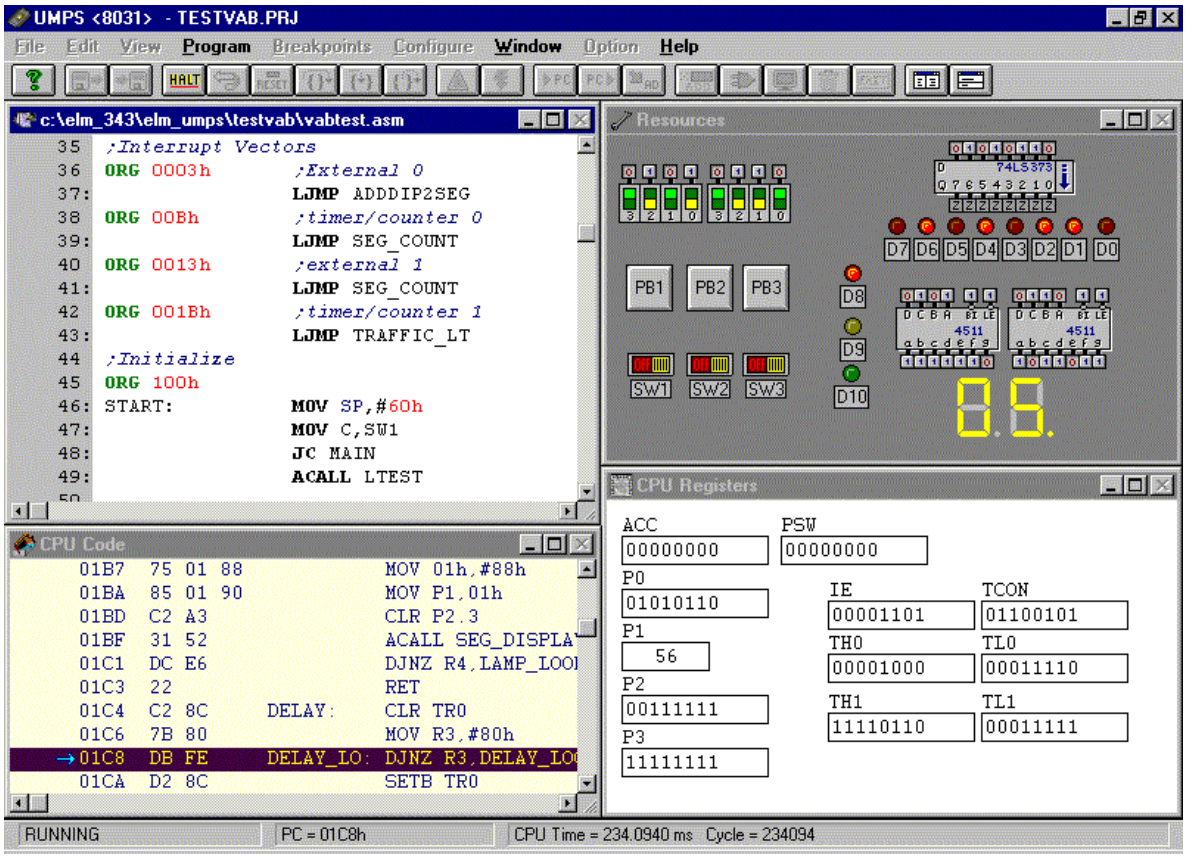
Application of Microcontrollers Manual
Part II - Intel 8051 and UMPS®
Version 2.1



Electronics Management
Department of Information Management Systems
Office of Off-Campus Academic Programs
College of Applied Sciences and Arts
Southern Illinois University Carbondale

Discussion For Our Non-SIU Users

The *Application of Microcontrollers* Manual and Labs were designed for off-campus students at military bases. During a 16-week period, the students take 3 lecture courses, each presented over 3 weekends. The first 2 courses cover analog and digital principles. The final course is on microcontrollers using the 8051 as an example.

The students independently perform the first part of this manual and the labs during the first 2 courses. Part I introduces controllers using the BASIC Stamp from Parallax, Inc. The students are provided the equipment to program and interface a microcontroller using a relatively simple language: BASIC. Part I is intended to reinforce the lecture material on basic electronics principles using the microcontroller. It may be downloaded from Parallax's education website at: http://www.stampsinclass.com

Part II, this material, is performed by the students while participating in their final lecture course covering the 8051. In this part students learn principles of programming a microcontroller using Assembler. UMPS from Virtual Micro Design is used to simulate the 8051 allowing students to see how instructions affect registers, memory and devices. A Virtual Activity Board with I/O was designed with the UMPS resources. UMPS projects were designed for the material allowing student to quickly load samples while reading about them.

We thank Philippe Techer for his generosity in allowing use to distribute UMPS demo versions to our students, use of the logos in our documents, and for developing such a wonderful development AND learning tool.

We hope you find the material developed useful. Please feel free to contact us. We enjoy feedback and hearing how our material is being used.

Martin Hebel
Southern Illinois University Carbondale

Application of Microcontrollers

Contact Information

E-mail:

Primary developer of the manual and labs:
Martin Hebel ..... mhebel@siu.edu

Contributing developer:
Will Devenport ..... willd@siu.edu

Director, Off-Campus Academic Programs:
Dr. Terry Bowman ..... tbowman@siu.edu

Chair, Department of Information Management Systems:
Dr. Janice Schoen Henry ..... jshenry@siu.edu

Mailing:

Electronics Management
College of Applied Sciences and Arts
Southern Illinois University, Carbondale
Carbondale, IL 62901-6614

The following people are thanked for their contributions: Ken Gracey and the gang at Parallax for their work helping make this possible for our students; Philippe Techer at Virtual Micro Design for designing a great simulation package and working with us; Myke Predko for his feedback and recommendations; I. Scott MacKenzie for a concise text on the 8051; students on campus and at Ft. Gordon, Cherry Point and New River for feedback; Cheri Barral for editing; and finally Terry Bowman and Jan Henry for budgeting the endeavor and wanting the best education for our students.

Key Web Sites:
Electronics Management Home Page: ..... www.siu.edu/~imsasa/elm

Off-Campus Programs Home Page: http://131.230.64.6/

Parallax Incorporated Home Page: ..... www.parallaxinc.com
..... www.stampsinclass.com

Virtual Micro Design Home Page (UMPS): ..... www.vmdesign.com

Distributors & Additional Information:
Digi-Key Electronics - Stamps, components ..... www.digikey.com

Jameco Electronics - Stamps, components ..... www.jameco.com

JDR Electronics - Stamps, components ..... www.jdr.com

Wirz Electronics - UMPS U.S. Sales..... www.wirz.com

Peter H. Anderson - General microcontroller information ... www.phanderson.com

SelmaWare Solutions - Specialized interfacing software ..... www.selmaware.com

Texts:
The 8051 Microcontroller, 3rd ed. 1999, Scott MacKenzie. Prentice-Hall
ISBN: 0-13-780008-8

Handbook of Microcontrollers. 1999, Myke Predko. McGraw-Hill
ISBN: 0-07-913716-4

Programming and Customizing the 8051 Microcontroller. 1999, Myke Predko. McGraw-Hill.
ISBN: 0-07-134192-7

The Microcontroller Idea Book. 1994, Jan Axelson. Lakeview Research.
ISBN: 096508190-7

Table of Contents

Section I: Introduction to the 8051

Reference:
A. MacKenzie, I.S., 1999. The 8051 microcontroller, 3rd ed. Prentice Hall.

Objectives:
1) Discuss the relationship between mnemonics and machine code.
2) List the features of the Intel 8051 including memory and ports.
3) Identify the function of opcodes and operands.
4) Use the 8051 RAM memory map in programming.
5) Discuss key Special Function Registers including the Accumulator, Program Status Word, Stack Pointer, Program Counter.
6) Use UMPS for CPU simulations.

The final sections of this manual concern programming microcontrollers in the most fundamental methods available. The BASIC Stamp II was programmed in PBASIC2. This is considered a fairly high level language. PBASIC2, and most BASIC languages, are interpreted languages. Code is written in pseudo-English code, and the interpreter performs numerous (sometimes hundreds) machine instructions to accomplish a single BASIC instruction.

As we saw with the BS2, there is a very finite amount of memory available in RAM and ROM to perform the operations that we desire. In fact, the entire BS2 ROM was used just to hold the PBASIC2 interpreter, and an external EEPROM was used to hold our PBASIC2 programs. Also, PBASIC2 is a very slow executing language. The interpreter must perform many machine instructions for a single command.

By writing programs directly in machine code, in symbolic mnemonics, or a low level Assembler language, we can increase program execution speed many fold and utilize less memory than a high level language. The drawback is that the code can be very cryptic to read and seemingly simple tasks may take dozens of machine instructions to accomplish. Additionally, each family of microcontrollers and microprocessors has a very unique instruction set that makes it impossible to move machine programs from one family to another without re-writing it.

Writing instructions in machine code means coding the data in the language of digital systems, which are 1's and 0's. To make things a little simpler, hexadecimal is normally used. An instruction to add 1 to the accumulator for the 8051 would look like the following in the ROM memory map:

24 01 (in hexadecimal)

Luckily, there are other methods. The instructions can be written in mnemonics, and software will assemble it directly into machine code. The instruction to add 1 to the accumulator would be:

ADD A, #01.

This code would be converted directly to '24 01'. Writing code in mnemonics is easier but can still be very tedious. Assembler language, a low level programming language, is the next step up. While the majority of the code is still written in machine code mnemonics, there exist methods to use constants (called symbols) and the means to simplify coding.

When programming the BS2 in PBASIC2, we used an application program (Stampw.exe) to edit our code. Stampw.exe then tokenized our code and transferred it to the BS2. From that point, with the exception of debugging data, the computer was no longer required. The PBASIC2 program on the PIC16C57 handled communications with the PC to accept the program being downloaded and to transfer it to EEPROM.

Normally, a special piece of hardware is required to program microcontrollers such as the PIC16C57 and the Intel 8051. If you are familiar with EPROM programming devices, it is very similar. The microcontroller is set in a latchable socket (ZIF socket), and the programming device handles accepting data from the computer and *burning it* into the microcontroller in much the same manner an EPROM or PROM is programmed. The binary machine code is transferred from the computer to the ROM space on the controller. Once programmed, the controller is placed in a circuit comprised of supporting interfacing electronics components. When power is applied, the microcontroller will begin reading ROM and executing the machine code instructions.

With the BS2 we had the benefit of an activity board to allow us to write programs that communicated with input and output devices. We could also have programmed the BS2 in the Activity Board, removed it and used it in a specialized circuit. Typically, any interfacing circuits will be of special design for the intended use of the controller.

It would be difficult to provide students with everything needed to program a microcontroller directly and test it in a circuit for independent study. Luckily, there exist simulation programs for this. The one we will be using is UMPS from Virtual Micro Design. This package allows a PC to simulate a microcontroller for programming and operation. Programs such as UMPS are not just for training purposes. Professional programmers simulate programs to test and debug them. Programming in machine code can be cumbersome, and there are dozens of registers to keep track of. A good simulation program can give the programmer insights into what is happening internally with the controller and allow correct of bugs prior to burning the program into an expensive IC.

While a license for a simulation package can cost hundreds to thousands of dollars, UMPS allows free distribution of a demonstration version with only limited reduced functionality. The associated labs outline these limitations. UMP also has a unique feature of allowing simulated devices (resources) to be connected to the microcontrollers.

In Part II of the manual we will look at machine code, mnemonics, assembler and various fundamentals of microcontroller programming using the 8051. UMPS will be used to explore coding the 8051 in both mnemonics and assembler to read and control simulated input and output devices called resources.

The 8051 Microcontroller & UMPS

In these sections we will be discussing a very popular microcontroller series, the Intel 8051. While it comes in different styles, such as the 8052, 8032 and so on, our discussion will focus on the 8051. This microcontroller has 4K bytes of internal ROM and 256 bytes internal RAM, 128 bytes of which are accessible for programming needs.

Figure I-1 shows the pinouts of the 8051. The majority of the pins are labeled PN.b, where 'N' is a port number (a register byte) and 'b' is the bit number in the port byte. For example P1.5: this indicates bit 5 on port 1. These are comparable to P0-P15 on theBS2. Many pins have additional names in parenthesis. Microcontroller I/O pins often have special dedicated functions. For example, P3.0 (RXD) is a special function pin that can be configured to capture incoming serial data in the RS-232 format. The 8051 can also be configured to work with external RAM and ROM. The pins with labels of 'AD' are used as address and data lines for external memory. The pins with

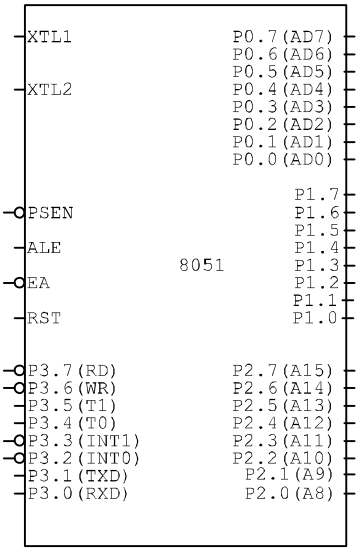


Figure I-1: The 8051 Microcontroller

only 'A' are only address lines. Since there are a total of 16 address lines, how much external memory could the processor address? Well, 2^{16} = 65536 bytes (8 data lines = 1 byte) of data.

UMPS will be used to simulate operation of the 8051. Figure I-2 is the Virtual Activity Board (VAB) designed for this manual using the UMPS component resources. Figure I-3 is the schematic representation of the VAB. Note that all switches bring the inputs LOW to ground and that the switches are debounced by one-shots that are not shown. All the LEDs light on a HIGH. Using latches, the data from the P1 port is multiplexed to both a set of 8 LEDs and/or two 7-segment displays, depending on the state of their enable (4511 active LOW) or the clock line (74LS373 active HIGH).

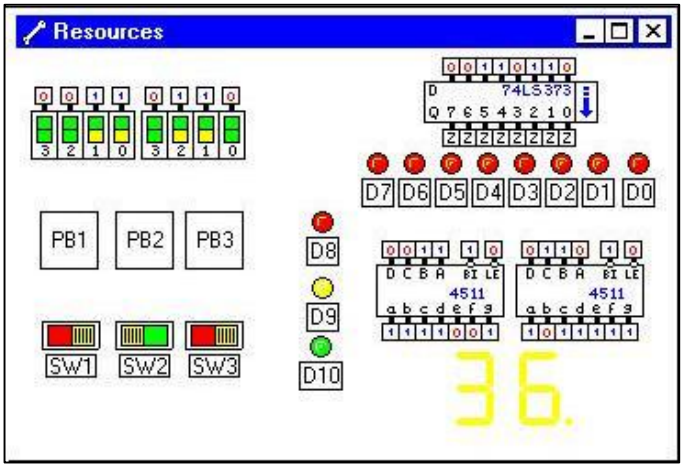


Figure I-2: Virtual Activity Board (VAB)

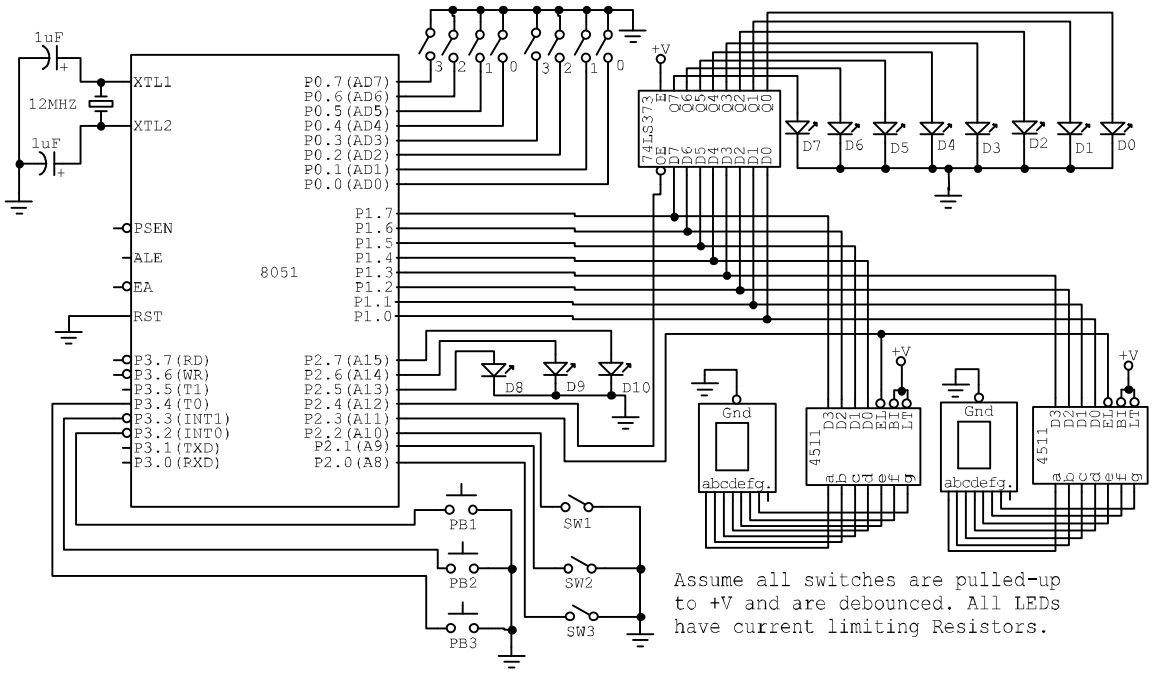


Figure I-3: Virtual Activity Board Schematic
Errata: P2.4 should connect to E on the '373 and OE should be grounded.

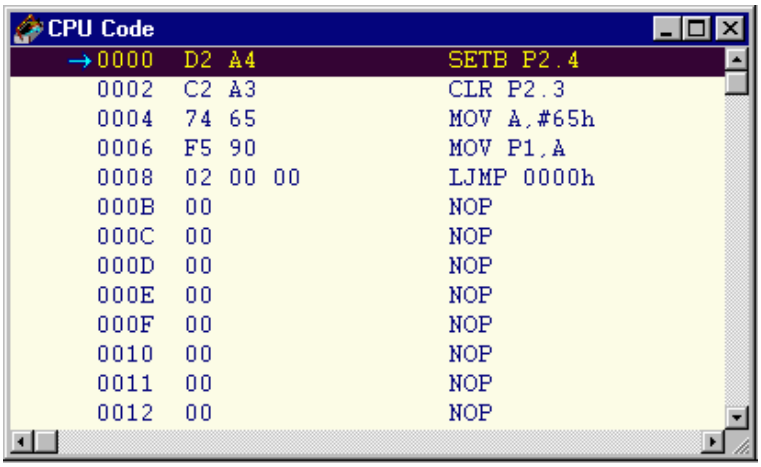
Opcodes & Operands

The most important fact that needs to be understood is that everything at machine level code is directly operating with bytes, bits, and in some cases 16-bit address words. The instructions are defined by a particular byte. The data that the instruction works with are bits, bytes or words. The instruction is known as the opcode. Each opcode has a unique byte that defines it to the microprocessor. In fact, there are many opcodes that have the same name, but have different associated bytes depending on the type of data they are working with. The data for the instruction is called the operand. This data may be a register, a bit in a register, a byte of data, or a 16-bit word address.

The second most important fact to understand is that of a register. We worked with registers on the BS2. They were used to define the direction and status of the I/O pins and to hold our data. The variables we declared were all held within these RAM registers. Microcontrollers have numerous registers (many more than microprocessors). Some of their functions include:

- Temporary storage of data.
- Math and logical operations accumulator.
- Math and logical result status.
- Control of specialized features such as interrupts, serial data transmission and timer operations.
- Holding return location pointers from subroutines calls.
- High speed access for data.

Let's take a look at a simple program. Figure I-4 is the CPU code window for UMPS proj_I-1. In this window we can directly enter mnemonic opcodes and operands which are assembled and converted directly to machine code.



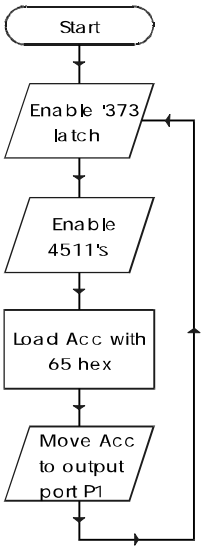


Figure I-4: Program I-1 CPU Code Window

The information in the window can be broken down as follows:

Memory Address	Machine Opcode and Operand		Mnemonic Opcode and Operand	
0000	D2	A4	SETB	P2.4
0002	C2	A3	CLR	P2.3

…. Etc

Mnemonics are entered in the right side. Upon moving to the next line, the code is assembled into machine language and updated on the left as bytes in hexadecimal. The memory location is the word of ROM in which that code is stored as bytes. In our program, the memory locations jump by 2's because most of the instructions took two bytes: one for the opcode, and one for the operand. The only exception to this is the last line of LJMP 0000h, which used 3 bytes because the operand is a 16-bit word using 2-bytes. Following the program, the rest of the memory is filled with 00's and translated as NOPs (No Operation).

8051 Tip

In PBASIC2 we prefixed a $ to indicate hex, and a % to indicate binary. The Intel standard is to suffix the number with an 'h' for hex or a 'b' for binary.

If the hex number starts with a letter (A-F), precede it with a zero. (C5h → 0C5h). UMPS requires the 0 to be used, but will strip it out upon assembly.

Figure I-5 is the ROM memory map. Compare it to the CPU code window of I-4. Can you see the same machine code in the locations specified? Let's analyze the program. Look back at Figure I-3 to see the electronics involved in our actions concerning the I/O pins.

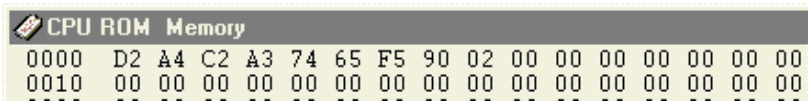


Figure I-5: Program I-1 Memory Map

OPCODE	OPERAND	Explanation
SETB	P2.4	Sets (HIGH) the specified bit, in this case I/O pin P2.4. This will enable the '373 latch for the 8 LEDs.
CLR	P2.3	Clears (LOW) the specified bit of pin P2.3 that will enable the 4511 7-segment decoder/drivers.
MOV	A, #65h	Move the number 65h into the Accumulator.
MOV	P1, A	Move the value in the Accumulator into the P1 memory location. This will set pins P1.0 - P1.7 equal to A (65h), our data display data lines.
LJMP	0000h	Make a long (16-bit address) jump to memory location 0000. The program will continue to loop.
NOP		This means 'No Operation'. It is used as fillers or for short delays (1 clock cycle).

Table I-1: Program I-1 Breakdown

Another window in the project displays certain registers and memory locations. Let's single step through the program and observe the contents of the registers affected by the program above. The program can be single stepped, one instruction at a time, by pressing F7 (Trace Into). In Table I-2 each change in a register is in bold. PC is the Program Counter. It is a 16-bit word that points to the current address in memory of the instruction *to be* executed. On power-up or reset, the PC begins executing the instructions at memory location 0000h.

Instruction	PC	ACC	P2	P1
Initial values:	0000	00000000	11111111	11111111
SETB P2.4	**0002**	00000000	111**1**1111 (was set)	11111111
CLR P2.3	**0004**	00000000	1111**0**111	11111111
MOV A, #65h	**0006**	**01100101** (65h)	11110111	11111111
MOV P1, A	**0008**	01100101	11110111	**01100101** (65h)
LJMP 0000h	**0000**	01100101	11110111	01100101

Table I-2: Program I-1 Register Values

And so it repeats. What action did it have on the VAB? The LEDs corresponding to 1's lit and 0's stayed off. Also, the 7-segment LEDs indicated the number 65. Not too bad, was it? Note that we went through an accumulator to load a byte and transfer it. Almost all instructions operate by using registers to move and manipulate data. The most important register is the Accumulator (Acc or A). It is the main arithmetic and logical register, and most math and logical operations are performed on data in the accumulator.

Look at the machine language codes for **MOV A, #65h** and **MOV P1,A** in Figure I-4. One instruction is **74 65,** and the next is **F5 90**. If the first number is the instruction code, why are they different for the same MOV instruction? It's because they *are* very different instructions**. MOV A, #65h** specified that we were moving a *data* value into the Accumulator (you can even see the 65 in the machine code). **MOV P1, A** specified a *move of the accumulator data* to I/O port 1. These are two very different tasks. We'll look more at how operations are classified depending on the operand in Section J.

Replacing **MOV A, #65h** in the third line with **MOV A, P0** (and moving down to the next line) changes the machine codes. Port 0 is now moved into the accumulator. It is now this value that is moved into P1. Referring to I-3, P0 is connected to the eight input DIP switches. Clicking the 'GO' button on the button bar allows the program to run. Changing the settings of the 8 DIP switches in the upper-left hand corner of the VAB changes the status of the 8 LEDs and the 7-segment display.

At some settings the 7-segment display shows no numbers. This occurs when the associated nibble exceeds 9. The 4511 is a *BDC* to 7-segment decoder. BCD is Binary Coded Decimal. Instead of being a binary number where the total byte is a number, the high and low order nibbles define decimal numbers. For example: 10010011 in binary would be 147 in decimal. In BCD, this digital value would equal 93. It's the same as converting it to hexadecimal, but neither nibble can exceed 9 or it would not be a valid decimal number. 7-segments refer to the 7 sections of the display which are manipulated to show our numbers.

RAM Memory Map and Registers.

Figure I-6 summarizes the 256 bytes of on-chip RAM of the 8051. The lower 128 bytes (00h-7Fh) are registers and RAM available to the programmer. Locations 00h - 1Fh are banks holding registers R0-R7 *in each*. R0 - R7 can be accessed faster than any other locations in RAM because their access is built into the instruction set for data manipulation in the same manner that accumulator was for MOV instructions. Only a single bank can be accessible at any one time. If Bank 0 (the default) is specified, R0-R7 will refer to RAM locations 00h-07h. If Bank1 is specified, R0-R7 will refer to memory locations 08h-0Fh and so on. By having four banks, there are 36 actual bytes available for fast access. Different sections of a program may utilize different banks for the memory location. The active bank is selected as part of the Program Status Word register discussed later.

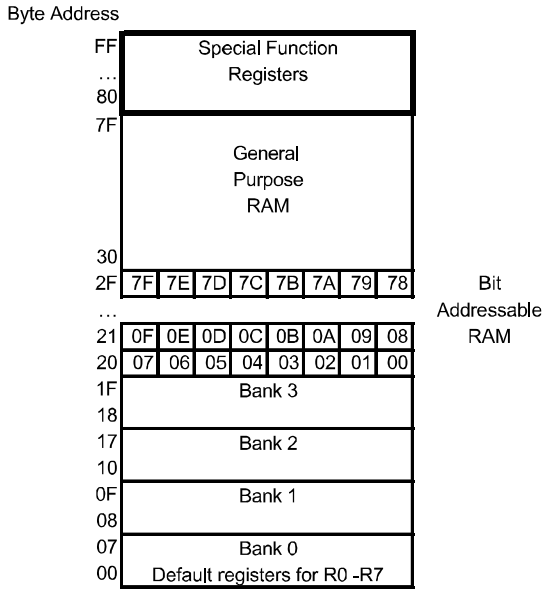


Figure I-6: 8051 RAM Memory Map

Memory locations 20h - 2Fh are bit addressable bytes. This allows the programmer to perform logic operations on the specified bits and do other operations such as set (SETB), clear (CLR). This increases execution speed by not having to perform bit masking on bytes. The bits are numbered 00h - 7Fh for a total of 128 individual bits. When memory is limited and operation must be maximized, the functionality of the resources such as these special registers are very important to programmers.

The remaining available memory from 30h - 7Fh are general-purpose byte addressable RAM for programmer data storage and to hold the *stack*. 80h-FFh are the special function registers, such as the Accumulator, the I/O ports, and many more.

Special Function Registers
Special Function Registers (SFRs) start at address 80h. This section describes the purpose of the most important registers. Discussion of the other registers will follow as need arises.

Accumulator (Acc or A): This is the main register in the controller. It is used for performing mathematical and logical operations.

Program Status Word (PSW): In many instances where registers are concerned, the byte is not as important as the bits within it. Table I-3 indicates the bits contained within the PSW, and Table I-4 discusses their purpose. For the majority of the bits, certain instructions return results in the form of flags within the PSW. For example, with addition we may want to know if the operation resulted in exceeding the limits of our number system. The auxiliary carry flag (AC) in the PSW would indicate if this had occurred.

CY	AC	FO	RS1	RS0	OV	--	P

Table I-3: PSW Bits

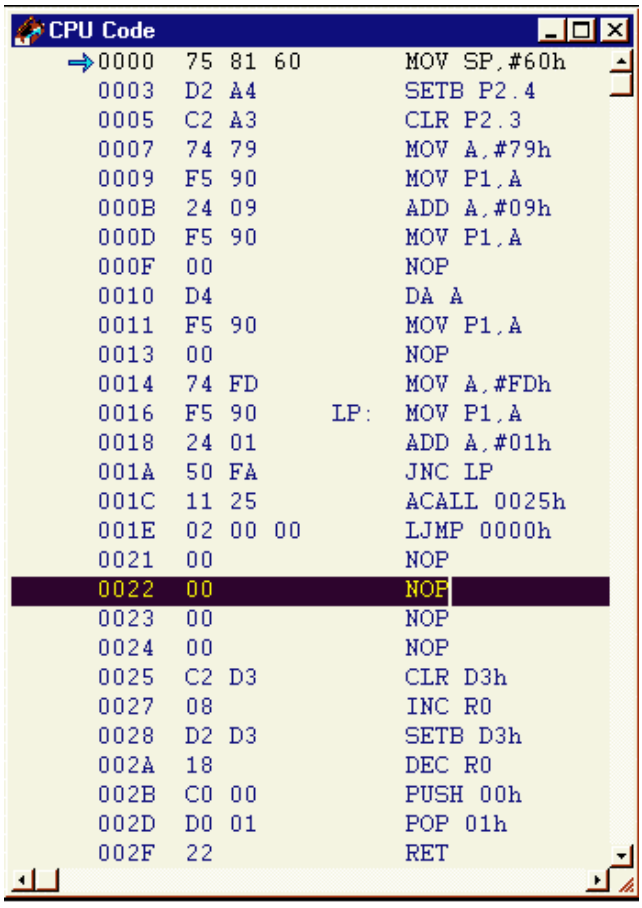
Bit	Discussion
P	Even Parity flag, usually used in serial transmissions. If the total number of HIGH bits in a byte is EVEN, P will be set to 1, otherwise it will be 0.
--	Not used
OV	Overflow flag. When adding or subtracting signed numbers (-128 to +127), this bit will indicate if the operation exceeded the limits (overflow).
RS0 RS1	This pair of bits determines the register bank in use for R0-R7. The binary number of the bits indicates the active bank (00b = 0, 01b =1, 10b = 2, 11b = 3).
F0	Flag 0. A general purpose flag bit for programmer's use.
AC	Auxiliary Carry Flag. Indicates an operation exceeded a binary-coded-decimal value of 9.
CY	Carry Flag. Indicates that a carry or a borrow operation took place for operations, such as adding 1 to 0FFh or subtracting 1 from 00h.

Table I-4: PSW Bit Descriptions

Stack Pointer (SP): The stack is used for two purposes in programming. On a jump to a subroutine instruction, the microcontroller must keep track of the return location. This is a 16-bit word. The high and low order bytes of this address get *pushed* into the stack. The stack pointer keeps track of the end of the stack so that a return call can *pop* the data off the stack. The stack is maintained in our accessible RAM. By default the SP is set to 07H. As bytes are pushed onto the stack, they will sequentially be pushed into location 08h then 09h then 0Ah and so forth (SP is incremented 1 prior to pushing). The SP is updated to reflect where the end of the stack is so the last data pushed-in is the first data popped out (*LIFO* - last in, first out). The second purpose of the stack is to allow us to manually push and pop data into the stack for fast, temporary storage.

If we do not manually reset SP to the upper memory area (such as 60h), the stack may make register banks 1 - 3 unusable along with our bit addressable registers. This depends on how many bytes are pushed into the stack at any one time and using this area of memory.

Let's take a look at a program to demonstrate some of these register concepts in Proj_i-2.



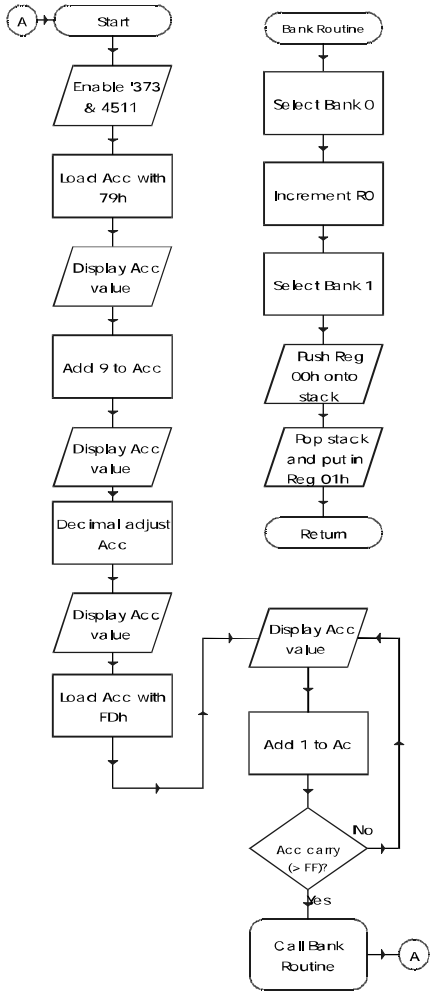


Figure I-7: Program I-2 CPU Code

Table I-5 describes the effects that Program I-2 in Project I-2 has on registers and provides explanations. It skips the NOPs and redundant MOV codes for the display. Use F7 to step through the Program I-2 in UMPS, and observe the effect on the program counter, accumulator, program status word flags and RAM.

Instruction	Register Effect	Explanation
MOV SP,#60h	SP = 60h	Set the Stack Pointer to upper RAM memory at 60h.
SETB P2.4	I/O P2.4 = 1	Enable the 373 latch for LEDs
CLR P2.3	I/O P2.3 = 0	Enable the 4511 for 7-segments.
MOV A,#79h	Acc = 79h	Store 79h in the accumulator.
MOV P1, A	P1 = A (79h)	Move accumulator to the P1 I/O port for displays.

ADD A, #09h	Acc = 82h AC = 1, OV = 1	Add 09h to the accumulator (79h + 09h =82h.) The addition resulted in a non-BCD result so Aux. Carry Flag is set. The addition resulted in exceeding the highest signed byte value (+127 or 7Fh) so the Overflow Flag is set.
DA, A	Acc = 88h	Since we had a non-BCD addition result, adjust the Acc. (79+9=88).
MOV A, #0FDh	Acc = 0FDh	Move 0FDh to the accumulator.
LP: MOV P1,A	P1 = A	Move Accumulator into P1 I/O port. LP is a label for looping.
ADD A, #01h	Acc = Acc + 1 When Acc = 00, CY =1, And AC = 0	Add data (1) to the Accumulator. When Acc goes from 0FFh to 00_{16}, the Aux Carry Flag is set. The Carry Flag is set, indicating a carry bit is generated.
JNC LP		Jump if no carry to the memory location defined. This will cause the program to loop back to LP until CY = 1.
ACALL 0025h	RAM address 61h = 1Eh RAM address 62h = 00h SP = SP + 2 (62h) PC = 0025h	Call to subroutine memory location 0025h (GOSUB to this location). This performs several functions. To be able to return from the call, the next PC position (001Eh) is pushed in the memory location defined by SP as low byte, high byte in 61h and 62h. SP is incremented by 2 to point to the last location to a push occurred. Finally, PC is updated with the called memory location so execution jumps there.
CLR D3h	Bit $D3_{16}$ (RS0) = 0	Clears the lower bank select bit. Register Bank 0 selected.
INC R0	R0 = R0 + 1 (RAM address 00h)	Increments R0 in the currently selected bank of 0. So R0 is located at RAM address 00h.
SETB D3h	Bit 0D3h (RSO) = 1	Sets the lower bank select bit. Register Bank 1 selected.
DEC R0	R0 = R0 –1 (RAM address 08h)	Decrements R0 in the currently selected bank of 1. So R0 is located at RAM address 08h.
PUSH 00h	SP = SP +1 (63h) RAM Address 62h = RAM address 00h.	The Stack Pointer (SP) is incremented one to point to the next available RAM address. The data in RAM location 00 is PUSHed onto the stack at the current stack location.
POP 01h	RAM address 01h = RAM Address 62h. SP = SP – 1	The top data on the stack is POPped off and stored in RAM address 01. SP is decremented by 1 to reflect the POP.
RET	PC = 001Eh. SP = SP - 2.	The program counter is updated by popping the stack twice to get the return location from the subroutine call. The stack pointer is decreased by 2 to reflect that 2 bytes were pulled off the stack.
LJMP 0000h	PC = 0000h.	Long jump to 0000h. Sets the PC counter back to 0000 to continuously loop.

Table I-5: Program I-2 Description

Clicking 'Go' will start the program; it will continuously loop. The displays on the VAB will flash rapidly, and the registers will change quickly as the program executes.

Section Summary

Microcontrollers are often programmed in a very low-level language, Assembler, instead of high level languages. Assembler can be converted directly to the binary machine code understood by the particular processor. In Assembler we deal with opcodes (the instructions for the processor) and operands (data the instruction uses).

Programming in Assembler requires the use of the numerous registers the microcontroller uses in storing data, manipulating data, setting flags, and accessing I/O pins. Special Function registers include the Accumlator for manipulating data, the Program Status Word containing status bits, and the Stack Pointer to keep track of data temporarily placed on the stack.

Section J: Assembler, Opcodes and Addressing.

Reference:
A. MacKenzie, I.S., 1999. The 8051 microcontroller, 3rd ed. Prentice Hall.

Objectives:
1) Discuss advantages and limitations of Assembler.
2) Write code in Assembler to use symbols, origin points, and macros.
3) Identify the differences between addressing modes.
4) Write code in Assembler to use various addressing modes.

Assembly Language Programming

The last section used programs that were written in instruction mnemonics and converted, or assembled, directly into machine code using the UMPS CPU Code window. This is a very quick method for entering code and executing it. As each line is entered, it is checked, and if it cannot be assembled, an error message pops up. It also allows the use of labels, therefore we do not have to worry so much about the memory locations for jumps. However, this method has its definite drawbacks in that it is very difficult to simply add a line of code, and we cannot use variables or 'symbols' other than those native to the 8051 (such as P1, Acc, PSW, etc) to represent memory locations or values. As we program more complex code, greater flexibility would be beneficial.

Take for example the following code in a CPU Code Window:

```
0000  75  90  50      MOV P1, #50h
0003  15  90    Loop: DEC P1
0005  E5  90          MOV A, P1
0007  70  FA          JNZ Loop
```

Starting at memory location 0000h, the program would move the value of 50h into the P1 register, decrement the value in P1, move it to the Accumulator, and if the Accumulator is not zero, jump back and repeat from the decrement on.

Not so bad. But what if we wanted this code to start at memory location 0030h instead? Or if we wanted to write 'DisplayBus', which is more descriptive for our circuit, instead of P1? What if we wanted to add comments? Or maybe do something as simple as putting SETB P2.4 after the DEC P1? Try it. Since each mnemonic gets assembled directly into the memory locations, it becomes quite a chore to simply insert or delete a line of code.

Enter Assembly Language programming. Assembly Language, or Assembler, is very powerful compared to mnemonic coding, but nowhere near the power of high level languages such as BASIC or C. The vast majority of the code in Assembler is still written in instruction mnemonics. Assembler is a compiled programming language. In PBASIC2 we saw that the code was tokenized and then interpreted as the program ran. In a compiled language, such as Assembler and C, the program code is converted directly to machine code for our processor, so no run-time interpretation is required.

A high level language such as C has compilers for many different processors whether we are programming for a Pentium in a IBM compatible, the PowerPC in a Macintosh, or even for the 8051! The same code can be compiled for whichever microprocessor or microcontroller it is intended to be run on. Assembler is not so powerful. Since each processor family has its own unique instruction set, there exists an Assembly language for each one.

Let's take a look at the following code in Assembler (to enter it, use FILE → NEW).

```
DisplayBus equ P1                          ;Symbol declaration for P1
StartValue equ 50h                         ;Symbol declaration for 50h

org 0030h                                  ;Originate compiled code at 0030h
            MOV Diplaybus,#StartValue      ;Move the start value into our LED display bus
Loop2:      DEC DisplayBus                 ;Decrement the value in the bus
            MOV A, DisplayBus              ;Move the value to the Accumulator
            JNZ Loop2                      ;Jump if not zero to Loop2.
```

In order to compile it, we first save the file with a name of our choosing File → Save. Then we instruct UMPS to compile our program (the lighting bolt button, ctrl-F9, or Program→Compile). The Assembler will analyze the program by performing 4 passes through it. After this, it is compiled directly into machine code. Looking in our CPU Code window starting at 0030h, we see the exact same mnemonics and machine code (except the label name) as the one we wrote at 0000h.

UMPS Tip
Using P1, A, PWS and such in Assembler programs are simply using predefined symbols corresponding to the 8051 memory map.

Org and equ are *compiler instructions*. They provide information to the Assembler during compiling. The symbols DisplayBus and StartValue are not variables; they are simply constants that are set equal to the values we assign. Nowhere in the 8051 memory are these being stored unless we explicitly write code to store them. When the program is compiled, the symbols of StartValue and DisplayBus are replaced by their assigned values. Placement of text is also very important. If a line starts at the far left, it is assumed by the language to be compiler instructions

Every program needs at least one origin point (org). This instructs the Assembler where to place the compiled code in ROM memory. We can write our Assembler code to use any number of origin points that we desire (when we discuss interrupts we have to do this).

Another feature of Assembler is the use of a *macro*. A macro is a predefined routine that can be referenced by simply using the macro's name. For example, the PROG_J-1 (in Proj_j-1) defines two macros and uses them. The first macro (Disp_LED) displays data in the Accumulator on the LEDs of the VAB. The second macro (Disp_7Seg) accepts an argument named 'data'. When the program is assembled, the value provided for the argument is used in the macro.

```
;*********************************************************************
;* PROG_J-1 – Demonstrates use of Macros
;*********************************************************************
LED_EN  equ P2.4              ;Symbol for LED enable line
Seg_EN  equ P2.3              ;Symbol for 7-Seg enable line
DIPS    equ P0
DisplayBus equ P1             ;Symbol for P1 port, display bus

MACRO Disp_Leds               ;Display Acc data on LEDs
      MOV DisplayBus, A       ;Move Acc Data to Display Bus
      SETB LED_EN             ;Enable LEDs
      CLR LED_EN              ;Disable LEDs
ENDMAC

MACRO Disp_7SEG (dispdata)          ;Display (dispdata) data on 7-Seg
      MOV DisplayBus, #dispdata     ;Move Acc data to Display Bus
      CLR Seg_EN                    ;Enable 7-Seg
      SETB Seg_EN                   ;Disable 7-Seg
ENDMAC

org 0000h                     ;Define starting address
Jump:
      Disp_7SEG (85h)         ;Display (value) in 7-seg using macro
      MOV A,DIPS              ;Load accum with DIP switch value
      Disp_Leds               ;Display in LEDs using macro
      SJMP Jump               ;Loop forever
```

NOTE: The resources in the VAB are normally refreshed every 75 cycles. For this example the refresh has been changed to 1 in order to allow the high-speed enabling/disabling of the LEDs and 7segments to be recognized, but it dramatically slows simulation execution. In most cases we will add a delay to the program to provide time for the resources to catch up.

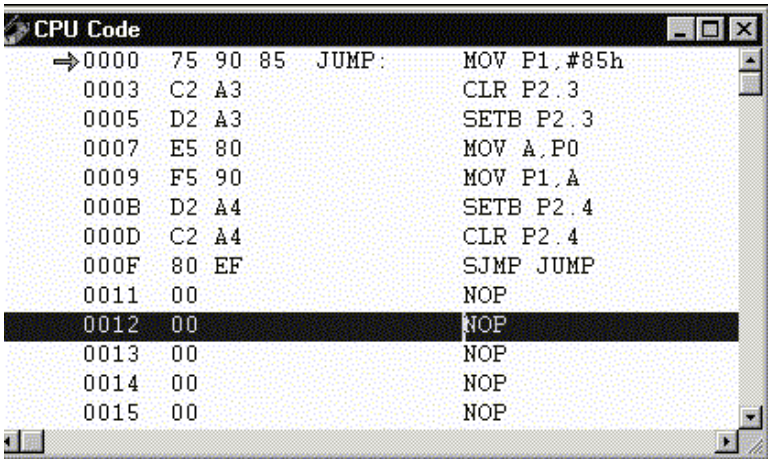


Figure J-1: Macro Compiled Code

Figure J-1 shows the compiled CPU code for Prog_J-1. Can you see how the macros, the defined symbols, and the macro argument 'Dispdata' are compiled?

Some Assemblers allow the use of *Pseudo-Code*. This is code which is not line-by-line compiled, such as mnemonics. Instead a line such as:

```
If Databus > 0
            Then Goto Loop2
```

may be compiled as:

```
    MOV A, Databus
    JNZ Loop2
```

The compiler takes the pseudo-code and creates machine code instructions that perform the required tasks. This is the manner in which high level compiled languages such as C, C++ and many more work.. UMPS does not support pseudo-code.

Opcodes

Let's take a look at opcodes and operands. These are the microprocessor instructions and the information on which they operate. An opcode, and sometimes all or part of the associated operand, is an instruction that is designed into the microcontroller. The 8051 is an 8-bit microcontroller. It works with bytes, which have 256 unique states, 00h - 0FFh. As such, an 8-bit controller may have up to 256 unique instructions. Each instruction has a unique byte that describes its function to the control unit of the microcontroller.

The instruction RET has an opcode of 22h. The data paths internal to the controller are set up to perform this function. If it were RL A, 23h, the paths would be set up for another instruction. You can think of the 8-bits of an instruction as traffic signals at a large intersection. Different combinations of 1's (green lights) and 0's (red lights) will allow traffic to move in different ways. In our case the movement of this traffic represents data being processed.

There are several issues involving every instruction:

- How big is the instruction?
- Where is the data coming from?
- How will the data be manipulated?
- Where are the results to go?
- How long will it take?
- Are any PSW flags set?

These issues are often tied together, as we'll see. First, let's take a look at the majority of instructions groups for the 8051. See reference A for a complete listing of 8051 instructions.

Instruction	General Description
ACALL	Absolute Call. Call a subroutine at a given 16-bit address. Return when done.
ADD A	Adds 2 bytes using the accumulator.
AJMP	Absolute Jump. Jump to a 16-bit address.
ANL	Logically AND with the accumulator or the carry bit

CJNE	Compare 2 bytes and jump if not equal to an address.
CLR	Clear (make 0's) a byte or bit.
DA A	Decimal adjust the accumulator if not in BCD.
DIV	Divide the accumulator by register B
DEC	Decrement (subtract 1)
DJNZ	Decrement and jump if not zero to an address.
INC	Increment (add 1)
JB	Jump of specified bit is set (1).
JBC	Jump and clear the bit if it is set.
JC	Jump if the carry bit in the PSW is set.
JMP	Perform a jump.
JNB	Jump if the bit is NOT set.
JNC	Jump if the carry is NOT set.
JNZ	Jump if the accumulator is NOT zero.
JZ	Jump if the accumulator is zero.
MOV	Move data to the destination from the source.
MUL AB	Multiply the contents of the accumulator and register B
ORL	Logically OR the contents of accumulator or the carrry bit.
POP	Pop a byte off the stack.
PUSH	Push a byte onto the stack.
RET	Return from a subroutine call.
RETI	Return from an interrupt routine.
RL A	Rotate the accumulator left.
RLC A	Rotate the accumulator left through the carry bit.
RR A	Rotate the accumulator right.
RRC A	Rotate the accumulator right through the carry bit.
SETB	Set a bit (1).
SJMP	Short jump, relative addressing.
SUBB A	Subtract from the accumulator.
SWAP A	Swap nibbles within the accumulator.
XCH A	Exchange Accumulator with a byte.
XRL A	Exclusive-OR (XOR) the accumulator with a byte.

Table J-1: Partial 8051 Instruction Set Summary

Whew! There's a few! But wait, there are only 37 instructions here; what about the rest of the 256 possible combinations for instructions? Some instructions, such as RET, are only used in one manner, others, such as a MOV can take on numerous forms. It mainly depends on where the information is coming from and where it is going. This is known as the *addressing mode*.

How long does an instruction take? It depends on how many *instruction cycles* are required for the microcontroller to perform it. **The 8051 requires 12 clock cycles to perform a single instruction cycle**. With the 8051 clocked at 12MHz, an instruction cycle will occur at a rate of 1 MHz or one instruction cycle every 1μS. An instruction, such as

UMPS Tip

With a microcomputer running at 233 MHz, UMPS will almost simulate an 8051 running at 4MHz in real time! (1 second of 8051 simulation time will take approximately 1 second of our time).

SWAP, swaps around the nibbles in the accumulator. That's all. No further information is required, so the instruction can be held in a single byte and requires 1 cycle to perform. The multiply instruction of MUL AB requires 4 cycles to perform.

It was mentioned earlier that the move (MOV) instruction could take on many forms. Different forms require different amounts of memory and execute in different numbers of clock cycles. For example, moving data from an R0-R7 register into the accumulator (MOV A, R3) has an operand associated with it (A, R3). The MOV instructions between the Accumulator and the eight registers each has it's own instruction byte (0DBh for MOV A, R3). It requires one byte and also one instruction cycle. Another move, such placing data (0F3h) in a general purpose RAM address, such as 54h (MOV 54h, #0F3h) requires 3 bytes. One is for the instruction (7Fh), one for the address (54h), and one for the data (0F3h) which makes a 3 byte-instruction (75h 54h F3h). The MOV using these operands takes 2 cycles, or 2μS, to perform with the CPU running at 12MHz. The way data is manipulated in order to minimize the program memory requirements and maximize execution speed can be very important when the requirements demand it.

The 8051 has 8 different methods of addressing. We'll look at each of these in terms of the destination byte, the source byte, bytes required, and the number of operations required using the MOV and JMP instructions.

Addressing

The general format for a MOV instruction is:

MOV <Destination byte> <Source byte>; (Destination) ← (Source).

Destination ← implies that the source byte will be moved into the destinations byte. Where the destination byte is located, and where the source byte resides, is determined by the addressing mode.

The general format for a JMP instruction is:

JMP <Memory Address>; (PC)← (Address)

Direct Addressing

The contents of the 256 bytes of RAM are directly manipulated.
The destination byte and the source byte are in the 256 bytes of RAM. W*here in memory* will determine how many bytes the instruction requires and the number of clock cycles. Remember that addressing the Accumulator and R0-R9 registers is built into the instruction set. Using symbols such as P1 and PWS are simply predefined locations within the special function registers (P1 = 90h). A value with no symbols (# or @) preceding it implies a memory location.

Instruction	Code	Action	Bytes Required	Cycles
MOV P1, A MOV A, 57h	F5 90 E5 57	(P1)←(Acc) (Acc) ← (57h)	**2.** One for MOV using the Accumulator, and one for a memory address.	2
MOV R1, P3 MOV A, 4Fh	A9 B0 E5 4F	(R1) ←(P3) (Acc) ←(4Fh)	**2.** One for MOV using R0-R7 registers and one for the memory address.	2
MOV A, R5 MOV R2, A	ED FA	(Acc)←(R5) (R2)←(Acc)	**1.** The MOV instruction byte defines that the accumulator and one of the registers (R0-R7) is used.	1
MOV P1, P2 MOV 54h,3Ch	85 A0 90 85 3C 54	(P1)←(P2) (54h)←(3Ch)	**3.** One for the MOV instruction, one for the source location, one for the destination location.	2

Indirect Addressing

The contents of a memory address POINTED-to are manipulated.
Instead of holding the data, the operand defines WHERE the data can be found. The source or destination points to the location of the byte. An '@' is used to identify an indirect memory pointer. For the following examples, assume:

- Register R1 holds 50h.
- RAM memory location 50h holds 7Ch.
- RAM memory location 40h holds 0FFh.

Instruction	Code	Action	Bytes Required	Cycles
MOV A, @R1	E7	(Acc)←((R1)) (Acc)←(50h - value in R1) Acc = #7Ch - value at 50h	**1.** One byte defines the Accumulator and that the location in the specified should be addressed.	1
MOV @R1, 40h	A7 40	((R1)) ←(40h) (50h) ←(40h) value in 50h = value in 40h 50h = #0FFh	**2.** One byte defines a move to the address defined by a register and the second defines the RAM address.	2

Immediate Addressing

The instruction holds the data to be manipulated.
Immediate address may be the easiest to understand. The operand defines a byte of data to be stored in the specified register or memory location. Note that a '#' defines data instead of an address.

Instruction	Code	Action	Bytes Required	Cycles
MOV A, #67h MOV R1, #0C3h	74 67 79 C3	(Acc) ← Data; Acc = 67h (R1) ← Data; R1 = C3h	**2.** One byte for a move to accumulator (or R0-R7 register) instruction, one for the data to move.	1
MOV P1, 85h MOV 55h,#30h	75 90 85 75 55 30	(P1) ← Data; P1 =85h (55h) ← Data; memory location 55h = 30h.	**3.** One for the move instruction, one for the address, one for the data.	2

Let's take a look at program (Program J-2 in Proj_j-2) that uses some of these addressing modes.

```
;*****************************************
;* PROG J-2
;* Count from 8 to 0 and display using
;* Direct and Indirect addressing modes
;*****************************************
org 0000h                       ;Set orgin
            SETB P2.4           ;Enable 373
            CLR P2.3            ;Enable 7-Seg

Start:      MOV 50h, #08h       ;Immediate move of 08h into RAM location 50h
```

```
Loop1:                          ;Loop testing direct mode
        MOV P1, 50h             ;Move direct contents of 50h to P1(Display bus)
        DEC 50h                 ;Decrement direct the contents of 50h
        MOV A, 50h              ;Move direct the contents of 50h to Acc
        JNZ Loop1               ;If Acc NOT zero, jump to Loop1
END1:   NOP

        MOV 50h, #08h           ;Load immediate 50h with 08h
        MOV R1, #50h            ;Load R1 with the value of 50h
LOOP2:                          ;Loop to test indirect mode
        MOV P1, @R1             ;Move indirect the contents of the memory location
                                ;that R1 is pointing to P1 (Display bus)
        DEC @R1                 ;Decrement indirect the contents of the memory
                                ;location that R1 is pointing to.
        MOV A, @R1              ;Move indirect the contents of the memory location
                                ;that R1 is pointing to Acc.
        JNZ Loop2               ;If Acc NOT zero, jump to Loop2
END2:   NOP
        SJMP Start              ;Repeat
```

Program J-2 uses 2 loops that perform the same task. They count down from 8 to 0 and display it in the 7-segment display. Loop1 uses immediate addressing to load memory location 50h with the value 8, it then displays it on port 1, decrements it, moves it into the Accumulator and jumps back to the beginning of the loop if it's not zero.

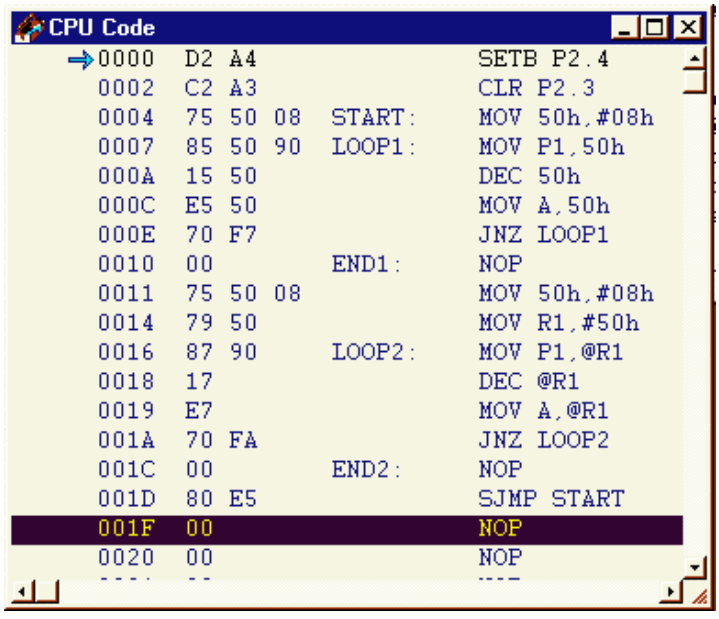


Figure J-2: Program J-2 CPU Code

Loop2 performs the same task, but in this case it uses indirect addressing where R1 is used as a pointer in manipulating memory location 50h. Figure J-2 is the assembled program.

Note the difference in the number of bytes within Loop1 and Loop2. Loop1 is accomplished in 9 bytes of code. Loop2 is performed using only 6 bytes! A savings of 3 bytes of our limited ROM memory. Single stepping through the program, Loop1 takes 48 cycles to accomplish. Loop2 also takes 48 cycles to complete, so even though we were able to save some bytes, there were no savings of execution speed.

Long Addressing

Address is defined by a 16-bit word.

Long Addressing is used in LJMP and LCALL instructions. In these instructions, a jump to a location or a subroutine call to a location, the full 16-bit address is specified. As such, 3 bytes used: one to hold the opcode and the 2 bytes of the address in the operand. For example, LJMP 0350h would cause execution to jump to ROM address 0350h.

Instruction	Code	Action	Bytes Required	Cycles
LJMP 0005h	02 00 05	PC ← address PC ← 0005h	**3.** One byte for the opcode, two bytes for the address.	2
LCALL 0100h	12 01 00	PC ← Address PC ← 0100h (the return 16-bit address is placed on stack)	**3.** One byte for the opcode, two bytes for the address.	2

Absolute Addressing

Address is within the current 2K-byte block.

In most cases we do not have to jump to an address outside of 2K of memory. The majority of our jumps and subroutine calls can be within 2K bytes of ROM. These 2K blocks are known as pages. Since the 8051 has 4K of internal ROM, it is broken up into 2 pages, from 0000h -07FFh and 0800h - 0FFFh. An absolute address is an 11-bit number (2048) which is the address within the current 2K page to which we will jump or call. The lower 8 bits of the address (A7 - A0) are contained in the operand. The upper 3 bits of the address (A10-A8) are the upper 3-bits of the instruction. In this manner a jump or call within a 2K page of memory can be addressed using only 2 bytes of memory.

Instruction	Code	Action	Bytes Required	Cycles
AJMP 0005h	01 05	PC ← 2k address within page 01h = 00000001b (A10-A8) 05h = 00000101b (A7-A0) 2k Address = 00000000101b PC ← 0005h	**2.** One byte for the opcode with upper 3 bits, one byte for lower address byte.	2
ACALL 0650h	D1 50	PC ←2k address within page D1h = 11000001b (A10-A8) 50h = 01010000b (A7-A0) 2K Address = 11001010000b PC ← 0650h (Return 16 bit address is placed on stack and stackpointer is incremented.)	**2.** One byte for the opcode with upper 3 bits, one byte for lower address byte.	2

Relative Addressing

Address lies within 256 bytes of current location.
In many cases, we do not even have to jump outside of a 256 byte range. For example, most loops and jumps occur to locations very near the jump call. Relative addressing adds or subtracts a signed 8-bit number from the current PC location plus 1. Let's look at some same compiled CPU code. Let's analyze it.

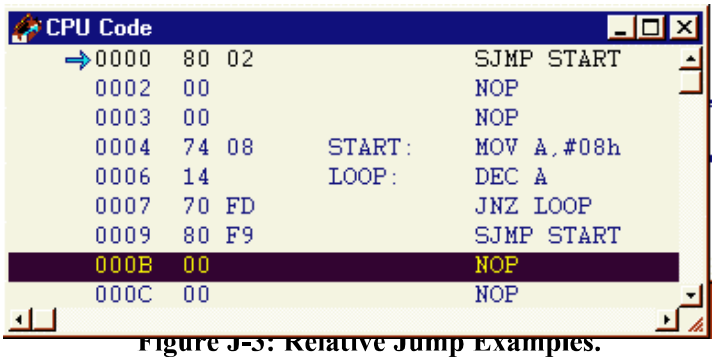


Figure J-3: Relative Jump Examples.

- **80 02 SJMP START**: The CPU code instructs the microcontroller to Short Jump (80) to a location 2 ahead after it adds one to the PC. 0001h is the final byte of the SJMP START instructions, so adding 3 (1 + 2) gives a final relative address of 0004h.
- **70 FD JNZ LOOP**: If the accumulator is not zero, jump to a location defined by 0FDh after adding one. 0FDh is the signed number for -3. From where the JNZ instruction finishes, 0008h + 1 -3 = 0006h, the location of LOOP.
- **80 F9 SJMP START**: Short Jump to relative location defined by F9 (-7). 000A + 1 - 7 = 0004h, the location of START.

See how relative addressing can add -128 or +127 bytes to jump within a section of code using a combination of branching instructions and one-byte operands?

Indexed Addressing

Indexed addressing can be used in jump and move instructions (MOVC). The simplest way to think about indexed addressing is adding a byte (the index) to an Absolute Address (16-bit address). Two registers can hold the address, the PC or DTPT (Data Pointer). DTPT is a 16-bit special function register (SFR) which is used with instructions that can use 16-bit addresses. The high and low bytes of DTPT can be addressed individually as DPH and DPL.

The following program demonstrates the use of indexed addressing. The program displays different patterns on the LEDs of the VAB. The display patterns are held in 19 ROM locations, starting at 0020h and labeled 'DATA'. DTPT is loaded with the address of DATA. Register R0 is used as a counter to count from 0 to 19. Each of these counts is moved into the accumulator. The Accumulator is then used as an index and added to DTPT to arrive at an indexed address holding the next display data byte (DB). The MOVC instruction loads the byte into the accumulator, which

is then placed on the port 1 (P1) and displayed on the LEDs. The program will loop while R0 is not equal to 19.

```
;*******************************************************
;* PROJ_J-3:  LED Display using indexed address
;*            look-up table
;*            Single step through for effects
;*******************************************************
org 0000h
Start:      SETB P2.4           ;Enable the 373 for LEDs

            MOV DPTR, #DATA     ;Move the address of Data label into Data pointer
            MOV R0, #00h        ;Load R0 with 00h.
Loop:
            MOV A, R0           ;Transfer R0 to Acc
            MOVC A, @A+DPTR     ;Load Acc with the indirect address of Acc + DTPR
                                ;(Data, Data+1, Data+2, etc)
            MOV P1, A           ;Move Acc data to P1, display bus
            INC R0              ;Increment R0
            CJNE R0, #19, Loop  ;Compare R0 to decimal 19, jump if not equal
            SJMP START          ;Relative jump back to start

org 0020h
Data:       DB 10000001b        ;Data Bytes for LED display in binary
            DB 01000010b
            DB 00100100b
            DB 00011000b
            DB 00100100b
            DB 01000010b
            DB 10000001b
            DB 11111111b
            DB 10000000b
            DB 01000000b
            DB 00100000b
            DB 00010000b
            DB 00001000b
            DB 00000100b
            DB 00000010b
            DB 00000001b
            DB 10101010b
            DB 01010101b
            DB 10101010b
            DB 01010101b
```

Assembler Tip

DB can also be used in the following manner:

DB 'HELLO WORLD!'

Where, just in the BS2 using the Data instruction, each character is a different byte.

Section Summary

A unique byte is used in defining an instruction to the microcontroller. The instruction may have a unique use, or one instruction may have multiple methods of being used depending on the addressing modes.

The 8051 has 8 addressing modes depending on how the operand is to used. The operand may contain the actual data or point to where the data is located in memory. The pointer may be an absolute memory address, an address within a 256-byte block, and address with a 2K-byte block, or be in a special register. Microcontrollors use the various addressing modes to provide the user with flexibility for size or speed depending on the needs.

Section K: Interrupts - External, Timers and Counters

Reference:
A. MacKenzie, I.S., 1999. The 8051 microcontroller, 3rd ed. Prentice Hall.

Objectives:
1) Discuss the use of interrupts in microcontrollers.
2) Discuss key elements in interrupt programming, such as enables and interrupt vectors.
3) Discuss the differences between external, timer and counter interrupts.
4) Control 8051 registers for interrupt programming.

General Interrupts

As we've seen, microcontrollers can operate very quickly. Individual instructions take only microseconds. Frequently, outside devices, such as people pressing buttons, need to be recognized by the microcontrollers. Take, for example, the microwave oven. Once started, it will heat our food until the heating program ends, or we interrupt the heating by pressing STOP or opening the door. No matter where in the heating program the microcontroller is, it is a safety concern to ensure it recognizes the fact the door was opened. Of course, an interrupt may not be as critical as this example. The controller goes about it's business of making sure the correct time is displayed until we interrupt it by punching buttons for a new heating cycle. If you've dealt with adding hardware to computers, you may have run across interrupt setting and conflicts for expansion cards.

Basically, a microcontroller or microprocessor deals with the main program task at hand unless it is interrupted by an external or internal interrupt request. Upon receiving an interrupt request, the processor will jump to a specific address, an *interrupt vector*. At this address we instruct the processor how to handle the interrupt. Once the interrupt routine is complete, execution returns to the previous task at the point it was interrupted.

The 8051 has the following sources for interrupts:
- Two I/O pins which can be used for external interrupts.
- Two I/O pins which can be used as counter interrupts
- Two internal timers used for timer interrupts or used in conjunction with the I/O pins for counter interrupts.
- I/O pin for Serial data interrupt.

In order to use interrupts, we need to understand two main principles: Interrupt Registers and Interrupt Vectors. Interrupt registers are used in configuring the interrupts. Do we want counter or timer interrupts? How many counts before interrupting? Is the interrupt enabled or not? Several registers used in configuring the 8051 interrupts and will be discussed later.

An interrupt vector is the memory address location that the Program Counter (PC) will jump to when an interrupt is set. Let's say, for example, we have a program with an external interrupt enabled. Our program may be going about its business and looping between addresses 0100h and 0200h. When the external interrupt is set (perhaps by an operator pressing a button), program execution will immediately jump to the interrupt vector address (0003h in this case). The program

will carry out the instructions starting at that address. When the interrupt routine is complete, the final instruction will 'return from interrupt', and program execution will resume where it left off within our 0100h - 0200h loop.

The 8051 has 5 interrupt vectors:

Address	**Purpose**
0000h	System Reset. When Pin RST is high, the system will reset.
0003h	External 0. Vector address for P3.2 INT0, external interrupt 0.
000Bh	Timer 0. Vector address for timer/counter (P3.4 T0) interrupt 0.
0013h	External 1. Vector address for P3.3 INT1, external interrupt 1.
001Bh	Timer 1. Vector address for timer/counter (P3.5 T1) interrupt 1.
0023h	Serial data received. Vector address for incoming serial data.

The serial interrupt is a nice example to look at although we will not delve into it further in this manual. On the BS2 we received serial data from another BS2 using SERIN command. The BS2 and PBASIC itself does not use interrupts (the 16C57 at its heart does though). When our receiver was expecting serial data, our program had to stop and wait for it. If it didn't arrive before we timed out, our program would have missed it. Additionally, we had the long wait of just having the program pause to see if any data was arriving.

In microcontrollers, such as the 8051, a serial data interrupt is quite a convenience. Our program can carry out any task it needs to without worrying about whether serial data is expected or not. If serial data DOES arrive, another part of the microcontroller collects the data and sets a flag indicating data has arrived. This flag would then cause our program to interrupted, branch to address 0023h, carry out any instructions we wrote to use the serial data, then return back to our main program when complete.

You may have noticed the available memory between each interrupt vector isn't very large (8 bytes). This doesn't leave much room for code, but it is sufficient to branch to a subroutine that does have a large amount of code. Let's look at a real simple program for processing serial data received. We'll keep it to just the basics.

```
Org 0000h
        LJMP Loop               'Jump down to our main program

Org 0023h                       'Serial Interrupt Vector
        LJMP ProcessSerial

Org 0100h
Start:
        (Configure Interrupts)
Loop:
        (code to light LEDs or whatever)
        LJMP Loop

ProcessSerial
        (code to read serial data from register and process it)
        RETI            'Return from interrupt command
```

The initial jump at 0000h will skip over our interrupt routines to 'Start' where the interrupts will be configured. Following that, our microcontroller will be busy with the tasks defined in 'Loop'. When serial data does arrive, the micro will be interrupted and branch to 0023h. There it is instructed to jump to 'ProcessSerial' where our serial data is collected and manipulated. Once complete RETI, (Return from Interrupt) will instruct the micro to go back about its business in Loop, where it left off.

Let's look at a program before we discuss the interrupts in depth. Program K-1 of Project K-1 will perform the following:

- The main loop (Loop) will read the data on the DIP switches and place the data on the LEDs D0-D7.
- PB1 on INT0 of the 8051 will trigger External Interrupt 0 and cause program execution to branch to 0003h. From there the routine 'Addone' will be performed. This will add 1 to our 7-segment display count. Only 1 will be added for each button press because this interrupt is defined as being *edge-triggered*. The transition from a HIGH to LOW will cause the interrupt to occur.
- PB2 on INT1 will trigger External Interrupt 1 and cause program execution to branch to 0013h. From there routine 'SubOne' will be performed. This will subtract 1 from our 7-segment display count. This time, though, the subtraction will continue as long as the button is pressed (some counts will result in a blank digit since the number may exceed the decimal values of 0-9). This interrupt is set up to be *state-triggered*. As long as the button is pressed and the input is LOW, the interrupt will continue to be set, giving us multiple decrementing counts.

```
;*********************************************************************
;* PROG_K-1  – Demonstrates external Interrupts
;* Program will loop putting DIPS on LED until
;* a PB1 or PB2 press calls interrupt to add/sub 1
;* from 7-segment
;*********************************************************************
LED_EN  equ P2.4            ;Symbol for LED enable line
Seg_EN  equ P2.3            ;Symbol for 7-Seg enable line
DIPS    equ P0              ;Symbol for DIP switches
DisplayBus equ P1           ;Symbol for P1 port, display bus

org 0000h
      SJMP START            ;Jump past interrupt vector address to start
org 0003h                   ;Ext. Int. 0 vector address
      LJMP AddOne           ;Jump down and add one to 7-seg
org 0013h                   ;Ext. Int. 1 vector address
      LJMP SubOne           ;Jump down and subtract one from 7-seg

org 0030h                   ;Start of program
START:
      MOV SP, #60h          ;Move Stack Pointer to upper RAM area
      SETB EX0              ;Enable Ext Int 0
      SETB IT0              ;Set Ext Int 0 to leading edge mode
      SETB EX1              ;Enable Ext Int 1
      CLR IT1               ;Set Ext Int 1 to state mode
      SETB EA               ;Set Global Interrupt
      MOV R0, #24h          ;Load up R0 (7-seg number)
```

```
        SETB IE0                ;Force interrupt Ext 0 to add one

Loop:
        MOV A, DIPS             ;Load Acc with DIPS
        MOV DisplayBus, A       ;Move Acc Data to Display Bus
        SETB LED_EN             ;Enable LEDs
        MOV R7,#08h             ;Short pause while VAB resources catch up
Ploop:
        DJNZ R7, Ploop
        CLR LED_EN              ;Disable LEDs
        SJMP Loop               ;Loop forever

AddOne:
       INC R0                   ;Increment R0
       SJMP Disp_7Seg           ;Jump down to display it

SubOne: DEC R0                  ;Decrement R0

Disp_7Seg:
        MOV DisplayBus, R0      ;Move R0 data to Display Bus
        CLR Seg_EN               ;Enable 4511's for 7-segments
        MOV R7, #30h            ;Pause while VAB resources catch up
iPause:  DJNZ R7, IPause
        SETB Seg_EN             ;Disable 7-Seg
        RETI                    ;Return from Interrupt routine
```

External Interrupts:

Let's look at the interrupt vectors first. When external interrupt 0 is enabled and occurs, execution will branch to 0003h. We have 8 bytes available at each interrupt vector to write code (at 000Bh is a timer interrupt vector). If the operation requires more than 8 bytes, the code at the interrupt vector may contain a jump to an actual interrupt routine. External Interrupt 1 vector is 0013h.

So how is an external interrupt enabled? There are 2 registers that need to be examined, IE and TCON. IE is the interrupt enable register. Its breakdown is as follows:

EA	--	--	ES	ET1	EX1	ET0	EX0

Table K-1: IE Bits

Bit	Discussion
EX0	Enables External Interrupt 0
ET0	Enables Timer Interrupt 0
EX1	Enables External Interrupt 1
ET1	Enables Timer Interrupt 1
ES	Enables Serial Interrupt
--	Not used in 8051
--	Not used in 8051
EA	Global Interrupt Enable.

Table K-2: IE Bit Descriptions

In order to enable the external interrupt 0, EX0 must be set. EX1 must be set for external interrupt 1. *For ANY of the interrupts to be enabled, EA -the global interrupt enable, must also be set.* EA can be used to quickly disable or enable all the interrupts. Once these bits are set, a low on INT0 or INT1 will cause the interrupt to occur. Upon completion of an interrupt routine, RETI will return program execution at the point the interrupt was called.

TCON is the second register controlling the external interrupts.

TF1	TR1	TFO	TRO	IE0	IT1	IE0	IT0

Table K-3: TCON Bits

Bit	Discussion
IT0 & IT1	Interrupt 0 and 1 mode controls. If 1, interrupt will occur on negative going edge only on INT0 and INT1. If 0, interrupt will occur on low level.
IE0 & IE1	These are the actual bits that are set on an interrupt. They are normally set by the external trigger on INT0 and INT1 but may be software set. (1= interrupt)
TR0 & TR1	These are the enable bits for Timers 0 and 1. If 1, the timers will be on.
TF0 &TF1	These are the flags which are set when Timers 0 and 1 overflow triggering an interrupt. (1 =interrupt).

Table K-4: PSW Bit Descriptions

The code in Program K-1 for setting up the interrupts is as follows:

```
SETB EX0        ;Enable Ext Int 0
SETB IT0        ;Set Ext Int 0 to leading edge mode
SETB EX1        ;Enable Ext Int 1
CLR IT1         ;Set Ext Int 1 to state mode
SETB EA         ;Set Global Interrupt
```

In our sample program, IT0 is set (1) and IT1 is clear (0). When PB1 is pressed, the 7-segment counts up one because it is on INT0, and IT0 defines it as edge triggered (1). PB2, on the other hand is on INT1, for which IT1 is clear (0), so that the interrupt is re-triggered as long as the button is pressed and LOW. This causes the program to decrement a high number of counts.

Running this program and pressing PB0 and PB1, you may notice the LEDs flicker as the interrupt routines occur. Why? The flickering resulting from our simple example brings up two very good issues to consider when using interrupts. First, how do you stop an interrupt when you don't want one to occur? And, secondly, how do you preserve critical data that may be present in registers when an interrupt occurs.

The LEDs flicker because during that short period of time that the LEDs enable line is HIGH, any data put on the DisplayBus will be captured by the '373 and displayed, including data from the interrupt routine. To prevent this we could have disabled EA prior to the LED write and re-enabled it following the write. If the interrupt occurred during the disabled period, the interrupt would take action as soon as EA was set HIGH again.

Caution needs to be taken when jumping to interrupts. If a register such as the Accumulator contain critical data, and disabling the interrupts would be undesirable, we need to ensure that the interrupt does not destroy the critical data. In this case, at the beginning of the interrupt we may want to PUSH critical registers onto the stack, and POP them off again at the end of the routine.

UMPS allows us to stop execution (break) on an interrupt call. Under OPTION→RUN MODE ... we are able to set UMPS to halt execution on an interrupt call (Halt on interrupt code = checked). Running the program, once an interrupt call is made and execution stops, we are able to use F7 to single step through it and see the actions taken.

Timer & Counter Interrupts

With external interrupts we used an external signal to interrupt the processor to perform certain tasks. Many times we may want to have routines that occur at specific time intervals or count external signals. These actions can be performed using timer and counter interrupts. Simply put, interrupt registers count, and when it overflows (exceeds all 1's), an interrupt is generated. The 8051 has 2 registers for this, Timer 0 and Timer 1. If the registers are incremented with each instruction cycle, they are considered timers. If they are incremented by an external pulse, they are considered counters. Whether they are acting as timers or counters, the registers are referred to as timers.

Both timers are 16-bit, comprised of high and low bytes (TH0 and TL0 for timer 0, TH1 and TL1 for timer 1). There exists four modes the timers in the 8051 can operate in, modes 0 - 3. We will focus our discussion on timer mode 1, 16-bit timer, and mode 2, 8-bit timer with auto-reload. The timers can be configured to count one bit each instruction cycle (for a timer) or count based off of an external signal (P3.4 -T0, P3.5 - T1).

In mode 1, 16-bit, the high and low bytes act as a16-bit word. When sufficient counts occur, the timer rolls over from FFFFh to 0000h, and a timer interrupt is generated. If we start at 0000h, and the timer is counting every instruction cycle, running at 12MHZ the time between interrupts would be 65,536 μS, or 65mS. We can preload the timer manual, to say 8000h, so the interrupts occur twice as fast. Reloading the interrupt to 8000h would need to be part of the interrupt routine, or the count would start from 0000h following the interrupt routine.

In mode 2, 8-bit auto-reload, only the lower byte is used for counting and interrupting. This provides a maximum number of counts of 256 between interrupts. The high byte is the reload value. When the low byte overflows, an interrupt is generated, and the low byte is automatically reloaded with the value of the high byte. Let's think about counter application for this.

We are given a flowmeter that provides 8 pulses or *clicks* per gallon of water flowing through it. If we wanted to interrupt the processor to count each gallon, we could load the value of 11111000b into the high bytes and set up the processor to count off external pulses. When 8 flow clicks have occurred, the low byte would roll over, generate an interrupt to count a gallon, and RELOAD the low byte with the value of 11111000b to be ready to count the next 8 clicks.

Project K-2 is an example of using both timers in the two different modes we have just discussed. It uses timer 1 in mode 1 as a 16-bit timer to control a traffic signal. As the timer 1 interrupt is

generated, the light will change and load the timer's high byte with a value depending on the light that is lit. The red light is on a relatively long time, green a short time, and yellow a very short time. Remember that the time length is based on how far below FFFFh the timer is set, since it starts at this value and counts to FFFFh.

PB3 is connected to T0. Timer 0 is configured in mode 2 for 8-bit auto-reload counter. In this example we are counting traffic. Cars have 2 axles and semi-trucks have 5 axles. The truck traffic at our intersection is light, so we will take an average of 3 axles as one vehicle. As the vehicles pass over a pressure hose, a click is sent to the processor and counted. Since we want 3 clicks/vehicle, we will use a preload value of 11111101 (253) so 3 counts will overflow the timer. PB3 connected to P3.4 (T0) is used to simulate the pressure hose.

```
;*************************************************
;* Prog_K-2 - Time/Counter interrupt Examples
;* 16-bit timer running stop lights
;* 8-bit auto load timer using external trigger for 7-seg counting
;*************************************************
DIPS   EQU   P0                          Set up symbols
PB3    EQU   P3.4
STOPLT  EQU   20h                        ;Hold stop light in byte 20 of bit addressable RAM
GREEN   EQU   07h                        ;lights are addressable bits in 20h
YELLOW  EQU   06h
RED     EQU   05h

SL_DELAY EQU   TH1
LED_EN  EQU   P2.4
LED_DAT EQU   00h                        ;Hold LED data in R0
SEG_EN  EQU   P2.3
SEG_DAT EQU   01h                        ;Hold Segment data in R1
DISP_BUS EQU   P1

ORG 0000h
        LJMP START
ORG 00Bh                                 ;Timer 0 interrupt vector
        LJMP SEG_COUNT
ORG 001Bh                                ;Timer 1 interrupt vector
        LJMP STOP_LIGHT
ORG 0030h
START:
        MOV SP,#60h                      ;Move stack to high RAM
        MOV TMOD, #00010110b             ;Set up timers,Timer 0 as 8-bit, ext counter
                                         ;              Timer 1 as 16 bit internal timer
        MOV TL0, #11111101b              ;Load low byte of timer 0
        MOV TH0, #11111101b              ;Load high byte(reload value) of Timer 0
        MOV TL1, #0FFh                   ;Load low byte of Timer 1
        MOV TH1, #01h                    ;Load high byte of Timer 1
        SETB TR0                         ;Start timer 0
        SETB TR1                         ;Start timer 1
        SETB ET0                         ;Enable Timer 0 Interrupt
        SETB ET1                         ;Enable Timer 1 Interrupt
        MOV P2, #00h                     ;Clear Stop light port
        SETB RED                         ;Turn on bits red light, Off yellow & green
        CLR GREEN
        CLR YELLOW
        MOV P2,STOPLT                    ;Move stop light data to P2 port for LEDs
        MOV SEG_DAT, #00                 ;Set 7-segments equal to 00
        ACALL SEG_DISPLAY                ;Display 7-seg
        SETB EA                          ;Enable global interrupts

LOOP:   MOV LED_DAT, DIPS                ;Read DIPS and put into LED_DAT
        ACALL LED_DISPLAY                ;Display DIP data
        LJMP LOOP                        ;Repeat
```

```
LED_DISPLAY:                    ;Display LED data on LEDs
        CLR EA                  ;Disable all interrupts
        MOV DISP_BUS, LED_DAT
        SETB LED_EN
        ACALL DELAY             ;Delay for VAB resources to catch up
        CLR LED_EN
        SETB EA                 ;Enable Global Interrupts
        RET                     ;Return from the call

SEG_DISPLAY:
        CLR SEG_EN
        ACALL DELAY
        SETB SEG_EN
        RET

SEG_COUNT:                      ;Count up on seg data
        PUSH ACC                ;Save Acc data on stack
        MOV A, SEG_DAT          ;Load Acc with seg_data
        ADD A, #1h              ;Add 1
        DA A                    ;Decimal Adjust
        MOV SEG_DAT,A           ;Move to display bus
        ACALL SEG_DISPLAY       ;Display it
        POP ACC                 ;Get Acc off stack
        RETI                    ;Return

STOP_LIGHT:
        JB RED, TURNGREEN       ;Jump if RED bit is set to green light
        JB GREEN, TURNYELLOW    ;Jump if Green bit is set to turn yellow
                                ;If neither above, must have been yellow
TURNRED:
        SETB RED                ;Turn on Red
        CLR YELLOW              ;Turn Off yellow
        MOV SL_DELAY, #80h      ;Load mid time for light delay (TH1)
        SJMP CHANGE             ;Go change the light

TURNYELLOW:
        SETB YELLOW             ;Turn on yellow
       CLR GREEN                ;Turn off green
       MOV SL_DELAY, #0D0h      ;Short value for (FFh-D0h) for yellow light
       SJMP CHANGE              ;Go change it

TURNGREEN:
        SETB GREEN              ;Turn on green from red
       CLR RED                  ;turn off red
       MOV SL_DELAY, #60h       ;Longer delay for light (FFh-60h)

CHANGE:                         ;Change the lights
        PUSH ACC                ;Store Acc in stack
        MOV A, P2               ;Get current value of P2 port
        ANL A, #00011111b       ;Mask out higher 3-bits (traffic lights)
        ORL A, STOPLT           ;Mask in the traffic light pattern
        MOV P2,A                ;Move it back to P2 port
        POP ACC                 ;Get Acc off stack
        RETI                    ;Return from this interrupt

DELAY:          MOV R3, #30h    ;short delay for resources
DELAY_LOOP:     DJNZ R3, DELAY_LOOP
        RET                     ;Return from routine call
```

Of course, when no interrupts are occurring, the program will be busy moving those darn DIP switch settings to the LEDs. Let's look at how the timers were set up. In the START section, the timers are configured and the interrupts enabled.

Let's first look at the TMOD (timer mode) register.

Gate	C/$\overline{T}$	M1	M0	Gate	C/$\overline{T}$	M1	MO

Table K-5: TMOD

The lower nibble controls Timer 0, and the higher nibble controls Timer 1, so we will look at only one set.

Bit	Discussion
M0 M1	These bits assign the mode number of the timer, 00,01,10,11 (0,1,2,3).
C/$\overline{T}$	Defines whether it will be a counter off the external input bit (HIGH) or a timer based on the internal instruction cycle clock.
Gate	If HIGH, INT0 or INT1 must be HIGH for the timer to run.

Table K-6: TMOD Bit Descriptions

For our example, TMOD is set to 00010110b. Breaking it down, we set up Timer 0 (low nibble) for mode 2 counter, and Timer 1 (high nibble) for a mode 1 timer. Other registers needed are ET0 and ET1 (enable interrupts) and EA for global interrupt enable. From TCON (Table K-3), TR0 and TR1 need to be high to enable the timers to count.

```
MOV TMOD, #00010110b    ;Set up timers,Timer 0 as 8-bit, ext counter
                        ;          Timer 1 as 16 bit internal timer
MOV TL0, #11111101b     ;Load low byte of timer 0
MOV TH0, #11111101b     ;Load high byte(reload value) of Timer 0
MOV TL1, #0FFh          ;Load low byte of Timer 1
MOV TH1, #01h           ;Load high byte of Timer 1
SETB TR0                ;Start timer 0
SETB TR1                ;Start timer 1
SETB ET0                ;Enable Timer 0 Interrupt
SETB ET1                ;Enable Timer 1 Interrupt
      .
      .
      .
SETB EA                 ;Enable global interrupts
```

Again, we can set UMPS to break on interrupts and follow the code as interrupts are generated.

Section Summary

Microcontrollers provide interrupts to the programmer. The interrupt allows branching to specific routines when an event occurs. Examples of these events include a special input changing, serial data arriving, or a timer overflowing. This allows immediate response to an event without having to continually check the status.

Timers can be used to either perform periodic events based on the system clock, or to count based on an external signal. In using timers and interrupts special registers must be configured to set and enable them to the needs of the program.