Using the Intel MCS®-51 Boolean Processing Capabilities

JOHN WHARTON
MICROCONTROLLER APPLICATIONS

April 1980
Information in this document is provided in connection with Intel products. Intel assumes no liability whatsoev-er, including infringement of any patent or copyright, for sale and use of Intel products except as provided in Intel’s Terms and Conditions of Sale for such products.

Intel retains the right to make changes to these specifications at any time, without notice. Microcomputer Products may have minor variations to this specification known as errata.

*Other brands and names are the property of their respective owners.

†Since publication of documents referenced in this document, registration of the Pentium, OverDrive and iCOMP trademarks has been issued to Intel Corporation.

Contact your local Intel sales office or your distributor to obtain the latest specifications before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

or call 1-800-879-4683
## CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2.0 BOOLEAN PROCESSOR OPERATION</td>
<td>2</td>
</tr>
<tr>
<td>Processing Elements</td>
<td>3</td>
</tr>
<tr>
<td>Direct Bit Addressing</td>
<td>5</td>
</tr>
<tr>
<td>Instruction Set</td>
<td>8</td>
</tr>
<tr>
<td>3.0 BOOLEAN PROCESSOR APPLICATIONS</td>
<td>12</td>
</tr>
<tr>
<td>4.0 SUMMARY</td>
<td>40</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>A-1</td>
</tr>
</tbody>
</table>

### 2.0 BOOLEAN PROCESSOR OPERATION

- Processing Elements
- Direct Bit Addressing
- Instruction Set
- Simple Instruction Combinations

### 3.0 BOOLEAN PROCESSOR APPLICATIONS

1. Bit Permutation
2. Software Serial I/O
3. Combinational Logic Equations
4. Automotive Dashboard Functions
5. Complex Control Functions

### Additional Functions and Uses
1.0 INTRODUCTION

The Intel microcontroller family now has three new members: the Intel 8031, 8051, and 8751 single-chip microcomputers. These devices, shown in Figure 1, will allow whole new classes of products to benefit from recent advances in Integrated Electronics. Thanks to Intel’s new HMOS technology, they provide larger program and data memory spaces, more flexible I/O and peripheral capabilities, greater speed, and lower system cost than any previous-generation single-chip microcomputer.

Table 1 summarizes the quantitative differences between the members of the MCS®-48 and 8051 families. The 8751 contains 4K bytes of EPROM program memory fabricated on-chip, while the 8051 replaces the EPROM with 4K bytes of lower-cost mask-programmed ROM. The 8031 has no program memory on-chip; instead, it accesses up to 64K bytes of program memory from external memory. Otherwise, the three new family members are identical. Throughout this Note, the term “8051” will represent all members of the 8051 Family, unless specifically stated otherwise.

The CPU in each microcomputer is one of the industry’s fastest and most efficient for numerical calculations on byte operands. But controllers often deal with bits, not bytes: in the real world, switch contacts can only be open or closed, indicators should be either lit or dark, motors are either turned on or off, and so forth. For such control situations the most significant aspect of the MCS®-51 architecture is its complete hardware support for one-bit, or Boolean variables (named in honor of Mathematician George Boole) as a separate data type.

The 8051 incorporates a number of special features which support the direct manipulation and testing of individual bits and allow the use of single-bit variables in performing logical operations. Taken together, these features are referred to as the MCS-51 Boolean Processor. While the bit-processing capabilities alone would be adequate to solve many control applications, their true power comes when they are used in conjunction with the microcomputer’s byte-processing and numerical capabilities.

Many concepts embodied by the Boolean Processor will certainly be new even to experienced microcomputer system designers. The purpose of this Application Note is to explain these concepts and show how they are used.

For detailed information on these parts refer to the Intel Microcontroller Handbook, order number 210918. The instruction set, assembly language, and use of the 8051 assembler (ASM51) are further described in the MCS®-51 Macro Assembler User’s Guide for DOS Systems, order number 122753.

<table>
<thead>
<tr>
<th>EPROM Program Memory</th>
<th>ROM Program Memory</th>
<th>External Program Memory</th>
<th>Program Memory (Int/Max)</th>
<th>Data Memory (Bytes)</th>
<th>Instr. Cycle Time (µs)</th>
<th>Input/Output Pins</th>
<th>Interrupt Sources</th>
<th>Reg. Banks</th>
</tr>
</thead>
<tbody>
<tr>
<td>8748</td>
<td>8048</td>
<td>8035</td>
<td>1K 4K</td>
<td>64</td>
<td>2.5</td>
<td>27</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>—</td>
<td>8049</td>
<td>8039</td>
<td>2K 4K</td>
<td>128</td>
<td>1.36</td>
<td>27</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8751</td>
<td>8051</td>
<td>8031</td>
<td>4K 64K</td>
<td>128</td>
<td>1.0</td>
<td>32</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 1. 8051 Family Pinout Diagram

Table 1. Features of Intel’s Single-Chip Microcomputers
2.0 BOOLEAN PROCESSOR OPERATION

The Boolean Processing capabilities of the 8051 are based on concepts which have been around for some time. Digital computer systems of widely varying designs all have four functional elements in common (Figure 2):

- a central processor (CPU) with the control, timing, and logic circuits needed to execute stored instructions:
- a memory to store the sequence of instructions making up a program or algorithm:
- data memory to store variables used by the program:
- and some means of communicating with the outside world.

The CPU usually includes one or more accumulators or special registers for computing or storing values during program execution. The instruction set of such a processor generally includes, at a minimum, operations to perform arithmetic or logical functions on program variables, move variables from one place to another, cause program execution to jump or conditionally branch based on register or variable states, and instructions to call and return from subroutines. The program and data memory functions sometimes share a single memory space, but this is not always the case. When the address spaces are separated, program and data memory need not even have the same basic word width.

A digital computer's flexibility comes in part from combining simple fast operations to produce more complex (albeit slower) ones, which in turn link together eventually solving the problem at hand. A four-bit CPU executing multiple precision subroutines can, for example, perform 64-bit addition and subtraction. The subroutines could in turn be building blocks for floating-point multiplication and division routines. Eventually, the four-bit CPU can simulate a far more complex "virtual" machine.

In fact, any digital computer with the above four functional elements can (given time) complete any algorithm (though the proverbial room full of chimpanzees at word processors might first re-create Shakespeare's classics and this Application Note)! This fact offers little consolation to product designers who want programs to run as quickly as possible. By definition, a real-time control algorithm must proceed quickly enough to meet the preordained speed constraints of other equipment.

One of the factors determining how long it will take a microcomputer to complete a given chore is the number of instructions it must execute. What makes a given computer architecture particularly well- or poorly-suited for a class of problems is how well its instruction set matches the tasks to be performed. The better the "primitive" operations correspond to the steps taken by the control algorithm, the lower the number of instructions needed, and the quicker the program will run. All else being equal, a CPU supporting 64-bit arithmetic directly could clearly perform floating-point math faster than a machine bogged-down by multiple-precision subroutines. In the same way, direct support for bit manipulation naturally leads to more efficient programs handling the binary input and output conditions inherent in digital control problems.

![Figure 2. Block Diagram for Abstract Digital Computer](image-url)
Processing Elements

The introduction stated that the 8051’s bit-handling capabilities alone would be sufficient to solve some control applications. Let’s see how the four basic elements of a digital computer—a CPU with associated registers, program memory, addressable data RAM, and I/O capability—relate to Boolean variables.

CPU. The 8051 CPU incorporates special logic devoted to executing several bit-wide operations. All told, there are 17 such instructions, all listed in Table 2. Not shown are 94 other (mostly byte-oriented) 8051 instructions.

Program Memory. Bit-processing instructions are fetched from the same program memory as other arithmetic and logical operations. In addition to the instructions of Table 2, several sophisticated program control features like multiple addressing modes, subroutine nesting, and a two-level interrupt structure are useful in structuring Boolean Processor-based programs.

Boolean instructions are one, two, or three bytes long, depending on what function they perform. Those involving only the carry flag have either a single-byte opcode or an opcode followed by a conditional-branch destination byte (Figure 3a). The more general instructions add a “direct address” byte after the opcode to specify the bit affected, yielding two or three byte encodings (Figure 3b). Though this format allows potentially 256 directly addressable bit locations, not all of them are implemented in the 8051 family.

Table 2. MCS-51 Boolean Processing Instruction Subset

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Byte</th>
<th>Cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETB C</td>
<td>Set Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SETB bit</td>
<td>Set direct Bit</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CLR C</td>
<td>Clear Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CLR bit</td>
<td>Clear direct bit</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CPL C</td>
<td>Complement Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CPL bit</td>
<td>Complement direct bit</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MOV C.bit</td>
<td>Move direct bit to Carry flag</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MOV bit.C</td>
<td>Move Carry flag to direct bit</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ANL C.bit</td>
<td>AND direct bit to Carry flag</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ANL bit</td>
<td>AND complement of direct bit to</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ORL C.bit</td>
<td>OR direct bit to Carry flag</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ORL bit</td>
<td>OR complement of direct bit to</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>bit to Carry flag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JC rel</td>
<td>Jump if Carry is flag is set</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JNC rel</td>
<td>Jump if No Carry flag</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JB bit.rel</td>
<td>Jump if direct Bit set</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>JNB bit.rel</td>
<td>Jump if direct Bit Not set</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>JBC bit.rel</td>
<td>Jump if direct Bit is set &amp;</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Clear bit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address mode abbreviations

C—Carry flag.
bit—128 software flags, any I/O pin, control or status bit.
rel—All conditional jumps include an 8-bit offset byte. Range is +127 – 128 bytes relative to first byte of the following instruction.

All mnemonics copyrighted © Intel Corporation 1980.
Data Memory. The instructions in Figure 3b can operate directly upon 144 general purpose bits forming the Boolean processor “RAM.” These bits can be used as software flags or to store program variables. Two operand instructions use the CPU’s carry flag (“C”) as a special one-bit register; in a sense, the carry is a “Boolean accumulator” for logical operations and data transfers.

Input/Output. All 32 I/O pins can be addressed as individual inputs, outputs, or both, in any combination. Any pin can be a control strobe output, status (Test) input, or serial I/O link implemented via software. An additional 33 individually addressable bits reconfigure, control, and monitor the status of the CPU and all on-chip peripheral functions (timer counters, serial port modes, interrupt logic, and so forth).

Figure 4. Bit Address Maps
### Symbol Position Name and Significance

#### PSW.7 Carry flag.
- Set/cleared by hardware or software during certain arithmetic and logical instructions.

#### PSW.6 Auxiliary Carry flag.
- Set/cleared by hardware during addition or subtraction instructions to indicate carry or borrow out of bit 3.

#### PSW.5 Flag 0.
- Set/cleared/tested by software as a user-defined status flag.

#### PSW.4 Register bank Select control bits.
- Set/cleared by software to determine working register bank (see Note).

#### PSW.3
- 1 & 0. Set/cleared by software to determine working register bank (see Note).

#### PSW.2 Overflow flag.
- Set/cleared by hardware during arithmetic instructions to indicate overflow conditions.

#### PSW.1 (reserved)

#### PSW.0 Parity flag.
- Set/cleared by hardware each instruction cycle to indicate an odd/even number of “one” bits in the accumulator, i.e., even parity.

**Note:**
- the contents of (RS1, RS0) enable the working register banks as follows:
  - (0,0) - Bank 0 (00H–07H)
  - (0,1) - Bank 1 (08H–0FH)
  - (1,0) - Bank 2 (10H–17H)
  - (1,1) - Bank 3 (18H–1FH)

---

### Symbol Position Name and Significance

#### P3.7 Read data control output.
- Active low pulse generated by hardware when external data memory is read.

#### P3.6 Write data control output.
- Active low pulse generated by hardware when external data memory is written.

#### P3.5 Timer/counter 1 external input or test pin.

#### P3.4 Timer/counter 0 external input or test pin.

#### P3.3 Interrupt 1 input pin.
- Low-level or falling-edge triggered.

#### P3.2 Interrupt 0 input pin.
- Low-level or falling-edge triggered.

#### P3.1 Transmit Data pin for serial port in UART mode. Clock output in shift register mode.

#### P3.0 Receive Data pin for serial port in UART mode. Data I/O pin in shift register mode.

---

### Direct Bit Addressing

The most significant bit of the direct address byte selects one of two groups of bits. Values between 0 and 127 (00H and 7FH) define bits in a block of 32 bytes of on-chip RAM, between RAM addresses 20H and 2FH (Figure 4a). They are numbered consecutively from the lowest-order byte’s lowest-order bit through the highest-order byte’s highest-order bit.

Bit addresses between 128 and 255 (80H and 0FFH) correspond to bits in a number of special registers, mostly used for I/O or peripheral control. These positions are numbered with a different scheme than RAM: the five high-order address bits match those of the register’s own address, while the three low-order bits identify the bit position within that register (Figure 4b).
Notice the column labeled “Symbol” in Figure 5. Bits with special meanings in the PSW and other registers have corresponding symbolic names. General-purpose (as opposed to carry-specific) instructions may access the carry like any other bit by using the mnemonic CY in place of C, P0, P1, P2, and P3 are the 8051’s four I/O ports: secondary functions assigned to each of the eight pins of P3 are shown in Figure 6.

Figure 7 shows the last four bit addressable registers. TCON (Timer Control) and SCON (Serial port Control) control and monitor the corresponding peripherals, while IE (Interrupt Enable) and IP (Interrupt Priority) enable and prioritize the five hardware interrupt sources. Like the reserved hardware register addresses, the five bits not implemented in IE and IP should not be accessed: they can not be used as software flags.

Addressable Register Set. There are 20 special function registers in the 8051, but the advantages of bit addressing only relate to the 11 described below. Five potentially bit-addressable register addresses (0C0H, 0C8H, 0D8H, 0E8H, & 0F8H) are being reserved for possible future expansion in microcomputers based on the MCS-51 architecture. Reading or writing non-existent registers in the 8051 series is pointless, and may cause unpredictable results. Byte-wide logical operations can be used to manipulate bits in all non-bit addressable registers and RAM.
Figure 7. Peripheral Configuration Registers

### a.) TCON—Timer/Counter Control/Status Register

<table>
<thead>
<tr>
<th>(MSB)</th>
<th>(LSB)</th>
<th>Symbol</th>
<th>Position</th>
<th>Name and Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF1</td>
<td>TR1</td>
<td>TF0</td>
<td>TR0</td>
<td>IE1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TF1</td>
<td>TCON.7</td>
<td>Timer 1 overflow Flag.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set by hardware on timer/counter overflow. Cleared when interrupt processed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TR1</td>
<td>TCON.6</td>
<td>Timer 1 Run control bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set/cleared by software to turn timer/counter on/off.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TF0</td>
<td>TCON.5</td>
<td>Timer 0 overflow Flag.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set by hardware on timer/counter overflow. Cleared when interrupt processed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TR0</td>
<td>TCON.4</td>
<td>Timer 0 Run control bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set/cleared by software to turn timer/counter on/off.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IE1</td>
<td>TCON.3</td>
<td>Interrupt 1 Edge flag.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set by hardware when external interrupt edge detected. Cleared when interrupt processed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT1</td>
<td>TCON.2</td>
<td>Interrupt 1 Type control bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set/cleared by software to specify falling edge/low level triggered external interrupts.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IE0</td>
<td>TCON.1</td>
<td>Interrupt 0 Edge flag.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set by hardware when external interrupt edge detected. Cleared when interrupt processed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT0</td>
<td>TCON.0</td>
<td>Interrupt 0 Type control bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set/cleared by software to specify falling edge/low level triggered external interrupts.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Note

- the state of (SM0, SM1) selects:
  - (0,0)—Shift register I/O expansion.
  - (0,1)—8-bit UART, variable data rate.
  - (1,0)—9-bit UART, fixed data rate.
  - (1,1)—9-bit UART, variable data rate.

### b.) SCON—Serial Port Control/Status Register

<table>
<thead>
<tr>
<th>(MSB)</th>
<th>(LSB)</th>
<th>Symbol</th>
<th>Position</th>
<th>Name and Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM0</td>
<td>SM1</td>
<td>SM2</td>
<td>REN</td>
<td>TB8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SM0</td>
<td>SCON.7</td>
<td>Serial port Mode control bit 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set/cleared by software (see note).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SM1</td>
<td>SCON.6</td>
<td>Serial port Mode control bit 1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set/cleared by software (see note).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SM2</td>
<td>SCON.5</td>
<td>Serial port Mode control bit 2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set by software to disable reception of frames for which bit 8 is zero.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REN</td>
<td>SCON.4</td>
<td>Receiver Enable control bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set/cleared by software to enable/disable serial data reception.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB8</td>
<td>SCON.3</td>
<td>Transmit Bit 8.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set/cleared by hardware to determine state of ninth data bit transmitted in 9-bit UART mode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB8</td>
<td>SCON.2</td>
<td>Receive Bit 8.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set/cleared by hardware to indicate state of ninth data bit received.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TI</td>
<td>SCON.1</td>
<td>Transmit Interrupt flag.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set by hardware when byte transmitted. Cleared by software after servicing.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RI</td>
<td>SCON.0</td>
<td>Receive Interrupt flag.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set by hardware when byte received. Cleared by software after servicing.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The accumulator and B registers (A and B) are normally involved in byte-wide arithmetic, but their individual bits can also be used as 16 general software flags. Added with the 128 flags in RAM, this gives 144 general purpose variables for bit-intensive programs. The program status word (PSW) in Figure 5 is a collection of flags and machine status bits including the carry flag itself. Byte operations acting on the PSW can therefore affect the carry.

### Instruction Set

Having looked at the bit variables available to the Boolean Processor, we will now look at the four classes of instructions that manipulate these bits. It may be helpful to refer back to Table 2 while reading this section.

#### State Control

Addressable bits or flags may be set, cleared, or logically complemented in one instruction cycle with the two-byte instructions SETB, CLR, and CPL. (The “B” affixed to SETB distinguishes it from the assembler “SET” directive used for symbol definition.) SETB and CLR are analogous to loading a bit with a constant: 1 or 0. Single byte versions perform the same three operations on the carry.

The MCS-51 assembly language specifies a bit address in any of three ways:

- by a number or expression corresponding to the direct bit address (0–255):
• by the name or address of the register containing the bit, the dot operator symbol (a period: "."), and the bit’s position in the register (7–0);  
• in the case of control and status registers, by the predefined assembler symbols listed in the first column of Figures 5–7.

Bits may also be given user-defined names with the assembler “BIT” directive and any of the above techniques. For example, bit 5 of the PSW may be cleared by any of the four instructions.

USR_FLG BIT FSW.5 ; User Symbol Definition
CLR OD6H ; Absolute Addressing
CLR FSW.5 ; Use of Dot Operator
CLR F0 ; Pre-Defined Assembler Symbol
CLR USR_FLG ; User-Defined Symbol

*Data Transfers.* The two-byte MOV instructions can transport any addressable bit to the carry in one cycle, or copy the carry to the bit in two cycles. A bit can be moved between two arbitrary locations via the carry by combining the two instructions. (If necessary, push and pop the PSW to preserve the previous contents of the carry.) These instructions can replace the multi-instruction sequence of Figure 8, a program structure appearing in controller applications whenever flags or outputs are conditionally switched on or off.

*Logical Operations.* Four instructions perform the logical-AND and logical-OR operations between the carry and another bit, and leave the results in the carry. The instruction mnemonics are ANL and ORL; the absence or presence of a slash mark (“/”) before the source operand indicates whether to use the positive-logic value or the logical complement of the addressed bit. (The source operand itself is never affected.)

*Bit-test Instructions.* The conditional jump instructions “JC rel” (Jump on Carry) and “JNC rel” (Jump on Not Carry) test the state of the carry flag, branching if it is a one or zero, respectively. (The letters “rel” denote relative code addressing.) The three-byte instructions “JB bit.rel” and “JNB bit.rel” (Jump on Bit and Jump on Not Bit) test the state of any addressable bit in a similar manner. A fifth instruction combines the Jump on Bit and Clear operations. “JBC bit.rel” conditionally branches to the indicated address, then clears the bit in the same two cycle instruction. This operation is the same as the MCS-48 “JTF” instructions.

All 8051 conditional jump instructions use program counter-relative addressing, and all execute in two cycles. The last instruction byte encodes a signed displacement ranging from $-128$ to $+127$. During execution, the CPU adds this value to the incremented program counter to produce the jump destination. Put another way, a conditional jump to the immediately following instruction would encode $00H$ in the offset byte.

A section of program or subroutine written using only relative jumps to nearby addresses will have the same machine code independent of the code’s location. An assembled routine may be repositioned anywhere in memory, even crossing memory page boundaries, without having to modify the program or recompute destination addresses. To facilitate this flexibility, there is an unconditional “Short Jump” (SJMP) which uses relative addressing as well. Since a programmer would have quite a chore trying to compute relative offset values from one instruction to another, ASM51 automatically computes the displacement needed given only the destination address or label. An error message will alert the programmer if the destination is “out of range.”

The so-called “Bit Test” instructions implemented on many other microprocessors simply perform the logical-AND operation between a byte variable and a constant mask, and set or clear a zero flag depending on the result. This is essentially equivalent to the 8051 “MOV C.bit” instruction. A second instruction is then needed to conditionally branch based on the state of the zero flag. This does not constitute abstract bit-addressing in the MCS-51 sense. A flag exists only as a field
within a register: to reference a bit the programmer
must know and specify both the encompassing register
and the bit’s position therein. This constraint severely
limits the flexibility of symbolic bit addressing and re-
duces the machine’s code-efficiency and speed.

Interaction with Other Instructions. The carry flag is
also affected by the instructions listed in Table 3. It can
be rotated through the accumulator, and altered as a
side effect of arithmetic instructions. Refer to the Us-
er’s Manual for details on how these instructions oper-
ate.

Simple Instruction Combinations

By combining general purpose bit operations with cer-
tain addressable bits, one can “custom build” several
hundred useful instructions. All eight bits of the PSW
can be tested directly with conditional jump instruc-
tions to monitor (among other things) parity and over-
flow status. Programmers can take advantage of 128
software flags to keep track of operating modes, re-
source usage, and so forth.

The Boolean instructions are also the most efficient
way to control or reconfigure peripheral and I/O regis-
ters. All 32 I/O lines become “test pins,” for example,
tested by conditional jump instructions. Any output pin
can be toggled (complemented) in a single instruction
cycle. Setting or clearing the Timer Run flags (TR0 and
TR1) turn the timer/counters on or off; polling the
same flags elsewhere lets the program determine if a
timer is running. The respective overflow flags (TF0
and TF1) can be tested to determine when the desired
period or count has elapsed, then cleared in preparation
for the next repetition. (For the record, these bits are all
part of the TCON register, Figure 7a. Thanks to sym-
bolic bit addressing, the programmer only needs to re-
member the mnemonic associated with each function.
In other words, don’t bother memorizing control word
layouts.)

In the MCS-48 family, instructions corresponding to
some of the above functions require specific opcodes.
Ten different opcodes serve to clear complement the
software flags F0 and F1, enable/disable each inter-
rupt, and start/stop the timer. In the 8051 instruction
set, just three opcodes (SETB, CLR, CPL) with a direct
bit address appended perform the same functions. Two
bit tests instructions (JB and JNB) can be combined with
test the software flags, the 8048 I/O pins T0, T1, and INT, and the eight accumulator bits,
replacing 15 more different instructions.

Table 4a shows how 8051 programs implement soft-
ware flag and machine control functions associated
with special opcodes in the 8048. In every case the
MCS-51 solution requires the same number of machine
cycles, and executes 2.5 times faster.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Byte</th>
<th>Cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD A,Rn</td>
<td>Add register to Accumulator</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADD A,dir</td>
<td>Add direct byte to Accumulator</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ADD A,@Ri</td>
<td>Add indirect RAM to Accumulator</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADD A,#data</td>
<td>Add immediate data to Accumulator</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ADDC A,Rn</td>
<td>Add register to Accumulator with Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADDC A,dir</td>
<td>Add direct byte to Accumulator with Carry flag</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ADDC A,@Ri</td>
<td>Add indirect RAM to Accumulator with Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADDC A,#data</td>
<td>Add immediate data to Acc with Carry flag</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,Rn</td>
<td>Subtract register from Accumulator with borrow</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,dir</td>
<td>Subtract direct byte from Acc with borrow</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,@Ri</td>
<td>Subtract indirect RAM from Acc with borrow</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SUBB A,#data</td>
<td>Subtract immediate data from Acc with borrow</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MUL AB</td>
<td>Multiply A &amp; B</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>DIV AB</td>
<td>Divide A by B</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>DA A</td>
<td>Decimal Adjust</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Accumulator Left through the Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RRC A</td>
<td>Rotate Accumulator Right through Carry flag</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CJNE A,dir.rel</td>
<td>Compare direct byte to Acc &amp; Jump if Not Equal</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CJNE A,#data.rel</td>
<td>Compare immediate to Acc &amp; Jump if Not Equal</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CJNE Rn,#data.rel</td>
<td>Compare imm to register &amp; Jump if Not Equal</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>CJNE @Ri,#data.rel</td>
<td>Compare imm to indirect &amp; Jump if Not Equal</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1980.
### Table 4a. Contrasting 8048 and 8051 Bit Control and Testing Instructions

<table>
<thead>
<tr>
<th>8048 Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
<th>μSec</th>
<th>8x51 Instruction</th>
<th>Bytes</th>
<th>Cycles &amp; μSec</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flag Control</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>Flag Control</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR C</td>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>CLR C</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CPL F0</td>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>CPL F0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Flag Testing</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>Flag Testing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JNC offset 2</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JNC rel 2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JF0 offset 2</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JB F0.rel 3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JB7 offset 2</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JB ACC.7.rel 3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Peripheral Polling</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>Peripheral Polling</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JT0 offset 2</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JB T0.rel 3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JN1 offset 2</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JNB INTO.rel 3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JTF offset 2</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>JBC TF0.rel 3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Machine and Peripheral Control</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>Machine and Peripheral Control</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRT T</td>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>SETB TR0 2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>EN 1</td>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>SETB EX0 2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DIS TCNT1</td>
<td>1</td>
<td>1</td>
<td>2.5</td>
<td>CLR ET0 2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 4b. Replacing 8048 Instruction Sequences with Single 8x51 Instructions

<table>
<thead>
<tr>
<th>8048 Instruction</th>
<th>Bytes</th>
<th>Cycles</th>
<th>μSec</th>
<th>8051 Instruction</th>
<th>Bytes</th>
<th>Cycles &amp; μSec</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flag Control</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>Flag Control</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set carry</td>
<td></td>
<td></td>
<td></td>
<td>Set carry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR C</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>SETB C</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CPL C</td>
<td></td>
<td></td>
<td></td>
<td>SETB F0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Set Software Flag</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>Set Software Flag</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR F0</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>SETB F0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CPL F0</td>
<td></td>
<td></td>
<td></td>
<td>SETB F0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Turn Off Output Pin</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>Turn Off Output Pin</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANL P1. #0FBH</td>
<td>2</td>
<td>2</td>
<td>5.0</td>
<td>CLR P1.2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Complement Output Pin</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>Complement Output Pin</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN A.P1</td>
<td></td>
<td></td>
<td></td>
<td>CPL P1.2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>XRL A. #04H</td>
<td></td>
<td></td>
<td></td>
<td>CPL P1.2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>OUTL P1. A</td>
<td>4</td>
<td>6</td>
<td>15.0</td>
<td>CPL P1.2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Clear Flag in RAM</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>Clear Flag in RAM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV R0. #FLGADR</td>
<td></td>
<td></td>
<td></td>
<td>CLR USER__FLG</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MOV A.@R0</td>
<td></td>
<td></td>
<td></td>
<td>CLR USER__FLG</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ANL A. #FLGMASK</td>
<td></td>
<td></td>
<td></td>
<td>CLR USER__FLG</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>MOV @R0.A</td>
<td>6</td>
<td>6</td>
<td>15.0</td>
<td>CLR USER__FLG</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
3.0 BOOLEAN PROCESSOR APPLICATIONS

So what? Then what does all this buy you?

Qualitatively, nothing. All the same capabilities could be (and often have been) implemented on other machines using awkward sequences of other basic operations. As mentioned earlier, any CPU can solve any problem given enough time.

Quantitatively, the differences between a solution allowed by the 8051 and those required by previous architectures are numerous. What the 8051 Family buys you is a faster, cleaner, lower-cost solution to microcontroller applications.

The opcode space freed by condensing many specific 8048 instructions into a few general operations has been used to add new functionality to the MCS-51 architecture—both for byte and bit operations. 144 software flags replace the 8048’s two. These flags (and the carry) may be directly set, not just cleared and complemented, and all can be tested for either state, not just one. Operating mode bits previously inaccessible may be read, tested, or saved. Situations where the 8051 instruction set provides new capabilities are contrasted with 8048 instruction sequences in Table 4b. Here the 8051 speed advantage ranges from 5x to 15x!

Combining Boolean and byte-wide instructions can produce great synergy. An MCS-51 based application will prove to be:

• simpler to write since the architecture correlates more closely with the problems being solved:
• easier to debug because more individual instructions have no unexpected or undesirable side-effects:
• more byte efficient due to direct bit addressing and program counter relative branching:
• faster running because fewer bytes of instruction need to be fetched and fewer conditional jumps are processed:
• lower cost because of the high level of system-integration within one component.

These rather unabashed claims of excellence shall not go unsubstantiated. The rest of this chapter examines less trivial tasks simplified by the Boolean processor. The first three compare the 8051 with other microprocessors; the last two go into 8051-based system designs in much greater depth.

Design Example #1—Bit Permutation

First off, we’ll use the bit-transfer instructions to permute a lengthy pattern of bits.
A steadily increasing number of data communication products use encoding methods to protect the security of sensitive information. By law, interstate financial transactions involving the Federal banking system must be transmitted using the Federal Information Processing Data Encryption Standard (DES).

Basically, the DES combines eight bytes of "plaintext" data (in binary, ASCII, or any other format) with a 56-bit "key", producing a 64-bit encrypted value for transmission. At the receiving end the same algorithm is applied to the incoming data using the same key, reproducing the original eight byte message. The algorithm used for these permutations is fixed; different user-defined keys ensure data privacy.

It is not the purpose of this note to describe the DES in any detail. Suffice it to say that encryption/decryption is a long, iterative process consisting of rotations, exclusive OR operations, function table look-ups, and an extensive (and quite bizarre) sequence of bit permutation, packing, and unpacking steps. (For further details refer to the June 21, 1979 issue of Electronics magazine.) The bit manipulation steps are included, it is rumored, to impede a general purpose digital supercomputer trying to "break" the code. Any algorithm implementing the DES with previous generation microprocessors would spend virtually all of its time fiddling bits.

The bit manipulation performed is typified by the Key Schedule Calculation represented in Figure 9. This step is repeated 16 times for each key used in the course of a transmission. In essence, a seven-byte, 56-bit "Shifted Key Buffer" is transformed into an eight-byte, "Permutation Buffer" without altering the shifted Key. The arrows in Figure 9 indicate a few of the translation steps. Only six bits of each byte of the Permutation Buffer are used; the two high-order bits of each byte are cleared. This means only 48 of the 56 Shifted Key Buffer bits are used in any one iteration.

Different microprocessor architectures would best implement this type of permutation in different ways. Most approaches would share the steps of Figure 10a:

- Initialize the Permutation Buffer to default state (ones or zeroes);
- Isolate the state of a bit of a byte from the Key Buffer. Depending on the CPU, this might be accomplished by rotating a word of the Key Buffer through a carry flag or testing a bit in memory or an accumulator against a mask byte;
- Perform a conditional jump based on the carry or zero flag if the Permutation Buffer default state is correct;
- Otherwise reverse the corresponding bit in the permutation buffer with logical operations and mask bytes.

Each step above may require several instructions. The last three steps must be repeated for all 48 bits. Most microprocessors would spend 300 to 3,000 microseconds on each of the 16 iterations.

Notice, though, that this flow chart looks a lot like Figure 8. The Boolean Processor can permute bits by simply moving them from the source to the carry to the destination—a total of two instructions taking four bytes and three microseconds per bit. Assume the Shifted Key Buffer and Permutation Buffer both reside in bit-addressable RAM, with the bits of the former assigned symbolic names SKB_1, SKB_2, . . . SKB_56, and that the bytes of the latter are named PB_1, . . . PB_8. Then working from Figure 9, the software for the permutation algorithm would be that of Example 1a. The total routine length would be 192 bytes, requiring 144 microseconds.
Figure 10a. Flowchart for Key Permutation Attempted with a Byte Processor
Figure 10b. DES Key Permutation with Boolean Processor
The algorithm of Figure 10b is just slightly more efficient in this time-critical application and illustrates the synergy of an integrated byte and bit processor. The bits needed for each byte of the Permutation Buffer are assimilated by loading each bit into the carry (1 µs.) and shifting it into the accumulator (1 µs.). Each byte is stored in RAM when completed. Forty-eight bits thus need a total of 112 instructions, some of which are listed in Example 1b.

Worst-case execution time would be 112 microseconds, since each instruction takes a single cycle. Routine length would also decrease, to 168 bytes. (Actually, in the context of the complete encryption algorithm, each permuted byte would be processed as soon as it is assimilated—saving memory and cutting execution time by another 8 µs.)

To date, most banking terminals and other systems using the DES have needed special boards or peripheral controller chips just for the encryption/decryption process, and still more hardware to form a serial bit stream for transmission (Figure 11a). An 8051 solution could pack most of the entire system onto the one chip (Figure 11b). The whole DES algorithm would require less than one-fourth of the on-chip program memory, with the remaining bytes free for operating the banking terminal (or whatever) itself.

Moreover, since transmission and reception of data is performed through the on-board UART, the unencrypted data (plaintext) never even exists outside the microcomputer! Naturally, this would afford a high degree of security from data interception.

<table>
<thead>
<tr>
<th>Example 1. DES Key Permutation Software.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.) “Brute Force” technique</td>
</tr>
<tr>
<td>MOV C,SKB_1</td>
</tr>
<tr>
<td>MOV FB_1.1,C</td>
</tr>
<tr>
<td>MOV C,SKB_2</td>
</tr>
<tr>
<td>MOV FB_4.0,C</td>
</tr>
<tr>
<td>MOV C,SKB_3</td>
</tr>
<tr>
<td>MOV FB_2.5,C</td>
</tr>
<tr>
<td>MOV C,SKB_4</td>
</tr>
<tr>
<td>MOV FB_1.0,C</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>MOV C,SKB_56</td>
</tr>
<tr>
<td>MOV FB_5.0,C</td>
</tr>
<tr>
<td>MOV C,SKB_56</td>
</tr>
<tr>
<td>MOV FB_7.2,C</td>
</tr>
<tr>
<td>b.) Using Accumulator to Collect Bits</td>
</tr>
<tr>
<td>CLR A</td>
</tr>
<tr>
<td>MOV C,SKB_14</td>
</tr>
<tr>
<td>RLC A</td>
</tr>
<tr>
<td>MOV C,SKB_17</td>
</tr>
<tr>
<td>RLC A</td>
</tr>
<tr>
<td>MOV C,SKB_11</td>
</tr>
<tr>
<td>RLC A</td>
</tr>
<tr>
<td>MOV C,SKB_24</td>
</tr>
<tr>
<td>RLC A</td>
</tr>
<tr>
<td>MOV C,SKB_1</td>
</tr>
<tr>
<td>RLC A</td>
</tr>
<tr>
<td>MOV C,SKB_5</td>
</tr>
<tr>
<td>RLC A</td>
</tr>
<tr>
<td>MOV FB_1,A</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>MOV C,SKB_29</td>
</tr>
<tr>
<td>RLC A</td>
</tr>
<tr>
<td>MOV C,SKB_32</td>
</tr>
<tr>
<td>RLC A</td>
</tr>
<tr>
<td>MOV FB_8,A</td>
</tr>
</tbody>
</table>
Design Example #2—Software Serial I/O

An exercise often imposed on beginning microcomputer students is to write a program simulating a UART. Though doing this with the 8051 Family may appear to be a moot point (given that the hardware for a full UART is on-chip), it is still instructive to see how it would be done, and maintains a product line tradition.

As it turns out, the 8051 microcomputers can receive or transmit serial data via software very efficiently using the Boolean instruction set. Since any I/O pin may be a serial input or output, several serial links could be maintained at once.

Figures 12a and 12b show algorithms for receiving or transmitting a byte of data. (Another section of program would invoke this algorithm eight times, synchronizing it with a start bit, clock signal, software delay, or timer interrupt.) Data is received by testing an input pin, setting the carry to the same state, shifting the carry into a data buffer, and saving the partial frame in internal RAM. Data is transmitted by shifting an output buffer through the carry, and generating each bit on an output pin.

A side-by-side comparison of the software for this common "bit-banging" application with three different microprocessor architectures is shown in Table 5a and 5b. The 8051 solution is more efficient than the others on every count!
Figure 12. Serial I/O Algorithms

a.) Reception

INPUT

PIN = 1

TEST PIN

PIN = 0

SET CARRY

CLEAR CARRY

LOAD BUFFER

ROTATE THRU C

STORE BUFFER

b.) Transmission

OUTPUT

LOAD BUFFER

ROTATE THRU C

STORE BUFFER

CARRY = 0

TEST CARRY

CARRY = 1

CLEAR OUTPUT

SET OUTPUT

203830-10

203830-11
Design Example #3—Combinatorial Logic Equations

Next we'll look at some simple uses for bit-test instructions and logical operations. (This example is also presented in Application Note AP-69.)

Virtually all hardware designers have solved complex functions using combinatorial logic. While the hardware involved may vary from relay logic, vacuum tubes, or TTL to more esoteric technologies like fluidics, in each case the goal is the same: to solve a problem represented by a logical function of several Boolean variables.

Figure 13 shows TTL and relay logic diagrams for a function of the six variables $U$ through $Z$. Each is a solution of the equation:

$$Q = (U \cdot (V + W)) + (X \cdot \overline{Y}) + Z$$

Equations of this sort might be reduced using Karnaugh Maps or algebraic techniques, but that is not the purpose of this example. As the logic complexity increases, so does the difficulty of the reduction process. Even a minor change to the function equations as the design evolves would require tedious re-reduction from scratch.
For the sake of comparison we will implement this function three ways, restricting the software to three proper subsets of the MCS-51 instruction set. We will also assume that U and V are input pins from different input ports, W and X are status bits for two peripheral controllers, and Y and Z are software flags set up earlier in the program. The end result must be written to an output pin on some third port. The first two implementations follow the flow-chart shown in Figure 14. Program flow would embark on a route down a test-and-branch tree and leaves either the "True" or "Not True" exit ASAP—as soon as the proper result has been determined. These exits then rewrite the output port with the result bit respectively one or zero.
Figure 14. Flow Chart for Tree-Branching Algorithm

The code which results is cumbersome and error prone. It would be difficult to prove whether the software worked for all input combinations in programs of this sort. Furthermore, execution time will vary widely with input data.

Thanks to the direct bit-test operations, a single instruction can replace each move mask conditional jump sequence in Example 2a, but the algorithm would be equally convoluted (see Example 2b). To lessen the confusion "a bit" each input variable is assigned a symbolic name.

A more elegant and efficient implementation (Example 2c) strings together the Boolean ANL and ORL functions to generate the output function with straight-line code. When finished, the carry flag contains the result, which is simply copied out to the destination pin. No flow chart is needed—code can be written directly from the logic diagrams in Figure 14. The result is simplicity itself: fast, flexible, reliable, easy to design, and easy to debug.

An 8051 program can simulate an N-input AND or OR gate with at most \( N + 1 \) lines of source program—one for each input and one line to store the results. To simulate NAND and NOR gates, complement the carry after computing the function. When some inputs to the gate have "inversion bubbles", perform the ANL or ORL operation on inverted operands. When the first input is inverted, either load the operand into the carry and then complement it, or use DeMorgan’s Theorem to convert the gate to a different form.

Example 2. Software Solutions to Logic Function of Figure 13.

a.) Using only byte-wide logical instructions

Example code:
```
:BFUNCI SOLVE RANDOM LOGIC
    FUNCTION OF 6 VARIABLES
    BY LOADING AND MASKING
    THE APPROPRIATE BITS IN
    THE Accumulator. Then
    Executing Conditional
    Jumps Based on Zero
    Condition. (Approach Used
    By Byte-Oriented
    Architectures.) Byte And
    Mask Values Correspond To
    Respective Byte Address
    And Bit Positions.
    OUTBUF DATA 22H
    ;Output Pin State Map
```
b.) Using only bit-test instructions

:BFUNC2 SOLVE A RANDOM LOGIC
:FUNCTION OF 6 VARIABLES
:BY DIRECTLY POLLING EACH
:BIT. (APPROACH USING
:MCS-51 UNIQUE BIT-TEST
:INSTRUCTION CAPABILITY.)
:SYMBOLS USED IN LOGIC
:DIAGRAM ASSIGNED TO
:CORRESPONDING 8x51 BIT
:ADDRESSES.

AP-70

TESTV: MOV A,P2
ANL A,#00000100B
JNZ TESTU
ANL A,#00100000B
JZ TESTX

TESTU: MOV A,P1
ANL A,#00000100B
JNZ SETQ

TESTX: MOV A,TCON
ANL A,#00100000B
JZ TESTX
MOV A,20H
ANL A,#00000010B
JZ SETQ

TESTZ: MOV A,21H
ANL A,#00000010B
JZ SETQ
CLRQ: MOV A,OUTBUF
ANL A,#11110111B
JMP OUTQ

SETQ: MOV A,OUTBUF
ANL A,#00001000B

OUTQ: MOV OUTBUF,A
MOV P3,A

U BIT P1.1
V BIT P2.2
W BIT TF0
X BIT IE1
Y BIT 20H.0
Z BIT 21H.1
Q BIT P3.3

b.) Using only bit-test instructions

b.) Using only bit-test instructions
An upper-limit can be placed on the complexity of software to simulate a large number of gates by summing the total number of inputs and outputs. The actual total should be somewhat shorter, since calculations can be “chained,” as shown. The output of one gate is often the first input to another, bypassing the intermediate variable to eliminate two lines of source.

**Design Example #4—Automotive Dashboard Functions**

Now let's apply these techniques to designing the software for a complete controller system. This application is patterned after a familiar real-world application which isn't nearly as trivial as it might first appear: automobile turn signals.

Imagine the three position turn lever on the steering column as a single-pole, triple-throw toggle switch. In its central position all contacts are open. In the up or down positions contacts close causing corresponding lights in the rear of the car to blink. So far very simple.

Two more turn signals blink in the front of the car, and two others in the dashboard. All six bulbs flash when an emergency switch is closed. A thermo-mechanical relay (accessible under the dashboard in case it wears out) causes the blinking.

Applying the brake pedal turns the tail light filaments on constantly ... unless a turn is in progress, in which case the blinking tail light is not affected. (Of course, the front turn signals and dashboard indicators are not affected by the brake pedal.) Table 6 summarizes these operating modes.

<table>
<thead>
<tr>
<th>Table 6. Truth Table for Turn-Signal Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Signals</strong></td>
</tr>
<tr>
<td>Brake Switch</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
But we’re not done yet. Each of the exterior turn signal (but not the dashboard) bulbs has a second, somewhat dimmer filament for the parking lights. Figure 15 shows TTL circuitry which could control all six bulbs. The signals labeled “High Freq.” and “Low Freq.” represent two square-wave inputs. Basically, when one of the turn switches is closed or the emergency switch is activated the low frequency signal (about 1 Hz) is gated through to the appropriate dashboard indicator(s) and turn signal(s). The rear signals are also activated when the brake pedal is depressed provided a turn is not being made in the same direction. When the parking light switch is closed the higher frequency oscillator is gated to each front and rear turn signal, sustaining a low-intensity background level. (This is to eliminate the need for additional parking light filaments.)

In most cars, the switching logic to generate these functions requires a number of multiple-throw contacts. As many as 18 conductors thread the steering column of some automobiles solely for turn-signal and emergency blinker functions. (The author discovered this recently to his astonishment and dismay when replacing the whole assembly because of one burned contact.)

A multiple-conductor wiring harness runs to each corner of the car, behind the dash, up the steering column, and down to the blinker relay below. Connectors at each termination for each filament lead to extra cost and labor during construction, lower reliability and safety, and more costly repairs. And considering the system’s present complexity, increasing its reliability or detecting failures would be quite difficult.

There are two reasons for going into such painful detail describing this example. First, to show that the messiest part of many system designs is determining what the controller should do. Writing the software to solve these functions will be comparatively easy. Secondly, to show the many potential failure points in the system. Later we’ll see how the peripheral functions and intelligence built into a microcomputer (with a little creativity) can greatly reduce external interconnections and mechanical part count.

The Single-Chip Solution

The circuit shown in Figure 16 indicates five input pins to the five input variables—left-turn select, right-turn select, brake pedal down, emergency switch on, and parking lights on. Six output pins turn on the front, rear, and dashboard indicators for each side. The microcomputer implements all logical functions through software, which periodically updates the output signals as time elapses and input conditions change.

![Figure 15. TTL Logic Implementation of Automotive Turn Signals](image)
Design Example #3 demonstrated that symbolic addressing with user-defined bit names makes code and documentation easier to write and maintain. Accordingly, we'll assign these I/O pins names for use throughout the program. (The format of this example will differ somewhat from the others. Segments of the overall program will be presented in sequence as each is described.)

; INPUT PIN DECLARATIONS:
; (ALL INPUTS ARE POSITIVE-TRUE LOGIC)
; BRAKE BIT P1.0 : BRAKE PEDAL
; DEPRESSED
; EMERG BIT P1.1 : EMERGENCY BLINKER
; ACTIVATED
; PARK BIT P1.2 : PARKING LIGHTS ON
; TURN BIT P1.3 : TURN LEVER DOWN
; R.TURN BIT P1.4 : TURN LEVER UP
;
; OUTPUT PIN DECLARATIONS:
; I.FRNT BIT P1.5 : FRONT LEFT-TURN
; I.INDICATOR
; R.FRNT BIT P1.6 : FRONT RIGHT-TURN
; R.INDICATOR
; I.DASH BIT P1.7 : DASHBOARD LEFT-TURN
; I.INDICATOR

Another key advantage of symbolic addressing will appear further on in the design cycle. The locations of cable connectors, signal conditioning circuitry, voltage regulators, heat sinks, and the like all affect P.C. board layout. It's quite likely that the somewhat arbitrary pin assignment defined early in the software design cycle will prove to be less than optimum; rearranging the I/O pin assignment could well allow a more compact module, or eliminate costly jumpers on a single-sided board. (These considerations apply especially to automotive and other cost-sensitive applications needing single-chip controllers.) Since other architectures mask bytes or use “clever” algorithms to isolate bits by rotating them into the carry, re-routing an input signal (from bit 1 of port 1, for example, to bit 4 of port 3) could require extensive modifications throughout the software.

The Boolean Processor’s direct bit addressing makes such changes absolutely trivial. The number of the port containing the pin is irrelevant, and masks and complex
program structures are not needed. Only the initial Boolean variable declarations need to be changed; ASMD1 automatically adjusts all addresses and symbolic references to the reassigned variables. The user is assured that no additional debugging or software verification will be required.

```
; ... ....
:INTERRUPT RATE SUBDIVIDER
SUB_DIV DATA 20H
:HIGH-FREQUENCY OSCILLATOR BIT
HI_FREQ BIT SUB_DIV,0
:LOW-FREQUENCY OSCILLATOR BIT
LO_FREQ BIT SUB_DIV,7
: ...
ORG 0000H
JMP INIT
: ...
ORG 100H
:PUT TIMER 0 IN MODE 1
INIT;
MOV TMOD,#00000001B
;INITIALIZE TIMER REGISTERS
MOV TL0,#0
MOV TH0,#-16
;SUBDIVIDE INTERRUPT RATE BY 244
MOV SUB_DIV,#244
;ENABLE TIMER INTERRUPTS
SETB ET0
;GLOBALLY ENABLE ALL INTERRUPTS
SETB EA
;START TIMER
SETB TR0
;
: (CONTINUE WITH BACKGROUND PROGRAM)
;
:PUT TIMER 0 IN MODE 1
;INITIALIZE TIMER REGISTERS
;
:SUBDIVIDE INTERRUPT RATE BY 244
;ENABLE TIMER INTERRUPTS
;GLOBALLY ENABLE ALL INTERRUPTS
;START TIMER
```

Timer 0 (one of the two on-chip timer counters) replaces the thermo-mechanical blinker relay in the dashboard controller. During system initialization it is configured as a timer in mode 1 by setting the least significant bit of the timer mode register (TMOD). In this configuration the low-order byte (TLO) is incremented every machine cycle, overflowing and incrementing the high-order byte (TH0) every 256 μs. Timer interrupt 0 is enabled so that a hardware interrupt will occur each time TH0 overflows.

An eight-bit variable in the bit-addressable RAM array will be needed to further subdivide the interrupts via software. The lowest-order bit of this counter toggles very fast to modulate the parking lights: bit 7 will be “tuned” to approximately 1 Hz for the turn- and emergency-indicator blinking rate.

Loading TH0 with -16 will cause an interrupt after 4.096 ms. The interrupt service routine reloads the high-order byte of timer 0 for the next interval, saves the CPU registers likely to be affected on the stack, and then decrements SUB_DIV. Loading SUB_DIV with 244 initially and each time it decrements to zero will produce a 0.999 second period for the highest-order bit.

```
ORG 000BH ;TIMER 0 SERVICE VECTOR
MOV TH0,#-16
PUSH PSW
PUSH ACC
PUSH B
DJNZ SUB_DIV,T0SERV
MOV SUB_DIV,#244
```

The code to sample inputs, perform calculations, and update outputs—the real “meat” of the signal controller algorithm—may be performed either as part of the interrupt service routine or as part of a background program loop. The only concern is that it must be executed at least several dozen times per second to prevent parking light flickering. We will assume the former case, and insert the code into the timer 0 service routine.

First, notice from the logic diagram (Figure 15) that the subterm (PARK • H_FREQ), asserted when the parking lights are to be on dimly, figures into four of the six output functions. Accordingly, we will first compute that term and save it in a temporary location named “DIM”. The PSW contains two general purpose flags: F0, which corresponds to the 8048 flag of the same name, and PSW.1. Since the PSW has been saved and will be restored to its previous state after servicing the interrupt, we can use either bit for temporary storage.

```
DIM BIT PSW.1 ;DECLARE TEMP STORAGE FLAG
; ... ....
MOV C,PARK ;GATE PARKING
ANL HI_FREQ ;WITH HIGH FREQUENCY SIGNAL
MOV DIM,C ;AND SAVE IN TEMP VARIABLE
```

This simple three-line section of code illustrates a remarkable point. The software indicates in very abstract terms exactly what function is being performed, inde-
dependent of the hardware configuration. The fact that these three bits include an input pin, a bit within a program variable, and a software flag in the PSW is totally invisible to the programmer.

Now generate and output the dashboard left turn signal.

```
: MOV C,L_TURN ;SET CARRY IF
          ;TURN
ORL C,EMERG ;OR EMERGENCY
          ;SELECTED
ANL C,LO_FREQ ;GATE IN 1 HZ
          ;SIGNAL
MOV I_DASH,C ;AND OUTPUT TO
          ;DASHBOARD
```

To generate the left front turn signal we only need to add the parking light function in F0. But notice that the function in the carry will also be needed for the rear signal. We can save effort later by saving its current state in F0.

```
: MOV F0,C ;SAVE FUNCTION
          ;SO FAR
ORL C,DIM ;ADD IN PARKING
          ;LIGHT FUNCTION
MOV L_FRTN,C ;AND OUTPUT TO
          ;TURN SIGNAL
```

Finally, the rear left turn signal should also be on when the brake pedal is depressed, provided a left turn is not in progress.

```
MOV C,BRAKE ;GATE BRAKE
ANL C,L_TURN ;WITH TURN
ORL C,F0 ;INCLUDE TEMP.
          ;VARIABLE FROM DASH
```

(The perceptive reader may notice that simply rearranging the steps could eliminate one instruction from each sequence.)

Now we have to go through a similar sequence for the right-hand equivalents to all the left-turn lights. This also gives us a chance to see how the code segments above look when combined.

```
: MOV C,R_TURN ;SET CARRY H-
          ;TURN
ORL C,EMERG ;OR EMERGENCY
          ;SELECTED
ANL C,LO_FREQ ;IF SO. GATE IN 1
          ;HZ SIGNAL
MOV R_DASH,C ;AND OUTPUT TO
          ;DASHBOARD
MOV F0,C ;SAVE FUNCTION
          ;SO FAR
ORL C,DIM ;ADD IN PARKING
          ;LIGHT FUNCTION
MOV R_FRTN,C ;AND OUTPUT TO
          ;TURN SIGNAL
MOV C.BRAKE ;GATE BRAKE
          ;PEDAL SWITCH
ANL C,R_TURN ;WITH TURN
ORL C,R ;INCLUDE TEMP.
          ;VARIABLE FROM
          ;DASH
ORL C,DIM ;AND OUTPUT TO
          ;TURN SIGNAL
```

(The perceptive reader may notice that simply rearranging the steps could eliminate one instruction from each sequence.)

Table 7. Non-Trivial Duty Cycles

<table>
<thead>
<tr>
<th>Sub_Div Bits</th>
<th>12.5%</th>
<th>25.0%</th>
<th>37.5%</th>
<th>50.0%</th>
<th>62.5%</th>
<th>75.0%</th>
<th>87.5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>X X X X X X</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
</tr>
<tr>
<td>X X X X X X</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
</tr>
<tr>
<td>X X X X X X</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
<td>On</td>
</tr>
<tr>
<td>X X X X X X</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
<td>Off</td>
<td>On</td>
<td>On</td>
<td>On</td>
</tr>
<tr>
<td>X X X X X X</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
<td>On</td>
<td>On</td>
<td>On</td>
<td>On</td>
</tr>
<tr>
<td>X X X X X X</td>
<td>Off</td>
<td>On</td>
<td>On</td>
<td>On</td>
<td>On</td>
<td>On</td>
<td>On</td>
</tr>
</tbody>
</table>
Program Refinements. The luminescence of an incandescent light bulb filament is generally non-linear: the 50% duty cycle of HI_FREQ may not produce the desired intensity. If the application requires, duty cycles of 25%, 75%, etc. are easily achieved by ANDing and ORing in additional low-order bits of SUB_DIV. For example, 30 H/ signals of seven different duty cycles could be produced by considering bits 2–0 as shown in Table 7. The only software change required would be to the code which sets-up variable DIM;

```
MOV C,SUB_DIV.1;START WITH 50
  ;PERCENT
ANL C,SUB_DIV.0;MASK DOWN TO 25
  ;PERCENT
ORL C,SUB_DIV.2;AND BUILD BACK TO
  ;62 PERCENT
MOV DIM,C;DUTY CYCLE FOR
  ;PARKING LIGHTS.
```

Interconnections increase cost and decrease reliability. The simple buffered pin-per-function circuit in Figure 16 is insufficient when many outputs require higher-than-TTL drive levels. A lower-cost solution uses the 8051 serial port in the shift-register mode to augment I/O. In mode 0, writing a byte to the serial port data buffer (SBUF) causes the data to be output sequentially through the "RXD" pin while a burst of eight clock pulses is generated on the "TXD" pin. A shift register connected to these pins (Figure 17) will load the data byte as it is shifted out. A number of special peripheral driver circuits combining shift-register inputs with high drive level outputs have been introduced recently.

Cascading multiple shift registers end-to-end will expand the number of outputs even further. The data rate in the I/O expansion mode is one megabaud, or 8 μs per byte. This is the mode which the serial port defaults to following a reset, so no initialization is required.

The software for this technique uses the B register as a "map" corresponding to the different output functions. The program manipulates these bits instead of the output pins. After all functions have been calculated the B register is shifted by the serial port to the shift-register driver. (While some outputs may glitch as data is shifted through them, at 1 Megabaud most people wouldn't notice. Some shift registers provide an "enable" bit to hold the output states while new data is being shifted in.)

This is where the earlier decision to address bits symbolically throughout the program is going to pay off. This major I/O restructuring is nearly as simple to implement as rearranging the input pins. Again, only the bit declarations need to be changed.

```
I_FRNT BIT B.0;FRONT LEFT-TURN
  ;INDICATOR
R_FRNT BIT B.1;FRONT RIGHT-TURN
  ;INDICATOR
I_DASH BIT B.2;DASHBOARD LEFT-TURN
  ;INDICATOR
R_DASH BIT B.3;DASHBOARD RIGHT-TURN
  ;INDICATOR
I_REAR BIT B.4;REAR LEFT-TURN
  ;INDICATOR
R_REAR BIT B.5;REAR RIGHT-TURN
  ;INDICATOR
```

Figure 17. Output Expansion Using Serial Port
The original program to compute the functions need not change. After computing the output variables, the control map is transmitted to the buffered shift register through the serial port.

MOV SBUF,B: LOAD BUFFER AND TRANSMIT

The Boolean Processor solution holds a number of advantages over older methods. Fewer switches are required. Each is simpler, requiring fewer poles and lower current contacts. The flasher relay is eliminated entirely. Only six filaments are driven, rather than 10. The wiring harness is therefore simpler and less expensive— one conductor for each of the six lamps and each of the five sensor switches. The fewer conductors use far fewer connectors. The whole system is more reliable.

And since the system is much simpler it would be feasible to implement redundancy or fault detection on the four main turn indicators. Each could still be a standard double filament bulb, but with the filaments driven in parallel to tolerate single-element failures.

Even with redundancy, the lights will eventually fail. To handle this inescapable fact current or voltage sensing circuits on each main drive wire can verify that each bulb and its high-current driver is functioning properly. Figure 18 shows one such circuit.

Assume all of the lights are turned on except one: i.e., all but one of the collectors are grounded. For the bulb which is turned off, if there is continuity from +12V through the bulb base and filament, the control wire, all connectors, and the P.C. board traces, and if the transistor is indeed not shorted to ground, then the collector will be pulled to +12V. This turns on the base of Q8 through the corresponding resistor, and grounds the input pin, verifying that the bulb circuit is operational. The continuity of each circuit can be checked by software in this way.
Now turn all the bulbs on, grounding all the collectors. Q7 should be turned off, and the Test pin should be high. However, a control wire shorted to +12V or an open-circuited drive transistor would leave one of the collectors at the higher voltage even now. This too would turn on Q7, indicating a different type of failure. Software could perform these checks once per second by executing the routine every time the software counter SUB_DIV is reload by the interrupt routine.

The complete assembled program listing is printed in Appendix A. The resulting code consists of 67 program statements, not counting declarations and comments, which assemble into 150 bytes of object code. Each pass through the service routine requires (coincidentally) 67 μs plus 32 μs once per second for the electrical test. If executed every 4 ms as suggested this software would typically reduce the throughput of the background program by less than 2%.

Once a microcomputer has been designed into a system, new features suddenly become virtually free. Software could make the emergency blinkers flash alternately or at a rate faster than the turn signals. Turn signals could override the emergency blinkers. Adding more bulbs would allow multiple tail light sequencing and syncopation—true flash factor, so to speak.

**Design Example # 5—Complex Control Functions**

Finally, we'll mix byte and bit operations to extend the use of 8051 into extremely complex applications.

Programmers can arbitrarily assign I/O pins to input and output functions only if the total does not exceed 32, which is insufficient for applications with a very large number of input variables. One way to expand the number of inputs is with a technique similar to multiplexed-keyboard scanning.

Figure 19 shows a block diagram for a moderately complex programmable industrial controller with the following characteristics:

- 64 input variable sensors:
- 12 output signals:
- Combinational and sequential logic computations:
- Remote operation with communications to a host processor via a high-speed full-duplex serial link:
- Two prioritized external interrupts:
- Internal real-time and time-of-day clocks.

While many microprocessors could be programmed to provide these capabilities with assorted peripheral support chips, an 8051 microcomputer needs no other integrated circuits!

The 64 input sensors are logically arranged as an 8x8 matrix. The pins of Port 1 sequentially enable each column of the sensor matrix; as each is enabled Port 0 reads in the state of each sensor in that column. An eight-byte block in bit-addressable RAM remembers the data as it is read in so that after each complete scan cycle there is an internal map of the current state of all sensors. Logic functions can then directly address the elements of the bit map.
Figure 19. Block Diagram of 64-Input Machine Controller

The computer’s serial port is configured as a nine-bit UART, transferring data at 17,000 bytes-per-second. The ninth bit may distinguish between address and data bytes.

The 8051 serial port can be configured to detect bytes with the address bit set, automatically ignoring all others. Pins INT0 and INT1 are interrupts configured respectively as high-priority, falling-edge triggered and low-priority, low-level triggered. The remaining 12 I/O pins output TTL-level control signals to 12 actuators.

There are several ways to implement the sensor matrix circuitry, all logically similar. Figure 20a shows one possibility. Each of the 64 sensors consists of a pair of simple switch contacts in series with a diode to permit multiple contact closures throughout the matrix.

The scan lines from Port 1 provide eight un-encoded active-high scan signals for enabling columns of the matrix. The return lines on rows where a contact is closed are pulled high and read as logic ones. Open return lines are pulled to ground by one of the 40 kΩ resistors and are read as zeros. (The resistor values must be chosen to ensure all return lines are pulled above the 2.0V logic threshold, even in the worst-case.
where all contacts in an enabled column are closed.) Since P0 is provided open-collector outputs and high-impedance MOS inputs its input loading may be considered negligible.

The circuits in Figures 20b–20d are variations on this theme. When input signals must be electrically isolated from the computer circuitry as in noisy industrial environments, phototransistors can replace the switch diode pairs and provide optical isolation as in Figure 20b. Additional opto-isolators could also be used on the control output and special signal lines.

The other circuits assume that input signals are already at TTL levels. Figure 20c uses octal three-state buffers enabled by active-low scan signals to gate eight signals onto Port 0. Port 0 is available for memory expansion or peripheral chip interfacing between sensor matrix scans. Eight-to-one multiplexers in Figure 20d select one of eight inputs for each return line as determined by encoded address bits output on three pins of Port 1. (Five more output pins are thus freed for more control functions.) Each output can drive at least one standard TTL or up to 10 low-power TTL loads without additional buffering.

Going back to the original matrix circuit, Figure 21 shows the method used to scan the sensor matrix. Two complete bit maps are maintained in the bit-addressable region of the RAM: one for the current state and one for the previous state read for each sensor. If the need arises, the program could then sense input transitions and/or debounce contact closures by comparing each bit with its earlier value.

The code in Example 3 implements the scanning algorithm for the circuits in Figure 20a. Each column is enabled by setting a single bit in a field of zeroes. The bit maps are positive logic: ones represent contacts that are closed or isolators turned on.

Example 3.

```assembly
Example 3.
INPUT_SCAN: ;SUBROUTINE TO READ
;CURRENT STATE
;OF 64 SENSORS AND
;SAVE IN RAM 20H-27H

MOV R0,#20H ;INITIALIZE
;POINTERS

MOV R1,#28H ;FOR BIT MAP
;BASES

MOV A,#80H ;SET FIRST BIT
;IN ACC

SCAN: MOV P1,A ;OUTPUT TO SCAN
;LINES

RR A ;SHIFT TO ENABLE
;NEXT COLUMN

MOV R2,A ;REMEMBER CUR-
;RENT SCAN
;POSITION

MOV A,P0 ;READ RETURN
;LINES

XCH A,@R0 ;SWITCH WITH
;PREVIOUS MAP
;BITS

MOV @R1,A ;SAVE PREVIOUS
;STATE AS WELL

INC R0 ;BUMP POINTERS

INC R1

MOV A,R2 ;RELOAD SCAN
;LINE MASK

JNB ACC,7;SCAN: ;LOOP UNTIL ALL
;EIGHT COLUMNS
;READ

RET
```
a.) Using Switch Contact/Diode Matrix

Figure 20. Sensor Matrix Implementation Methods
b.) Using Optically-Coupled Isolators

Figure 20. Sensor Matrix Implementation Methods (Continued)
c.) Using TTL Three-State Buffers

Figure 20. Sensor Matrix Implementation Methods (Continued)
d.) Using TTL Data Selectors

Figure 20. Sensor Matrix Implementation Methods (Continued)
What happens after the sensors have been scanned depends on the individual application. Rather than inventing some artificial design problem, software corresponding to commonplace logic elements will be discussed.

**Combinatorial Output Variables.** An output variable which is a simple (or not so simple) combinational function of several input variables is computed in the spirit of Design Example 3. All 64 inputs are represented in the bit maps: in fact, the sensor numbers in Figure 20 correspond to the absolute bit addresses in RAM! The code in Example 4 activates an actuator connected to P2.2 when sensors 12, 23, and 34 are closed and sensors 45 and 56 are open.

**Example 4.**
Simple Combinatorial Output Variables.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>;SET P2.2=(12) (23) (34) ( 45) ( 56)</td>
<td>Sets P2.2 to the status of sensors 12, 23, 34, 45, and 56</td>
</tr>
<tr>
<td>MOV C,12</td>
<td>Sets C to the value of sensor 12</td>
</tr>
<tr>
<td>ANL C,23</td>
<td>ANDs C with the value of sensor 23</td>
</tr>
<tr>
<td>ANL C,34</td>
<td>ANDs C with the value of sensor 34</td>
</tr>
<tr>
<td>ANL C,45</td>
<td>ANDs C with the value of sensor 45</td>
</tr>
<tr>
<td>ANL C,56</td>
<td>ANDs C with the value of sensor 56</td>
</tr>
<tr>
<td>MOV P2.2,C</td>
<td>Sets P2.2 to the value of C</td>
</tr>
</tbody>
</table>

**Intermediate Variables.** The examination of a typical relay-logic ladder diagram will show that many of the rungs control not outputs but rather relays whose contacts figure into the computation of other functions. In effect, these relays indicate the state of intermediate variables of a computation.

The MCS-51 solution can use any directly addressable bit for the storage of such intermediate variables. Even when all 128 bits of the RAM array are dedicated (to input bit maps in this example), the accumulator, PSW, and B register provide 18 additional flags for intermediate variables.

For example, suppose switches 0 through 3 control a safety interlock system. Closing any of them should deactivate certain outputs. Figure 22 is a ladder diagram for this situation. The interlock function could be recomputed for every output affected, or it may be computed once and save (as implied by the diagram). As the program proceeds this bit can qualify each output.
Example 5. Incorporating Override signal into actuator outputs.

```
; CALL INPUT_SCAN
MOV C,0
ORL C,1
ORL C,2
ORL C,3
MOV F0,C
; .... ....
; COMPUTE FUNCTION 0
:
ANL C, F0
MOV P0,C
; .... ....
; COMPUTE FUNCTION 1
:
ANL C, F0
MOV P1,1,C
; .... ....
; COMPUTE FUNCTION 2
:
ANL C, F0
MOV P1,2,C
; .... ....
```

Latching Relays. A latching relay can be forced into either the ON or OFF state by two corresponding input signals, where it will remain until forced onto the opposite state—alogous to a TTL Set/Reset flip-flop. The relay is used as an intermediate variable for other calculations. In the previous example, the emergency condition could be remembered and remain active until an “emergency cleared” button is pressed.

Any flag or addressable bit may represent a latching relay with a few lines of code (see Example 6).

Example 6. Simulating a latching relay.

```
;I_SET SET FLAG 0 IF C=1
I_SET: ORL C,F0
       MOV F0,C
;
;I_RST RESET FLAG 0 IF C=1
I_RST: CPS C
       ANL C,F0
       MOV F0,C
;
```

Time Delay Relays. A time delay relay does not respond to an input signal until it has been present (or absent) for some predefined time. For example, a ballast or load resistor may be switched in series with a D.C. motor when it is first turned on, and shunted from the circuit after one second. This sort of time delay may be simulated by an interrupt routine driven by one of the two 8051 timer counters. The procedure followed by the routine depends heavily on the details of the exact function needed: time-outs or time delays with resettable or non-resettable inputs are possible. If the interrupt routine is executed every 10 milliseconds the code in Example 7 will clear an intermediate variable set by the background program after it has been active for two seconds.

Example 7. Code to clear USRFLG after a fixed time delay.

```
JNB USR_FLG,NXTTST
DJNZ DLAY_COUNT,NXTTST
CLR USR_FLG
MOV DLAY_COUNT,#200
NXTTST: ;.; ....
```
**Serial Interface to Remote Processor.** When it detects emergency conditions represented by certain input combinations (such as the earlier Emergency Override), the controller could shut down the machine immediately and/or alert the host processor via the serial port. Code bytes indicating the nature of the problem could be transmitted to a central computer. In fact, at 17,000 bytes-per-second, the entire contents of both bit maps could be sent to the host processor for further analysis in less than a millisecond! If the host decides that conditions warrant, it could alert other remote processors in the system that a problem exists and specify which shut-down sequence each should initiate. For more information on using the serial port, consult the MCS-51 User’s Manual.

**Response Timing**

One difference between relay and programmed industrial controllers (when each is considered as a “black box”) is their respective reaction times to input changes. As reflected by a ladder diagram, relay systems contain a large number of “rungs” operating in parallel. A change in input conditions will begin propagating through the system immediately, possibly affecting the output state within milliseconds.

Software, on the other hand, operates sequentially. A change in input states will not be detected until the next time an input scan is performed, and will not affect the outputs until that section of the program is reached. For that reason the raw speed of computing the logical functions is of extreme importance.

Here the Boolean processor pays off. Every instruction mentioned in this Note completes in one or two microseconds—the minimum instruction execution time for many other microcontrollers! A ladder diagram containing a hundred rungs, with an average of four contacts per rung can be replaced by approximately five hundred lines of software. A complete pass through the entire matrix scanning routine and all computations would require about a millisecond: less than the time it takes for most relays to change state.

A programmed controller which simulates each Boolean function with a subroutine would be less efficient by at least an order of magnitude. Extra software is needed for the simulation routines, and each step takes longer to execute for three reasons: several byte-wide logical instructions are executed per user program step (rather than one Boolean operation); most of those instructions take longer to execute with microprocessors performing multiple off-chip accesses; and calling and returning from the various subroutines requires overhead for stack operations.

In fact, the speed of the Boolean Processor solution is likely to be much faster than the system requires. The CPU might use the time left over to compute feedback parameters, collect and analyze execution statistics, perform system diagnostics, and so forth.

**Additional Functions and Uses**

With the building-block basics mentioned above many more operations may be synthesized by short instruction sequences.

**Exclusive-OR.** There are no common mechanical devices or relays analogous to the Exclusive-OR operation, so this instruction was omitted from the Boolean Processor. However, the Exclusive-OR or Exclusive-NOR operation may be performed in two instructions by conditionally complementing the carry or a Boolean variable based on the state of any other testable bit.

```plaintext
;EXCLUSIVE-OR FUNCTION IMPOSED ON CARRY
;USING F0 IS INPUT VARIABLE.
:XOR_F0: JNB F0,XORCNT ;("JB" FOR X-NOR)
    CPL C
:XORCNT: ...
```

**XCH.** The contents of the carry and some other bit may be exchanged (switched) by using the accumulator as temporary storage. Bits can be moved into and out of the accumulator simultaneously using the Rotate-
through-carry instructions, though this would alter the accumulator data.

```
;EXCHANGE CARRY WITH USRFLG
XCHBIT: RLC A
        MOV C,USR_FLAG
        RRC A
        MOV USR_FLAG,C
        RLC A
```

Extended Bit Addressing. The 8051 can directly address 144 general-purpose bits for all instructions in Figure 3b. Similar operations may be extended to any bit anywhere on the chip with some loss of efficiency.

The logical operations AND, OR, and Exclusive-OR are performed on byte variables using six different addressing modes, one of which lets the source be an immediate mask, and the destination any directly addressable byte. Any bit may thus be set, cleared, or complemented with a three-byte, two-cycle instruction if the mask has all bits but one set or cleared.

Byte variables, registers, and indirectly addressed RAM may be moved to a bit addressable register (usually the accumulator) in one instruction. Once transferred, the bits may be tested with a conditional jump, allowing any bit to be polled in 3 microseconds—still much faster than most architectures—of logical calculations. (This technique can also simulate additional bit addressing modes with byte operations.)

Parity of bytes or bits. The parity of the current accumulator contents is always available in the PSW, from whence it may be moved to the carry and further processed. Error-correcting Hamming codes and similar applications require computing parity on groups of isolated bits. This can be done by conditionally complementing the carry flag based on those bits or by gathering the bits into the accumulator (as shown in the DES example) and then testing the parallel parity flag.

Multiple byte shift and CRC codes

Though the 8051 serial port can accommodate eight- or nine-bit data transmissions, some protocols involve much longer bit streams. The algorithms presented in Design Example 2 can be extended quite readily to 16 or more bits by using multi-byte input and output buffers.

Many mass data storage peripherals and serial communications protocols include Cyclic Redundancy (CRC) codes to verify data integrity. The function is generally computed serially by hardware using shift registers and Exclusive-OR gates, but it can be done with software. As each bit is received into the carry, appropriate bits in the multi-byte data buffer are conditionally complemented based on the incoming data bit. When finished, the CRC register contents may be checked for zero by ORing the two bytes in the accumulator.

4.0 SUMMARY

A truly unique facet of the Intel MCS-51 microcomputer family design is the collection of features optimized for the one-bit operations so often desired in real-world, real-time control applications. Included are 17 special instructions, a Boolean accumulator, implicit and direct addressing modes, program and mass data storage, and many I/O options. These are the world's first single-chip microcomputers able to efficiently manipulate, operate on, and transfer either bytes or individual bits as data.

This Application Note has detailed the information needed by a microcomputer system designer to make full use of these capabilities. Five design examples were used to contrast the solutions allowed by the 8051 and those required by previous architectures. Depending on the individual application, the 8051 solution will be easier to design, more reliable to implement, debug, and verify, use less program memory, and run up to an order of magnitude faster than the same function implemented on previous digital computer architectures.

Combining byte- and bit-handling capabilities in a single microcomputer has a strong synergistic effect: the power of the result exceeds the power of byte- and bit-processors laboring individually. Virtually all user applications will benefit in some way from this duality.

Data intensive applications will use bit addressing for test pin monitoring or program control flags; control applications will use byte manipulation for parallel I/O expansion or arithmetic calculations.

It is hoped that these design examples give the reader an appreciation of these unique features and suggest ways to exploit them in his or her own application.
APPENDIX A

Automobile Turn-Indicator Controller Program Listing

IS18-II MCS-51 MACRO ASSEMBLER V1.0
OBJECT MODULE PLACED IN FD AP70 HEX
ASSEMBLER INVOKED BY: f1.asm51 ap70 src date(328)

1 *REF TITLE(AP-70 APPENDIX)
2 ***********************************************
3 4 THE FOLLOWING PROGRAM USES THE BOOLEAN INSTRUCTION SET
5 6 OF THE INTEL 8051 MICROCOMPUTER TO PERFORM A NUMBER OF
7 8 AUTOMOTIVE DASHBOARD CONTROL FUNCTIONS RELATING TO
9 10 TURN SIGNAL CONTROL, EMERGENCY BLINKERS, BRAKE LIGHT
11 12 CONTROL, AND PARKING LIGHT OPERATION.
13 14 THE ALGORITHMS AND HARDWARE ARE DESCRIBED IN DESIGN
15 16 "USING THE INTEL MCS-51(TM)
17 18 BOOLEAN PROCESSING CAPABILITIES"
19
20 INPUT PIN DECLARATIONS
21 (ALL INPUTS ARE POSITIVE-TRUE LOGIC
22 INPUTS ARE HIGH WHEN RESPECTIVE SWITCH CONTACT IS CLOSED)
23
24 BRAKE BIT P1 0 ; BRAKE PEDAL DEPRESSED
25 EMERG BIT P1 1 ; EMERGENCY BLINKER ACTIVATED
26 PARK BIT P1 2 ; PARKING LIGHTS ON
27 L_TURN BIT P1 3 ; TURN LEVER DOWN
28 R_TURN BIT P1 4 ; TURN LEVER UP
29
30 OUTPUT PIN DECLARATIONS
31 (ALL OUTPUTS ARE POSITIVE TRUE LOGIC
32 BULB IS TURNED ON WHEN OUTPUT PIN IS HIGH)
33
34 LPRINT BIT P1 5 ; FRONT LEFT-TURN INDICATOR
35 FRONT BIT P1 6 ; FRONT RIGHT-TURN INDICATOR
36 DASH BIT P1 7 ; DASHBOARD LEFT-TURN INDICATOR
37 R_DASH BIT P2 0 ; DASHBOARD RIGHT-TURN INDICATOR
38 L_REAR BIT P2 1 ; REAR LEFT-TURN INDICATOR
39 R_REAR BIT P2 2 ; REAR RIGHT-TURN INDICATOR
40
41 S_FAIL BIT P2 3 ; ELECTRICAL SYSTEM FAULT INDICATOR
42
43 INTERNAL VARIABLE DEFINITIONS
44
45 SUB_DIV DATA 20H ; INTERRUPT RATE SUBDIVIDER
46 HI_FREQ BIT SUB_DIV 0 ; HIGH-FREQUENCY OSCILLATOR BIT
47 LO_FREQ BIT SUB_DIV 7 ; LOW-FREQUENCY OSCILLATOR BIT
48
49 DIM BIT PSW 1 ; PARKING LIGHTS ON FLAG
50
51 $EJECT

203830-26
LOC OBJ  LINE SOURCE

0000 020040 49  ORG O000H  .:  RESET VECTOR
0000 020040 50  1, JMP INIT
0008 020048 52  ORG O008H  .:  TIMER 0 SERVICE VECTOR
0008 73BCF0 53  MOV TH0,#16  .:  HIGH TIMER BYTE ADJUSTED TO CONTROL INT RATE
0008 C000 54  MOV PSH #16  .:  EXECUTE CODE TO SAVE ANY REGISTERS USED BELOW
0010 0154 55  JMP UPDATE  .: (CONTINUE WITH REST OF ROUTINE)
0040 020070 57  ORG O040H
0040 75BA00 58  INIT  MOV TLO,#0  .:  ZERO LOADED INTO LOW-ORDER BYTE AND
0040 75BCF0 59  MOV TH0,#16  .:  #16 IN HIGH-ORDER BYTE GIVES # MSSEC PERIOD
0046 758961 60  MOV TMDD,#010000B  .:  8-BIT AUTO RELOAD COUNTER MODE FOR TIMER 1.
0046 7520F4 61  MOV SUB_DIV,#244  .:  16-BIT TIMER MODE FOR TIMER 0 SELECTED
0046 759204 62  MOV SUB_DIV,#244  .:  SUBDIVIDE INTERRUPT RATE BY #244 FOR #244
004F D2AF 63  SETB ETO  .:  USE TIMER 0 OVERFLOWS TO INTERRUPT PROGRAM
004F D2AF 64  SETB EA  .:  CONFIGURE IE TO GLOBALY ENABLE INTERRUPTS
0050 D2BC 65  SETB TR0  .:  KEEP INSTRUCTION CYCLE COUNT UNTIL OVERFLOW
0052 80FE 66  SJMP $  .:  START BACKGROUND PROGRAM EXECUTION
0054 D5203B 67  DCBj  .:  SUB_DIV, 8ifers  .:  EXECUTE SYSTEM TEST ONLY ONCE PER SECOND
0057 7520F4 68  DCBj  .:  SUB_DIV, 8ifers  .:  GET VALUE FOR NEXT ONE SECOND DELAY AND
0057 7520F4 69  DCBj  .:  SUB_DIV, 8ifers  .:  GO THROUGH ELECTRICAL SYSTEM TEST CODE:
005A 439063 70  ORL P1,#1110000B  .:  SET CONTROL OUTPUTS HIGH
005B 43A007 71  ORL P2,#00000011B
0060 C295 72  CLR  L_FRNRT  .:  FLOAT DRIVE COLLECTOR
0062 208420 73  JB TO_FAULT  .:  TO SHOULD BE PULLED LOW
0065 D295 74  SETB L_FRNRT  .:  FULL COLLECTOR BACK DOWN
0067 C297 75  CLR  L_DASH  .:  REPEAT SEQUENCE FOR L_DASH.
0069 208421 76  JB TO_FAULT  .:  SETB L_DASH
006C D297 77  CLR  L_REAR  .:  L_REAR.
006E C2A1 78  JB TO_FAULT  .:  SETB L_REAR
0070 20841A 79  CLR  R_FRNRT  .:  R_FRNRT.
0072 D2A1 80  JB TO_FAULT  .:  SETB R_FRNRT
0075 C296 81  CLR  R_DASH  .:  R_DASH.
0077 208413 82  JB TO_FAULT  .:  SETB R_DASH
007A D2A6 83  SETB R_FRNRT  .:  AND R_REAR
007C D2A0 84  CLR  R_DASH  .:  R_REAR
007E 20840C 85  JB TO_FAULT  .:  SETB R_DASH
0081 D2A0 86  SETB R_DASH
0083 C2A2 87  CLR  R_REAR  .:  AND R_REAR
0085 208405 88  JB TO_FAULT  .:  SETB R_REAR
0088 D2A2 89  SETB R_REAR
008A 208402 90  JB TO_FAULT  .:  WITH ALL COLLECTORS GROUNDED, TO SHOULD BE HIGH
008D 72A0 91  JC NO  .:  IF NO, CONTINUE WITH INTERRUPT ROUTINE
008D 208A02 92  JB TO_FAULT  .:  ELECTRICAL FAILURE PROCESSING ROUTINE
008D 208A3 93  FAULT CPL S_FAIL  .:  TOGGLE INDICATOR ONCE PER SECOND
008F +1 $EJECT

AP-70
203830-27
LOC OBJ LINE SOURCE
100 ; CONTINUE WITH INTERRUPT PROCESSING
101 ;
102 ; 1) COMPUTE LOW BULB INTENSITY WHEN PARKING LIGHTS ARE ON.
103 ;
008F A201 104 TOSERV MOV C.SUB_DIV 1 ; START WITH 50 PERCENT.
0091 B200 105 ANL C.SUB_DIV 0 ; MASK DOWN TO 25 PERCENT.
0093 7202 106 ORL C.SUB_DIV 2 ; BUILD BACK TO 62.5 PERCENT.
0095 8292 107 ANL C.PARK ; GATE WITH PARKING LIGHT SWITCH.
0097 92D1 108 MOV DIM.C ; AND SAVE IN TEMP. VARIABLE
109 ;
110 ; 2) COMPUTE AND OUTPUT LEFT-HAND DASHBOARD INDICATOR.
111 ;
0099 A293 112 MOV C.L.TURN ; SET CARRY IF TURN
009B 7291 113 ORL C.EMERG ; OR EMERGENCY SELECTED.
009D B207 114 ANL C.LD_FREQ ; IF GD. GATE IN 1 Hz SIGNAL
009F 9297 115 MOV L_DASH.C ; AND OUTPUT TO DASHBOARD
116 ;
117 ; 3) COMPUTE AND OUTPUT LEFT-HAND FRONT TURN SIGNAL.
118 ;
00A1 92D5 119 MOV FO.C ; SAVE FUNCTION SO FAR
00A3 72D1 120 ORL C.DIM ; ADD IN PARKING LIGHT FUNCTION
00A5 9295 121 MOV L.FRNT.C ; AND OUTPUT TO TURN SIGNAL
122 ;
123 ; 4) COMPUTE AND OUTPUT LEFT-HAND REAR TURN SIGNAL.
124 ;
00A7 A290 125 MOV C.BRAKE ; GATE BRAKE PEDAL SWITCH
00AA B293 126 ANL C.L.TURN ; WITH TURN LEVER
00AB 72D5 127 ORL C.F0 ; INCLUDE TEMP. VARIABLE FROM DASH
00AD 72D1 128 ORL C.DIM ; AND PARKING LIGHT FUNCTION
00AF 92A1 129 MOV L.REAR.C ; AND OUTPUT TO TURN SIGNAL
130 ;
131 ; 5) REPEAT ALL OF ABOVE FOR RIGHT-HAND COUNTERPARTS.
132 ;
00B1 A294 133 MOV C.R.TURN ; SET CARRY IF TURN
00B3 7291 134 ORL C.EMERG ; OR EMERGENCY SELECTED.
00B5 B207 135 ANL C.LD_FREQ ; IF GD. GATE IN 1 Hz SIGNAL
00B7 92D0 136 MOV R_DASH.C ; AND OUTPUT TO DASHBOARD
00B9 92D5 137 MOV FO.C ; SAVE FUNCTION SO FAR
00BB 72D1 138 ORL C.DIM ; ADD IN PARKING LIGHT FUNCTION
00BF 9296 139 MOV R.FRNT.C ; AND OUTPUT TO TURN SIGNAL
008F A290 140 MOV C.BRAKE ; GATE BRAKE PEDAL SWITCH
00C1 B044 141 ANL C.R.TURN ; WITH TURN LEVER
00C3 72D5 142 ORL C.F0 ; INCLUDE TEMP. VARIABLE FROM DASH
00C5 72D1 143 ORL C.DIM ; AND PARKING LIGHT FUNCTION
00C7 92A2 144 MOV R.REAR.C ; AND OUTPUT TO TURN SIGNAL
145 ;
146 ; RESTORE STATUS REGISTER AND RETURN
147 ;
00C9 D060 148 POP PSW ; RESTORE PSW
00C0 32 149 RETI ; AND RETURN FROM INTERRUPT ROUTINE
150 ;
151 ; END