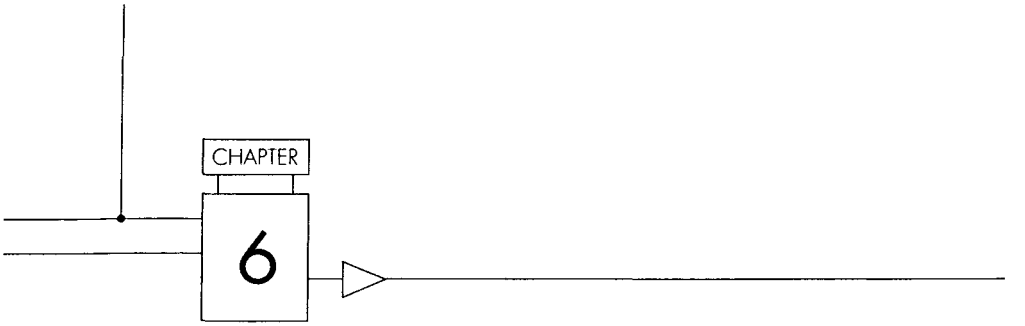


Jump and Call Opcodes

Chapter Outline

Introduction

The opcodes that have been examined and used in the preceding chapters may be thought of as action codes. Each instruction performs a single operation on bytes of data.

The jumps and calls discussed in this chapter are *decision* codes that alter the flow of the program by examining the results of the action codes and changing the contents of the program counter. A jump permanently changes the contents of the program counter if certain program conditions exist. A call temporarily changes the program counter to allow another part of the program to run. These decision codes make it possible for the programmer to let the program adapt itself, as it runs, to the conditions that exist at the time.

While it is true that computers can't "think" (at least as of this writing), they can make decisions about events that the programmer can foresee, using the following decision opcodes:

Jump on bit conditions
Compare bytes and jump if *not* equal
Decrement byte and jump if zero
Jump unconditionally
Call a subroutine
Return from a subroutine

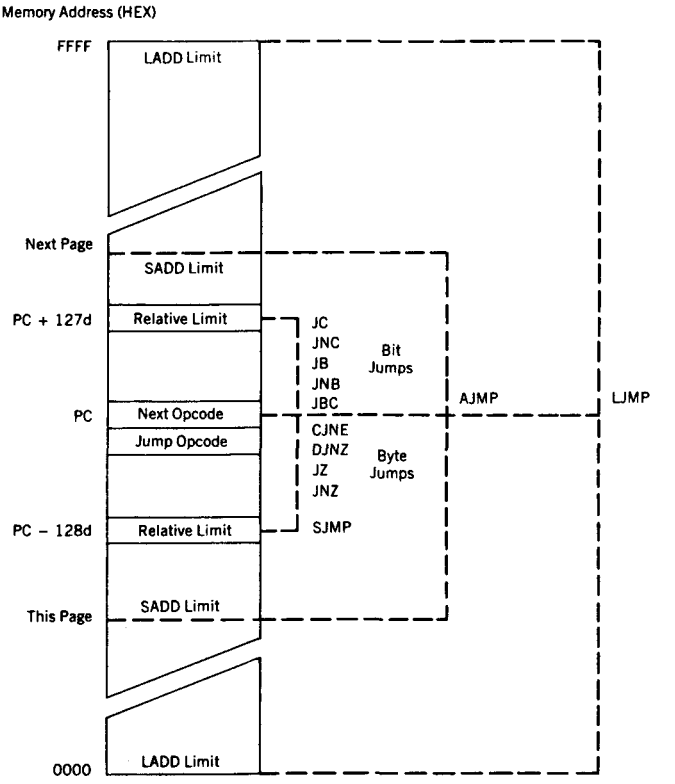
Jumps and calls may also be generically referred to as "branches," which emphasizes that two divergent paths are made possible by this type of instruction.

The Jump and Call Program Range

A jump or call instruction can replace the contents of the program counter with a new program address number that causes program execution to begin at the code located at the new address. The difference, in bytes, of this new address from the address in the program where the jump or call is located is called the *range* of the jump or call. For example, if a jump instruction is located at program address 0100h, and the jump causes the program counter to become 0120h, then the range of the jump is 20h bytes.

Jump or call instructions may have one of three ranges: a *relative* range of +127d, −128d bytes from the instruction *following* the jump or call instruction; an *absolute* range on the same 2K byte page as the instruction *following* the jump or call; or a *long* range of any address from 0000h to FFFFh, anywhere in program memory. Figure 6.1 shows the relative range of all the jump instructions.

FIGURE 6.1 Jump Instruction Ranges



Relative Range

Jumps that replace the program counter contents with a new address that is greater than the address of the instruction *following* the jump by 127d or less than the address of the instruction following the jump by 128d are called *relative* jumps. They are so named because the address that is placed in the program counter is relative to the address where the jump occurs. If the absolute address of the jump instruction changes, then the jump address changes also but remains the same distance away from the jump instruction. The address following the jump is used to calculate the relative jump because of the action of the PC. The PC is incremented to point to the *next* instruction *before* the current instruction is executed. Thus, the PC is set to the following address before the jump instruction is executed, or in the vernacular: "before the jump is taken."

Relative jumping has two advantages. First, only one byte of data need be specified, either in positive format for jumps ahead in the program or in 2's complement negative format for jumps behind. The jump address displacement byte can then be added to the PC to get the absolute address. Specifying only one byte saves program bytes and speeds up program execution. Second, the program that is written using relative jumps can be located anywhere in the program address space without re-assembling the code to generate absolute addresses.

The disadvantage of using relative addressing is the requirement that all addresses jumped be within a range of +127d, −128d bytes of the jump instruction. This range is not a serious problem. Most jumps form program loops over short code ranges that are within the relative address capability. Jumps are the only branch instructions that can use the relative range.

If jumps beyond the relative range are needed, then a relative jump can be done to another relative jump until the desired address is reached. This need is better handled, however, by the jumps that are covered in the next sections.

Short Absolute Range

Absolute range makes use of the concept of dividing memory into logical divisions called "pages." Program memory may be regarded as one continuous stretch of addresses from 0000h to FFFFh. Or, it may be divided into a series of pages of any convenient binary size, such as 256 bytes, 2K bytes, 4K bytes, and so on.

The 8051 program memory is arranged as 2K byte pages, giving a total of 32d (20h) pages. The hexadecimal address of each page is shown in the following table:

PAGE	ADDRESS(HEX)	PAGE	ADDRESS(HEX)	PAGE	ADDRESS(HEX)
00	0000–07FF	0B	5800–5FFF	16	B000–B7FF
01	0800–0FFF	0C	6000–67FF	17	B800–BFFF
02	1000–17FF	0D	6800–6FFF	18	C000–C7FF
03	1800–1FFF	0E	7000–77FF	19	C800–CFFF
04	2000–27FF	0F	7800–7FFF	1A	D000–D7FF
05	2800–2FFF	10	8000–87FF	1B	D800–DFFF
06	3000–37FF	11	8800–8FFF	1C	E000–E7FF
07	3800–3FFF	12	9000–97FF	1D	E800–EFFF
08	4000–47FF	13	9800–9FFF	1E	F000–F7FF
09	4800–4FFF	14	A000–A7FF	1F	F800–FFFF
0A	5000–57FF	15	A800–AFFF		

Inspection of the page numbers shows that the upper five bits of the program counter hold the page *number*, and the lower eleven bits hold the *address* within each page. An absolute address is formed by taking the page number of the instruction *following* the

branch and attaching the absolute page range address of eleven bits to it to form the 16-bit address.

Branches on page *boundaries* occur when the jump or call instruction finishes at X7FFh or XFFFh. The next instruction starts at X800h or X000h, which places the jump or call address on the same page as the *next* instruction after the jump or call. The page change presents no problem when branching ahead but could be troublesome if the branch is *backwards* in the program. The assembler should flag such problems as errors, so adjustments can be made by the programmer to use a different type of range.

Absolute range addressing has the same advantages as relative addressing; fewer bytes are needed and the code is relocatable as long as the relocated code begins at the start of a page. Absolute addressing has the advantage of allowing jumps or calls over longer programming distances than does relative addressing.

Long Absolute Range

Addresses that can access the entire program space from 0000h to FFFFh use long range addressing. Long-range addresses require more bytes of code to specify and are relocatable only at the beginning of 64K byte pages. Since we are limited to a nominal ROM address range of 64K bytes, the program must be re-assembled every time a long-range address changes and these branches are not generally relocatable.

Long-range addressing has the advantage of using the entire program address space available to the 8051. It is most likely to be used in large programs.

Jumps

The ability of a program to respond quickly to changes in conditions depends largely upon the number and types of jump instructions available to the programmer. The 8051 has a rich set of jumps that can operate at the bit and byte levels. These jump opcodes are one reason the 8051 is such a powerful microcontroller.

Jumps operate by testing for conditions that are specified in the jump mnemonic. If the condition is *true*, then the jump is taken—that is, the program counter is altered to the address that is part of the jump instruction. If the condition is *false*, then the instruction immediately following the jump instruction is executed because the program counter is not altered. Keep in mind that the condition of *true* does *not* mean a binary 1 and that *false* does *not* mean binary 0. The *condition* specified by the mnemonic is either true or false.

Bit Jumps

Bit jumps all operate according to the status of the carry flag in the PSW or the status of any bit-addressable location. All bit jumps are relative to the program counter.

Jump instructions that test for bit conditions are shown in the following table:

Mnemonic	Operation
JC radd	Jump relative if the carry flag is set to 1
JNC radd	Jump relative if the carry flag is reset to 0
JB b,radd	Jump relative if addressable bit is set to 1
JNB b,radd	Jump relative if addressable bit is reset to 0
JBC b,radd	Jump relative if addressable bit is set, and clear the addressable bit to 0

Note that no flags are affected unless the bit in JBC is a flag bit in the PSW. When the bit used in a JBC instruction is a port bit, the SFR latch for that port is read, tested, and altered.

The following program example makes use of bit jumps:

```
ADDRESS   MNEMONIC        COMMENT
LOOP:     MOV A,#10h      ;A = 10h
          MOV R0,A        ;R0 = 10h
ADDA:     ADD A,R0        ;add R0 to A
          JNC ADDA        ;if the carry flag is 0, then no carry is
                          ;true; jump to address ADDA; jump until A
                          ;is F0h; the C flag is set to
                          ;1 on the next ADD and no carry is
                          ;false; do the next instruction
          MOV A,#10h      ;A = 10h; do program again using JNB
ADDR:     ADD A,R0        ;add R0 to A (R0 already equals 10h)
          JNB 0D7h,ADDR   ;D7h is the bit address of the carry flag
          JBC 0D7h,LOOP   ;the carry bit is 1; the jump to LOOP
                          ;is taken, and the carry flag is cleared
                          ;to 0
```

CAUTION

All jump addresses, such as ADDA and ADDR, must be within +127d, −128d of the instruction following the jump opcode.

If the addressable bit is a flag bit and JBC is used, the flag bit will be cleared.

Do not use any label names that are also the names of registers in the 8051. These are called "reserved" words and will cause great agitation in the assembler.

Byte Jumps

Byte jumps—jump instructions that test bytes of data—behave as bit jumps. If the condition that is tested is *true*, the jump is taken; if the condition is *false*,the instruction after the jump is executed. All byte jumps are relative to the program counter.

The following table lists examples of byte jumps:

Mnemonic	Operation
CJNE A,add,radd	Compare the contents of the A register with the contents of the direct address; if they are *not* equal, then jump to the relative address; set the carry flag to 1 if A is less than the contents of the direct address; otherwise, set the carry flag to 0
CJNE A,#n,radd	Compare the contents of the A register with the immediate number n; if they are *not* equal, then jump to the relative address; set the carry flag to 1 if A is less than the number; otherwise, set the carry flag to 0
CJNE Rn,#n,radd	Compare the contents of register Rn with the immediate number n; if they are *not* equal, then jump to the relative address; set the carry flag to 1 if Rn is less than the number; otherwise, set the carry flag to 0
CJNE @Rp,#n,radd	Compare the contents of the address contained in register Rp to the number n; if they are *not* equal, then jump to the relative address; set the carry flag to 1 if the contents of the address in Rp are less than the number; otherwise, set the carry flag to 0

DJNZ Rn,radd	Decrement register Rn by 1 and jump to the relative address if the result is *not* zero; no flags are affected
DJNZ add,radd	Decrement the direct address by 1 and jump to the relative address if the result is *not* 0; no flags are affected unless the direct address is the PSW
JZ radd	Jump to the relative address if A is 0; the flags and the A register are not changed
JNZ radd	Jump to the relative address if A is *not* 0; the flags and the A register are not changed

Note that if the direct address used in a DJNZ is a port, the port SFR is decremented and tested for 0.

Unconditional Jumps

Unconditional jumps do not test any bit or byte to determine whether the jump should be taken. The jump is *always* taken. All jump ranges are found in this group of jumps, and these are the only jumps that can jump to any location in memory.

The following table shows examples of unconditional jumps:

Mnemonic	Operation
JMP @A+DPTR	Jump to the address formed by adding A to the DPTR; this is an unconditional jump and will always be done; the address can be anywhere in program memory; A, the DPTR, and the flags are unchanged
AJMP sadd	Jump to absolute short range address sadd; this is an unconditional jump and is always taken; no flags are affected
LJMP ladd	Jump to absolute long range address ladd; this is an unconditional jump and is always taken; no flags are affected
SJMP radd	Jump to relative address radd; this is an unconditional jump and is always taken; no flags are affected
NOP	Do nothing and go to the next instruction; NOP (no operation) is used to waste time in a software timing loop; or to leave room in a program for later additions; no flags are affected

The following program example uses byte and unconditional jumps:

```
ADDRESS   MNEMONIC          COMMENT
          .ORG 0100h        ;begin program at 0100h
BGN:      MOV A,#30h        ;A = 30h
          MOV 50h,#00h      ;RAM location 50h = 00h
AGN:      CJNE A,50h,AEQ    ;compare A and the contents of 50h in RAM
          SJMP NXT          ;SJMP will be executed if (50h) = 30h
AEQ:      DJNZ 50h,AGN      ;count RAM location 50h down until (50h) =
          NOP               ;A; (50h) will reach 30h before 00h
NXT:      MOV R0,#0FFh      ;R0 = FFh
DWN:      DJNZ R0,DWN       ;count R0 to 00h; loop here until done
          MOV A,R0          ;A = R0 = 00h
          JNZ ABIG          ;the jump will not be taken
          JZ AZR0           ;the jump will be taken
```

Continued

ADDRESS	MNEMONIC	COMMENT
Continued		
ABIG:	NOP	;this address will not be reached
	ORG 1000h	;start this segment of program code at ;1000h
AZRO:	MOV A,#08h	;A = 08h (code at 1000,1h)
	MOV DPTR,#1000h	;DPTR = 1000h (code at 1002,3,4h)
	JMP @A+DPTR	;jump to location 1008h (code at 1005h)
	NOP	;(code at 1006h)
	NOP	;(code at 1007h)
HERE:	AJMP AZRO	;(code at 1008h, all code on page 2)

CAUTION

DJNZ decrements *first, then* checks for 0. A location set to 00h and then decremented goes to FFh, then FEh, and so on, down to 00h.

CJNE does not change the contents of any register or RAM location. It can change the carry flag to 1 if the destination byte is less than the source byte.

There is no zero flag; the JZ and JNZ instructions check the contents of the A register for 0.

JMP @A+DPTR does not change A, DPTR, or any flags.

Calls and Subroutines

The life of a microcontroller would be very tranquil if all programs could run with no thought as to what is going on in the real world outside. However, a microcontroller is specifically intended to interact with the real world and to react, very quickly, to events that require program attention to correct or control.

A program that does not have to deal unexpectedly with the world outside of the microcontroller could be written using jumps to alter program flow as external conditions require. This sort of program can determine external conditions by moving data from the port pins to a location and jumping on the conditions of the port pin data. This technique is called "polling" and requires that the program does not have to respond to external conditions quickly. (Quickly means in microseconds; slowly means in milliseconds.)

Another method of changing program execution is using "interrupt" signals on certain external pins or internal registers to automatically cause a branch to a smaller program that deals with the specific situation. When the event that caused the interruption has been dealt with, the program resumes at the point in the program where the interruption took place. Interrupt action can also be generated using software instructions named *calls*.

Call instructions may be included explicitly in the program as mnemonics or implicitly included using hardware interrupts. In both cases, the call is used to execute a smaller, stand-alone program, which is termed a *routine* or, more often, a *subroutine*.

Subroutines

A *subroutine* is a program that may be used many times in the execution of a larger program. The subroutine could be written into the body of the main program everywhere it is needed, resulting in the fastest possible code execution. Using a subroutine in this manner has several serious drawbacks.

Common practice when writing a large program is to divide the total task among many programmers in order to speed completion. The entire program can be broken into smaller parts and each programmer given a part to write and debug. The main program

can then call each of the parts, or subroutines, that have been developed and tested by each individual of the team.

Even if the program is written by one individual, it is more efficient to write an oft-used routine once and then call it many times as needed. Also, when writing a program, the programmer does the main part first. Calls to subroutines, which will be written later, enable the larger task to be defined before the programmer becomes bogged down in the details of the application.

Finally, it is quite common to buy "libraries" of common subroutines that can be called by a main program. Again, buying libraries leads to faster program development.

Calls and the Stack

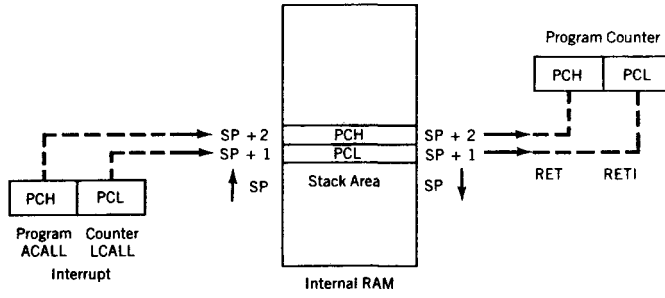
A call, whether hardware or software initiated, causes a jump to the address where the called subroutine is located. At the end of the subroutine the program resumes operation at the opcode address immediately *following* the call. As calls can be located anywhere in the program address space and used many times, there must be an automatic means of storing the address of the instruction following the call so that program execution can continue after the subroutine has executed.

The *stack* area of internal RAM is used to automatically store the address, called the return address, of the instruction found immediately after the call. The stack pointer register holds the address of the *last* space used on the stack. It stores the return address above this space, adjusting itself upward as the return address is stored. The terms "stack" and "stack pointer" are often used interchangeably to designate the *top* of the stack area in RAM that is pointed to by the stack pointer.

Figure 6.2 diagrams the following sequence of events:

1. A call opcode occurs in the program software, or an interrupt is generated in the hardware circuitry.
2. The return address of the next instruction after the call instruction or interrupt is found in the program counter.
3. The return address bytes are pushed on the stack, *low* byte *first*.
4. The stack pointer is incremented for each push on the stack.
5. The subroutine address is placed in the program counter.
6. The subroutine is executed.
7. A RET (return) opcode is encountered at the end of the subroutine.

FIGURE 6.2 Storing and Retrieving the Return Address



8. Two pop operations restore the return address to the PC from the stack area in internal RAM.
9. The stack pointer is decremented for each address byte pop.

All of these steps are automatically handled by the 8051 hardware. It is the *responsibility* of the programmer to ensure that the subroutine ends in a RET instruction *and* that the stack does not grow up into data areas that are used by the program.

Calls and Returns

Calls use short- or long-range addressing; returns have no addressing mode specified but are always long range. The following table shows examples of call opcodes:

Mnemonic	Operation
ACALL sadd	Call the subroutine located on the same page as the address of the opcode immediately following the ACALL instruction; push the address of the instruction immediately after the call on the stack
LCALL ladd	Call the subroutine located anywhere in program memory space; push the address of the instruction immediately following the call on the stack
RET	Pop two bytes from the stack into the program counter

Note that no flags are affected unless the stack pointer has been allowed to erroneously reach the address of the PSW special-function register.

Interrupts and Returns

As mentioned previously, an *interrupt* is a hardware-generated call. Just as a call opcode can be located within a program to automatically access a subroutine, certain pins on the 8051 can cause a call when external electrical signals on them go to a low state. Internal operations of the timers and the serial port can also cause an interrupt call to take place.

The subroutines called by an interrupt are located at fixed hardware addresses discussed in Chapter 2. The following table shows the interrupt subroutine addresses.

INTERRUPT	ADDRESS (HEX) CALLED
IE0	0003
TF0	000B
IE1	0013
TF1	001B
SERIAL	0023

When an interrupt call takes place, hardware interrupt disable flip-flops are set to prevent another interrupt of the same priority level from taking place until an interrupt return instruction has been executed in the interrupt subroutine. The action of the interrupt routine is shown in the table below.

Mnemonic	Operation
RETI	Pop two bytes from the stack into the program counter and reset the interrupt enable flip-flops

Note that the only difference between the RET and RETI instructions is the enabling of the interrupt logic when RETI is used. RET is used at the ends of subroutines called by an opcode. RETI is used by subroutines called by an interrupt.

The following program example use a call to a subroutine.

```
ADDRESS   MNEMONIC        COMMENT
MAIN:     MOV 81h,#30h    ;set the stack pointer to 30h in RAM
          LCALL SUB       ;push address of NOP; PC = #SUB; SP = 32h
          NOP             ;return from SUB to this opcode
          . . .
          . . .
SUB:      MOV A,#45h      ;SUB loads A with 45h and returns
          RET             ;pop return address to PC; SP = 30h
```

CAUTION

Set the stack pointer above any area of RAM used for additional register banks or data memory.

The stack may only be 128 bytes *maximum;* which limits the number of successive calls with no returns to 64.

Using RETI at the end of a *software* called subroutine may enable the interrupt logic erroneously.

To jump out of a subroutine (not recommended), adjust the stack for the two return address bytes by POPing it twice or by moving data to the stack pointer to reset it to its original value.

Use the LCALL instruction if your subroutines are normally placed at the end of your program.

In the following example of an interrupt call to a routine, timer 0 is used in mode 0 to overflow and set the timer 0 interrupt flag. When the interrupt is generated, the program vectors to the interrupt routine, resets the timer 0 interrupt flag, stops the timer, and returns.

```
ADDRESS   MNEMONIC        COMMENT
          .ORG 0000h      ;begin program at 0000
          AJMP OVER       ;jump over interrupt subroutine
          .ORG 000Bh      ;put timer 0 interrupt subroutine here
          CLR 8Ch         ;stop timer 0; set TR0 = 0
          RETI            ;return and enable interrupt structure
          .
          .
OVER:     MOV 0A8h,#82h   ;enable the timer 0 interrupt in the IE
          MOV 89h,#00h    ;set timer operation, mode 0
          MOV 8Ah,#00h    ;clear TL0
          MOV 8Ch,#00h    ;clear TH0
          SET 8Ch         ;start timer 0; set TR0 = 1
;
;
;
;the program will continue on and be interrupted when the timer has
;timed out
```

CAUTION

The programmer must enable any interrupt by setting the appropriate enabling bits in the IE register.

Example Problems

We now have all of the tools needed to write powerful, compact programs. The addition of the decision jump and call opcodes permits the program to alter its operation as it runs.

EXAMPLE PROBLEM 6.1

Place any number in internal RAM location 3Ch and increment it until the number equals 2Ah.

▪ **Thoughts on the Problem** The number can be incremented and then tested to see whether it equals 2Ah. If it does, then the program is over; if not, then loop back and decrement the number again.

Three methods can be used to accomplish this task.

▪ ***Method 1:***

```
ADDRESS   MNEMONIC      COMMENT
ONE:      CLR C         ;this program will use SUBB to detect equality
          MOV A,#2Ah    ;put the target number in A
          SUBB A,3Ch    ;subtract the contents of 3Ch; C is cleared
          JZ DONE       ;if A = 00h, then the contents of 3Ch = 2Ah
          INC 3Ch       ;if A is not zero, then loop until it is
          SJMP ONE      ;loop to try again
DONE:     NOP           ;when finished, jump here and continue
```

As there is no compare instruction for the 8051, the SUBB instruction is used to compare A against a number. The SUBB instruction subtracts the C flag also, so the C flag has to be cleared before the SUBB instruction is used.

▪ **Method 2:**

```
ADDRESS   MNEMONIC      COMMENT
TWO:      INC 3Ch       ;incrementing 3Ch first saves a jump later
          MOV A,#2Ah    ;this program will use XOR to detect equality
          XRL A,3Ch     ;XOR with the contents of 3Ch; if equal, A = 00h
          JNZ TWO       ;this jump is the reverse of program one
          NOP           ;finished when the jump is false
```

Many times if the loop is begun with the action that is to be repeated until the loop is satisfied, only one jump, which repeats the loop, is needed.

▪ **Method 3:**

```
ADDRESS   MNEMONIC          COMMENT
THREE:    INC 3Ch           ;begin by incrementing the direct address
          MOV A,#2Ah        ;this program uses the very efficient CJNE
          CJNE A,3Ch,THREE  ;jump if A and (3Ch) are not equal
          NOP               ;all done
```

CJNE combines a compare and a jump into one compact *instruction.*

EXAMPLE PROBLEM 6.2

The number A6h is placed somewhere in external RAM between locations 0100h and 0200h. Find the address of that location and put that address in R6 (LSB) and R7 (MSB).

■ **Thoughts on the Problem** The DPTR is used to point to the bytes in external memory, and CJNE is used to compare and jump until a match is found.

```
ADDRESS   MNEMONIC          COMMENT
          MOV 20h,#0A6h     ;load 20h with the number to be found
          MOV DPTR, #00FFh  ;start the DPTR below the first address
MOR:      INC DPTR          ;increment first and save a jump
          MOVX A,@DPTR      ;get a number from external memory to A
          CJNE A,20h,MOR    ;compare the number against (20h) and
                            ;loop to MOR if not equal
          MOV R7,83h        ;move DPH byte to R7
          MOV R6,82h        ;move DPL byte to R6; finished
```

This program might loop forever unless we know the number will be found; a check to see whether the DPTR has exceeded 0200h can be included to leave the loop if the number is not found before DPTR = 0201h.

EXAMPLE PROBLEM 6.3

Find the address of the first two internal RAM locations between 20h and 60h which contain consecutive numbers. If so, set the carry flag to 1, else clear the flag.

■ **Thoughts on the Problem** A check for end of memory will be included as a Called routine, and CJNE and a pointing register will be used to search memory.

```
ADDRESS   MNEMONIC          COMMENT
          MOV 81h,#65h      ;set the stack above memory area
          MOV R0,#20h       ;load R0 with address of memory start
NXT:      MOV A,@R0         ;get first number
          INC A             ;increment and compare to next number
          MOV 1Fh,A         ;store incremented number at 1Fh
          INC R0            ;point to next number
          CALL DUN          ;see if R0 greater than 60h
          JNC THRU          ;DUN returns C = 0 if over 60h
          MOV A,@R0         ;get next number
          CJNE A,1Fh,NXT    ;if not equal then look at next pair
          SETB 0D7h         ;set the carry to 1; finished
THRU:     SJMP THRU         ;jump here if beyond 60h
DUN:      PUSH A            ;save A on the stack
          CLR C             ;clear the carry
          MOV A,#61h        ;use XOR as a compare
          XRL A,R0          ;A will be 0 if equal
          JNZ BCK           ;if not 0 then continue
          RET               ;A 0, signal calling routine
BCK:      POP A             ;get A back
          CPL C             ;A not 0, set C to indicate not done
          RET
```

Set the stack pointer to put the stack out of the memory area in use.

Summary

Jumps

Jumps alter program flow by replacing the PC counter contents with the address of the jump address. Jumps have the following ranges:

Relative: up to PC +127 bytes, PC −128 bytes away from the PC

Absolute short: anywhere on a 2K-byte page

Absolute long: anywhere in program memory

Jump opcodes can test an individual bit, or a byte, to check for conditions that make the program jump to a new program address. The bit jumps are shown in the following table:

INSTRUCTION TYPE	RESULT
JC radd	Jump relative if carry flag set to 1
JNC radd	Jump relative if carry flag cleared to 0
JB b,radd	Jump relative if addressable bit set to 1
JNB b,radd	Jump relative if addressable bit cleared to 0
JBC b,radd	Jump relative if addressable bit set to 1 and clear bit to 0

Byte jumps are shown in the following table:

INSTRUCTION TYPE	RESULT
CJNE destination,source,address	Compare destination and source; jump to address if *not* equal
DJNZ destination,address	Decrement destination by one; jump to address if the result is *not* zero
JZ radd	Jump A = 00h to relative address
JNZ radd	Jump A > 00h to relative address

Unconditional jumps make no test and are always made. They are shown in the following table:

INSTRUCTION TYPE	RESULT
JMP @A+DPTR	Jump to 16-bit address formed by adding A to the DPTR
AJMP sadd	Jump to absolute short address
LJMP ladd	Jump to absolute long address
SJMP radd	Jump to relative address
NOP	Do nothing and go to next opcode

Call and Return

Software calls may use short- and long-range addressing; returns are to any long-range address in memory. Interrupts are calls forced by hardware action and call subroutines located at predefined addresses in program memory. The following table shows calls and returns:

INSTRUCTION TYPE	RESULT
ACALL sadd	Call the routine located at absolute short address
LCALL ladd	Call the routine located at absolute long address
RET	Return to anywhere in the program at the address found on the top two bytes of the stack
RETI	Return from a routine called by a hardware interrupt and reset the interrupt logic

Problems

Write programs for each of the following problems using as few lines of code as you can. Place comments on each line of code.

1. Put a random number in R3 and increment it until it equals E1h.
2. Put a random number in address 20h and increment it until it equals a random number put in R5.
3. Put a random number in R3 and decrement it until it equals E1h.
4. Put a random number in address 20h (LSB) and 21h (MSB) and decrement them as if they were a single 16-bit counter until they equal random numbers in R2 (LSB) and R3 (MSB).
5. Random unsigned numbers are placed in registers R0 to R4. Find the largest number and put it in R6.
6. Repeat Problem 3, but find the smallest number.
7. If the lower nibble of any number placed in A is larger than the upper nibble, set the C flag to one; otherwise clear it.
8. Count the number of ones in any number in register B and put the count in R5.
9. Count the number of zeroes in any number in register R3 and put the count in R5.
10. If the signed number placed in R7 is negative, set the carry flag to 1; otherwise clear it.
11. Increment the DPTR from any initialized value to ABCDh.
12. Decrement the DPTR from any initialized value to 0033h.
13. Use R4 (LSB) and R5 (MSB) as a single 16-bit counter, and decrement the pair until they equal 0000h.
14. Get the contents of the PC to the DPTR.
15. Get the contents of the DPTR to the PC.
16. Get any two bytes you wish to the PC.
17. Write a simple subroutine, call it, and jump back to the calling program after adjusting the stack pointer.
18. Put one random number in R2 and another in R5. Increment R2 and decrement R5 until they are equal.
19. Fill external memory locations 100h to 200h with the number AAh.
20. Transfer the data in internal RAM locations 10h to 20h to internal RAM locations 30h to 40h.
21. Set every third byte in internal RAM from address 20h to 7Fh to zero.
22. Count the number of bytes in external RAM locations 100h to 200h that are greater than the random unsigned number in R3 *and* less than the random unsigned number in R4. Use registers R6 (LSB) and R7 (MSB) to hold the count.
23. Assuming the crystal frequency is 10 megahertz, write a program that will use timer 1 to interrupt the program after a delay of 2 ms.
24. Put the address of every internal RAM byte from 50h to 70h in the address; for instance, internal RAM location 6Dh would contain 6Dh.
25. Put the byte AAh in all internal RAM locations from 20h to 40h, then read them back and set the carry flag to 1 if any byte read back is not AAh.