Memory Addressing

Chapter 2

INTRODUCTION

In a microcomputer system, the microprocessor serves to manipulate the binary information which is stored in memory. The ease and speed with which the microprocessor can gain access to any particular word stored in the memory will determine the speed at which a particular program can be executed, and also the size of the program. It follows that the structure and execution time of the various instructions, which require interaction with memory, have great bearing on the overall size and efficiency of the complete microcomputer system, and are therefore of vital importance. A microprocessor with a powerful "instruction set" yields a design which will use the minimum number of memory elements, and therefore cost less. In practice, the memory is a much larger portion of a complete system than the microprocessor itself, and in even the smallest systems there are usually two memory elements (one for program and one for data) to one microprocessor, and it is not unusual to find 30 or more memory packages supporting one microprocessor. Clearly, the microprocessor system designer needs to study the instruction set very carefully to ensure that it gives the smallest program for the particular application under consideration. However, there is no single criterion which covers all jobs, and each situation must be assessed on its own merits.

Instructions which work with information stored in memory are usually known as "memory reference instructions," and they fall into two broad categories; those which address data, and those which address the program memory. A data reference instruction

might be "Store contents of accumulator at location ABCD," and a jump instruction is one type of instruction which causes interaction with the program memory. Generally speaking, once a program has been loaded into a microcomputer system it does not modify itself: this means that it is not normal to write information into the program memory during program run time and therefore most program memory reference instructions are of the type "Read contents of program memory location ABCD." There are two reasons for this restriction on the program memory: firstly, it is poor programming practice to have a program which modifies itself because it makes system crashes more likely, and secondly, program memory is often the "read only" type in which write operations are impossible.

ARITHMETIC AND LOGICAL OPERATIONS

There is a wide range of arithmetic and logical instructions that can be found in microprocessors. Some processors have only an elementary set such as add, subtract and shift; others include instructions such as exclusive-OR, AND, compare, etc. In general, most microprocessors carry out binary arithmetic using two's complement number representation, and one remarkable feature of nearly all units is that they include facilities for operations on BCD numbers as well as on binary numbers. This is something that many of the larger computers do not have.

The minimum set of arithmetic and logical instructions would probably be:

(1) Add
(2) Subtract
(3) Shift left through carry-link flag
(4) Shift right through carry-link flag
(5) Clear carry-link flag
(6) Clear accumulator
(7) Complement accumulator
(8) Complement carry.

Using this basic set of instructions it would be possible to build up other more sophisticated operations such as AND, OR, exclusive-OR, etc. Note in particular that instructions 1 and 2

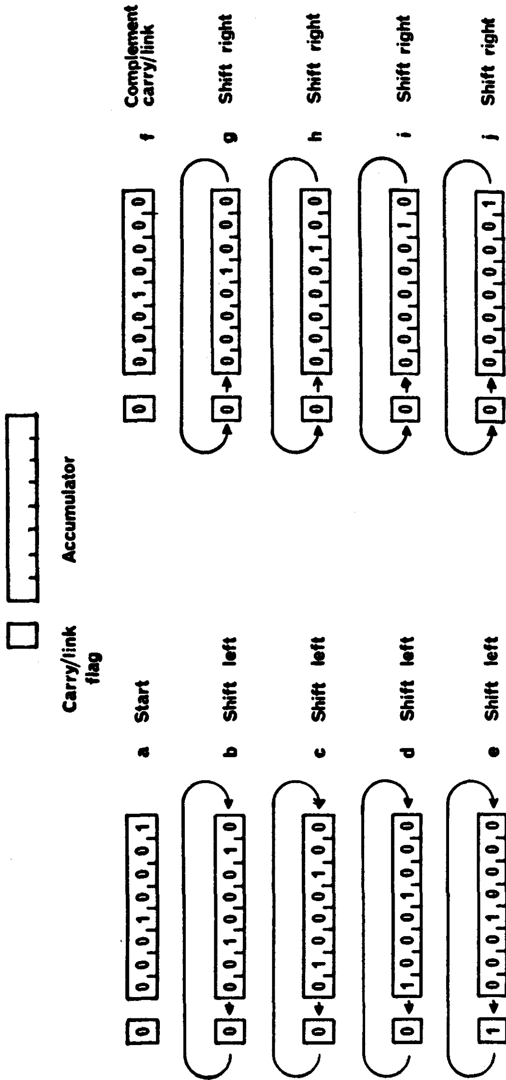


Figure 2-1. Complementing bit 4 by rotating through carry/link flag

above require two operands, with one most probably held in the accumulator and the other held in memory, whereas the remainder of the instructions operate on information already held within the microprocessor. This means that the instruction cycle for items 3 to 8 will be the simple procedure:

(a) Send out p.c. contents and fetch instruction
(b) Increment p.c.
(c) Execute the instruction.

But the instruction cycle for items 1 and 2 will be more complex. If the instruction is "Add the contents of the address specified by the index register to the accumulator," then the instruction cycle would be:

(a) Send out p.c. contents and fetch instruction
(b) Increment p.c.
(c) Send out index register contents as an address and fetch data to microprocessor
(d) Add accumulator and the data just fetched.

A memory reference instruction is implied by the add instruction because one of the operands is held in memory. In this particular case the appropriate address is specified by the index register, but there are many ways of specifying a particular data address and these will be the subject of much of this chapter.

The carry-link flag can in many cases be considered as a 1-bit extension of the accumulator register. By rotating the accumulator through the carry-link flag and using those instructions which set, clear and complement the carry-link flag, it is possible to gain access to and modify any single bit within the accumulator. Clearly this can be quite a slow process when, for example, it is required to complement bit 4 in an 8-bit accumulator. Such a program would involve 4 shifts left, complement carry-link, and 4 shifts right – a total of 9 instructions which is quite wasteful in program memory storage space (see Figure 2-1). A much neater operation would be to exclusive-OR the contents of the accumulator with 00010000. This would achieve the same result and probably use only 2 bytes of program storage. The more sophisticated arithmetic and logical instructions can therefore save quite a lot of program storage space. Instructions in this category include:

Increment
Decrement
AND
OR
Exclusive-OR
Compare
Logical and Arithmetic shifts.
(A logical shift is one in which the bit shifted in is determined by the state of the carry flag, whereas for an arithmetic shift, the bit shifted in is determined by the number convention in use – usually two's complement.)

The ability to operate with numbers in BCD format is an important feature because microprocessors are usually quite close to the man-machine interface, and it is often faster and more convenient to work with BCD numbers throughout than do conversions to and from binary. There are several levels of "BCD ability," ranging from simple instructions which convert the result of a BCD addition back into BCD right up to processors which can operate throughout in either a BCD or two's complement binary mode. For example, consider the case where 28 is added to 39, both numbers being represented in 8-bit BCD format. The result after a straightforward binary addition is shown below:

```
  0010:1000
  0011:1001
  ---------
  0110:0001
```

In order to restore the number to its true BCD result, the microprocessor has to execute a special instruction which converts the number in the accumulator back into BCD format – this instruction is usually called "decimal adjust accumulator." The decimal adjust accumulator instruction (usual mnemonic DAA) uses the information that a carry took place from the low order BCD character to the high order character to restore the least significant digit to its correct value of 0111. So for each BCD addition using the process described above the programmer has to write two instructions:

(1) Add
(2) Decimal adjust accumulator.

Carry flag before DAA	Upper half-byte	Half carry before DAA	Lower half-byte	Number added to Acc.	Carry flag after DAA
0	0 to 9	0	0 to 9	00	0
0	0 to 8	0	A to F	06	0
0	0 to 9	1	0 to 3	06	0
0	A to F	0	0 to 9	60	1
0	9 to F	0	A to F	66	1
0	A to F	1	0 to 3	66	1
1	0 to 2	0	0 to 9	60	1
1	0 to 2	0	A to F	66	1
1	0 to 3	1	0 to 3	66	1

Figure 2-2. DAA (decimal adjust accumulator) algorithm for an 8-bit microprocessor

Machines with separate binary and BCD addition instructions do not require the decimal adjust accumulator instruction and this gives a saving in program size. Figure 2-2 gives the full algorithm which the DAA instruction uses: it should be noted that it relies on the carry and half-carry flags for its operation. These are explained below.

ARITHMETIC FLAGS

The accumulator of a microprocessor usually has a number of flags (or flip-flops) associated with it. These flags signify that certain events have taken place and the operation of some instructions depends upon the state of these flags. The flags are often referred to as "status flags" and the binary word made up of their various states is called the "status word." The set of flags may be considered to be an additional register within the MPU called the "status register."

The status flags vary from processor to processor. The most fundamental flags are:

(a) Carry-link flag
(b) Half-carry flag (this is necessary for the DAA instruction).

The carry-link flag is set if an arithmetic operation yields a carry.

Alternatively it may be used with shift operations as described above. The half-carry flag is set if a carry propagates from the 4 least significant bits to the 4 most significant bits in the course of an arithmetic operation. Its main function is to aid the operation of the "decimal adjust accumulator" instruction, but it can be used for other purposes.

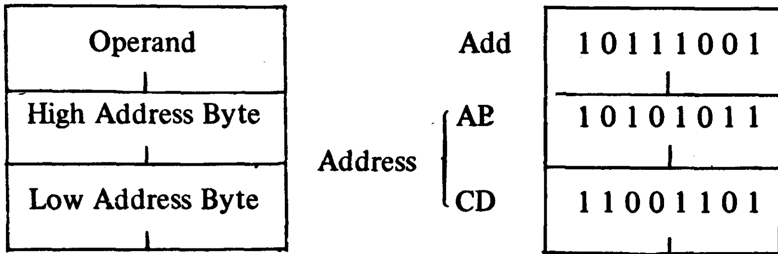
Other flags can include a parity flag, a sign flag, a zero flag relating to the number in the accumulator, and also an overflow flag denoting an overflow as the result of an arithmetic operation. It is important to recognise the difference between an overflow and a carry. The carry flag is set when an arithmetic operation results in a carry; the overflow flag is only set when an arithmetic operation yields a true arithmetic overflow. For example, +5 minus +3 in two's complement gives a carry-out of the high order end. However, the result +2 is still within the number range of the machine so the overflow is not set. Conversely, minus 100 plus minus 64 on an 8-bit machine gives a carry-out plus an overflow because the result is a positive 8-bit number even though the result should be negative in true arithmetic terms.

Machines with an interrupt system also have an interrupt mask flag which is usually included as part of the status register. This particular flag is related to input-output operations and is not directly affected by arithmetic operations (see Chapter 3).

MEMORY REFERENCE INSTRUCTIONS – CLASSICAL APPROACH

DIRECT, INDIRECT AND IMMEDIATE ADDRESSING

A memory reference instruction is one which refers to memory in order to obtain an operand. The instruction "Add contents of memory address ABCD to accumulator" is an example of a memory reference instruction. In an 8-bit microprocessor this instruction would probably have the structure as follows:



The type of instruction given here, where the address of the relevant data is contained explicity within the instruction, is known as direct addressing. This mode of addressing has severe limitations because, every time a data address is used, it has to be included in the instruction. For example, if it is required to find the average of four numbers stored in addresses ABCA, ABCB, ABCC and ABCD (hexadecimal notation), then the program using direct addressing would be:

Clear Accumulator
Add [ABCA]
Add [ABCD]
Add [ABCC]
Add [ABCD]
Shift Right
Shift Right

This program would leave the average value in the accumulator.

In Chapter 1 it was shown how it is possible to reduce the amount of program storage space by utilising the fact that the numbers are stored in consecutive locations. The index register was used to "point" to each data item and the instruction simply specified that the relevant data address was to be found in the index register. This type of instruction, where the actual data address is not explicit in the instruction but is held in some other location, is known as indirect addressing. In its most general form the pointer location can be any memory address and it can point to any other location.

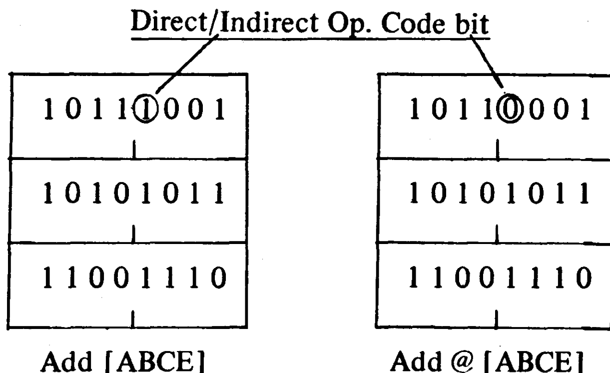
In the averaging example used above, it would be possible to use memory location ABCE as the address pointer by using the

instruction "Add @ [ABCE]." The @ sign denotes that indirect addressing is to be used. If it is assumed that when the program begins memory location ABCE holds the value ABCA, then the program using indirect addressing will be:

Clear Accumulator
Add @ [ABCE]
Increment [ABCE]
Add @ [ABCE]
Increment [ABCE]
Add @ [ABCE]
Increment [ABCE]
Add @ [ABCE]
Shift Right
Shift Right

Instead of using the index register to point to the consecutive memory addresses, the program has used memory location ABCE to store the pointer. As it stands, the program would in fact use more program memory space with indirect addressing than it would with direct addressing. However, as will be shown later, use may be made of the fact that the instruction sequence "Add @ [ABCE]" followed by "Increment [ABCE]" occurs three times.

The binary op-code for the instruction "Add @ [ABCE]" would not be the same as that for "Add [ABCE]"; it would most likely differ by 1 bit. The two instruction codes might be as shown below:



As indicated below, the instruction cycle for direct addressing is

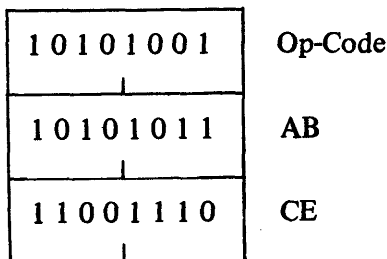
shorter than that for indirect addressing: this will mean that the cycle time for direct addressing is less than that for indirect addressing. The instruction cycle for direct addressing is:

Send out p.c. contents and fetch op-code
Increment p.c.
Send out p.c. contents and fetch high order address bytes
Increment p.c.
Send out p.c. contents and fetch low order address bytes
Increment p.c.
Send out data address, fetch data and execute instruction.

The instruction cycle for indirect addressing is:

Send out p.c. contents and fetch op-code
Increment p.c.
Send out p.c. contents and fetch high order pointer address bytes
Increment p.c.
Send out p.c. contents and fetch low order pointer address bytes
Increment p.c.
Send out pointer address and fetch data address
Send out data address, fetch data and execute instruction.

Apart from direct and indirect addressing, there is one other fundamental addressing mode known as immediate addressing. In immediate addressing the operand is contained within the instruction itself; a typical immediate addressing instruction might be "Add the constant ABCE (hexadecimal notation) to the accumulator." The binary instruction format would be



and a typical mnemonic would be "Add # ABCE." The symbol

# denotes immediate addressing. Since the relevant data is contained in the instruction itself, immediate addressing does not need to reference the data memory. Immediate addressing is useful where it is required to introduce constants into a program. Setting up address ABCE to contain ABCA for the start of the indirect addressing program above would probably be done using immediate addressing as follows:

Clear accumulator
Add # ABCA
Store [ABCE]

The value ABCA is added to the accumulator and then the contents of the accumulator are stored at address ABCE. In the above example the "store" instruction provides another example of direct addressing; it is of course possible to have store instructions which use indirect addressing.

REFINEMENTS TO MEMORY ADDRESSING MODES

Direct and indirect addressing can be further enhanced by including additional features such as autoincrement, autodecrement and indexed addressing. These refinements are additional to the basic addressing modes so that it is possible to have, for example, an indirect, autoincrement addressing mode. In the average program using indirect addressing, the sequence of instructions

Add @ [ABCE]
Increment [ABCE]

occurred several times. Some processors can combine these two instructions into one so that a single instruction adds the contents of the address specified by the address stored at location ABCE, and also automatically increments the contents of location ABCE. The automatic incrementing feature is known as autoincrement and a similar facility for decrementing is known as autodecrement. Automatic incrementing and decrementing are particularly useful where the program works with large quantities of data which are stored in consecutive memory locations. This occurs in matrix operations. Another refinement which is particularly useful in complicated programs is indexed addressing. With indexed

addressing the correct data address is calculated by adding an offset value to a specified address. Usually, the offset is stored in the index register and the specified address can be obtained by direct or indirect addressing. For example, if the index register contains 0005 and the instruction is "Add [0A00] indexed," then the correct data address would be obtained by adding 0A00 to 0005 to give the correct address at 0A05. It is possible to have an indexed autoincrement indirect addressing mode, where the correct address is obtained by adding the index register to the indirectly specified address and then the index register is automatically incremented: sometimes the indirect address is incremented rather than the index register.

RELATIVE ADDRESSING

Relative addressing is very similar to indexed addressing in that the correct address is calculated by adding an offset to some base address. The instruction contains the offset value and the program counter usually provides the base address. This relationship with the program counter means that relative addressing is often used in connection with jump instructions. Relative addressing is also important when writing programs which may be subsequently moved to another portion of the program memory.

CLASSICAL STRUCTURE OF THE OP-CODE FOR MEMORY REFERENCE INSTRUCTIONS

Very often the binary op-code for memory reference instructions is built up according to a neat logical pattern. This makes it much easier to write assembler programs. (An assembler is a computer program, which reads a program written in mnemonics and converts it into a program written in binary or hexadecimal characters.)

Figure 2-3 shows a possible structure for the op-code of an 8-bit microprocessor. This op-code would of course occur together with the appropriate address bytes.

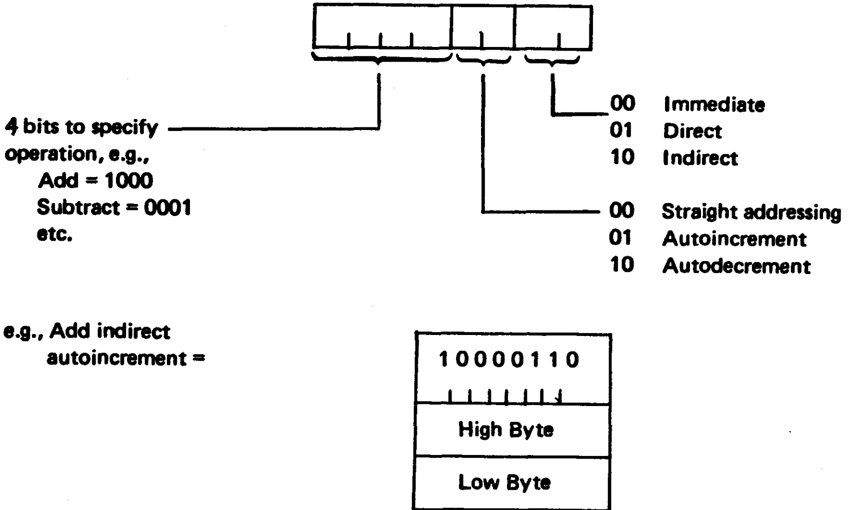


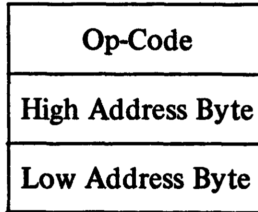
Figure 2-3 Possible op-code structure

MEMORY REFERENCE INSTRUCTIONS FOR MICROPROCESSORS

Microprocessors try to meet two criteria which often conflict; the need for a powerful instruction set and the requirement for instructions which use the minimum amount of program storage. In order to meet these goals various methods are used to reduce the number of bytes in a particular instruction. Usually this entails reducing the addressing capability of the memory reference instructions whilst retaining the essential character of the classical instruction. Some of the more common approaches are given below. It should be remembered that reducing the number of bytes in an instruction also reduces the cycle time of an instruction and therefore speeds up the program.

REGISTER ADDRESSING

The 3-byte indirect addressing instruction of the form



contains the ability to use any memory location as a pointer to the appropriate data address. In practice, this complete flexibility is usually not required and it is adequate to have only one or two "address pointers." By doing this it is possible to reduce the instruction from a three-byte instruction to a single-byte instruction where the actual address pointer is specified within the op-code. Most microprocessors have at least one register within the MPU which can be used as an address pointer and this is usually the index register. The example in Chapter 1 shows how it is possible to achieve a form of indirect addressing using the index register. Some MPU's have only one index register whereas others have several within the processor.

Register addressing is used as a standard approach for computers much larger than microprocessors because of the tremendous saving in program storage and the resultant improvement in instruction cycle time.

PAGING

Register addressing gives a considerable improvement for indirect addressing operations, but it is not applicable to direct addressing. One way of speeding up direct addressing operations is to use paging. With paging, the high address byte is stored within the microprocessor in some register and only the low address byte is specified by the direct addressing instruction. This approach makes use of the fact that most of the addresses which the microprocessor is working with at any one time are fairly close to each other, and therefore will have the same high order address byte. The splitting up of the memory address into a high order byte and a low order byte is akin to the division of a book into a series of pages, and hence the term paging.

The efficient use of paging normally requires at least one extra register in the MPU to specify the page number and also some additional instructions which make it possible to change page numbers whenever necessary. It also entails taking extra care when programming to make sure that page boundaries are not crossed inadvertently. An alternative to having an extra register for storing the page number is to restrict direct addressing to a single page, say page 0, so that when a direct addressing instruction is received by the MPU, it knows that the high order bits are all logical zero. This method also saves having extra instructions for changing page numbers.

Paging is a very powerful tool and has been applied in many different ways. For example, it can be used in conjunction with the program counter to reduce the number of bytes in a jump instruction. Broadly speaking, paging finds greatest application in 4- and 8-bit microprocessors and where the number of available pins on a microprocessor is severely restricted.

ON-CHIP REGISTER ADDRESSING

Some microprocessors have several additional general purpose registers within the MPU as shown in Figure 2-4. Usually special instructions are provided so that these registers can be directly addressed with a single-byte instruction. The term "inherent" addressing has been applied to this type of operation because the

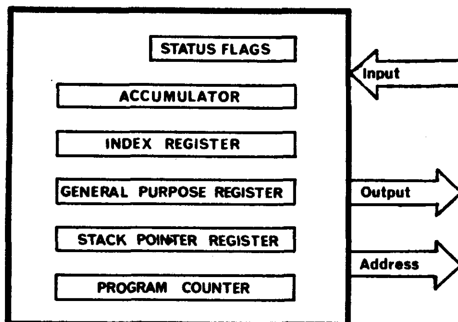


Figure 2-4. Microprocessor with multiple internal registers

data address is inherent in the single-byte op-code. These general purpose registers may be used for data storage or as address pointers in the same way as the index register is used.

BUILDING UP SOPHISTICATED MEMORY REFERENCE INSTRUCTIONS

Usually a microprocessor does not have all the addressing modes mentioned above, and in assessing a microprocessor it is worthwhile checking how many bytes of program are required to implement all the classical memory reference instructions. For example, quite a few microprocessors do not have any direct addressing facilities and a direct addressing instruction might have to be built up as follows:

Load the index register with the required address (3 bytes)
Add contents of address specified by index register to accumulator (1 byte).

Indirect addressing with autoincrement can be very difficult if there are no facilities for incrementing the on-chip registers. In this case, incrementing would have to be done using the accumulator so that the contents of the accumulator would have to be temporarily stored whilst the accumulator is used. In such a case the two instructions

Add contents of memory location specified by index register to accumulator (1 byte)
Increment index register (1 byte)

might have to be written

Add contents of location specified by index register to accumulator	(1 byte)
Exchange contents of accumulator and index register	(1 byte)
Increment accumulator	(1 byte)
Exchange contents of accumulator and index register	(1 byte)
Total	4 bytes.

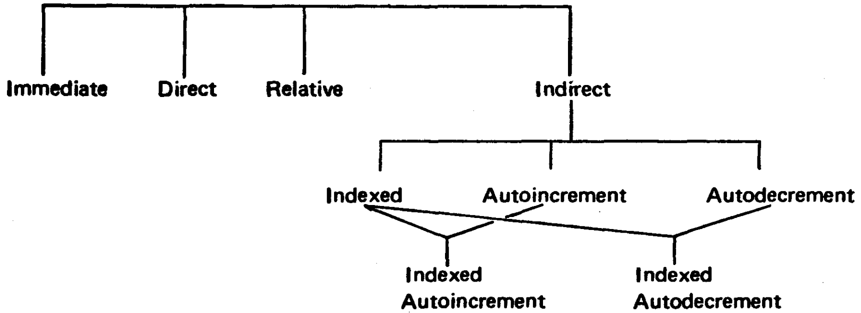


Figure 2-5. Fundamental addressing modes

It is unlikely that a microprocessor has ever been designed with such a restrictive set of instructions, but the above example does serve to illustrate the problems that can easily arise. Figure 2-5 gives a table of the various classical addressing modes for easy reference.

JUMP AND CONDITIONAL JUMP INSTRUCTIONS

Jump instructions are ones which can cause the program counter to be loaded with a new value instead of allowing it to continue through the program. This is an important point because the manner and the type of instructions which can change the program counter vary widely. The second, and equally important feature of jump instructions, is that "conditional jump" instructions give the program the ability to make decisions. An example of this might be "Jump to specified address if accumulator is zero, otherwise continue with normal program flow." Such an instruction would test the contents of the accumulator to see if it is zero. If it is, the program counter would be loaded with the address specified in the instruction; if the accumulator was not zero, the program counter contents would remain untouched and the program would continue in a straight numerical sequence. To illustrate this, consider the example in this chapter which finds the average of four numbers stored in locations ABCA, ABCB, ABCC, and ABCD. Assume that the microprocessor which is to be used has the internal register structure shown in Figure 2-4. Note in particular that a general purpose register (GPR) has been added to

the fundamental architecture used so far. This general purpose register will be used for counting the number of times the program executes the add instruction. The averaging program now becomes:

	Instruction No.	Program memory store
(1)	Load immediate index register with ABCA	3 bytes
(2)	Load immediate GPR with 0004	3 bytes
(3)	Clear accumulator	1 byte
(4)	Add contents of location specified by index register	1 byte
(5)	Increment index register	1 byte
(6)	Decrement GPR	1 byte
(7)	If GPR ≠ zero jump back to add instruction	3 bytes
(8)	Shift right	1 byte
(9)	Shift right	1 byte
	Total	15 bytes.

Figure 2-6 shows the corresponding program flow chart.

The conditional jump instruction "If GPR does not equal zero, jump back to add instruction" provides the program with the ability to make the elementary decision on whether to go around the loop once more, or whether to finish the addition. In practice the conditional jump contains an address to which the jump should take place rather than the indefinite statement "Jump to add instruction." The final program pattern in program memory would be as shown in Figure 2-7, where the jump instruction now specifies a jump to program memory location 0007. If the general purpose register is not equal to zero, the program counter will be loaded with the value 0007 and its current contents 000D will be lost.

If the general-purpose register had not been included, it would have been much more difficult to write the program, because the results of the addition and the contents of the loop counter would have had to be continually exchanged and moved around to utilise the accumulator, and any advantage in using the addition

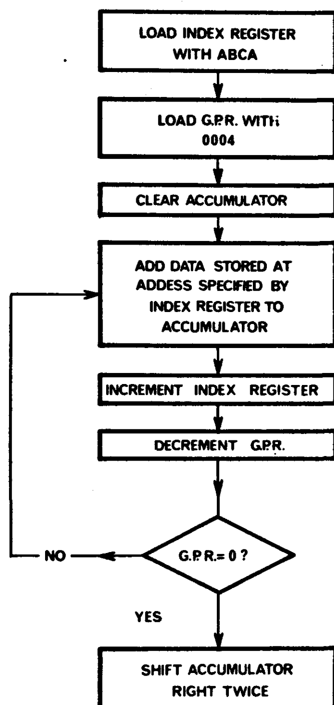


Figure 2-6. Program flow-chart for averaging example

loop approach would have been lost. For this reason all microprocessors include at least one general-purpose register for loop counting and other functions. Another point to note from the above example is that great use is made of the indirect addressing approach using the index register. The ability to dynamically create a new address from existing information within the program is essential for most applications.

The address to which the program jump takes place does not have to be explicit in the program as it was above. The particular type of jump used above was a version of a conditional immediate addressing instruction: "If condition is satisfied, load program counter with data which is contained in this instruction." In just the same way it is possible to use direct and indirect addressing to specify the value with which the program counter must be loaded.

Instruction No.	Program memory contents	Program memory address (hex.)
	Load IR immediate	0000
1	A B	0001
	C A	0002
	Load GPR immediate	0003
2	0 0	0004
	0 4	0005
3	Clear accumulator ←	0006
4	Add @ IR	0007
5	Increment IR	0008
6	Decrement GPR	0009
	If GPR ≠ 0 Jump ┘	000A
7	0 0	000B
	0 7	000C
8	Shift right	000D
9	Shift right	000E

Figure 2-7. Program memory contents for averaging program

For example, the instruction "If condition is satisfied load program counter with the contents of address specified by the index register" is an indirect addressing jump instruction. These types of jumps add great power to the instruction set, and furthermore can reduce the number of bytes in an instruction. A jump instruction with immediate or direct addressing requires three bytes of memory, whereas an instruction with indirect addressing using the index register can be accommodated in a single program byte.

In some processors the program counter, the index register, the general-purpose register(s) and the accumulator are all regarded as general-purpose registers for some operations. Therefore the instruction "Load immediate ABCD to register A" could have quite different effects depending on what function register A is used for. For example, if the registers are lettered as follows,

A – Accumulator
B – Index register
C – General purpose register
D – Program counter

then "Load immediate ABCD to register A" will load the accumulator with value ABCD, but "Load immediate ABCD to register D" would cause a program jump to the program location ABCD. This approach can give the suprising impression that at first glance some microprocessors appear to have no specific jump instructions.

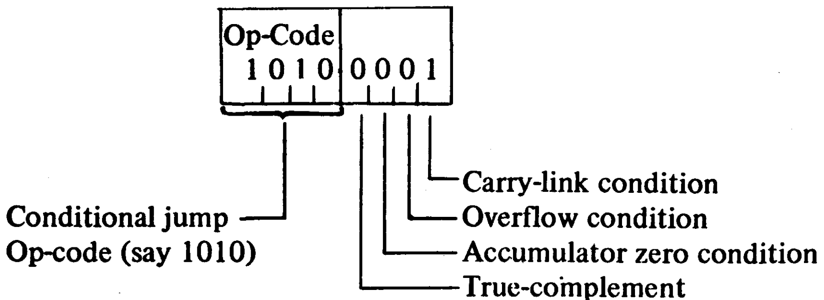
CONDITIONAL JUMP INSTRUCTIONS

The unconditional jump causes a jump to another section of program without having to test for any particular condition existing. By comparison the conditional jump (sometimes called a conditional branch instruction) requires that a condition be satisfied before the jump can take place, and it is this condition that gives the power of decision.

Conditional jumps usually base their decisions on the condition of the various arithmetic flags. The following is a list of some of the conditions that are available:

Carry-link flag set	Carry-link flag not set
Half-carry flag set	Half-carry flag not set
Accumulator zero	Accumulator not zero
Overflow flag set	Overflow flag not set
Parity flag set	Parity flag not set
Accumulator positive	Accumulator negative.

It is normal for the binary op-code of a conditional jump instruction to have an ordered structure such as that shown below.



For example, "Jump to ABCD if carry-link set," would be:

1 0 1 0 0 0 0 1
1 0 1 0 1 0 1 1
1 1 0 0 1 1 0 1

The true-complement-bit gives the instruction the power to negate the condition so that the instruction "Jump to 0007 if accumulator not zero," would become:

1 0 1 0 1 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 1

The "1" in the true-complement-bit signifies that the complement of the condition should be satisfied, which in this case is a zero condition in the accumulator. Another common feature is the power to AND conditions together. For example, the instruction "Jump to ABBA if accumulator is zero AND carry-link is set" would become:

1 0 1 0 0 1 0 1
1 0 1 0 1 0 1 1
1 0 1 1 1 0 1 0

REDUCING THE NUMBER OF BYTES IN A JUMP INSTRUCTION

In previous sections various methods were discussed for reducing the number of bytes in memory reference instructions. The methods of indirect register addressing and paging are applicable

to jump instructions. Several forms of paging can be used with jump instructions: the most simple form is where the high order program counter address bytes remain unchanged by the jump instruction, and only the low order bytes are loaded from the jump instruction. This means that it is only possible to jump within a program page but this is often adequate. An alternative approach is to add some number to the program counter. This number can either be specified in the jump instruction (i.e., immediate addressing) or in a register. The term "relative addressing" (see section RELATIVE ADDRESSING) is used to describe this operation since the jump is by some displacement relative to the program counter.

In microprocessors using 8-bit instruction bytes, and a 16-bit address, relative addressing makes it possible to reduce a jump instruction from three bytes to two bytes.

One particular type of instruction which can achieve a conditional jump with only a single instruction byte is the skip instruction which has the form "If condition XYZ is satisified then skip the next n instructions." This makes it possible for the program to jump over the n instructions without executing them. The skip instruction has not been used much in microprocessors to date although some of the early devices used it to reduce the amount of program bytes in loop-counting applications.

SUBROUTINES

The instruction "Jump to subroutine" is much the same as a normal jump instruction. However, when a subroutine jump takes place, the program counter contents are not lost, but are temporarily stored in some special location where they can be recovered for later use. Subroutine jumps can be conditional or unconditional. A subroutine is used where a section of program such as multiplication is repeated several times during the course of the overall task. To write the same program each time it is needed is very wasteful in program storage space. The subroutine jump provides a method of holding the subroutine (e.g., multiply) in program memory only once. Each time the program is required, the main program jumps to the subroutine, and once the sub-task

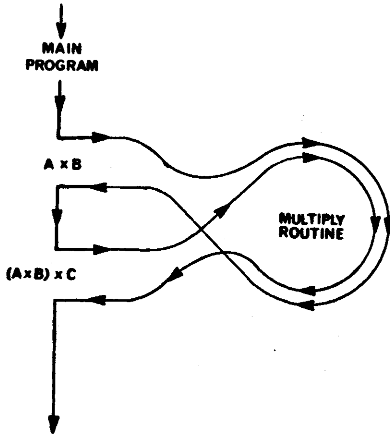


Figure 2-8a.

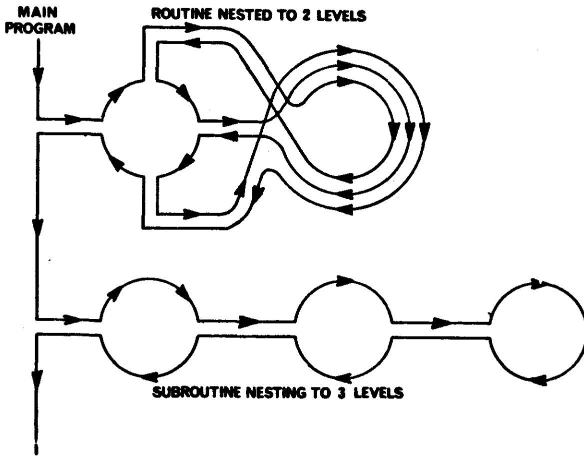


Figure 2-8b.

Figure 2-8. Illustrations of program flow using subroutines

is completed, control is returned to the program at the point it left off. Figure 2-8a shows schematically how a subroutine would be used in the case of a program to multiply three numbers together. Often it is required to "nest" subroutines so that one subroutine can call another subroutine. For example, an integration subroutine might call a division subroutine, and this

situation of nested subroutines is schematically shown in Figure 2-8b. The problem in these circumstances is how to store the program counter contents, so that each time a return from subroutine is made the program counter is reloaded with the correct value.

In order to overcome this difficulty the concept of a subroutine "stack" is usually used. Each time a jump to subroutine is made, the current program counter contents are loaded onto a stack in the same way as one would stack plates. The value loaded onto the stack is always placed at the top of the stack. When a return from subroutine is made, the value stored at the top of the stack is loaded back into the program counter. These two operations are usually called "Push" for loading onto the stack, and "Pop" for taking values from off the stack. A "Jump to subroutine" instruction would have the following basic instruction cycle:

Send out p.c. contents and fetch instruction
Increment p.c.
Load p.c. contents onto stack
Load p.c. with address of subroutine specified in instruction.

Stacks can be implemented in two basic ways. One is to use a shift-register (sometimes called "Pushdown stack") where a "Push" corresponds to shifting the register one direction and a "Pop" corresponds to shifting in the opposite direction. The other method is to use a random access memory plus a stack pointer. The stack pointer is a register within the MPU which is specifically reserved for the purpose of keeping track of the next memory location available on the stack. A "Jump to subroutine" instruction with a microprocessor which uses a stack pointer has the instruction cycle:

Send out p.c. contents and fetch instruction
Increment p.c.
Load p.c. contents at memory location specified by stack pointer register
Increment stack pointer
Load p.c. with start address of subroutine.

The corresponding "Return from subroutinc" would have the following instruction cycle:

Send out p.c. contents and fetch instruction
Decrement stack pointer
Load p.c. from address specified by stack pointer.

The shift register type of stack has been chiefly used where the stack is an integral part of the microprocessor chip, but it has the severe limitation that the number of stages in the "shift register" are limited, and therefore the number of levels to which subroutines can be nested are limited. With the stack-pointer approach, the stack can be any size the system designer chooses since the stack is usually exterior to the processor and can be part of the same memory as the data storage memory.

Push and Pop instructions can be quite powerful in their own right, particularly if instructions provide for Pushing and Popping other registers, apart from the program counter, onto the stack. Furthermore, in some cases it is desirable to store not only the p.c. contents on the stack when a jump to subroutine occurs, but also the contents of all the other registers. This is particularly true in the case of real-time interrupts (see Chapter 3) and re-entrant subroutines. A re-entrant subroutine is one which can be interrupted by one program and then called by another program. Once the second call has been completed, the interrupted subroutine is restarted where it left off. Since a subroutine usually changes the contents of the various registers, it is necessary to save all register contents before starting another run of the same subroutine. The Push and Pop instructions for various registers give this ability.

In addition to the stack architectures described above, several other methods have been used to save the program counter contents when jumping to a subroutine. One approach is to use multiple program counters and simply switch from one program counter to another when a subroutine call is made: this method gives a very fast change over. Another method is to exchange the program counter contents with the contents of a general-purpose register. Where limited subroutine nesting is required this technique is adequate, but subroutine nesting to several levels usually has to be done by means of software and this slows down the overall program flow.

MULTIPLE ADDRESS MACHINES

To date the text has concentrated on machines where each addressing instruction specified one data address: where two operands are required, as in an add instruction, the second operand is provided by the accumulator. More general instructions would be of the type:

(i) Add contents of address A to contents of address B and place results at address C

or

(ii) Add contents of address A to contents of address B and place results at address B.

Type (i) is for a 3-address machine and is only found in very large computers. Type (ii) is for a 2-address machine. Instructions like this are much more powerful than those for single-address machines discussed so far, but in order to specify more than one address more bits are needed in the instruction, and so the amount of program storage for each instruction increases. Of course the addresses do not have to be directly addressed and, in fact, indirect addressing via registers is often preferable. This gives a significant reduction in instruction size and complexity. A machine operating in the 2-address mode might typically have 8 general-purpose registers. An add instruction could take the form:

Add contents of address specified by register 1 to contents of address specified by register 2 and place result at address specified by register 2 and increment register 1.

Such an instruction uses indirect addressing via registers for both addresses with autoincrement applied to register 1.

Multiple-address machines are of particular advantage in applications that process large amounts of data: the advantage is less pronounced where the emphasis is on real-time data handling.

SUMMARY

This chapter has attempted to describe the types of instruction

usually used with microprocessors. Because microprocessors are small machines with a restricted number of pin connections, certain economies have to be made and various techniques are employed so as to obtain maximum computing power within these constraints. Inevitably different microprocessors use different instructions and architectures to achieve their goal. A microprocessor intended for "number crunching" applications such as intelligent calculators, would place emphasis on sophisticated addressing modes, whereas a device intended for, say, domestic control purposes, like an automatic washing machine, would be more concerned with obtaining a low-cost product with easy input-output facilities. The microprocessor designer chooses the best compromise compatible with his terms of reference, and consequently, no two microprocessors have identical instruction sets.