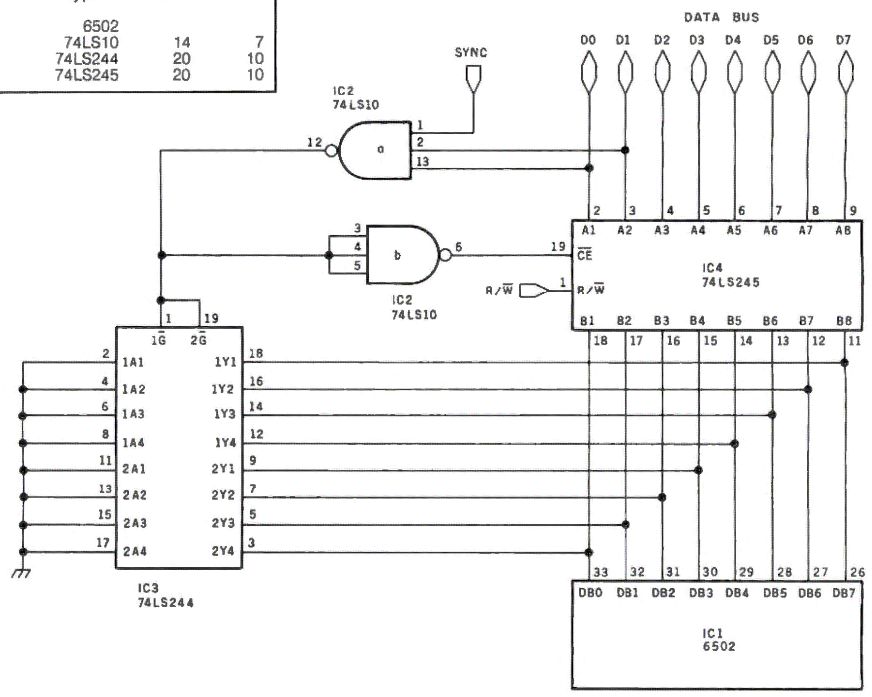

Figure 1: *Schematic diagram of a circuit that forces all lines on the data bus into a logic 0 (ie: low) condition whenever the 6502 attempts to fetch an operation code with the two least significant bits both set to 1. The low-order hexadecimal digit of such an op code will be 3, 7, B, or F. When all data bus lines are forced low, the 6502 executes the BRK (break) instruction, for which the op code is 00. The BRK instruction causes the processor to go through an interrupt sequence under program control.*

address stored in the highest locations in memory (FFFE and FFFF), the IRQ interrupt vector. This address must indicate the starting location of the interrupt-service routine.

Software Action

The first thing the interrupt-service routine must do is determine whether it was invoked by a hardware or a software interrupt. This is accomplished by examining the B bit in the processor status word. Having determined that it was a software interrupt (B=1), the program uses the stack pointer as an index to get the return address off of the stack. This is accomplished with the help of the TSX instruction to transfer the stack pointer to the X index register. Then it can load the accumulator using the indirect address mode and examine the actual instruction that caused the interrupt. Since the processor is not fetching the op code for *execution*, the circuit of figure 1 does not interfere.

Once the interrupt-service routine has established that it was not an actual BRK instruction that caused the interrupt, the v-code can be used as an index into a table of addresses of subroutines. There is one entry in the table for each of the v-codes. The subroutine performs the v-code operation.

By using the byte following the v-code as an additional instruction byte to be decoded by the v-code executing subroutine, each v-code can act as a gateway to 256 more virtual instructions.

Do not be intimidated by the prospect of over 16,000 user-defined instructions. Instead, welcome the ability to microprogram any or all of your favorite machine architectures into the virtual machine now available.

Instruction Characteristics

At this point I should probably clarify a poorly documented fact about the 6502 BRK instruction. One reason the processor treats BRK as a 2-byte instruction is that the BRK can be followed by a 1-byte code to be interpreted by the interrupt-service routine, much the same way that a supervisor call (SVC) instruction works in the IBM 360/370 computers. Using the circuit of figure 1 enables you to save one byte of code by eliminating the BRK instruction itself. This allows the interrupt-service routine and v-code subroutine to access 2 bytes of user data (the v-code and a data byte) without having to adjust the return address on the stack.

Taking a tip from Steve Wozniak (see "SWEET16: The 6502 Dream Machine" November 1977 BYTE, page 150), you can reserve 16 bytes on page 0 of memory to serve as eight 16-bit registers, each of which may be

used to contain either data or the address of data. The byte following the v-code can then be divided into two 4-bit nybbles to specify *source* and *destination* registers for the virtual operation. The low-order 3 bits of the nybble contain the register number (0 thru 7), while the high bit indicates direct or indirect mode. When the high-order bit has a value of 0, it means the register contains the *data*; a value of 1 means the register contains the *address* of the data.

If you wish, you can use 2 bytes for source and destination information, in which case 1 nybble of each byte can be used as an index-pointer (meaning it specifies which register to use as an index register). However, this requires the user to assume responsibility for fixing up the return address from the interrupt. On the other hand, if you use a real BRK instruction followed by a v-code and one or more data-bytes, the return address has to be manipulated anyway.

Hardware Interrupt Vectors

If you are truly ambitious, a circuit based on one by Yogesh M Gupta ("True Confessions: How I Relate to KIM" August 1976 BYTE, page 44) intended for the vectoring of hardware interrupts, which the 6502 lacks, can be modified and added to the circuit in figure 1 to provide hardware vectoring of the software interrupts. Gupta's circuit generates vectored addresses that are 4 bytes apart. This is a compromise. All you really need is 3 bytes to contain a JMP op code and a 16-bit destination address, but hardware-generated addresses are most conveniently generated in positive powers of 2.

I suggest that the addresses from such a device be 16 bytes apart, so that the service routines can be entered with three assumptions:

- The routine was called by a jump to subroutine (JSR) instruction, which is not strictly true, but this will be clarified in a moment.
- All of the 6502 registers are stored on the stack, directly beneath the return address, so that any register may be used with impunity.
- The routine can exit from anywhere by a return from subroutine (RTS) instruction, and all registers will be restored to the pre-interrupt state before a return from interrupt (RTI) instruction is executed.

Listing 1: *Example of an interrupt-service routine. It saves the contents of the 6502 registers on the stack, calls a subroutine (by JSR) to operate the interrupting device, restores the registers, and finally returns to the interrupted task. Some instruction mnemonics are from a macro-assembler of the author's own design. For example, the PSH X instruction mnemonic causes the macro-assembler to generate two machine instructions, TXA and PHA.*

```
        ABCD        SRVRTN    EQU     $ABCD

XX00                          ORG     $XX00
        XX00        ENTRY     EQU     .
XX00    48                    PSH     A          ;SAVE REGISTERS
XX01    8A 48                 PSH     X
XX03    98 48                 PSH     Y
XX05    20 CD AB              JSR     SRVRTN     ;CALL SERVICE ROUTINE
XX08    68 A8                 PUL     Y          ;RESTORE REGISTERS
XX0A    68 AA                 PUL     X
XX0C    68                    PUL     A
XX0D    40                    RTI                ;RETURN FROM INTERRUPT
        XX0E                  END     ENTRY
```

Listing 2: *An interrupt-service routine that uses the* soft-coded vector *technique of subroutine calling. To call a subroutine, this routine places a return address on the 6502 stack, and then branches to the subroutine by executing a JMP instruction in the indirect addressing mode. The subroutine can return normally to this calling routine by executing an RTS instruction. This procedure compensates for the inability of the 6502 processor to execute the JSR instruction using the indirect addressing mode.*

```
YY00                          ORG     $YY00
        YY00        SRVADR    EQU     .
        ABCD        SRVRTN    EQU     $ABCD
YY00    CD AB                 ADR     SRVRTN

XX00                          ORG     $XX00
        XX00        ENTRY     EQU     .
XX00    48                    PSH     A          ;SAVE REGISTERS
XX01    8A 48                 PSH     X
XX03    98 48                 PSH     Y
XX05    20 8B XX              JRS     JMPAT      ;SIMULATE JSR @ADDR
XX08    4C 00 ZZ              JMP     COMRTI     ;GOTO REGISTER RESTORE
        XX0B        JMPAT     EQU     .
XX0B    6C 00 YY              JMP     @SRVADR    ;GOTO SERVICE ROUTINE
        XX0E                  END     ENTRY

ZZ00                          ORG     $ZZ00
        ZZ00        COMRTI    EQU     .          ;COMMON RTI ROUTINE
ZZ00    68 A8                 PUL     Y          ;RESTORE REGISTERS
ZZ02    68 AA                 PUL     X
ZZ04    68                    PUL     A
ZZ05    40                    RTI                ;RETURN FROM INTERRUPT
        XX06                  END     COMRTI
```

Interrupt Service Routines

The program of listing 1 is an example of how to service an interrupt. It first saves the contents of the 6502 registers on the stack. Next, it calls a subroutine to service the interrupt, restores the registers, and finally returns to the interrupted task. My example is for a 6502, but the instruction mnemonics are for a macro-assembler of my own design; for example, the push X index register on stack (PUSH X) mnemonic generates two instructions: transfer X to register A (TXA) and push A onto stack (PHA).

The program in listing 2 also saves the registers on the stack, but instead of calling a subroutine in the normal fashion, this program places a return address on the stack. It then executes a jump (JMP) instruction in the indirect addressing mode to reach the subroutine. This means that the program in listing 2 will look in a location in memory to find the address of the subroutine to jump to. The impli-

cit assumption is made that the subroutine will exit with an RTS instruction. Thus, the subroutine thinks it was entered by a JSR instruction, when actually, the way there was wormed by a circuitous path.

This method for simulating use of the JSR instruction in indirect mode was developed by Tom Pittman, and I thank him for suggesting this *soft-coded vector* technique.

Benefits of Indirect-Mode Entry

There are several good reasons for entering an interrupt-service routine (or any monitor-service routine, for that matter) with a JMP indirect rather than by a JSR instruction. The first is that the choice of routine to service a given interrupt can be easily changed. Instead of being forced to use a particular routine in response to a particular interrupt, you need alter only the address value contained in the JMP-indirect vector location, which can be located in programmable memory. Thus, a string of characters to any of several different peripheral devices can be output using the same microcoded v-code. Simply place the starting address of the desired device-driver routine in the appropriate vector location.

Another reason is that if the locations of service routines are to be changed (perhaps because of additions that make a routine too big for the space it used to occupy), only the entry in the vector address table need be updated. The vector address table can be stored in an inexpensive 256-byte programmable read-only memory. Should the need arise, it is much easier to replace the 256-byte device containing the table than to find all references to a routine in a 2,048-byte programmable read-only memory, change them, and burn a new 2 K-byte device.

The reason for having each routine utilize a common return sequence is that the user may desire to have classes of routines which all need different sets of common operations done before returning to the calling routine. Such an operation could be to transfer the saved register values from the stack into the registers before returning. It may also be used to check if completion of an interrupt service (or v-code instruction) should reset a timer/counter or initiate some other action before truly returning to the pre-interrupt state.

Other Ideas

A truly innovative approach, and one that saves software overhead at the cost of more circuitry, is to *latch* the v-code, using the same circuit that detects the v-code, so that when the 6502 attempts to fetch the IRQ vector address from hexadecimal locations FFFE and FFFF, it gets an address that has been stored in a 128 by 8 programmable read-only memory. This approach, while limited in its flexibility, is ideally suited to "black box" or turnkey systems, where it is assumed that the end user has no desire to know (let alone alter) the internal operations of the machine.

I have set forth these ideas to enlighten my fellow computer experimenters. I owe a debt to the authors of the other articles I have mentioned; without their work, I could not have completely developed the ideas discussed here. I assume that many of you will improve on my work. I merely ask that you write to me and keep me informed of your progress. ■

References

1. Gordon, H T, "Decoding 650X Op Codes" *Dr Dobbs Journal of Computer Calisthenics and Orthodontia*, Volume 2, Issue 7, August 1977, pages 20 thru 22.
2. Gordon, H T, "Decoding Efficiency and Speed" *Dr Dobbs Journal*, Volume 3, Issue 2, February 1978, pages 5 thru 7.
3. Gordon, H T, "Software and Correction" *Dr Dobbs Journal*, Volume 2, Issue 9, October 1977, pages 42 thru 44.
4. Gordon, H T, "Use of NOP Codes as Executable Labels" *Dr Dobbs Journal*, Volume 3, Issue 8, September 1978, page 29.
5. Gordon, H T, "The XF and X7 Instructions of the MOS Technology 6502", BYTE, Volume 2, Number 12, December 1977, page 72.
6. Gupta, Yogesh M, "True Confessions: How I Relate to KIM" BYTE, Volume 1, Number 12, August 1976, pages 44 thru 48.
7. *MCS6500 Microcomputer Family Programming Manual*, MOS Technology, Norristown PA, 1976, pages 144 thru 147, pages 87 thru 92.
8. Moser, C W, "Add a Trap Vector for Unimplemented Op Codes" *Dr Dobbs Journal*, Volume 4, Issue 1, January 1979, pages 32 thru 34.
9. Wozniak, Stephen, "SWEET16: The 6502 Dream Machine" BYTE, Volume 2, Number 11, November 1977, pages 150 thru 159.