An optimized loop mode of operation substantially improves a microprocessor's throughput while retaining full compatibility with its predecessor's software.

Built-in tight-loop mode raises μP's performance

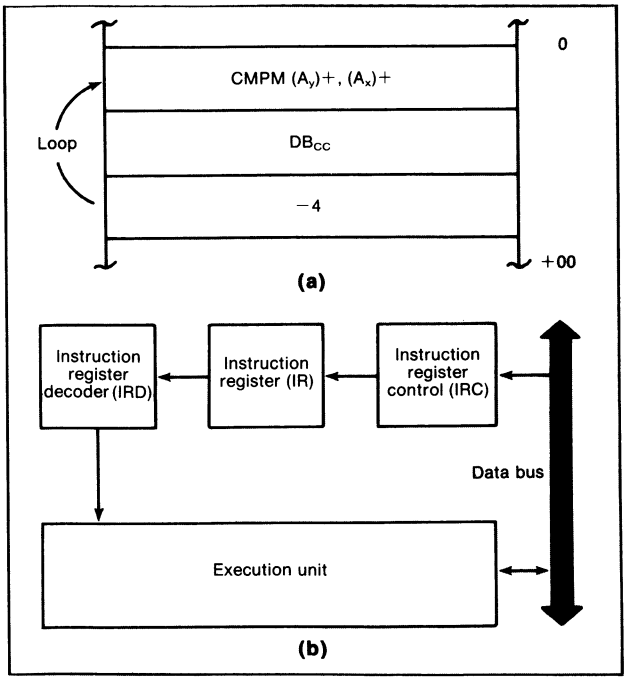
SOFTWARE

When a microprocessor family becomes widely established and its users have made a large investment in software, new members in the family offering improved performance need to be program-compatible with existing family members. With this constraint, upgrading requires that the new unit be designed with minimal change to the internal micromachine—generally imposing a severe limitation on the improvements that can be made.

One change, however, that does not affect the compatibility of the 68000 with its more powerful 68010 upgrade but that still dramatically improves performance is a built-in automatic method of efficiently executing "tight loops." The method's elegance and generality make it particularly appropriate: a less general solution would have been glaringly inconsistent with the powerful instructions, addressing modes, and word sizes available in the 68000 microprocessor family. The 68010's hardware and instruction set are basically the same as the 68000's except for the 68010's enhancements for implementing virtual-memory and virtual-machine systems; two new general-purpose data-moving instructions; and of course, the tight-loop mode of operation.

Looping is an operation heavily associated with branching—an extensively used microprocessor function. When an instruction sequence branches a very short distance from a particular (or the current) location in the sequence and returns to that location, the instuction sequence is called a tight loop. More specifically, the 68010 automatically recognizes a tight-loop sequence when the destination of the branch instruction falls within the length of the processor's instruction pipeline—a displacement value of −4. The 68010 instruction pipeline, like that of the 68000, overlaps the fetching of succeeding instructions with decoding and executing the current instruction.

The tight loop method is transparent to the user, because the instruction stream requires no modification. The processor merely interprets the situation—certainly a most compatible method of



1. The loop example (a) compares memory values A_y and A_x. Because the destination of the loop's conditional-branching-with-counter (DB_{cc}) operation falls within the length of the instruction pipe (b)—a displacement of −4—it is considered to be a tight loop.

Douglas MacGregor, Microcomponent Design Engineer
Bill Moyer, Microcomponent Design Engineer
Motorola Inc.
3501 Ed Bluestein Blvd., Austin, Texas 78721

enhancing performance. Moreover the method inherently provides a large degree of generality. Should the length of the microprocessor's pipeline ever be extended, the distance branched could be increased without refetching the instruction stream.

Thus, when the 68010 processor detects a tight-loop sequence, it automatically alters its normal execution activity to take advantage of the length of its internal instruction pipeline. The instruction stream is circulated within the machine, thereby eliminating the need to make superfluous instruction accesses. The result is nearly a 50% reduction in tight-loop execution time without the need to alter the instruction set at all.

Something about branch instructions

The 68000 includes several types of branch instructions: a conditional (1 of 15 conditions), and a sixteenth unconditional type, a subroutine type and a conditional-branch-with-counter type. The conditional and the unconditional branch instructions are used most frequently, although the subroutine type is very important in certain applications.

One instruction, the conditional branch with counter (DB_{cc}), although it is particularly interesting and powerful, has attained little popularity, largely because of its complex nature (see "The DB_{cc} Instruction," p. 228).

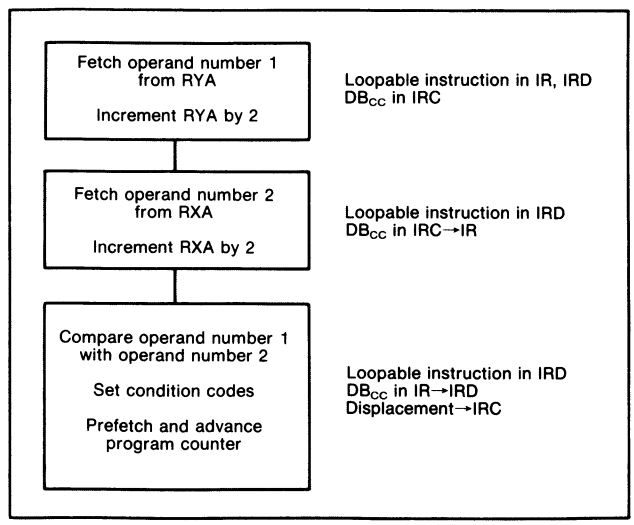
The DB_{cc} type of branching is primarily used in short, or tightly looped, branching operations which, in turn, find heavy applications in the implementation of such high-level functions as array arithmetic, extended-precision arithmetic, and string and block operations. These functions typically require the execution of very few instructions per loop iteration. Of course, the powerful addressing modes and rich instruction set of the 68000 family help keep the loops short by enabling operations such as initialization, arithmetic, logical, array and extended multiprecision data procedures to be carried out by brief code sequences.

Naturally, the more general execution of a large loop containing a sequence of many instructions per iteration can be efficiently carried out by the DB_{cc} instruction too. Depending on the size of the loop, either a byte or 16-bit word displacement can be added to the program counter (PC) to form the branch destination address. For a short branch—one involving a destination that lies within ±128 bytes of the current PC location—only a single 16-bit instruction word is required. With a 16-bit displacement word, however, loops as large as ±32 kbytes can be controlled with a single branch instruction. Accordingly, the branching overhead time is minimal when compared with the total in-

Loopable instructions

Op codes	Applicable addressing modes	
MOVE [BWL]	(A_y) to (A_x) (A_y) to $(A_x)+$ (A_y) to $-(A_x)$ $(A_y)+$ to (A_x) $(A_y)+$ to $(A_x)+$ $(A_y)+$ to $-(A_x)$	$-(A_y)$ to (A_x) $-(A_y)$ to $(A_x)+$ $-(A_y)$ to $-(A_x)$ R_y to (A_x) R_y to $(A_x)+$
ADD [BWL] AND [BWL] CMP [BWL] OR [BWL] SUB [BWL]	(A_y) to D_x $(A_y)+$ to D_x $-(A_y)$ to D_x	
ADDA [WL] CMPA [WL] SUBA [WL]	(A_y) to A_x $-(A_y)$ to A_x $(A_y)+$ to A_x	
ADD [BWL] AND [BWL] EOR [BWL] OR [BWL] SUB [BWL]	D_x to (A_y) D_x to $(A_y)+$ D_x to $-(A_y)$	
ABCD [B] ADDX [BWL] SBCD [B] SUBX [BWL]	$-(A_y)$ to $-A_x$	
CMP [BWL]	$(A_y)+$ to $(A_x)+$	
CLR [BWL] NEG [BWL] NEGX [BWL] NOT [BWL] TST [BWL] NBCD [B]	(A_y) $(A_y)+$ $-(A_y)$	
ASL [W] ASR [W] LSL [W] LSR [W] ROL [W] ROR [W] ROXL [W] ROXR [W]	(A_y) by #1 $(A_y)+$ by #1 $-(A_y)$ by #1	

Note: B, W, and L indicate an operand size of byte, word, or long word.



2. The first time the loopable instruction (see Fig. 1) is executed, it is treated as it would be in the 68000; without counting the instructions in the loop.

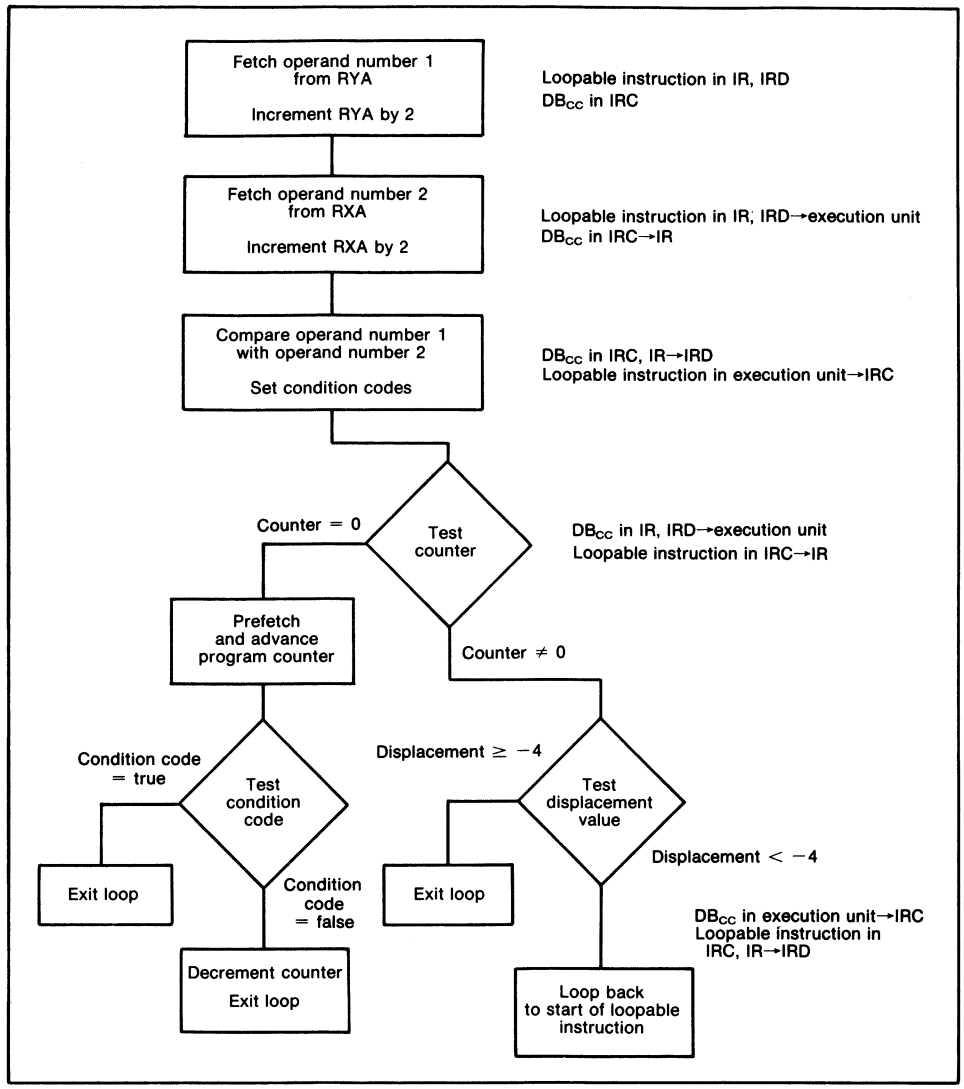
struction execution time within the loop.

With the 68010, when the destination of a DB_{cc} instruction falls within the length of the processor's instruction pipe, and when the instruction that is the destination of the branch is considered a loopable instruction (see the table, p. 226), the processor automatically executes both the loop and the DB_{cc} instruction as an integral operation. At the end of each iteration, the processor executes the normal activity associated with the beginning of a new instruction (which includes the sampling of interrupts, as well as the conducting of tracing, if required). If after a check for such "exceptions" the condition codes and the counter value indicate that another iteration is required, the combined loopable and DB_{cc} instructions are executed again.

Neither the loopable instruction nor the DB_{cc} instruction, as well as its associated displacement word, are fetched again until the loop-mode operation is terminated. The only external references made during loop-mode operation are operand transfers, thus eliminating the overhead of instruction fetching.

For example, consider the Compare Memory with Memory [CMPM W (A_g) + (A_x) +] instruction in the tight loop of Fig. 1a. The instruction pipe of the 68010 consists of three registers and a path into the execution unit (Fig. 1b). In the figure, IRD is a decoding register that contains data for various residual decoding operations, whereas the IR and IRC instruction registers provide sequencing control and a limited amount of residual control.

The first time that this tight loop is executed, the loopable instruction and the DB_{cc} instruction are



3. After the first loop, if the displacement is within the length of the instruction pipe, subsequent iterations consist of a combination of DB_{cc} and loop back to the first instruction (in the so-called loop mode). However, because instructions need to be fetched again, considerable time is saved over that of normal looping.

treated normally (Fig. 2). At the end of the execution of the DB_{cc} instruction, however, the sequence determines whether the loop mode should be entered. If the displacement value does not exceed −4 and the branch conditions are met, then the loop mode is entered and an internal status bit is set. Thereafter, the 68010 executes the distinct microroutine that is the composite of both the loopable instruction and the DB_{cc} instruction (Fig. 3).

In order to present an identical programmer's model when dealing with exceptions, the loop status bit is cleared at the termination of each normal instruction, and only at the end of a DB_{cc} instruction and a subsequent loop instruction routine is the loop bit set. Accordingly, if the program is being traced, the trace microsequence is executed before the end of the DB_{cc} operation, the status bit is not set, which automatically prevents the loop mode

The DB_{cc} instruction

The DB_{cc} instruction is a general branching primitive that allows loop control with two exit conditions. DB_{cc} tests the condition code register for one of sixteen conditions, which includes always true, or unconditional, branching:

Carry clear	Low or same
Carry set	Less
Equal	Minus
Never true	Not equal
Greater or equal	Plus
Greater	Always true
High	No overflow
Less or equal	Overflow

Based on the result of the test, either the loop is exited or a word-sized loop counter in one of eight user-specified data registers is decremented. When the result of the decrementing is −1, the looping is terminated. As long as the condition tested remains false and the counter has not reached −1, the branch is made and the looping continues. Thus the DB_{cc} can be thought of simply as an iteration counter combined with a branch conditional instruction.

Consider a simple example, where N consecutive 8-bit data items are to be initialized to zero. A typical loop would clear the first item, decrement N by one, test for zero, and continue the loop if N is not equal to −1. A simple loop using the DBRA instruction in the DB_{cc} operation can carry out this procedure:

```
        LEA      data item, AO  /* point to first item */
        MOVE.W   #N-1, DO       /* get loop count in DO */
LOOP    CLR. B   (AO)+          /* clear data item, advance
                                   pointer */
        DBRA     DO, LOOP       /* decrement count, check
                                   for −1 */
```

The loop terminates only when the count of items has expired, because a true always (false condition) branching condition is in effect.

If the loop is to be terminated after a certain condition is found, the count field of a data register may be set to −1 before the loop is entered. The loop then continues until the termination condition arrives, and then the loop is exited. By inverting the value in the data register, the number of iterations through the loop may be obtained. This use of DB_{cc} is helpful in counting certain types of data items or in searching for a specific value through a contiguous list of values where the search requires an actual count of iterations.

Suppose the length of a string terminated by a data byte of $00 is to be calculated. This is a typical operation in the C programming environment where strings are represented as a sequence of characters (bytes) terminated by a null byte equal to $00. A typical assembly language sequence for strings up to 64 kbytes might be:

```
        LEA      string, AO     /* get address of first char-
                                   acter in string */
        MOVEQ    #−1, DO
LOOP    TST. B   (AO)+          /* test the current byte in
                                   string */
        DBEG     DO, LOOP       /* if last byte was zero, exit,
                                   else loop */
        NOT. L   DO             /* DO now contains length
                                   of string */
```

Note that the DB_{cc} combines the normal count (add or subtract) and the conditional branch instruction into a single primitive. Combining this string with the previous looping example permits searching a string of a known length for a particular character or a value. This use of the DB_{cc} uses both conditions as a basis for loop exiting.

Many systems place an upper limit on the length of a string or the index values of an array of elements. Then when the end of the string or array is reached, the loop action must terminate. The example that follows searches an array of elements designated ARRAY[0:N], for a specific value, where N is the number of elements in the array and 0 is the value to be matched:

```
        LEA      string, AO     /* get address of first char-
                                   acter in string */
        MOVE. L  #N, DO         /* get number of elements
                                   into DO */
        ADDA. L  DO, AO         /* get pointer set up */
        MOVEQ    #Q, D1         /* value to be matched into
                                   D1 */
LOOP    CMP. B   −(AO), D1      /* test the current byte in
                                   string */
        DBEQ     DO, LOOP       /* if last byte was zero, exit,
                                   else loop */
                                /* AO points to element
                                   matched or to ARRAY[O] */
                                /* DO is element # in array
                                   that matches */
```

When the value is found, general-purpose register DO contains the index of the value in the array, and the zero bit (Z) in the condition code register will be set once the loop terminates.

from executing. In this way, the integrity of a tracing operation is maintained, even when the loop mode is called for. Otherwise, the trace mode, which also starts at the end of a DB_{cc} instruction would interfere with the loop mode.

String functions are easily written to take advantage of the loop-mode capability of the 68010. Loops which manipulate such strings are particularly common in high-level-language programming. For example, consider the character string functions in the high-level C programming language—STRLEN, STRCMP, and STRCPY—which determine the length of a string, compare two strings, and copy one string to another, respectively. A character string—a sequence of bytes terminated with a byte equal to zero—represents an array of characters. Indexing provides successive characters of the string, and the length of a string need not be fixed; however, an upper limit may be imposed by the string definition.

A single DB_{cc} loop can implement the string-length function, STRLEN, up to 64 kbytes long with:

```
      LEA      string, AO  /* get address of first character in
                           string*
      MOVEQ    #−1, DO
LOOP  TST. B   (AO) +      /* test the current byte in string*/
      DBEQ     DO, LOOP    /* if last byte was zero, exit, else loop
                           */
      NOT. W   DO          /* DO now contains length of string
                           */
```

Moreover, the string can be extended to occupy the entire 32-bit address space of the 68010 with the addition of only a single outer loop.

With only eight clocks of overhead required before entering the loop, the 68010 could then execute this function for a typical 80 byte string in 905 clock cycles, or 91 μs, at a clock frequency of 10 MHz. The equivalent code on a 68000 running at 10 MHz would take 146 μs for the same string length. Each iteration takes 12 clocks on the 68010 vs 18 clocks on the 68000—a significant performance improvement.

Similarly the string copy function, STRCPY, can be coded to copy strings of up to 64 kbytes long:

```
      LEA      string1, A0     /* get address of first string */
      LEA      string2, A1     /* get address of second string
                               */
      MOVEQ    £*−1, DO
LOOP  MOVE. B  (A0)+, (A1)+    /* copy character from string1
                               to string2 */
      DBEQ     DO, LOOP        /* if last byte was zero, exit, else
                               loop */
```

An average string length of 80 characters would execute in 116 μs on the 10-MHz 68010, with each loop iteration requiring just 14 clock cycles. At the same clock frequency, the 68000 performs the function in about 195 μs, with a 24-clock-loop iteration.

To demonstrate this increase in speed, consider a single dimensional array of 32-bit data with index numbers running from zero to some value N [0:N]. The first step, generally is to initialize the array. Some languages require the array elements to be cleared upon allocation. The automatic loop mode supports such an operation function quite nicely:

```
      LEA      ARRAY, AO   /* get address of first element
                           of array */
      MOVE.W   #N, DO      /* set DO to max array dimen-
                           sion */
LOOP  CLR.L    (AO)+       /* clear the current 32-bit array
                           element */
      DBRA     DO, LOOP    /* loop through array indices */
```

For an array of 100 elements of 32 bits each, the sequence executes in 143 ms on the 68010 operating at 10 MHz, with each loop iteration requiring just 14 clocks. The 68000, however, at the same clock frequency, performs the identical function in about 302 ms with a loop iteration time of 30 clocks. Because of its fast tight-loop mode, the 68010, therefore, is more than twice as fast at executing the array initialization as the 68000.

Another use of the loop mode is in extended-precision arithmetic. In addition to the usual handling of 32-bit data with single instructions, the 68000 architecture provides extended instructions ADDX (Add Extended), SUBX (Subtract Extended), ROXL/ROXR (Rotate Extended), ABCD/SBCD/NBCD (Add/Sub/Negate BCD), and NEGX (Negate Extended). With these instructions, words of unlimited size may be constructed and operated on. For example, consider the use of the loop mode in the addition of one 512-bit integer to another, where the integers are represented by N (16 in this case) consecutive 32-bit memory locations. The loop appears to be very similar to the last example:

```
      LEA      digit1, AO      /* initialize pointer for first inte-
                               ger */
      LEA      digit2, A1      /* initialize pointer for second in-
                               teger */
      MOVE.W   #N−1, DO        /* get loop iteration count ini-
                               tialized */
      ANDI. B  #$0, CCR        /* clear condition codes for
                               extended operation */
LOOP  ADDX. L  −(AO), −(A1)    /* add integer 1 to integer 2
                               using extended arithmetic */
      DBRA     DO, LOOP        /* loop until add complete */
```

The 68010, operating at 10 MHz, performs the addition in 56 μs, and each loop iteration requires 32 clock cycles for each additional 32 bits of operand. The 68000 can execute the same loop in 71 μs and requires 40 clock cycles per additional iteration.□

MOTOROLA ***Semiconductor Products Inc.***

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721 • A SUBSIDIARY OF MOTOROLA INC.

A16684-1 PRINTED IN USA 4-85 IMPERIAL LITHO C30282 5,000

AR-211