MATCH-MAKING

We have already considered the use of indexed addressing on the 6809 processor. Here we examine how this is used to perform simple arithmetic on values in the index registers and discuss the use of subroutines in a string-matching program.

In the previous instalment of the course we took our first look at indexed addressing on the 6809 processor. In indexed mode addressing, the effective address specified by, for example, OFFSET,X is formed as the sum of the offset (which can be a constant or the contents of a memory location) and the current value held in the index register specified (in this case, the X register). We saw that in some common situations the offset may be zero, in which case we can write ,X (although 0,X would also work). In special cases, one of the accumulators A, B or D can be used for the offset (e.g. B,X). And we took a look at how one of the most common uses of indexing — stepping through a table of values — can be made easier by the use of auto-increment and auto-decrement mode. This mode increments a register by one or two after the instruction has been carried out (,X+ and ,X++), or decrements the register by one or two before the instruction is carried out (,-Y and ,--Y).

Now we can briefly look at how indexed addressing can be used to perform some simple arithmetic on values in the index registers using the LEA (Load Effective Address) instruction. The normal arithmetic instructions will not work on the values in registers other than the accumulators. Although it is possible to transfer the contents of the index register into the D accumulator, perform the arithmetic and then transfer the result back, this is an awkward and slow procedure. The LEA instruction (which can be applied to the X, Y, S and U registers only) will perform any necessary address calculations and then load the effective address value. Normally the contents of an effective address would be loaded, so this is a useful alternative.

Let's take a look at an example. The instruction:

```
LEAX   -1,X
```

will calculate the effective address as the sum of −1 and the current contents of the X register. This address is then loaded back into X, effectively decrementing the value in that register. This is not the only use of this instruction; it could be used, for example, to carry out an address calculation once and save the result, rather than perform that same calculation a number of times.

It is also possible to do a certain amount of arithmetic on the X register using the ABX (Add B to X) instruction, which does an unsigned addition of the contents of B to the contents of X. However, this is not as generally useful as LEA.

SUBROUTINES

A subroutine is a self-contained section of code that is called from the main program (or another subroutine) to perform a specific task. Once that job has been done, control is automatically transferred back to the calling program at the instruction immediately following the original subroutine call. There are three main reasons for using subroutines:

1) To save writing the same piece of code more than once. It is more convenient to write an often used piece of code as a subroutine and call this when it is required.
2) So that a library of common routines can be built up, and then used in a number of different programs.
3) To break a program down into smaller, more manageable sections.

The most significant thing to remember about using subroutines in Assembly language is that both the calling program and the subroutine will be using the same registers. One of the most common errors in machine code programming occurs when, having stored a value in one of the registers, a program calls a subroutine and on its return finds that the contents of that register have been altered by the subroutine. Therefore, it is vital to know, and to document, the registers that a subroutine uses. It is particularly essential to save the contents of the registers being used when a subroutine is called, and restore those contents when control returns from the subroutine.

Later in the course we will look at how the stacks are used both as a convenient way of saving such data, and as a means of passing values and addresses (parameters) to the subroutine. For the moment, however, we shall assume that the subroutine uses the same data as the calling program (global variables) and any other values that it needs will actually be in the registers. A subroutine call is made by means of one of these instructions:

- BSR: Branch to SubRoutine
- JSR: Jump to SubRoutine

The BSR command causes a relative branch — it finds the subroutine at a certain offset from the current value of the program counter. This instruction is normally used for subroutines written as part of the program.

The JSR instruction calls a subroutine at a certain specified address. This would be used for a