Transporting M68HC11 Code to M68HC12 Devices

By James M. Sibigtroth

1 INTRODUCTION

In general, the CPU12 is a proper superset of the M68HC11 CPU. Significant changes have been made to improve the efficiency and capabilities of the CPU without sacrificing compatibility with the popular M68HC11 family. This note provides information that will allow the large number of programmers familiar with the M68HC11 to evaluate moving from an M68HC11 system to an M68HC12 system. For more detailed information, please refer to the *CPU12 Reference Manual,* Freescale Publication Number CPU12RM/AD. The manual is available on the Freeware Data Systems website: http://www.freeware.aus.sps.mot.com/.

1.1 CPU12 Design Goals

The primary goals of the CPU12 design were:

- **ABSOLUTE** source code compatibility with the M68HC11
- Same programming model
- Same stacking operations
- Upgrade to 16-bit architecture
- Eliminate extra byte/extra cycle penalty for using index register Y
- Improve performance
- Improve compatibility with high level languages

2 SOURCE CODE COMPATIBILITY

Every M68HC11 instruction mnemonic and source code statement can be assembled directly with a CPU12 assembler with no modi cations. CPU12 instructions affect condition code bits in the same way as M68HC11 instructions. The CPU12 supports all M68HC11 addressing modes and several new variations of indexed addressing mode.

CPU12 object code is similar to but not identical to M68HC11 object code. Some primary objectives, such as the elimination of the penalty for using Y, could not be achieved without object code differences. While the object code has been changed, the majority of the opcodes are identical to those of the M6800, which was developed more than 20 years earlier.

The CPU12 assembler automatically translates a few M68HC11 instruction mnemonics into functionally equivalent CPU12 instructions. For example, the CPU12 does not have an increment stack pointer (INS) instruction, so the INS mnemonic is translated to LEAS 1,S. The CPU12 does provide single-byte DEX, DEY, INX, and INY instructions because the LEAX and LEAY instructions do not affect the condition codes, while the M68HC11 instructions update the Z bit to according to the result of the operation.

Table 1 shows M68HC11 instruction mnemonics that are automatically translated into equivalent CPU12 instructions. The translation is performed by the assembler so there is no need to modify old M68HC11 code in order to assemble it for the CPU12. In fact, M68HC11 mnemonics can be used in new CPU12 programs.

Table 1 Translated M68HC11 Mnemonics

M68HC11 Mnemonic	Equivalent CPU12 Instruction	Comments
ABX ABY	LEAX B,X LEAY B,Y	Since CPU12 has accumulator offset indexing, ABX and ABY are rarely used in new CPU12 programs. ABX was one byte on M68HC11 but ABY was two bytes. The LEA substitutes are two bytes.
CLC CLI CLV SEC SEI SEV	ANDCC #$FE ANDCC #$EF ANDCC #$FD ORCC #$01 ORCC #$10 ORCC #$02	ANDCC and ORCC now allow more control over the CCR, including the ability to set or clear multiple bits in a single instruction. These instructions took one byte each on M68HC11 while the ANDCC and ORCC equivalents take two bytes each.
DES INS	LEAS –1,S LEAS 1,S	Unlike DEX and INX, DES and INS did not affect CCR bits in the M68HC11, so the LEAS equivalents in CPU12 duplicate the function of DES and INS. These instructions were one byte on M68HC11 and two bytes on CPU12.
TAP TPA TSX TSY TXS TYS XGDX XGDY	TFR A,CCR TFR CCR,A TFR S,X TFR S,Y TFR X,S TFR Y,S EXG D,X EXG D,Y	The M68HC11 had a small collection of specific transfer and exchange instructions. CPU12 expanded this to allow transfer or exchange between any two CPU registers. For all but TSY and TYS (which take two bytes on either CPU), the CPU12 transfer/exchange costs one extra byte compared to M68HC1. The substitute instructions execute in one cycle rather than two.

All of the translations produce the same amount of or slightly more object code than the original M68HC11 instructions. However, there are offsetting savings in other instructions. Y-indexed instructions in particular assemble into one byte less object code than the same M68HC11 instruction.

The CPU12 has a two-page opcode map, rather than the four-page M68HC11 map. This is largely due to redesign of the indexed addressing modes. Most of pages 2, 3, and 4 of the M68HC11 opcode map are required because Y-indexed instructions use different opcodes than X-indexed instructions.

Approximately two-thirds of the M68HC11 page 1 opcodes are unchanged in the CPU12. Some opcodes that are on other pages of the M68HC11 opcode map have been moved to page 1 of the CPU12 map. CPU12 object code for each of these instructions is one byte smaller than object code for the equivalent M68HC11 instruction. **Table 2** shows these instructions.

Instruction set changes offset each other to a certain extent. Programming style also affects the rate at which instructions appear. As a test, the BUFFALO monitor, an 8-Kbyte M68HC11 assembly code program, was reassembled for the CPU12. The resulting object code is six bytes smaller than the M68HC11 code. It is fair to conclude that M68HC11 code can be reassembled with very little change in size.

The relative size of M68HC11 and CPU12 code has also been tested by rewriting several smaller assembly programs from scratch. In these cases, the CPU12 code is typically about 30% smaller. These savings are mostly due to improved indexed addressing.

It is useful to compare the relative sizes of C programs. A C program compiled for the CPU12 is about 30% smaller than the same program compiled for the M68HC11. The difference is largely attributable to better indexing.

Table 2 Instructions With Smaller Object Code

Instruction	Comments
DEY INY	Page 2 opcodes in M68HC11 but page 1 in CPU12.
INST n,Y	For values of n less than 16 (the majority of cases). Were on page 2, now are on page 1. Applies to BSET, BCLR, BRSET, BRCLR, NEG, COM, LSR, ROR, ASR, ASL, ROL, DEC, INC, TST, JMP, CLR, SUB, CMP, SBC, SUBD, ADDD, AND, BIT, LDA, STA, EOR, ADC, ORA, ADD, JSR, LDS, and STS. If X is the index reference and the offset is greater than 15 (much less frequent than offsets of 0, 1, and 2), the CPU12 instruction assembles to one byte more of object code than the equivalent M68HC11 instruction.
PSHY PULY	Were on page 2, now are on page 1.
LDY STY CPY	Were on page 2, now are on page 1.
CPY n,Y LDY n,Y STY n,Y	For values of n less than 16 (the majority of cases). Were on page 3, now are on page 1.
CPD	Was on page 2, 3, or 4, now on page 1. In the case of indexed with offset greater than 15, CPU12 and M68HC11 object code are the same size.

3 PROGRAMMER'S MODEL AND STACKING

The CPU12 programming model (**Figure 1**) is identical to that of the M68HC11.

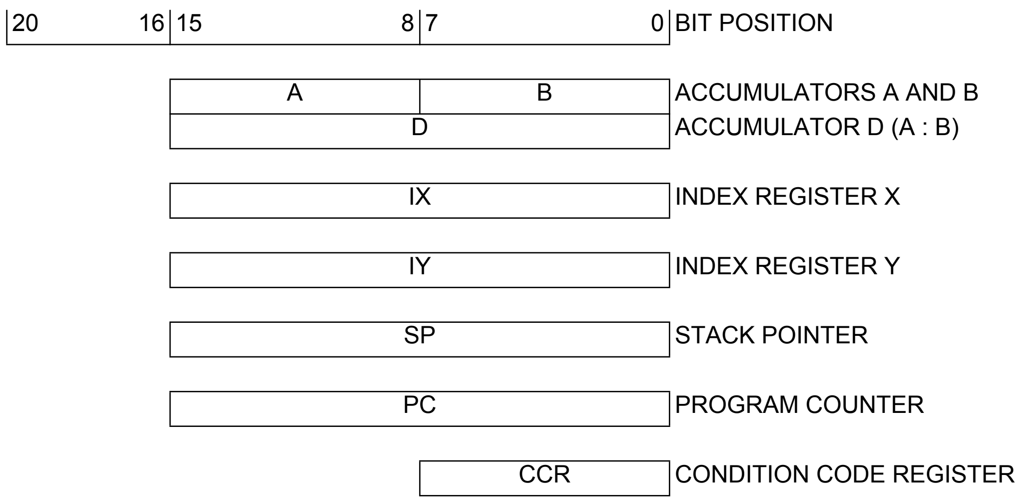


Figure 1 CPU12 Programming Model

Both the M68HC11 and the CPU12 stack nine bytes of system resources when an interrupt occurs. The stacking order is identical. However, since this is an odd number of bytes, there is no practical way to assure that the CPU12 stack will stay aligned. To assure that instructions take a fixed number of cycles regardless of stack alignment, the internal RAM in M68HC12 MCUs is designed to allow single cycle 16-bit accesses to misaligned addresses. As long as the stack is located in this special RAM, stacking and unstacking operations take the same amount of execution time, regardless of stack alignment. If the stack is located in an external 16-bit RAM, a PSHX instruction can take two or three cycles depending upon the alignment of the stack. This extra access time is transparent to the CPU because the integration module freezes the CPU clocks while it performs the extra 8-bit bus cycle required for a misaligned stack operation.

4 TRUE 16-BIT ARCHITECTURE

The M68HC11 is a direct descendant of the M6800, one of the first microprocessors, which was introduced in 1974. The M6800 was strictly an 8-bit machine, with 8-bit data buses and 8-bit instructions. As Freescale devices evolved from the M6800 to the M68HC11, a number of 16-bit instructions were added, but the data buses remained 8 bits wide, so these instructions were performed as sequences of 8-bit operations. The CPU12 is a true 16-bit implementation, but it retains the ability to work with the mostly 8-bit M68HC11 instruction set. The larger ALU of the CPU12 (it can perform some 20-bit operations) is used to calculate 16-bit pointers and to speed up math operations.

The CPU12 is a 16-bit processor with 16-bit data paths. Typical M68HC12 devices have internal and external 16-bit data paths, but some derivatives incorporate operating modes that allow for an 8-bit data bus, so that a system can be built with low-cost 8-bit program memory. M68HC12 MCUs include an on-chip integration module that manages the external bus interface. When the CPU makes a 16-bit access to a resource that is served by an 8-bit bus, the integration module performs two 8-bit accesses, freezes the CPU clocks for part of the sequence, and assembles the data into a 16-bit word. As far as the CPU is concerned, there is no difference between this access and a 16-bit access to an internal resource via the16-bit data bus. This is similar to the way an MC68HC11 can stretch clock cycles to accommodate slow peripherals.

5 INSTRUCTION QUEUE

The CPU12 has a two-word instruction queue and a 16-bit holding buffer, which sometimes acts as a third word for queueing program information. All program information is fetched from memory as aligned 16-bit words, even though there is no requirement for instructions to begin or end on even word boundaries. There is no penalty for misaligned instructions. If a program begins on an odd boundary (if the reset vector is an odd address), program information is fetched to fill the instruction queue, beginning with the aligned word at the next address below the misaligned reset vector. The instruction queue logic starts execution with the opcode in the low order half of this word.

The instruction queue causes three bytes of program information (starting with the instruction opcode) to be directly available to the CPU at the beginning of every instruction. As it executes, each instruction performs enough additional program fetches to refill the space it took up in the queue. Alignment information is maintained by the logic in the instruction queue. The CPU provides signals that tell the queue logic when to advance a word of program information, and when to toggle the alignment status.

The CPU is not aware of instruction alignment. The queue logic includes a multiplexer that sorts out the information in the queue to present the opcode and the next two bytes of information as CPU inputs. The multiplexer determines whether the opcode is in the even or odd half of the word at the head of the queue. Alignment status is also available to the ALU for address calculations. The execution sequence for all instructions is independent of the alignment of the instruction.

The only situation where alignment can affect the number of cycles an instruction takes occurs in devices that have a narrow (8-bit) external data bus, and is related to optional program fetch cycles (O type cycles). O cycles are always performed, but serve different purposes determined by instruction size and alignment.

Each instruction includes one program fetch cycle for every two bytes of object code. Instructions with an odd number of bytes can use an O cycle to fetch an extra word of object code. If the queue is aligned at the start of an instruction with an odd byte count, the last byte of object code shares a queue word with the opcode of the next instruction. Since this word holds part of the next instruction, the queue cannot advance after the odd byte executes, or the first byte of the next instruction would be lost. In this case, the O cycle appears as a free cycle since the queue is not ready to accept the next word of program information. If this same instruction had been misaligned, the queue would be ready to advance and the O cycle would be used to perform a program word fetch.

In a single-chip system or in a system with the program in16-bit memory, both the free cycle and the program fetch cycle take one bus cycle. In a system with the program in an external 8-bit memory, the O cycle takes one bus cycle when it appears as a free cycle, but it takes two bus cycles when used to perform a program fetch. In this case, the on-chip integration module freezes the CPU clocks long enough to perform the cycle as two smaller accesses. The CPU handles only 16-bit data, and is not aware that the 16-bit program access is split into two 8-bit accesses.

In order to allow development systems to track events in the CPU12 instruction queue, two status signals (IPIPE[1:0]) provide information about data movement in the queue and about the start of instruction execution. A development system can use this information along with address and data information to externally reconstruct the queue. This representation of the queue can also track both the data and address buses.

6 STACK FUNCTION

The CPU12 has a “last-used” stack rather than a “next-available” stack like the M68HC11 CPU. That is, the stack pointer points to the last 16-bit stack address used, rather than to the address of the next available stack location. This generally has very little effect, because it is very unusual to access stacked information using absolute addressing.

The change does allow a 16-bit word of data to be removed from the stack without changing the value of the SP twice. To illustrate, consider the operation of a PULX instruction. With the next-available M68HC11 stack, if the SP=$01F0 when execution begins, the sequence of operations is: SP=SP+1; load X from $01F1:01F2; SP=SP+1; and the SP ends up at $01F2. With the last-used CPU12 stack, if the SP=$01F0 when execution begins, the sequence is: load X from $01F0:01F1; SP=SP+2; and the SP again ends up at $01F2. The second sequence requires one less stack pointer adjustment.

The stack pointer change also affects operation of the TSX and TXS instructions. In the M68HC11, TSX increments the SP by one during the transfer, so that the the X index points to the last stack location used. The TXS instruction decrements the SP by one during the transfer for the same reason. CPU12 TSX and TXS instructions are ordinary transfers — the CPU12 stack requires no adjustment.

For ordinary uses of the stack, such as pushes, pulls, and manipulations involving TSX and TXS, the M68HC11 and CPU12 stacks appear identical. However, there is one very subtle difference.

The LDS #$xxxx instruction is normally used to initialize the stack pointer. In the M68HC11, the address specified in the LDS instruction is the first stack location used. In the CPU12, the first stack location used is one address lower than the address specified in the LDS instruction. Since the stack builds downward, M68HC11 programs re-assembled for the CPU12 operate normally, but stacked values are located one physical address lower in memory.

In very uncommon situations, such as test programs used to verify CPU operation, a program could initialize the SP, stack data, and then read the stack via an extended mode read (it is normally improper to read stack data from an absolute extended address). To make an M68HC11 source program that contains such a sequence work on the CPU12, the programmer must change either the initial LDS #$xxxx, or the absolute extended address used to read the stack.

7 IMPROVED INDEXING

The CPU12 has significantly improved indexed addressing capability, yet retains compatibility with the M68HC11. The one cycle and one byte cost of doing Y-related indexing in the M68HC11 has been eliminated. In addition, high level language requirements, including stack relative indexing and the ability to perform pointer arithmetic directly in the index registers, have been accommodated.

The M68HC11 has one variation of indexed addressing that works from X or Y as the reference pointer. For X indexed addressing, an 8-bit unsigned offset in the instruction is added to the index pointer to arrive at the address of the operand for the instruction. A load accumulator instruction assembles into two bytes of object code, the opcode and a 1-byte offset. Using Y as the reference, the same instruction assembles into three bytes (a page prebyte, the opcode, and a one-byte offset.) Analysis of M68HC11 source code indicates that the offset is most frequently zero and very seldom greater than four.

The CPU12 indexed addressing scheme uses a postbyte plus 0, 1, or 2 extension bytes after the instruction opcode. These bytes specify which index register is used, determine whether an accumulator is used as the offset, implement automatic pre/post increment/decrement of indices, and allow a choice of 5-, 9-, or 16-bit signed offsets. This approach eliminates the differences between X and Y register use and dramatically enhances indexed addressing capabilities.

Major improvements that result from this new approach are:

- Stack pointer can be used as an index register in all indexed operations
- Program counter can be used as index register in all but auto inc/dec modes
- Accumulator offsets allowed using A, B, or D accumulators
- Automatic pre- or post-, increment or decrement (by –8 to +8)
- 5-bit, 9-bit, or 16-bit signed constant offsets
- 16-bit offset indexed-indirect and accumulator D offset indexed-indirect

The change completely eliminates pages three and four of the M68HC11 opcode map and eliminates almost all instructions from page two of the opcode map. For offsets of +0 to +15 from the X index register, the object code is the same size as it was for the M68HC11. For offsets of +0 to +15 from the Y index register, the object code is one byte smaller than it was for the M68HC11.

7.1 Constant Offset Indexing

The CPU12 offers three variations of constant offset indexing in order to optimize the efficiency of object code generation.

The most common constant offset is zero. Offsets of 1, 2…4 are used fairly often, but with less frequency than zero.

The 5-bit constant offset variation covers the most frequent indexing requirements by including the offset in the postbyte. This reduces a load accumulator indexed instruction to two bytes of object code, and matches the object code size of the smallest M68HC11 indexed instructions, which can only use X as the index register. The CPU12 can use X, Y, SP, or PC as the index reference with no additional object code size cost.

The signed 9-bit constant offset indexing mode covers the same positive range as the M6HC11 8-bit unsigned offset. The size was increased to nine bits with the sign bit (ninth bit) included in the postbyte, and the remaining 8-bits of the offset in a single extension byte.

The 16-bit constant offset indexing mode allows indexed access to the entire normal 64-Kbyte address space. Since the address consists of 16 bits, the 16-bit offset can be regarded as a signed (–32,768 to +32767) or unsigned (0 to 65,535) value. In 16-bit constant offset mode, the offset is supplied in two extension bytes after the opcode and postbyte.

7.2 Auto-Increment Indexing

The CPU12 provides greatly enhanced auto increment and decrement modes of indexed addressing. In the CPU12, the index modification may be specified for before the index is used (pre-), or after the index is used (post-), and the index can be incremented or decremented by any amount from one to eight, independent of the size of the operand that was accessed. X, Y, and SP can be used as the index reference, but this mode does not allow PC to be the index reference (this would interfere with proper program execution).

This addressing mode can be used to implement a software stack structure, or to manipulate data structures in lists or tables, rather than manipulating bytes or words of data. Anywhere an M68HC11 program has an increment or decrement index register operation near an indexed mode instruction, the increment or decrement operation can be combined with the indexed instruction with no cost in object code size, as shown in the following code comparison.

HC11	HC12

`18 A6 00           LDAA  0,Y`
`18 08              INY`
`18 08              INY`
	`A6 71              LDAA  2,Y+`

The M68HC11 object code requires seven bytes, while the CPU12 requires only two bytes to accomplish the same functions. Three bytes of M68HC11 code were due to the page prebyte for each Y related instruction ($18). CPU12 post increment indexing capability allowed the two INY instructions to be absorbed into the LDAA indexed instruction. The replacement code is not identical to the original three instruction sequence because the Z condition code bit is affected by the M68HC11 INY instructions, while the Z bit in the CPU12 would be determined by the value loaded into A.

7.3 Accumulator Offset Indexing

This indexed addressing variation allows use of either an 8-bit accumulator (A or B), or of the 16-bit D accumulator as an offset for indexed addressing. This supports program-generated offsets, which are more difficult to achieve in the M68HC11. The following code compares M68HC11 and CPU12 operation.

HC11	HC12

`C6 05              LDAB  #$5[2]`
`CE 10 00      LOOP  LDX   #$1000[3]`
`3A                 ABX      [3]`
`A6 00              LDAA  0,X[4]`
`                   |`
`5A                 DECB     [2]`
`26 F7              BNE   LOOP[3]`
	`C6 05              LDAB  #$5[1]`
	`CE 10 00           LDX   #$1000[2]`
	`A6 E5         LOOP  LDAA  B,X[3]`
	`                   |`
	`04 31 FB           DBNE  B,LOOP[3]`

The CPU12 object code is only one byte smaller, but the LDX # instruction is outside the loop. It is not necessary to reload the base address in the index register on each pass through the loop because the LDAA B,X instruction does not alter the index register. This reduces loop execution time from 15 cycles to 6 cycles. This reduction, combined with the 8 MHz bus speed of the M68HC12 family, can have significant effects.

7.4 Indirect Indexing

The CPU12 allows some forms of indexed indirect addressing where the instruction points to a location in memory where the address of the operand is stored. This is an extra level of indirection compared to ordinary indexed addressing. The two forms of indexed indirect addressing are 16-bit constant offset indexed indirect and D accumulator indexed indirect. The reference index register can be X, Y, SP, or PC as in other CPU12 indexed addressing modes. PC-relative indirect addressing is one of the more common uses of indexed indirect addressing. The indirect variations of indexed addressing help in the implementation of pointers. D accumulator indexed indirect addressing can be used to implement a runtime computed GOTO function. Indirect addressing is also useful in high level language compilers. For instance, PC-relative indirect indexing can be used to efficiently implement some C case statements.

8 IMPROVED PERFORMANCE

The CPU12 improves on M68HC11 performance in several ways. M68HC12 devices are designed using sub-micron design rules, and fabricated using advanced semiconductor processing, the same methods used to manufacture the M68HC16 and M68300 families of modular microcontrollers. M68HC12 devices have a base bus speed of 8 MHz, and are designed to operate over a wide range of supply voltages. The 16-bit wide architecture also increases performance. Beyond these obvious improvements, the CPU12 uses a reduced number of cycles for many of its instructions, and a 20-bit ALU makes certain CPU12 math operations much faster.

8.1 Reduced Cycle Counts

No M68HC11 instruction takes less than two cycles, but the CPU12 has more than 50 opcodes that take only one cycle. Some of the reduction comes from the instruction queue, which assures that several program bytes are available at the start of each instruction. Other cycle reductions occur because the CPU12 can fetch 16 bits of information at a time, rather than eight bits at a time.

8.2 Fast Math

The CPU12 has some of the fastest math ever designed into a Freescale general-purpose MCU. Much of the speed is due to a 20-bit ALU that can perform two smaller operations simultaneously. The ALU can also perform two operations in a single bus cycle in certain cases. **Table 3** compares the speed of CPU12 and M68HC11 math instructions. The CPU12 require much fewer cycles to perform an operation, and the cycle time is half that of the M68HC11.

Table 3 Comparison of Math Instruction Speeds

Instruction Mnemonic	Math Operation	M68HC11 1 cycle = 250 ns	M68HC11 w/co-processor 1 cycle = 250 ns	CPU12 1 cycle = 125 ns
MUL	$8 \times 8 = 16$ (signed)	10 cycles	—	3 cycles
EMUL	$16 \times 16 = 32$ (unsigned)	—	20 cycles	3 cycles
EMULS	$16 \times 16 = 32$ (signed)	—	20 cycles	3 cycles
IDIV	$16 \div 16 = 16$ (unsigned)	41 cycles	—	12 cycles
FDIV	$16 \div 16 = 16$ (fractional)	41 cycles	—	12 cycles
EDIV	$32 \div 16 = 16$ (unsigned)	—	33 cycles	11 cycles
EDIVS	$32 \div 16 = 16$ (signed)	—	37 cycles	12 cycles
IDIVS	$16 \div 16 = 16$ (signed)	—	—	12 cycles
EMACS	$16 \times 16 \Rightarrow 32$ (signed MAC)	—	20 cycles per iteration	12 cycles per iteration

The IDIVS instruction is included specifically for C compilers, where word-sized operands are divided to produce a word-sized result (unlike the 32÷16=16 EDIV). The EMUL and EMULS instructions place the result in registers so a C compiler can choose to use only 16 bits of the 32-bit result.

8.3 Code Size Reduction

CPU12 assembly language programs written from scratch tend to be 30% smaller than equivalent programs written for the M68HC11. This figure has been independently qualified by Freescale programmers and an independent C compiler vendor. The major contributors to the reduction appear to be improved indexed addressing and the universal transfer/exchange instruction.

In some specialized areas, the reduction is much greater. A fuzzy logic inference kernel requires about 250 bytes in the M68HC11, and the same program for the CPU12 requires about 50 bytes. The CPU12 fuzzy logic instructions replace whole subroutines in the M68HC11 version. Table lookup instructions also greatly reduce code space.

Other CPU12 code space reductions are more subtle. Memory to memory moves are one example. The CPU12 move instruction requires almost as many bytes as an equivalent sequence of M68HC11 instructions, but the move operations themselves do not require the use of an accumulator. This means that the accumulator often need not be saved and restored, which saves instructions.

Arithmetic on index pointers is another example. The M68HC11 usually requires that the content of the index register be moved into accumulator D, where calculations are performed, then back to the index register before indexing can take place. In the CPU12, the LEAS, LEAX, and LEAY instructions perform arithmetic operations directly on the index pointers. The pre-/post-increment/decrement variations of indexed addressing also allow index modification to be incorporated into an existing indexed instruction rather than performing the index modification as a separate operation.

Transfer and exchange operations often allow register contents to be temporarily saved in another register rather than having to save the contents in memory. Some CPU12 instructions such as MIN and MAX combine the actions of several M68HC11 instructions into a single operation.

9 ADDITIONAL FUNCTIONS

The CPU12 incorporates a number of new instructions that provide added functionality and code efficiency. Among other capabilities, these new instructions allow efficient processing for fuzzy logic applications and support subroutine processing in extended memory beyond the standard 64-Kbyte address map for M68HC12 devices incorporating this feature. The following paragraphs discuss the most significant of these enhancements. For detailed information, please refer to the *CPU12 Reference Manual,* Freescale Publication Number CPU12RM/AD

9.1 Memory-to-Memory Moves

The CPU12 has both 8- and 16-bit variations of memory-to-memory move instructions. The source address can be specified with immediate, extended, or indexed addressing modes. The destination address can be specified by extended or indexed addressing mode. The indexed addressing mode for move instructions is limited to modes that require no extension bytes (9- and 16-bit constant offsets are not allowed), and indirect indexing is not allowed for moves. This leaves a 5-bit signed constant offset, accumulator offsets, and the automatic increment/decrement modes. The following simple loop is a block move routine capable of moving up to 256 words of information from one memory area to another.

```
LOOP   MOVW   2,X+ , 2,Y+   ;move a word and update pointers
       DBNE   B,LOOP        ;repeat B times
```

The move immediate to extended is a convenient way to initialize a register without using an accumulator or affecting condition codes.

9.2 Universal Transfer and Exchange

The M68HC11 has only six transfer instructions and two exchange instructions. The CPU12 has a universal transfer/exchange instruction that can be used to transfer or exchange data between any two CPU registers. The operation is obvious when the two registers are the same size, but some of the other combinations provide very useful results. For example when an 8-bit register is transferred to a 16-bit register, a sign-extend operation is performed. Other combinations can be used to perform a zero-extend operation.

These instructions are used often in CPU12 assembly language programs. Transfers can be used to make extra copies of data in another register, and exchanges can be used to temporarily save data during a call to a routine that expects data in a specific register. This is sometimes faster and smaller (object code) than saving data to memory with pushes or stores.

9.3 Loop Construct

The CPU12 instruction set includes a new family of six loop primitive instructions thatdecrement, increment, or test a loop count in a CPU register and then branch based on a zero or non-zero test result. The CPU registers that can be used for the loop count are A, B, D, X, Y, or SP. The branch range is a 9-bit signed value (–512 to +511) which gives these instructions twice the range of a short branch instruction.

9.4 Long Branches

All of the branch instructions from the M68HC11 are also available with 16-bit offsets which allows them to reach any location in the 64K address space.

9.5 Minimum and Maximum Instructions

Control programs often need to restrict data values within upper and lower limits. The CPU12 facilitates this function with 8- and 16-bit versions of MIN and MAX instructions. Each of these instructions has a version that stores the result in either the accumulator or in memory.

For example, in a fuzzy logic inference program, rule evaluation consists of a series of MIN and MAX operations. The min operation is used to determine the smallest rule input (the running result is held in an accumulator), and the max operation is used to store the largest rule truth value (in an accumulator) or the previous fuzzy output value (in a RAM location), to the fuzzy output in RAM. The following code demonstrates how min and max instructions can be used to evaluate a rule with four inputs and two outputs.

```
LDY     #OUT1      ;Point at first output
LDX     #IN1       ;Point at first input value
LDAA    #$FF       ;start with largest 8-bit number in A
MINA    1,X+       ;A=MIN(A,IN1)
MINA    1,X+       ;A=MIN(A,IN2)
MINA    1,X+       ;A=MIN(A,IN3)
MINA    1,X+       ;A=MIN(A,IN4) so A holds smallest input
MAXM    1,Y+       ;OUT1=MAX(A,OUT1) and A is unchanged
MAXM    1,Y+       ;OUT1=MAX(A,OUT2) A still has min input
```

Before this sequence is executed, the fuzzy outputs must be cleared to zeros (not shown). M68HC11 min or max operations are performed by executing a compare followed by a conditional branch around a load or store operation.

These instructions can also be used to limit a data value prior to using it as an input to a table lookup or other routine. Suppose a table is valid for input values between $20 and $7F. An arbitrary input value can be tested against these limits and be replaced by the largest legal value if it is too big, or the smallest legal value if too small using the following two CPU12 instructions.

```
HILIMIT   FCB   $7F           ;comparison value needs to be in mem
LOWLIMIT  FCB   $20           ;so it can be referenced via indexed
          MINA  HILIMIT,PCR   ;A=MIN(A,$7F)
          MAXA  LOWLIMIT,PCR  ;A=MAX(A,$20)
                              ;A now within the legal range $20 to $7F
```

The ",PCR" notation is also new for the CPU12. This notation indicates the programmer wants an appropriate offset from the PC reference to the memory location (HILIMIT or LOWLIMIT in this example), and then to assemble this instruction into a PC-relative indexed MIN or MAX instruction.

9.6 Fuzzy Logic Support

The CPU12 includes four instructions (MEM, REV, REVW, and WAV) specifically designed to support fuzzy logic programs. These instructions have a very small impact on the size of the CPU, and even less impact on the cost of a complete MCU. At the same time these instructions dramatically reduce the object code size and execution time for a fuzzy logic inference program. A kernel written for M68HC11 required about 250 bytes and executed in about 750 milliseconds. The CPU12 kernel uses about 50 bytes and executes in about 50 microseconds.

9.7 Table Lookup and Interpolation

The CPU12 instruction set includes two instructions (TBL and ETBL) for lookup and interpolation of compressed tables. Consecutive table values are assumed to be the x coordinates the endpoints of a line segment. The TBL instruction uses 8-bit table entries (y-values) and returns an 8-bit result. The ETBL instruction uses 16-bit table entries (y-values) and returns a 16-bit result.

An indexed addressing mode is used to identify the effective address of the data point at the beginning of the line segment, and the data value for the end point of the line segment is the next consecutive memory location (byte for TBL and word for ETBL). In both cases, the B accumulator represents the ratio of (the x-distance from the beginning of the line segment to the lookup point) to (the x-distance from the beginning of the line segment to the end of the line segment). B is treated as an 8-bit binary fraction with radix point left of the MSB, so each line segment is effectively divided into 256 pieces. During execution of the TBL or ETBL instruction, the difference between the end point y-value and the beginning point y-value (a signed byte for TBL or a signed word for ETBL) is multiplied by the B accumulator to get an intermediate delta-y term. The result is the y-value of the beginning point, plus this signed intermediate delta-y value.

9.8 Extended Bit Manipulation

The M68HC11 CPU only allows direct or indexed addressing. This typically causes the programmer to dedicate an index register to point at some memory area such as the on-chip registers. The CPU12 allows all bit manipulation instructions to work with direct, extended or indexed addressing modes.

9.9 Push and Pull D and CCR

The CPU12 includes instructions to push and pull the D accumulator and the CCR. It is interesting to note that the order in which 8-bit accumulators A and B are stacked for interrupts is the opposite of what would be expected for the upper and lower bytes of the 16-bit D accumulator. The order used originated in the M6800, an 8-bit microprocessor developed long before anyone thought 16-bit single-chip devices would be made. The interrupt stacking order for accumulators A and B is retained for code compatibility.

9.10 Compare SP

This instruction was added to the CPU12 instruction set to improve orthogonality and high-level language support. One of the most important requirements for C high level language support is the ability to do arithmetic on the stack pointer for such things as allocating local variable space on the stack. The LEAS –5,SP instruction is an example of how the compiler could easily allocate five bytes on the stack for local variables. LDX 5,SP+ loads X with the value on the bottom of the stack and deallocates five bytes from the stack in a single operation that takes only two bytes of object code.

.11 Support for Memory Expansion

Bank switching is a common method of expanding memory, but there are some known difficulties associated with it. One problem is that interrupts cannot take place during the bank switching operation. This increases worst case interrupt latency and requires extra programming space and execution time.

Some M68HC12 variants include a built-in bank switching scheme that expands the address space beyond the standard 64 Kbytes, but eliminates many of the problems associated with external switching logic. The CPU12 includes CALL and return from call (RTC) instructions that manage the interface to the bank-switching system. These instructions are analogous to the JSR and RTS instructions, except that the bank page number is saved and restored automatically during execution. Since the page change operation is part of an uninterruptable instruction, many of the difficulties associated with bank switching are eliminated. On M68HC12 derivatives with expanded memory capability, bank numbers are specified by on-chip control registers. Since the addresses of these control registers may not be the same in all M68HC12 derivatives, the CPU12 has a dedicated control line to the on-chip integration module that indicates when a memory-expansion register is being read or written. This allows the CPU to access the PPAGE register without knowing the register address.

The indexed indirect versions of the CALL instruction access the address of the called routine and the destination page value indirectly. For other addressing mode variations of the CALL instruction, the destination page value is provided as immediate data in the instruction object Code. CALL and RTC execute correctly in the normal 64-Kbyte address space, thus providing for portable code.

10 INSTRUCTION SET REFERENCE

Table 4 is a quick reference to the CPU12 instruction set. The table shows source form, describes the operation performed, lists the addressing modes used, gives machine encoding in hexadecimal form, and describes the effect of execution on the Condition Code bits.

Table 4 Instruction Set Summary

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
ABA	(A) + (B) ⇒ A Add Accumulators A and B	INH	18 06	2	–	–	Δ	–	Δ	Δ	Δ	Δ
ABX	(B) + (X) ⇒ X *Translates to* LEAX B,X	IDX	1A E5	2	–	–	–	–	–	–	–	–
ABY	(B) + (Y) ⇒ Y *Translates to* LEAY B,Y	IDX	19 ED	2	–	–	–	–	–	–	–	–
ADCA *opr*	(A) + (M) + C ⇒ A Add with Carry to A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	89 ii 99 dd B9 hh ll A9 xb A9 xb ff A9 xb ee ff A9 xb A9 xb ee ff	1 3 3 3 3 4 6 6	–	–	Δ	–	Δ	Δ	Δ	Δ
ADCB *opr*	(B) + (M) + C ⇒ B Add with Carry to B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C9 ii D9 dd F9 hh ll E9 xb E9 xb ff E9 xb ee ff E9 xb E9 xb ee ff	1 3 3 3 3 4 6 6	–	–	Δ	–	Δ	Δ	Δ	Δ

Table 4 Instruction Set Summary (Continued)

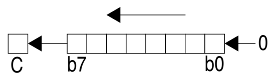
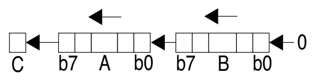
Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
ADDA *opr*	(A) + (M) ⇒ A	IMM	8B ii	1	–	–	Δ	–	Δ	Δ	Δ	Δ
	Add without Carry to A	DIR	9B dd	3								
		EXT	BB hh ll	3								
		IDX	AB xb	3								
		IDX1	AB xb ff	3								
		IDX2	AB xb ee ff	4								
		[D,IDX]	AB xb	6								
		[IDX2]	AB xb ee ff	6								
ADDB *opr*	(B) + (M) ⇒ B	IMM	CB ii	1	–	–	Δ	–	Δ	Δ	Δ	Δ
	Add without Carry to B	DIR	DB dd	3								
		EXT	FB hh ll	3								
		IDX	EB xb	3								
		IDX1	EB xb ff	3								
		IDX2	EB xb ee ff	4								
		[D,IDX]	EB xb	6								
		[IDX2]	EB xb ee ff	6								
ADDD *opr*	(A:B) + (M:M+1) ⇒ A:B	IMM	C3 jj kk	2	–	–	–	–	Δ	Δ	Δ	Δ
	Add 16-Bit to D (A:B)	DIR	D3 dd	3								
		EXT	F3 hh ll	3								
		IDX	E3 xb	3								
		IDX1	E3 xb ff	3								
		IDX2	E3 xb ee ff	4								
		[D,IDX]	E3 xb	6								
		[IDX2]	E3 xb ee ff	6								
ANDA *opr*	(A) • (M) ⇒ A	IMM	84 ii	1	–	–	–	–	Δ	Δ	0	–
	Logical And A with Memory	DIR	94 dd	3								
		EXT	B4 hh ll	3								
		IDX	A4 xb	3								
		IDX1	A4 xb ff	3								
		IDX2	A4 xb ee ff	4								
		[D,IDX]	A4 xb	6								
		[IDX2]	A4 xb ee ff	6								
ANDB *opr*	(B) • (M) ⇒ B	IMM	C4 ii	1	–	–	–	–	Δ	Δ	0	–
	Logical And B with Memory	DIR	D4 dd	3								
		EXT	F4 hh ll	3								
		IDX	E4 xb	3								
		IDX1	E4 xb ff	3								
		IDX2	E4 xb ee ff	4								
		[D,IDX]	E4 xb	6								
		[IDX2]	E4 xb ee ff	6								
ANDCC *opr*	(CCR) • (M) ⇒ CCR Logical And CCR with Memory	IMM	10 ii	1	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓
ASL *opr*		EXT	78 hh ll	4	–	–	–	–	Δ	Δ	Δ	Δ
		IDX	68 xb	3								
		IDX1	68 xb ff	4								
		IDX2	68 xb ee ff	5								
	Arithmetic Shift Left	[D,IDX]	68 xb	6								
		[IDX2]	68 xb ee ff	6								
ASLA	Arithmetic Shift Left Accumulator A	INH	48	1								
ASLB	Arithmetic Shift Left Accumulator B	INH	58	1								
ASLD	 Arithmetic Shift Left Double	INH	59	1	–	–	–	–	Δ	Δ	Δ	Δ

Table 4 Instruction Set Summary (Continued)

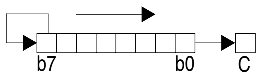
Source Form	Operation	Addr. Mode	Machine Coding (hex)	~1	S	X	H	I	N	Z	V	C
ASR *opr* ASRA ASRB	 Arithmetic Shift Right Arithmetic Shift Right Accumulator A Arithmetic Shift Right Accumulator B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	77 hh ll 67 xb 67 xb ff 67 xb ee ff 67 xb 67 xb ee ff 47 57	4 3 4 5 6 6 1 1	–	–	–	–	Δ	Δ	Δ	Δ
BCC *rel*	Branch if Carry Clear (if C = 0)	REL	24 rr	3/1	–	–	–	–	–	–	–	–
BCLR *opr*, *msk*	(M) • $(\overline{mm})$ ⇒ M Clear Bit(s) in Memory	DIR EXT IDX IDX1 IDX2	4D dd mm 1D hh ll mm 0D xb mm 0D xb ff mm 0D xb ee ff mm	4 4 4 4 6	–	–	–	–	Δ	Δ	0	–
BCS *rel*	Branch if Carry Set (if C = 1)	REL	25 rr	3/1	–	–	–	–	–	–	–	–
BEQ *rel*	Branch if Equal (if Z = 1)	REL	27 rr	3/1	–	–	–	–	–	–	–	–
BGE *rel*	Branch if Greater Than or Equal (if N ⊕ V = 0) (signed)	REL	2C rr	3/1	–	–	–	–	–	–	–	–
BGND	Place CPU in Background Mode see Background Mode section.	INH	00	5	–	–	–	–	–	–	–	–
BGT *rel*	Branch if Greater Than (if Z ✛ (N ⊕ V) = 0) (signed)	REL	2E rr	3/1	–	–	–	–	–	–	–	–
BHI *rel*	Branch if Higher (if C ✛ Z = 0) (unsigned)	REL	22 rr	3/1	–	–	–	–	–	–	–	–
BHS *rel*	Branch if Higher or Same (if C = 0) (unsigned) same function as BCC	REL	24 rr	3/1	–	–	–	–	–	–	–	–
BITA *opr*	(A) • (M) Logical And A with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	85 ii 95 dd B5 hh ll A5 xb A5 xb ff A5 xb ee ff A5 xb A5 xb ee ff	1 3 3 3 3 4 6 6	–	–	–	–	Δ	Δ	0	–
BITB *opr*	(B) • (M) Logical And B with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C5 ii D5 dd F5 hh ll E5 xb E5 xb ff E5 xb ee ff E5 xb E5 xb ee ff	1 3 3 3 3 4 6 6	–	–	–	–	Δ	Δ	0	–
BLE *rel*	Branch if Less Than or Equal (if Z ✛ (N ⊕ V) = 1) (signed)	REL	2F rr	3/1	–	–	–	–	–	–	–	–
BLO *rel*	Branch if Lower (if C = 1) (unsigned) same function as BCS	REL	25 rr	3/1	–	–	–	–	–	–	–	–
BLS *rel*	Branch if Lower or Same (if C ✛ Z = 1) (unsigned)	REL	23 rr	3/1	–	–	–	–	–	–	–	–
BLT *rel*	Branch if Less Than (if N ⊕ V = 1) (signed)	REL	2D rr	3/1	–	–	–	–	–	–	–	–
BMI *rel*	Branch if Minus (if N = 1)	REL	2B rr	3/1	–	–	–	–	–	–	–	–
BNE *rel*	Branch if Not Equal (if Z = 0)	REL	26 rr	3/1	–	–	–	–	–	–	–	–
BPL *rel*	Branch if Plus (if N = 0)	REL	2A rr	3/1	–	–	–	–	–	–	–	–

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	$\sim^{1}$	S	X	H	I	N	Z	V	C
BRA *rel*	Branch Always (if 1 = 1)	REL	20 rr	3	–	–	–	–	–	–	–	–
BRCLR *opr*, *msk*, *rel*	Branch if (M) • (mm) = 0 (if All Selected Bit(s) Clear)	DIR EXT IDX IDX1 IDX2	4F dd mm rr 1F hh ll mm rr 0F xb mm rr 0F xb ff mm rr 0F xb ee ff mm rr	4 5 4 6 8	–	–	–	–	–	–	–	–
BRN *rel*	Branch Never (if 1 = 0)	REL	21 rr	1	–	–	–	–	–	–	–	–
BRSET *opr*, *msk*, *rel*	Branch if $(\overline{M})$ • (mm) = 0 (if All Selected Bit(s) Set)	DIR EXT IDX IDX1 IDX2	4E dd mm rr 1E hh ll mm rr 0E xb mm rr 0E xb ff mm rr 0E xb ee ff mm rr	4 5 4 6 8	–	–	–	–	–	–	–	–
BSET *opr*, *msk*	(M) ✣ (mm) ⇒ M Set Bit(s) in Memory	DIR EXT IDX IDX1 IDX2	4C dd mm 1C hh ll mm 0C xb mm 0C xb ff mm 0C xb ee ff mm	4 4 4 4 6	–	–	–	–	Δ	Δ	0	–
BSR *rel*	(SP) – 2 ⇒ SP; $RTN_H:RTN_L \Rightarrow M_{(SP)}:M_{(SP+1)}$ Subroutine address ⇒ PC Branch to Subroutine	REL	07 rr	4	–	–	–	–	–	–	–	–
BVC *rel*	Branch if Overflow Bit Clear (if V = 0)	REL	28 rr	3/1	–	–	–	–	–	–	–	–
BVS *rel*	Branch if Overflow Bit Set (if V = 1)	REL	29 rr	3/1	–	–	–	–	–	–	–	–
CALL *opr*, *page*	(SP) – 2 ⇒ SP; $RTN_H:RTN_L \Rightarrow M_{(SP)}:M_{(SP+1)}$ (SP) – 1 ⇒ SP; (PPG) ⇒ $M_{(SP)}$; pg ⇒ PPAGE register; Program address ⇒ PC Call Subroutine in extended memory (Program may be located on another expansion memory page.)	EXT IDX IDX1 IDX2	4A hh ll pg 4B xb pg 4B xb ff pg 4B xb ee ff pg	8 8 8 9	–	–	–	–	–	–	–	–
CALL [D,*r*] CALL [*opr*,*r*]	Indirect modes get program address and new pg value based on pointer. *r* = X, Y, SP, or PC	[D,IDX] [IDX2]	4B xb 4B xb ee ff	10 10	–	–	–	–	–	–	–	–
CBA	(A) – (B) Compare 8-Bit Accumulators	INH	18 17	2	–	–	–	–	Δ	Δ	Δ	Δ
CLC	0 ⇒ C *Translates to* ANDCC #$FE	IMM	10 FE	1	–	–	–	–	–	–	–	0
CLI	0 ⇒ I *Translates to* ANDCC #$EF (enables I-bit interrupts)	IMM	10 EF	1	–	–	–	0	–	–	–	–
CLR *opr* CLRA CLRB	0 ⇒ M Clear Memory Location 0 ⇒ A Clear Accumulator A 0 ⇒ B Clear Accumulator B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	79 hh ll 69 xb 69 xb ff 69 xb ee ff 69 xb 69 xb ee ff 87 C7	3 2 3 3 5 5 1 1	–	–	–	–	0	1	0	0
CLV	0 ⇒ V *Translates to* ANDCC #$FD	IMM	10 FD	1	–	–	–	–	–	–	0	–

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
CMPA *opr*	(A) – (M)	IMM	81 ii	1	–	–	–	–	Δ	Δ	Δ	Δ
	Compare Accumulator A with Memory	DIR	91 dd	3								
		EXT	B1 hh ll	3								
		IDX	A1 xb	3								
		IDX1	A1 xb ff	3								
		IDX2	A1 xb ee ff	4								
		[D,IDX]	A1 xb	6								
		[IDX2]	A1 xb ee ff	6								
CMPB *opr*	(B) – (M)	IMM	C1 ii	1	–	–	–	–	Δ	Δ	Δ	Δ
	Compare Accumulator B with Memory	DIR	D1 dd	3								
		EXT	F1 hh ll	3								
		IDX	E1 xb	3								
		IDX1	E1 xb ff	3								
		IDX2	E1 xb ee ff	4								
		[D,IDX]	E1 xb	6								
		[IDX2]	E1 xb ee ff	6								
COM *opr*	$(\overline{M}) \Rightarrow M$ *equivalent to* \$FF – (M) ⇒ M	EXT	71 hh ll	4	–	–	–	–	Δ	Δ	0	1
	1's Complement Memory Location	IDX	61 xb	3								
		IDX1	61 xb ff	4								
		IDX2	61 xb ee ff	5								
		[D,IDX]	61 xb	6								
		[IDX2]	61 xb ee ff	6								
COMA	$(\overline{A}) \Rightarrow A$ Complement Accumulator A	INH	41	1								
COMB	$(\overline{B}) \Rightarrow B$ Complement Accumulator B	INH	51	1								
CPD *opr*	(A:B) – (M:M+1)	IMM	8C jj kk	2	–	–	–	–	Δ	Δ	Δ	Δ
	Compare D to Memory (16-Bit)	DIR	9C dd	3								
		EXT	BC hh ll	3								
		IDX	AC xb	3								
		IDX1	AC xb ff	3								
		IDX2	AC xb ee ff	4								
		[D,IDX]	AC xb	6								
		[IDX2]	AC xb ee ff	6								
CPS *opr*	(SP) – (M:M+1)	IMM	8F jj kk	2	–	–	–	–	Δ	Δ	Δ	Δ
	Compare SP to Memory (16-Bit)	DIR	9F dd	3								
		EXT	BF hh ll	3								
		IDX	AF xb	3								
		IDX1	AF xb ff	3								
		IDX2	AF xb ee ff	4								
		[D,IDX]	AF xb	6								
		[IDX2]	AF xb ee ff	6								
CPX *opr*	(X) – (M:M+1)	IMM	8E jj kk	2	–	–	–	–	Δ	Δ	Δ	Δ
	Compare X to Memory (16-Bit)	DIR	9E dd	3								
		EXT	BE hh ll	3								
		IDX	AE xb	3								
		IDX1	AE xb ff	3								
		IDX2	AE xb ee ff	4								
		[D,IDX]	AE xb	6								
		[IDX2]	AE xb ee ff	6								
CPY *opr*	(Y) – (M:M+1)	IMM	8D jj kk	2	–	–	–	–	Δ	Δ	Δ	Δ
	Compare Y to Memory (16-Bit)	DIR	9D dd	3								
		EXT	BD hh ll	3								
		IDX	AD xb	3								
		IDX1	AD xb ff	3								
		IDX2	AD xb ee ff	4								
		[D,IDX]	AD xb	6								
		[IDX2]	AD xb ee ff	6								

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
DAA	Adjust Sum to BCD Decimal Adjust Accumulator A	INH	18 07	3	–	–	–	–	Δ	Δ	?	Δ
DBEQ *cntr, rel*	(cntr) – 1⇒ cntr if (cntr) = 0, then Branch else Continue to next instruction Decrement Counter and Branch if = 0 (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 lb rr	3	–	–	–	–	–	–	–	–
DBNE *cntr, rel*	(cntr) – 1 ⇒ cntr If (cntr) not = 0, then Branch; else Continue to next instruction Decrement Counter and Branch if ≠ 0 (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 lb rr	3	–	–	–	–	–	–	–	–
DEC *opr* DECA DECB	(M) – $01 ⇒ M Decrement Memory Location (A) – $01 ⇒ A Decrement A (B) – $01 ⇒ B Decrement B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	73 hh ll 63 xb 63 xb ff 63 xb ee ff 63 xb 63 xb ee ff 43 53	4 3 4 5 6 6 1 1	–	–	–	–	Δ	Δ	Δ	–
DES	(SP) – $0001 ⇒ SP *Translates to* LEAS –1,SP	IDX	1B 9F	2	–	–	–	–	–	–	–	–
DEX	(X) – $0001 ⇒ X Decrement Index Register X	INH	09	1	–	–	–	–	–	Δ	–	–
DEY	(Y) – $0001 ⇒ Y Decrement Index Register Y	INH	03	1	–	–	–	–	–	Δ	–	–
EDIV	(Y:D) ÷ (X) ⇒ Y Remainder ⇒ D 32 × 16 Bit ⇒ 16 Bit Divide (unsigned)	INH	11	11	–	–	–	–	Δ	Δ	Δ	Δ
EDIVS	(Y:D) ÷ (X) ⇒ Y Remainder ⇒ D 32 × 16 Bit ⇒ 16 Bit Divide (signed)	INH	18 14	12	–	–	–	–	Δ	Δ	Δ	Δ
EMACS *sum*	$(M_{(X)}:M_{(X+1)}) \times (M_{(Y)}:M_{(Y+1)}) + (M\sim M+3) \Rightarrow M\sim M+3$ 16 × 16 Bit ⇒ 32 Bit Multiply and Accumulate (signed)	Special	18 12 hh ll	13	–	–	–	–	Δ	Δ	Δ	Δ
EMAXD *opr*	MAX((D), (M:M+1)) ⇒ D MAX of 2 Unsigned 16-Bit Values N, Z, V and C status bits reflect result of internal compare ((D) – (M:M+1))	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1A xb 18 1A xb ff 18 1A xb ee ff 18 1A xb 18 1A xb ee ff	4 4 5 7 7	–	–	–	–	Δ	Δ	Δ	Δ
EMAXM *opr*	MAX((D), (M:M+1)) ⇒ M:M+1 MAX of 2 Unsigned 16-Bit Values N, Z, V and C status bits reflect result of internal compare ((D) – (M:M+1))	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1E xb 18 1E xb ff 18 1E xb ee ff 18 1E xb 18 1E xb ee ff	4 5 6 7 7	–	–	–	–	Δ	Δ	Δ	Δ
EMIND *opr*	MIN((D), (M:M+1)) ⇒ D MIN of 2 Unsigned 16-Bit Values N, Z, V and C status bits reflect result of internal compare ((D) – (M:M+1))	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1B xb 18 1B xb ff 18 1B xb ee ff 18 1B xb 18 1B xb ee ff	4 4 5 7 7	–	–	–	–	Δ	Δ	Δ	Δ

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
EMINM *opr*	MIN((D), (M:M+1)) ⇒ M:M+1 MIN of 2 Unsigned 16-Bit Values N, Z, V and C status bits reflect result of internal compare ((D) – (M:M+1))	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1F xb 18 1F xb ff 18 1F xb ee ff 18 1F xb 18 1F xb ee ff	4 5 6 7 7	–	–	–	–	Δ	Δ	Δ	Δ
EMUL	(D) × (Y) ⇒ Y:D 16 × 16 Bit Multiply (unsigned)	INH	13	3	–	–	–	–	Δ	Δ	–	Δ
EMULS	(D) × (Y) ⇒ Y:D 16 × 16 Bit Multiply (signed)	INH	18 13	3	–	–	–	–	Δ	Δ	–	Δ
EORA *opr*	(A) ⊕ (M) ⇒ A Exclusive-OR A with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	88 ii 98 dd B8 hh ll A8 xb A8 xb ff A8 xb ee ff A8 xb A8 xb ee ff	1 3 3 3 3 4 6 6	–	–	–	–	Δ	Δ	0	–
EORB *opr*	(B) ⊕ (M) ⇒ B Exclusive-OR B with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C8 ii D8 dd F8 hh ll E8 xb E8 xb ff E8 xb ee ff E8 xb E8 xb ee ff	1 3 3 3 3 4 6 6	–	–	–	–	Δ	Δ	0	–
ETBL *opr*	(M:M+1)+ [(B)×((M+2:M+3) – (M:M+1))] ⇒ D 16-Bit Table Lookup and Interpolate Initialize B, and index before ETBL. <ea> points at first table entry (M:M+1) and B is fractional part of lookup value (no indirect addr. modes allowed)	IDX	18 3F xb	10	–	–	–	–	Δ	Δ	–	?
EXG *r1*, *r2*	(r1) ⇔ (r2) (if r1 and r2 same size) *or* $00:(r1) ⇒ r2 (if r1=8-bit; r2=16-bit) *or* ($r1_{low}$) ⇔ (r2) (if r1=16-bit; r2=8-bit) r1 and r2 may be A, B, CCR, D, X, Y, or SP	INH	B7 eb	1	–	–	–	–	–	–	–	–
FDIV	(D) ÷ (X) ⇒ X; r ⇒ D 16 × 16 Bit Fractional Divide	INH	18 11	12	–	–	–	–	–	Δ	Δ	Δ
IBEQ *cntr*, *rel*	(cntr) + 1⇒ cntr If (cntr) = 0, then Branch else Continue to next instruction Increment Counter and Branch if = 0 (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 lb rr	3	–	–	–	–	–	–	–	–
IBNE *cntr*, *rel*	(cntr) + 1⇒ cntr if (cntr) not = 0, then Branch; else Continue to next instruction Increment Counter and Branch if ≠ 0 (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 lb rr	3	–	–	–	–	–	–	–	–
IDIV	(D) ÷ (X) ⇒ X; r ⇒ D 16 × 16 Bit Integer Divide (unsigned)	INH	18 10	12	–	–	–	–	–	Δ	0	Δ

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
IDIVS	(D) ÷ (X) ⇒ X; r ⇒ D 16 × 16 Bit Integer Divide (signed)	INH	18 15	12	–	–	–	–	Δ	Δ	Δ	Δ
INC *opr* INCA INCB	(M) + $01 ⇒ M Increment Memory Byte (A) + $01 ⇒ A Increment Acc. A (B) + $01 ⇒ B Increment Acc. B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	72 hh ll 62 xb 62 xb ff 62 xb ee ff 62 xb 62 xb ee ff 42 52	4 3 4 5 6 6 1 1	–	–	–	–	Δ	Δ	Δ	–
INS	(SP) + $0001 ⇒ SP *Translates to* LEAS 1,SP	IDX	1B 81	2	–	–	–	–	–	–	–	–
INX	(X) + $0001 ⇒ X Increment Index Register X	INH	08	1	–	–	–	–	–	Δ	–	–
INY	(Y) + $0001 ⇒ Y Increment Index Register Y	INH	02	1	–	–	–	–	–	Δ	–	–
JMP *opr*	Subroutine address ⇒ PC Jump	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	06 hh ll 05 xb 05 xb ff 05 xb ee ff 05 xb 05 xb ee ff	3 3 3 4 6 6	–	–	–	–	–	–	–	–
JSR *opr*	(SP) – 2 ⇒ SP; $RTN_H{:}RTN_L \Rightarrow M_{(SP)}{:}M_{(SP+1)}$; Subroutine address ⇒ PC Jump to Subroutine	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	17 dd 16 hh ll 15 xb 15 xb ff 15 xb ee ff 15 xb 15 xb ee ff	4 4 4 4 5 7 7	–	–	–	–	–	–	–	–
LBCC *rel*	Long Branch if Carry Clear (if C = 0)	REL	18 24 qq rr	4/3	–	–	–	–	–	–	–	–
LBCS *rel*	Long Branch if Carry Set (if C = 1)	REL	18 25 qq rr	4/3	–	–	–	–	–	–	–	–
LBEQ *rel*	Long Branch if Equal (if Z = 1)	REL	18 27 qq rr	4/3	–	–	–	–	–	–	–	–
LBGE *rel*	Long Branch Greater Than or Equal (if N ⊕ V = 0) (signed)	REL	18 2C qq rr	4/3	–	–	–	–	–	–	–	–
LBGT *rel*	Long Branch if Greater Than (if Z ✢ (N ⊕ V) = 0) (signed)	REL	18 2E qq rr	4/3	–	–	–	–	–	–	–	–
LBHI *rel*	Long Branch if Higher (if C ✢ Z = 0) (unsigned)	REL	18 22 qq rr	4/3	–	–	–	–	–	–	–	–
LBHS *rel*	Long Branch if Higher or Same (if C = 0) (unsigned) same function as LBCC	REL	18 24 qq rr	4/3	–	–	–	–	–	–	–	–
LBLE *rel*	Long Branch if Less Than or Equal (if Z ✢ (N ⊕ V) = 1) (signed)	REL	18 2F qq rr	4/3	–	–	–	–	–	–	–	–
LBLO *rel*	Long Branch if Lower (if C = 1) (unsigned) *same function as LBC*S	REL	18 25 qq rr	4/3	–	–	–	–	–	–	–	–
LBLS *rel*	Long Branch if Lower or Same (if C ✢ Z = 1) (unsigned)	REL	18 23 qq rr	4/3	–	–	–	–	–	–	–	–
LBLT *rel*	Long Branch if Less Than (if N ⊕ V = 1) (signed)	REL	18 2D qq rr	4/3	–	–	–	–	–	–	–	–
LBMI *rel*	Long Branch if Minus (if N = 1)	REL	18 2B qq rr	4/3	–	–	–	–	–	–	–	–
LBNE *rel*	Long Branch if Not Equal (if Z = 0)	REL	18 26 qq rr	4/3	–	–	–	–	–	–	–	–
LBPL *rel*	Long Branch if Plus (if N = 0)	REL	18 2A qq rr	4/3	–	–	–	–	–	–	–	–
LBRA *rel*	Long Branch Always (if 1=1)	REL	18 20 qq rr	4	–	–	–	–	–	–	–	–

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
LBRN *rel*	Long Branch Never (if 1 = 0)	REL	18 21 qq rr	3	–	–	–	–	–	–	–	–
LBVC *rel*	Long Branch if Overflow Bit Clear (if V=0)	REL	18 28 qq rr	4/3	–	–	–	–	–	–	–	–
LBVS *rel*	Long Branch if Overflow Bit Set (if V = 1)	REL	18 29 qq rr	4/3	–	–	–	–	–	–	–	–
LDAA *opr*	(M) ⇒ A	IMM	86 ii	1	–	–	–	–	Δ	Δ	0	–
	Load Accumulator A	DIR	96 dd	3								
		EXT	B6 hh ll	3								
		IDX	A6 xb	3								
		IDX1	A6 xb ff	3								
		IDX2	A6 xb ee ff	4								
		[D,IDX]	A6 xb	6								
		[IDX2]	A6 xb ee ff	6								
LDAB *opr*	(M) ⇒ B	IMM	C6 ii	1	–	–	–	–	Δ	Δ	0	–
	Load Accumulator B	DIR	D6 dd	3								
		EXT	F6 hh ll	3								
		IDX	E6 xb	3								
		IDX1	E6 xb ff	3								
		IDX2	E6 xb ee ff	4								
		[D,IDX]	E6 xb	6								
		[IDX2]	E6 xb ee ff	6								
LDD *opr*	(M:M+1) ⇒ A:B	IMM	CC jj kk	2	–	–	–	–	Δ	Δ	0	–
	Load Double Accumulator D (A:B)	DIR	DC dd	3								
		EXT	FC hh ll	3								
		IDX	EC xb	3								
		IDX1	EC xb ff	3								
		IDX2	EC xb ee ff	4								
		[D,IDX]	EC xb	6								
		[IDX2]	EC xb ee ff	6								
LDS *opr*	(M:M+1) ⇒ SP	IMM	CF jj kk	2	–	–	–	–	Δ	Δ	0	–
	Load Stack Pointer	DIR	DF dd	3								
		EXT	FF hh ll	3								
		IDX	EF xb	3								
		IDX1	EF xb ff	3								
		IDX2	EF xb ee ff	4								
		[D,IDX]	EF xb	6								
		[IDX2]	EF xb ee ff	6								
LDX *opr*	(M:M+1) ⇒ X	IMM	CE jj kk	2	–	–	–	–	Δ	Δ	0	–
	Load Index Register X	DIR	DE dd	3								
		EXT	FE hh ll	3								
		IDX	EE xb	3								
		IDX1	EE xb ff	3								
		IDX2	EE xb ee ff	4								
		[D,IDX]	EE xb	6								
		[IDX2]	EE xb ee ff	6								
LDY *opr*	(M:M+1) ⇒ Y	IMM	CD jj kk	2	–	–	–	–	Δ	Δ	0	–
	Load Index Register Y	DIR	DD dd	3								
		EXT	FD hh ll	3								
		IDX	ED xb	3								
		IDX1	ED xb ff	3								
		IDX2	ED xb ee ff	4								
		[D,IDX]	ED xb	6								
		[IDX2]	ED xb ee ff	6								
LEAS *opr*	Effective Address ⇒ SP	IDX	1B xb	2	–	–	–	–	–	–	–	–
	Load Effective Address into SP	IDX1	1B xb ff	2								
		IDX2	1B xb ee ff	2								

Table 4 Instruction Set Summary (Continued)

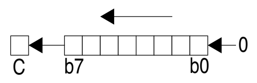
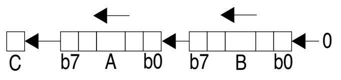
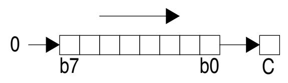
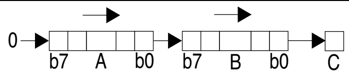
Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
LEAX *opr*	Effective Address ⇒ X	IDX	1A xb	2	–	–	–	–	–	–	–	–
	Load Effective Address into X	IDX1	1A xb ff	2								
		IDX2	1A xb ee ff	2								
LEAY *opr*	Effective Address ⇒ Y	IDX	19 xb	2	–	–	–	–	–	–	–	–
	Load Effective Address into Y	IDX1	19 xb ff	2								
		IDX2	19 xb ee ff	2								
LSL *opr*		EXT	78 hh ll	4	–	–	–	–	Δ	Δ	Δ	Δ
		IDX	68 xb	3								
		IDX1	68 xb ff	4								
	Logical Shift Left	IDX2	68 xb ee ff	5								
	same function as ASL	[D,IDX]	68 xb	6								
		[IDX2]	68 xb ee ff	6								
LSLA	Logical Shift Accumulator A to Left	INH	48	1								
LSLB	Logical Shift Accumulator B to Left	INH	58	1								
LSLD		INH	59	1	–	–	–	–	Δ	Δ	Δ	Δ
	Logical Shift Left D Accumulator											
	same function as ASLD											
LSR *opr*		EXT	74 hh ll	4	–	–	–	–	0	Δ	Δ	Δ
		IDX	64 xb	3								
		IDX1	64 xb ff	4								
	Logical Shift Right	IDX2	64 xb ee ff	5								
		[D,IDX]	64 xb	6								
		[IDX2]	64 xb ee ff	6								
LSRA	Logical Shift Accumulator A to Right	INH	44	1								
LSRB	Logical Shift Accumulator B to Right	INH	54	1								
LSRD		INH	49	1	–	–	–	–	0	Δ	Δ	Δ
	Logical Shift Right D Accumulator											
MAXA	MAX((A), (M)) ⇒ A	IDX	18 18 xb	4	–	–	–	–	Δ	Δ	Δ	Δ
	MAX of 2 Unsigned 8-Bit Values	IDX1	18 18 xb ff	4								
		IDX2	18 18 xb ee ff	5								
	N, Z, V and C status bits reflect result of	[D,IDX]	18 18 xb	7								
	internal compare ((A) – (M)).	[IDX2]	18 18 xb ee ff	7								
MAXM	MAX((A), (M)) ⇒ M	IDX	18 1C xb	4	–	–	–	–	Δ	Δ	Δ	Δ
	MAX of 2 Unsigned 8-Bit Values	IDX1	18 1C xb ff	5								
		IDX2	18 1C xb ee ff	6								
	N, Z, V and C status bits reflect result of	[D,IDX]	18 1C xb	7								
	internal compare ((A) – (M)).	[IDX2]	18 1C xb ee ff	7								
MEM	μ (grade) ⇒ $M_{(Y)}$;	Special	01	5	–	–	?	–	?	?	?	?
	(X) + 4 ⇒ X; (Y) + 1 ⇒ Y; A unchanged											
	if (A) < P1 or (A) > P2 then μ = 0, else											
	μ =MIN[((A) – P1)×S1, (P2 – (A))×S2, $FF]											
	where:											
	A = current crisp input value;											
	X points at 4 byte data structure that describes a trapezoidal membership function (P1, P2, S1, S2);											
	Y points at fuzzy input (RAM location).											
	See instruction details for special cases.											

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
MINA	MIN((A), (M)) ⇒ A	IDX	18 19 xb	4	–	–	–	–	Δ	Δ	Δ	Δ
	MIN of 2 Unsigned 8-Bit Values	IDX1	18 19 xb ff	4								
		IDX2	18 19 xb ee ff	5								
	N, Z, V and C status bits reflect result of	[D,IDX]	18 19 xb	7								
	internal compare ((A) – (M)).	[IDX2]	18 19 xb ee ff	7								
MINM	MIN((A), (M)) ⇒ M	IDX	18 1D xb	4	–	–	–	–	Δ	Δ	Δ	Δ
	MIN of 2 Unsigned 8-Bit Values	IDX1	18 1D xb ff	5								
		IDX2	18 1D xb ee ff	6								
	N, Z, V and C status bits reflect result of	[D,IDX]	18 1D xb	7								
	internal compare ((A) – (M)).	[IDX2]	18 1D xb ee ff	7								
MOVB *opr1*, *opr2*	$(M_1) \Rightarrow M_2$	IMM-EXT	18 0B ii hh ll	4	–	–	–	–	–	–	–	–
	Memory to Memory Byte-Move (8-Bit)	IMM-IDX	18 08 xb ii	4								
		EXT-EXT	18 0C hh ll hh ll	6								
		EXT-IDX	18 09 xb hh ll	5								
		IDX-EXT	18 0D xb hh ll	5								
		IDX-IDX	18 0A xb xb	5								
MOVW *opr1*, *opr2*	$(M{:}M{+}1_1) \Rightarrow M{:}M{+}1_2$	IMM-EXT	18 03 jj kk hh ll	5	–	–	–	–	–	–	–	–
	Memory to Memory Word-Move (16-Bit)	IMM-IDX	18 00 xb jj kk	4								
		EXT-EXT	18 04 hh ll hh ll	6								
		EXT-IDX	18 01 xb hh ll	5								
		IDX-EXT	18 05 xb hh ll	5								
		IDX-IDX	18 02 xb xb	5								
MUL	(A) × (B) ⇒ A:B	INH	12	3	–	–	–	–	–	–	–	Δ
	8 × 8 Unsigned Multiply											
NEG *opr*	0 – (M) ⇒ M *or* $(\overline{M})$ + 1 ⇒ M	EXT	70 hh ll	4	–	–	–	–	Δ	Δ	Δ	Δ
	2's Complement Negate	IDX	60 xb	3								
		IDX1	60 xb ff	4								
		IDX2	60 xb ee ff	5								
		[D,IDX]	60 xb	6								
		[IDX2]	60 xb ee ff	6								
NEGA	0 – (A) ⇒ A *equivalent to* $(\overline{A})$ + 1 ⇒ B	INH	40	1								
	Negate Accumulator A											
NEGB	0 – (B) ⇒ B *equivalent to* $(\overline{B})$ + 1 ⇒ B	INH	50	1								
	Negate Accumulator B											
NOP	No Operation	INH	A7	1	–	–	–	–	–	–	–	–
ORAA *opr*	(A) ✛ (M) ⇒ A	IMM	8A ii	1	–	–	–	–	Δ	Δ	0	–
	Logical OR A with Memory	DIR	9A dd	3								
		EXT	BA hh ll	3								
		IDX	AA xb	3								
		IDX1	AA xb ff	3								
		IDX2	AA xb ee ff	4								
		[D,IDX]	AA xb	6								
		[IDX2]	AA xb ee ff	6								
ORAB *opr*	(B) ✛ (M) ⇒ B	IMM	CA ii	1	–	–	–	–	Δ	Δ	0	–
	Logical OR B with Memory	DIR	DA dd	3								
		EXT	FA hh ll	3								
		IDX	EA xb	3								
		IDX1	EA xb ff	3								
		IDX2	EA xb ee ff	4								
		[D,IDX]	EA xb	6								
		[IDX2]	EA xb ee ff	6								
ORCC *opr*	(CCR) ✛ M ⇒ CCR	IMM	14 ii	1	⇑	–	⇑	⇑	⇑	⇑	⇑	⇑
	Logical OR CCR with Memory											

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
PSHA	$(SP) - 1 \Rightarrow SP; (A) \Rightarrow M_{(SP)}$ Push Accumulator A onto Stack	INH	36	2	–	–	–	–	–	–	–	–
PSHB	$(SP) - 1 \Rightarrow SP; (B) \Rightarrow M_{(SP)}$ Push Accumulator B onto Stack	INH	37	2	–	–	–	–	–	–	–	–
PSHC	$(SP) - 1 \Rightarrow SP; (CCR) \Rightarrow M_{(SP)}$ Push CCR onto Stack	INH	39	2	–	–	–	–	–	–	–	–
PSHD	$(SP) - 2 \Rightarrow SP; (A:B) \Rightarrow M_{(SP)}:M_{(SP+1)}$ Push D Accumulator onto Stack	INH	3B	2	–	–	–	–	–	–	–	–
PSHX	$(SP) - 2 \Rightarrow SP; (X_H:X_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$ Push Index Register X onto Stack	INH	34	2	–	–	–	–	–	–	–	–
PSHY	$(SP) - 2 \Rightarrow SP; (Y_H:Y_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$ Push Index Register Y onto Stack	INH	35	2	–	–	–	–	–	–	–	–
PULA	$(M_{(SP)}) \Rightarrow A; (SP) + 1 \Rightarrow SP$ Pull Accumulator A from Stack	INH	32	3	–	–	–	–	–	–	–	–
PULB	$(M_{(SP)}) \Rightarrow B; (SP) + 1 \Rightarrow SP$ Pull Accumulator B from Stack	INH	33	3	–	–	–	–	–	–	–	–
PULC	$(M_{(SP)}) \Rightarrow CCR; (SP) + 1 \Rightarrow SP$ Pull CCR from Stack	INH	38	3	Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ
PULD	$(M_{(SP)}:M_{(SP+1)}) \Rightarrow A:B; (SP) + 2 \Rightarrow SP$ Pull D from Stack	INH	3A	3	–	–	–	–	–	–	–	–
PULX	$(M_{(SP)}:M_{(SP+1)}) \Rightarrow X_H:X_L; (SP) + 2 \Rightarrow SP$ Pull Index Register X from Stack	INH	30	3	–	–	–	–	–	–	–	–
PULY	$(M_{(SP)}:M_{(SP+1)}) \Rightarrow Y_H:Y_L; (SP) + 2 \Rightarrow SP$ Pull Index Register Y from Stack	INH	31	3	–	–	–	–	–	–	–	–
REV[2]	MIN-MAX rule evaluation Find smallest rule input (MIN). Store to rule outputs unless fuzzy output is already larger (MAX). For rule weights see REVW. Each rule input is an 8-bit offset from the base address in Y. Each rule output is an 8-bit offset from the base address in Y. $FE separates rule inputs from rule outputs. $FF terminates the rule list. REV may be interrupted.	Special	18 3A	3 per rule byte	–	–	–	–	–	–	Δ	–

Table 4 Instruction Set Summary (Continued)

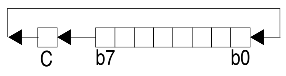
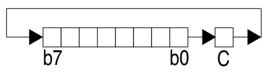
Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
REVW[2]	MIN-MAX rule evaluation Find smallest rule input (MIN), Store to rule outputs unless fuzzy output is already larger (MAX). Rule weights supported, optional. Each rule input is the 16-bit address of a fuzzy input. Each rule output is the 16-bit address of a fuzzy output. The value $FFFE separates rule inputs from rule outputs. $FFFF terminates the rule list. REVW may be interrupted.	Special	18 3B	3 per rule byte; 5 per wt.	–	–	?	–	?	?	Δ	!
ROL *opr* ROLA ROLB	 Rotate Memory Left through Carry Rotate A Left through Carry Rotate B Left through Carry	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	75 hh ll 65 xb 65 xb ff 65 xb ee ff 65 xb 65 xb ee ff 45 55	4 3 4 5 6 6 1 1	–	–	–	–	Δ	Δ	Δ	Δ
ROR *opr* RORA RORB	 Rotate Memory Right through Carry Rotate A Right through Carry Rotate B Right through Carry	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	76 hh ll 66 xb 66 xb ff 66 xb ee ff 66 xb 66 xb ee ff 46 56	4 3 4 5 6 6 1 1	–	–	–	–	Δ	Δ	Δ	Δ
RTC	$(M_{(SP)}) \Rightarrow$ PPAGE; $(SP) + 1 \Rightarrow$ SP; $(M_{(SP)}:M_{(SP+1)}) \Rightarrow PC_H:PC_L$; $(SP) + 2 \Rightarrow$ SP Return from Call	INH	0A	6	–	–	–	–	–	–	–	–
RTI	$(M_{(SP)}) \Rightarrow$ CCR; $(SP) + 1 \Rightarrow$ SP $(M_{(SP)}:M_{(SP+1)}) \Rightarrow$ B:A; $(SP) + 2 \Rightarrow$ SP $(M_{(SP)}:M_{(SP+1)}) \Rightarrow X_H:X_L$; $(SP) + 4 \Rightarrow$ SP $(M_{(SP)}:M_{(SP+1)}) \Rightarrow PC_H:PC_L$; $(SP) – 2 \Rightarrow$ SP $(M_{(SP)}:M_{(SP+1)}) \Rightarrow Y_H:Y_L$; $(SP) + 4 \Rightarrow$ SP Return from Interrupt	INH	0B	8	Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ
RTS	$(M_{(SP)}:M_{(SP+1)}) \Rightarrow PC_H:PC_L$; $(SP) + 2 \Rightarrow$ SP Return from Subroutine	INH	3D	5	–	–	–	–	–	–	–	–
SBA	(A) – (B) ⇒ A Subtract B from A	INH	18 16	2	–	–	–	–	Δ	Δ	Δ	Δ
SBCA *opr*	(A) – (M) – C ⇒ A Subtract with Borrow from A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	82 ii 92 dd B2 hh ll A2 xb A2 xb ff A2 xb ee ff A2 xb A2 xb ee ff	1 3 3 3 3 4 6 6	–	–	–	–	Δ	Δ	Δ	Δ

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~1	S	X	H	I	N	Z	V	C
SBCB *opr*	(B) – (M) – C ⇒ B Subtract with Borrow from B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C2 ii D2 dd F2 hh ll E2 xb E2 xb ff E2 xb ee ff E2 xb E2 xb ee ff	1 3 3 3 3 4 6 6	–	–	–	–	Δ	Δ	Δ	Δ
SEC	1 ⇒ C *Translates to* ORCC #$01	IMM	14 01	1	–	–	–	–	–	–	–	1
SEI	1 ⇒ I; (inhibit I interrupts) *Translates to* ORCC #$10	IMM	14 10	1	–	–	–	1	–	–	–	–
SEV	1 ⇒ V *Translates to* ORCC #$02	IMM	14 02	1	–	–	–	–	–	–	1	–
SEX *r1, r2*	$00:(r1) ⇒ r2 if r1, bit 7 is 0 *or* $FF:(r1) ⇒ r2 if r1, bit 7 is 1 Sign Extend 8-bit r1 to 16-bit r2 r1 may be A, B, or CCR r2 may be D, X, Y, or SP *Alternate mnemonic for* TFR r1, r2	INH	B7 eb	1	–	–	–	–	–	–	–	–
STAA *opr*	(A) ⇒ M Store Accumulator A to Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5A dd 7A hh ll 6A xb 6A xb ff 6A xb ee ff 6A xb 6A xb ee ff	2 3 2 3 3 5 5	–	–	–	–	Δ	Δ	0	–
STAB *opr*	(B) ⇒ M Store Accumulator B to Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5B dd 7B hh ll 6B xb 6B xb ff 6B xb ee ff 6B xb 6B xb ee ff	2 3 2 3 3 5 5	–	–	–	–	Δ	Δ	0	–
STD *opr*	(A) ⇒ M, (B) ⇒ M+1 Store Double Accumulator	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5C dd 7C hh ll 6C xb 6C xb ff 6C xb ee ff 6C xb 6C xb ee ff	2 3 2 3 3 5 5	–	–	–	–	Δ	Δ	0	–
STOP[2]	(SP) – 2 ⇒ SP; RTN_H:RTN_L ⇒ $M_{(SP)}$:$M_{(SP+1)}$; (SP) – 2 ⇒ SP; (Y_H:Y_L) ⇒ $M_{(SP)}$:$M_{(SP+1)}$; (SP) – 2 ⇒ SP; (X_H:X_L) ⇒ $M_{(SP)}$:$M_{(SP+1)}$; (SP) – 2 ⇒ SP; (B:A) ⇒ $M_{(SP)}$:$M_{(SP+1)}$; (SP) – 1 ⇒ SP; (CCR) ⇒ $M_{(SP)}$; STOP All Clocks If S control bit = 1, the STOP instruction is disabled and acts like a two-cycle NOP. Registers stacked to allow quicker recovery by interrupt.	INH	18 3E	9 +5 or +2	–	–	–	–	–	–	–	–

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
STS *opr*	$(SP_H:SP_L) \Rightarrow M:M+1$ Store Stack Pointer	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5F dd 7F hh ll 6F xb 6F xb ff 6F xb ee ff 6F xb 6F xb ee ff	2 3 2 3 3 5 5	–	–	–	–	Δ	Δ	0	–
STX *opr*	$(X_H:X_L) \Rightarrow M:M+1$ Store Index Register X	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5E dd 7E hh ll 6E xb 6E xb ff 6E xb ee ff 6E xb 6E xb ee ff	2 3 2 3 3 5 5	–	–	–	–	Δ	Δ	0	–
STY *opr*	$(Y_H:Y_L) \Rightarrow M:M+1$ Store Index Register Y	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5D dd 7D hh ll 6D xb 6D xb ff 6D xb ee ff 6D xb 6D xb ee ff	2 3 2 3 3 5 5	–	–	–	–	Δ	Δ	0	–
SUBA *opr*	$(A) – (M) \Rightarrow A$ Subtract Memory from Accumulator A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	80 ii 90 dd B0 hh ll A0 xb A0 xb ff A0 xb ee ff A0 xb A0 xb ee ff	1 3 3 3 3 4 6 6	–	–	–	–	Δ	Δ	Δ	Δ
SUBB *opr*	$(B) – (M) \Rightarrow B$ Subtract Memory from Accumulator B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C0 ii D0 dd F0 hh ll E0 xb E0 xb ff E0 xb ee ff E0 xb E0 xb ee ff	1 3 3 3 3 4 6 6	–	–	–	–	Δ	Δ	Δ	Δ
SUBD *opr*	$(D) – (M:M+1) \Rightarrow D$ Subtract Memory from D (A:B)	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	83 jj kk 93 dd B3 hh ll A3 xb A3 xb ff A3 xb ee ff A3 xb A3 xb ee ff	2 3 3 3 3 4 6 6	–	–	–	–	Δ	Δ	Δ	Δ
SWI	$(SP) – 2 \Rightarrow SP;$ $RTN_H:RTN_L \Rightarrow M_{(SP)}:M_{(SP+1)};$ $(SP) – 2 \Rightarrow SP; (Y_H:Y_L) \Rightarrow M_{(SP)}:M_{(SP+1)};$ $(SP) – 2 \Rightarrow SP; (X_H:X_L) \Rightarrow M_{(SP)}:M_{(SP+1)};$ $(SP) – 2 \Rightarrow SP; (B:A) \Rightarrow M_{(SP)}:M_{(SP+1)};$ $(SP) – 1 \Rightarrow SP; (CCR) \Rightarrow M_{(SP)}$ $1 \Rightarrow I$; (SWI Vector) $\Rightarrow$ PC Software Interrupt	INH	3F	9	–	–	–	1	–	–	–	–
TAB	$(A) \Rightarrow B$ Transfer A to B	INH	18 0E	2	–	–	–	–	Δ	Δ	0	–

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
TAP	(A) ⇒ CCR *Translates to* TFR A , CCR	INH	B7 02	1	Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ
TBA	(B) ⇒ A Transfer B to A	INH	18 0F	2	–	–	–	–	Δ	Δ	0	–
TBEQ *cntr*, *rel*	If (cntr) = 0, then Branch; else Continue to next instruction Test Counter and Branch if Zero (cntr = A, B, D, X,Y, or SP)	REL (9-bit)	04 lb rr	3	–	–	–	–	–	–	–	–
TBL *opr*	(M) + [(B) × ((M+1) – (M))] ⇒ A 8-Bit Table Lookup and Interpolate Initialize B, and index before TBL. <ea> points at first 8-bit table entry (M) and B is fractional part of lookup value. (no indirect addressing modes allowed.)	IDX	18 3D xb	8	–	–	–	–	Δ	Δ	–	?
TBNE *cntr*, *rel*	If (cntr) not = 0, then Branch; else Continue to next instruction Test Counter and Branch if Not Zero (cntr = A, B, D, X,Y, or SP)	REL (9-bit)	04 lb rr	3	–	–	–	–	–	–	–	–
TFR *r1, r2*	(r1) ⇒ r2 *or* $00:(r1) ⇒ r2 *or* (r1[7:0]) ⇒ r2 Transfer Register to Register r1 and r2 may be A, B, CCR, D, X, Y, or SP	INH	B7 eb	1	– or Δ	– ⇓	– Δ	– Δ	– Δ	– Δ	– Δ	– Δ
TPA	(CCR) ⇒ A *Translates to* TFR CCR , A	INH	B7 20	1	–	–	–	–	–	–	–	–
TRAP	(SP) – 2 ⇒ SP; $RTN_H:RTN_L \Rightarrow M_{(SP)}:M_{(SP+1)}$; (SP) – 2 ⇒ SP; $(Y_H:Y_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$; (SP) – 2 ⇒ SP; $(X_H:X_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$; (SP) – 2 ⇒ SP; (B:A) ⇒ $M_{(SP)}:M_{(SP+1)}$; (SP) – 1 ⇒ SP; (CCR) ⇒ $M_{(SP)}$ 1 ⇒ I; (TRAP Vector) ⇒ PC Unimplemented opcode trap	INH	18 tn tn = $30–$39 or $40–$FF	10	0	0	0	1	0	0	0	0
TST *opr* TSTA TSTB	(M) – 0 Test Memory for Zero or Minus (A) – 0 Test A for Zero or Minus (B) – 0 Test B for Zero or Minus	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	F7 hh ll E7 xb E7 xb ff E7 xb ee ff E7 xb E7 xb ee ff 97 D7	3 3 3 4 6 6 1 1	–	–	–	–	Δ	Δ	0	0
TSX	(SP) ⇒ X *Translates to* TFR SP,X	INH	B7 75	1	–	–	–	–	–	–	–	–
TSY	(SP) ⇒ Y *Translates to* TFR SP,Y	INH	B7 76	1	–	–	–	–	–	–	–	–
TXS	(X) ⇒ SP *Translates to* TFR X,SP	INH	B7 57	1	–	–	–	–	–	–	–	–
TYS	(Y) ⇒ SP *Translates to* TFR Y,SP	INH	B7 67	1	–	–	–	–	–	–	–	–

Table 4 Instruction Set Summary (Continued)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	~[1]	S	X	H	I	N	Z	V	C
WAI[2]	(SP) – 2 ⇒ SP; RTN_H:RTN_L ⇒ $M_{(SP)}$:$M_{(SP+1)}$; (SP) – 2 ⇒ SP; (Y_H:Y_L) ⇒ $M_{(SP)}$:$M_{(SP+1)}$; (SP) – 2 ⇒ SP; (X_H:X_L) ⇒ $M_{(SP)}$:$M_{(SP+1)}$; (SP) – 2 ⇒ SP; (B:A) ⇒ $M_{(SP)}$:$M_{(SP+1)}$; (SP) – 1 ⇒ SP; (CCR) ⇒ $M_{(SP)}$; WAIT for interrupt	INH	3E	8 (in) + 5 (int)	– or – or –	– – 1	– – –	– 1 1	– – –	– – –	– – –	– – –
WAV[2]	$\sum_{i=1}^{B} S_i F_i \Rightarrow Y{:}D$ $\sum_{i=1}^{B} F_i \Rightarrow X$ Calculate Sum of Products and Sum of Weights for Weighted Average Calculation. Initialize B, X, and Y before WAV. B specifies number of elements. X points at first element in S_i list. Y points at first element in F_i list. All S_i and F_i elements are 8-bits. If interrupted, 6 extra bytes of stack used for intermediate values	Special	18 3C	8 per lable	–	–	?	–	?	∆	?	?
wavr[2] pseudo-instruction	*see* WAV Resume executing an interrupted WAV instruction (recover intermediate results from stack rather than initializing them to 0)	Special	3C		–	–	?	–	?	∆	?	?
XGDX	(D) ⇔ (X) *Translates to* EXG D, X	INH	B7 C5	1	–	–	–	–	–	–	–	–
XGDY	(D) ⇔ (Y) *Translates to* EXG D, Y	INH	B7 C6	1	–	–	–	–	–	–	–	–

Notes:
1. Each cycle (~) is typically 125ns for an 8MHz bus (16MHz oscillator).
2. Refer to *CPU12 Reference Manual* for additional information.

freescale™ semiconductor